Convey Reference Manual

September 2009

Version 2.00

900-000001-000

Revisions

Version	Description
1.0	May 2008. Original printing.
1.05	December 2008. Preliminary release for customers.
2.0	September 2009. Split base architecture and individual personalities into separate documents.

Table of Contents

1 Overview

1.1 Introduction

This manual describes the Convey Computer Architecture. The Convey Computer Architecture integrates two types of processor architecture in one system: the Intel 64 architecture implemented by an Intel microprocessor and a reconfigurable architecture implemented as a coprocessor designed and implemented by Convey.

To serve as an explanation of a coprocessor within the Convey architecture the following is provided by the Wikipedia entry for the Intel 8087 coprocessor (produced in 1980).

> “The 8087 differed from subsequent Intel coprocessors in that it was directly connected to the address and data buses. The 8088/86 looked for instructions that commenced with the '11011' sequence and relinquished control to the coprocessor. The coprocessor handed control back once the sequence of coprocessor instructions ended. “(http://en.wikipedia.org/wiki/Intel_8087)

The Convey coprocessor implements multiple instruction sets (referred to as *personalities*) by incorporating FPGAs within the hardware system architecture. To the user, one integrated instruction set is provided. One executable image is produced. To provide this capability, the Convey coprocessor shares the physical address and data busses of the x86 processor, and is cache coherent with the x86. To the x86 memory system, the Convey coprocessor acts and looks like another x86 processor. This sharing means that x86 and the coprocessor share the same virtual address space. Instruction dispatch is achieved by the x86 storing coprocessor instructions in a shared area of virtual memory. When coprocessor instructions exist in this area; the coprocessor begins executing these instructions. Both the x86 *host processor* and the coprocessor are needed to completely execute an application.

In the Convey system architecture, each application can have instructions that are unique to its application. Potentially each application could have its own instruction set implemented as a Convey personality. In practice, Convey is implementing a series of personalities with instruction sets that are optimized for certain types of algorithms, and can support multiple applications that use those types of algorithms. More specialized personalities may be developed by Convey or its partners and customers that address specific high value applications.

Examples could be:

- Digital Signal Processing
    - 32 bit floating point
    - ALU structures and instructions that optimally execute operations like: FFT, Convolution, Filters, and Pre-Stack Migration
- Finite Element, Computational Fluid Dynamics, and Financial Modeling
    - 64 bit floating point

  - ALU structures and instructions that optimally perform high-speed operations on sparse data structures and the operations that are specified in industry standard libraries like LAPACK
  - Optimal support for solving the Black-Sholes Equation
- Data Mining
  - Integer Data Type and Tree Data Structures
    - 32 and 64 bit integer and logical
    - Memory operations that traverse Complex Tree structures
- Cryptography
  - Integer and Logical Data Structures
  - Instructions that perform operations on extremely long bit strings

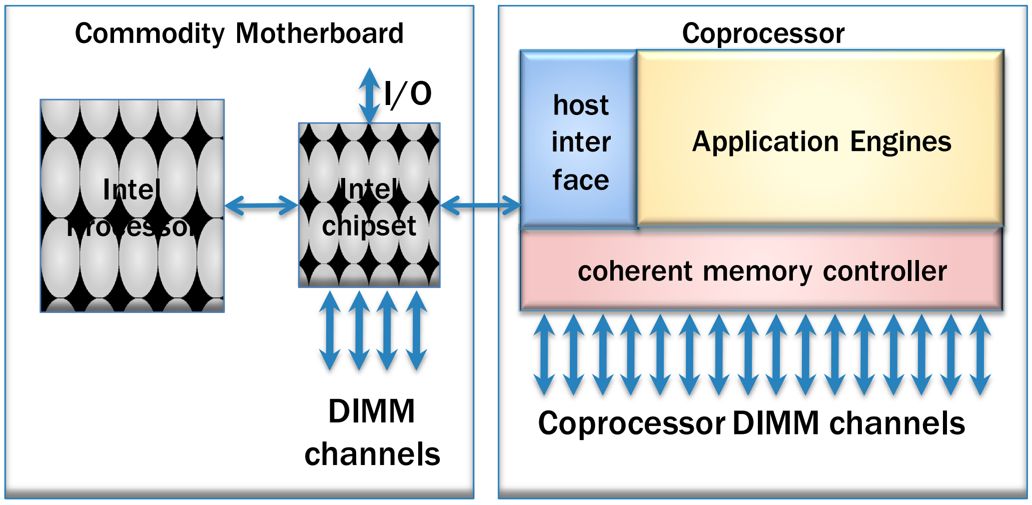
In the above examples, the 4 instruction sets are mutually exclusive. However they do share several things in common.

- The coprocessor instructions needed to reference virtual memory. These load and store instructions adhere to the protocols established by INTEL. This is necessary for cache coherency, sharing the same process space as defined by the x86 and for the creation of one executable image. One of the benefits of this approach is the elimination of superfluous data movement from main memory. Since the physical memory between the x86 and the coprocessor are common and shared, only the data that is referenced is loaded into the internal state of either the x86 or the coprocessor. The memory reference load and store models of the x86 and Convey coprocessor are identical.
- The software development environment is common across all supported instruction sets. There is one compiler that supports multiple machine state models. Each machine state model is an instruction set. This approach provides the same level of software productivity as experienced on an x86 microprocessor. In fact, the Convey compilers, to support the creation of one executable image, generate both x86 and coprocessor instructions.

2 HC-1 System Organization

2.1 System Architecture

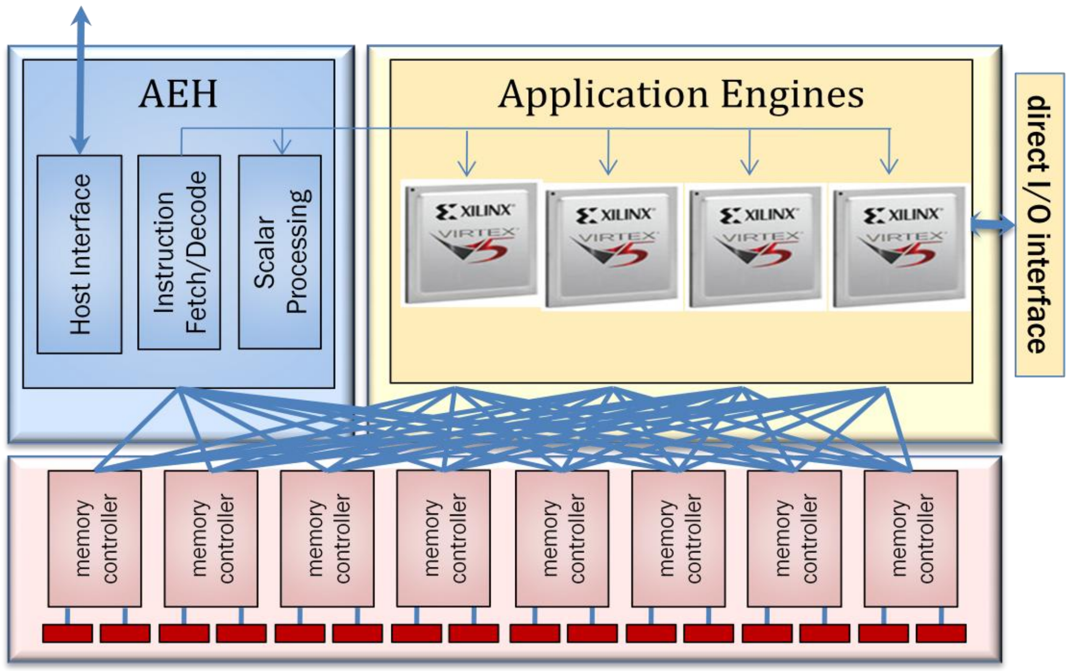
Convey HC-1 systems utilize a commodity two socket motherboard that includes an Intel 64 host processor and standard Intel I/O chipset, along with a reconfigurable coprocessor based on FPGA technology connected to the second processor socket. This coprocessor can be reloaded dynamically while the system is running with instructions that are optimized for different workloads. It includes its own high bandwidth memory subsystem that is incorporated into the Intel coherent global memory space. Coprocessor instructions can therefore be thought of as extensions to the Intel instruction set – an executable can contain both Intel and coprocessor instructions, and those instructions exist in the same virtual and physical address space.



The coprocessor supports multiple instruction sets (referred to as "personalities"). Each personality includes a base set of instructions that are common to all personalities. This base set includes instructions that perform scalar operations on integer and floating point data, address computations, conditionals and branches, as well as miscellaneous control and status operations. A personality also includes a set of extended instructions that are designed for a particular workload. The extended instructions for a personality designed for signal processing, for instance, may implement a SIMD model and include vector instructions for 32-bit complex arithmetic.

2.2 Coprocessor

The coprocessor has three major sets of components, referred to as the Application Engine Hub (AEH), the Memory Controllers (MCs), and the Application Engines (AEs).



The AEH is the central hub for the coprocessor. It implements the interface to the host processor and to the Intel I/O chipset, fetches and decodes instructions, and executes scalar instructions (see Personalities below). It processes coherence and data requests from the host processor, routing requests for addresses in coprocessor memory to the MCs. A cache for memory requests from the coprocessor to host memory reduces latency for remote references. Scalar instructions are executed in the AEH, while extended instructions are passed to the AEs for execution.

To support the bandwidth demands of the coprocessor, 8 Memory Controllers support a total of 16 DDR2 memory channels, providing an aggregate of over 80GB/sec of bandwidth to ECC protected memory. The MCs translate virtual to physical addresses on behalf of the AEs, and include snoop filters to minimize snoop traffic to the host processor. The Memory Controllers support standard DIMMs as well as Convey designed Scatter-Gather DIMMs. The Scatter-Gather DIMMs are optimized for transfers of 8-byte bursts, and provide near peak bandwidth for non-sequential 8-byte accesses. The coprocessor therefore not only has a much higher peak bandwidth than is available to commodity processors, but also delivers a much higher percentage of that peak for non-sequential accesses.

Together the AEH and the MC’s implement features that are present in all personalities. This ensures that important features such as memory protection, access to coprocessor memory, and communication with the host processor are always available.

The Application Engines (AEs) implement the extended instructions that deliver performance for a personality. The AEs are connected to the AEH by a command bus that transfers opcodes and scalar operands, and to the memory controllers via a network of point-to-point links that provide very high sustained bandwidth. Each AE instruction is passed to all four AEs. How they process the instructions depends on the personality. For instance, a personality that implements a vector model might implement multiple arithmetic pipelines in each AE, and divide the elements of a vector across all the pipelines to be processed in parallel.

2.3 Personalities

A personality defines the set of instructions supported by the coprocessor and their behavior. A system may have multiple personalities installed and can switch between them dynamically to execute different types of code, but only one personality is active on a coprocessor at any one time. Each installed personality includes the loadable bit files that implement a coprocessor instruction set, a list of the instructions supported by that personality, and an ID used by the application to load the correct image at runtime.

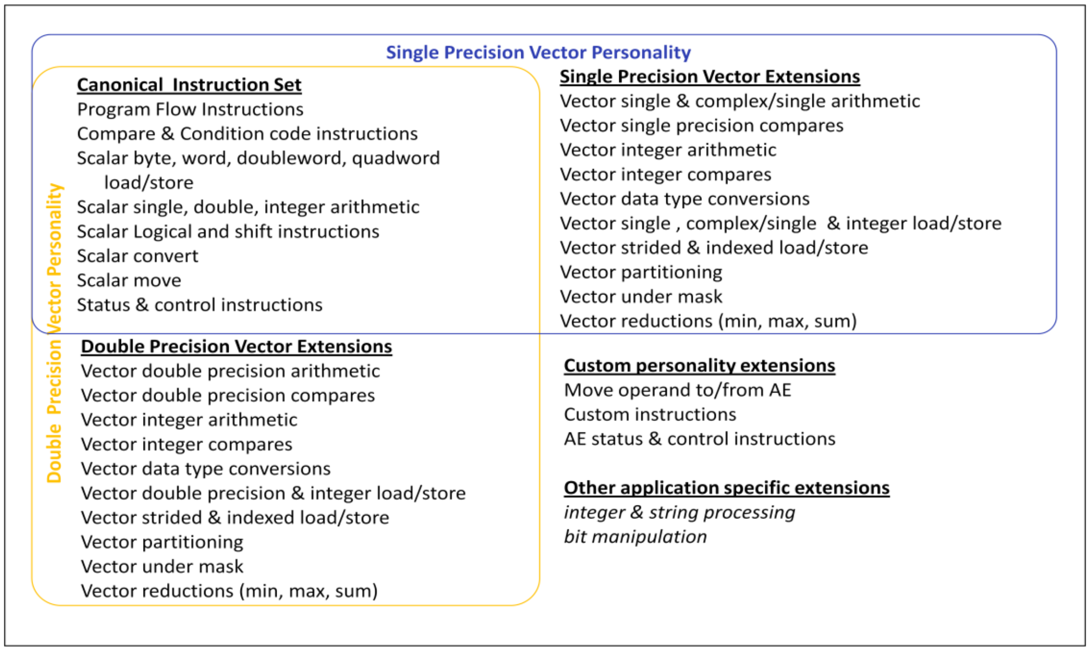
All personalities implement a base set of instructions referred to as the scalar instruction set. These instructions include scalar arithmetic, conditionals, branches, and other operations required to implement loops and manage the operation of the coprocessor. In addition to the scalar instructions, each personality includes extended instructions that may be unique to that personality. Extended instructions are designed for particular workloads, and may include only the operations that represent the largest portion of the execution time for an application. For instance, a personality designed for seismic processing may implement 32-bit complex vector arithmetic instructions.

All personalities have some elements in common:

- Coprocessor execution is initiated and controlled via instructions, as defined by the Convey Instruction Set Architecture.
- All personalities use a common host interface to dispatch coprocessor instructions and return status. This interface uses shared memory and leverages the cache coherency protocol to minimize latency.
- Coprocessor instructions use virtual addresses compatible with the Intel® 64 specification, and coherently share memory with the host processor. The host processor and I/O system can access coprocessor memory and the coprocessor can access host memory. The virtual memory implementation provides protection for process address spaces as in a conventional system.
- All personalities support a common set of instructions called the scalar instruction set. These instructions implement basic scalar and control flow operations required to implement interfaces and manage the operation of the AEs.

These common elements ensure that compilers and other tools can be leveraged across multiple personalities, while still allowing customization for different workloads.

The following figure shows some of the types of instructions that might be included in different personalities:

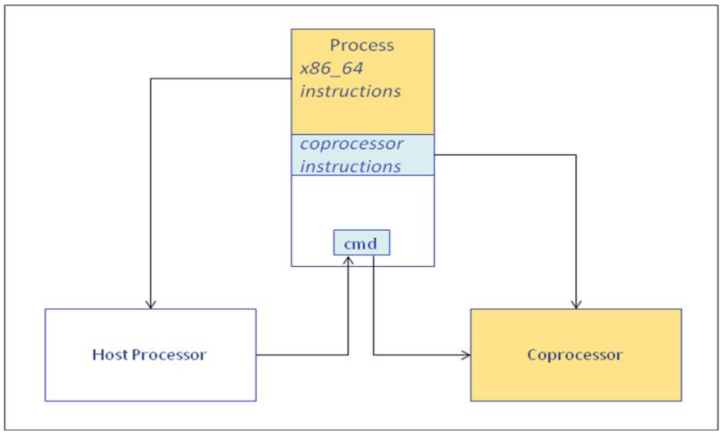

Single Precision Vector Personality
Canonical Instruction Set
Program Flow Instructions
Compare & Condition code instructions
Scalar byte, word, doubleword, quadword load/store
Scalar single, double, integer arithmetic
Scalar Logical and shift instructions
Scalar convert
Scalar move
Status & control instructions
Single Precision Vector Extensions
Vector single & complex/single arithmetic
Vector single precision compares
Vector integer arithmetic
Vector integer compares
Vector data type conversions
Vector single , complex/single & integer load/store
Vector strided & indexed load/store
Vector partitioning
Vector under mask
Vector reductions (min, max, sum)
Double Precision Vector Personality
Double Precision Vector Extensions
Vector double precision arithmetic
Vector double precision compares
Vector integer arithmetic
Vector integer compares
Vector data type conversions
Vector double precision & integer load/store
Vector strided & indexed load/store
Vector partitioning
Vector under mask
Vector reductions (min, max, sum)
Custom personality extensions
Move operand to/from AE
Custom instructions
AE status & control instructions
Other application specific extensions
integer & string processing
bit manipulation

The Convey Instruction Set Architecture includes all instructions implemented for all personalities. Each personality, therefore, implements a subset of the entire instruction set.

The user defined personality extension is different from the other extensions in that it includes instructions whose behavior is not specified by the architecture. This extension provides an architected interface to user defined logic within the AEs. Instructions are provided to transfer data and status to or from the AEs and a set of generic instructions that initiate execution. The behavior of these generic instructions is implementation dependent, and could be as simple as a single instruction that executes an entire algorithm. Like all personalities, user defined personalities support the address and scalar instruction set and register model, support virtual memory, and use the common dispatch and return model. This allows the user defined logic to execute within the confines of a Linux process, sharing a protected virtual address space with an x86 application.

2.4 Coprocessor Programming Model

Coprocessor instructions are considered extensions to the Intel 64 instruction set, in a manner analogous to the way an 8087 coprocessor extended the 8086 instruction set. The instruction set available to a thread executing on a Convey system is therefore a superset of the Intel 64 instruction set. Processes that use the coprocessor contain both Intel 64 host processor and coprocessor instructions and transfer control between the two via a mechanism that utilizes the cache coherence hardware to minimize latency. Since the coprocessor and the host processor share the same coherent view of memory, this transfer of control is seamless and does not require copying data. For performance reasons, however, it may be desirable to place data in the memory region associated with the host processor or coprocessor that uses it most frequently (similar to other NUMA architectures). The Convey operating system and compilers provide mechanisms to allocate data in the appropriate memory, and/or migrate data at runtime.

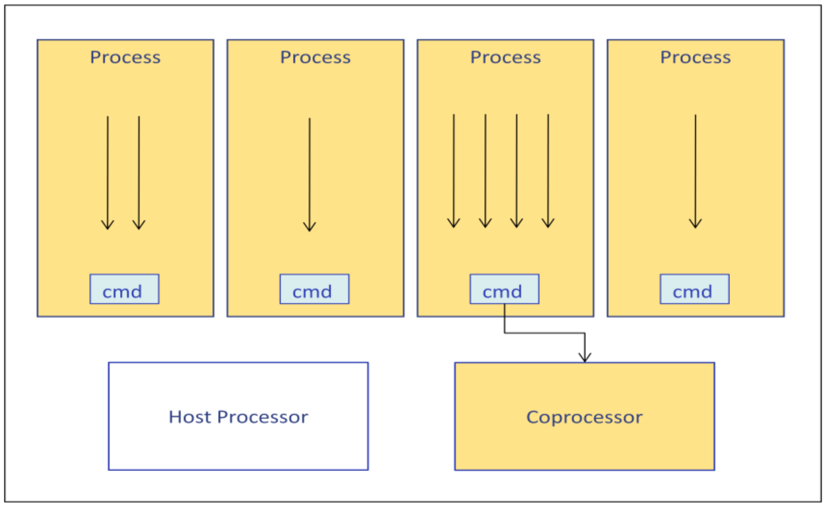


Dispatching work to the coprocessor requires the following steps:

- Attach a process to the coprocessor, allocating a command and status area in the process' address space.
- Load any required personalities into the coprocessor personality cache.
- Dispatch a block of instructions to the coprocessor by writing the address of the coprocessor code region and any arguments to be loaded into coprocessor registers into a command block in the command and status area
- The host processor can continue executing asynchronously, or wait for completion of the command to be posted by the coprocessor into the status block associated with the command block.

Shared library routines that perform the above functions are automatically inserted into an application if compiled for the coprocessor.

The coprocessor is a shared resource: it executes one dispatch at a time, but multiple threads can share the coprocessor by queuing requests.



Each process is given its own command area, with two command blocks to queue coprocessor dispatch requests. Multiple threads within a process queue dispatch requests on the command blocks within that process' command area. Different dispatch modes instruct the coprocessor to empty the command queue within the current process or examine the command queues in different processes on a round robin basis.

Since the system supports a limited number of command pages, and the coprocessor contains a significant amount of state in the form of cached personalities, address translations, and register contents, the coprocessor may only be allocated by processes executing as part of the same application.

The combination of parallelization of an application on multiple host processors and execution of key loops or kernels on the coprocessor increases overall utilization by ensuring the coprocessor always has work queued, and by allowing multiple host processors to execute non-coprocessor code in parallel.

3 Data Types

The set of data types handled by the coprocessor were chosen to reflect the data types provided by high level programming languages that are used by Server Class Computing applications (C, C++ and FORTRAN).

The sections below talk about three sets of data types: fundamental, numeric and native. The fundamental data types represent the sizes of operands that the coprocessor is able to read and write from memory. The numeric data types are the set of numeric data representations that the coprocessor recognizes. The native data types are the subset of numeric data types that the coprocessor instruction set is able to manipulate with arithmetic and logical operations.

3.1 Fundamental Data Types

The fundamental data types represent the size of operands that the coprocessor reads and writes from memory. The fundamental data types are listed in the table below with the required memory alignment.

Fundamental Data Type	*Data Size (in bytes)*	*Alignment Requirement*
Byte	1	Any
Word	2	2
Double Word	4	4
Dual Double Word	8	4
Quad Word	8	8

Table 1 - Fundamental Data Types

Note that all fundamental data types must be aligned on a natural boundary for that data type except Dual Double Words. Dual Double Words are able to be aligned on any 4-byte boundary. Accesses to memory on unaligned boundaries cause a trap to the operating system.

The figure below shows how the fundamental data types can be located in memory.

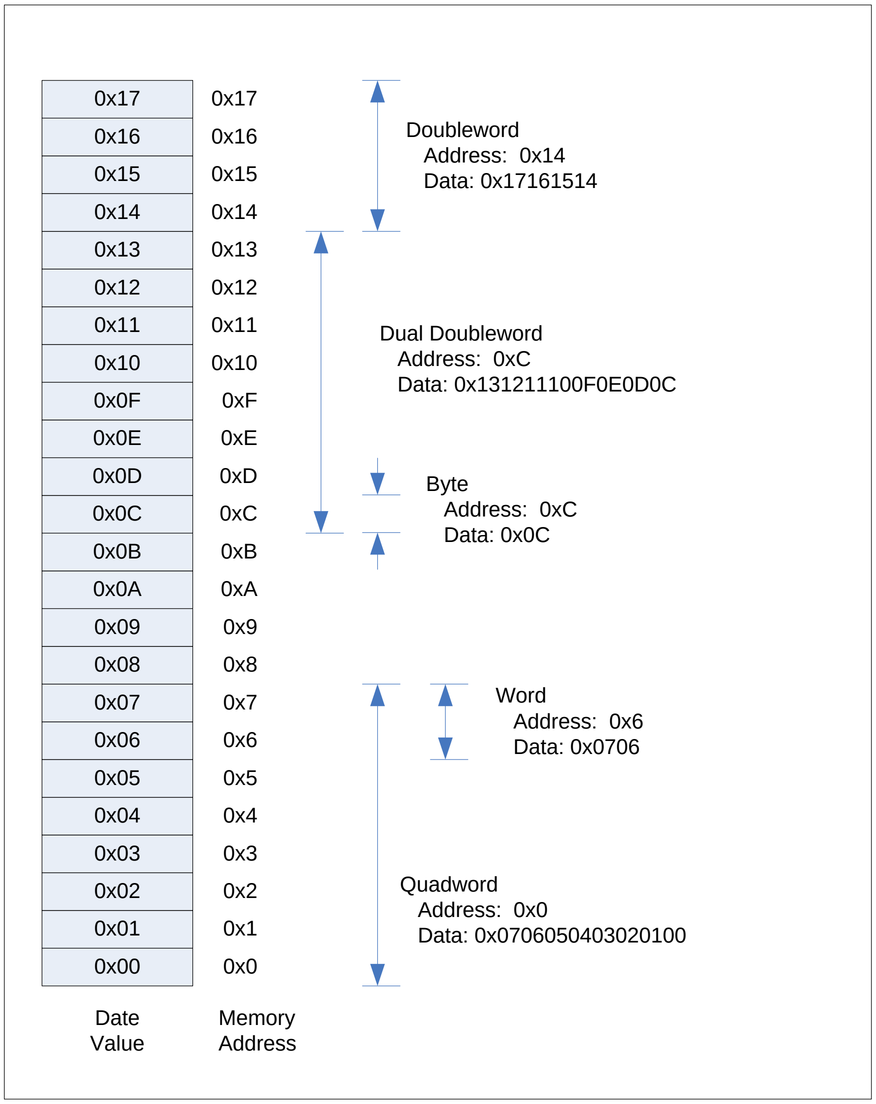


Figure 1 - Fundamental Data Types in Memory

3.2 Numeric Data Types

The following table lists the numeric data types and abbreviations used by assembly level instructions.

Data Type	*Opcode Suffix*	*Storage Width*	*Range of Values*
Signed Byte Integer	SB	8-bits	-128 to 127
Unsigned Byte Integer	UB	8-bits	0 to 255
Signed Word Integer	SW	16-bits	-32768 to 32767
Unsigned Word Integer	UW	16-bits	0 to 65535
Signed Double Word Integer	SD	32-bits	-2147483648 to 2147483647
Unsigned Double Word Integer	UD	32-bits	0 to 4294967295
Signed Quad Word Integer	SQ	64-bits	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
Unsigned Quad Word Long	UQ	64-bits	18,446,744,073,709,551,615
Single Precision IEEE Floating Point	FS	32-bits	FSmin To FSmax
Double Precision IEEE Floating Point	FD	64-bits	FDmin To FDmax
Complex Single Precision Floating Point	CS	64-bits	Two values, each FSmin To FSmax, Fundamental data type is Double Word allowing four byte memory alignment
Complex Double Precision Floating Point	CD	128-bits	Two values, each FDmin To FDmax, Fundamental data type is Quad Word allowing eight byte memory alignment

Table 2 – Numeric Data Types

The figure below shows the format of the supported numeric data types.

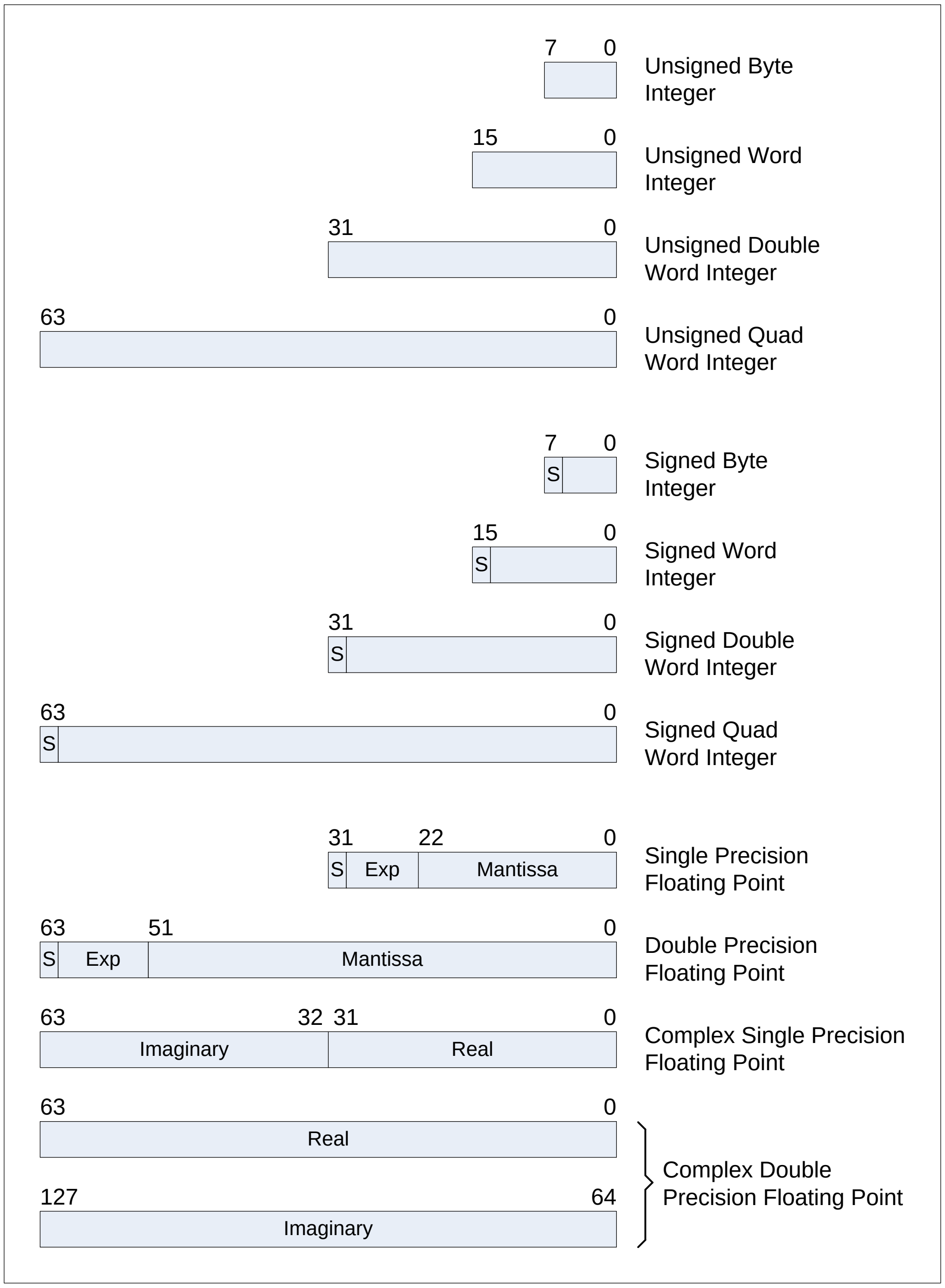


Figure 2 - Supported Numeric Data Types

Unsigned integer values are ordinal numbers from zero to the largest binary number that fits within the size of the data type. Integer numbers have both positive and negative values and are in 2's complement format. Floating point arithmetic adheres to the IEEE754 Standard.

3.3 Native Data Types

Data is accessed from memory in the fundamental memory units (1, 2, 4 or 8 bytes). Once written to a coprocessor internal register, all operations are performed on the native data types. The native data types include 64-bit integers, single precision, double precision, complex single precision and complex double precision.

The coprocessor loads integers from memory as 1, 2, 4 and 8 byte quantities. The data loaded from memory is either zero or sign extended to 64-bits as it is being written to an internal register. All integer operations are performed on the native 64-bit integer operand size. All arithmetic operations are checked for overflow/underflow for 64-bit values. Store operations store the native 64-bit values to the fundamental memory sizes of 1, 2, 4 or 8 bytes. Underflow/overflow is checked as the 64-bit values are read from the internal registers and converted to the smaller fundamental memory sizes.

3.4 Pointer Data Types

All pointers are 64-bits and are required to be in the IA32e 64-bit form. IA32e 64-bit form implies that virtual addresses are 64-bits, with bits 48-63 being the same value as bit 47. Since a pointer is an unsigned integer value, then with IA32e 64-bit addressing, half of the valid virtual address space is at the upper end of the 64-bit address range, and half is at the lower end of the 64-bit address range. The figure below illustrates the valid address range assuming the host processor supports a 48-bit virtual address. Future versions of processors that support the IA32e 64-bit architecture may support virtual addresses wider than 48-bits.

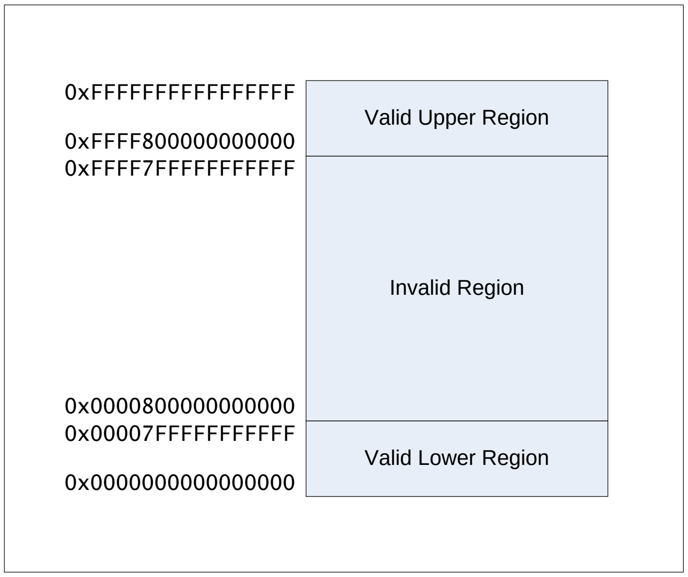


Figure 3 - Valid Virtual Address Regions

3.5 Floating Point Data Types

Floating point data types are defined for coprocessor personalities that support floating point arithmetic operations.

3.5.1 Floating Point Formats

The Convey coprocessor supports four floating point types:

- Single Precision
- Double Precision
- Complex Single Precision
- Complex Double Precision

Figure 2 above shows the formats for these data types. Note that Complex Single Precision is the placement of two single precision values in a single 64-bit data type. Similarly, Complex Double Precision is the placement of two double precision values in a 128-bit data type. Having a complex data types allow both values (real and imaginary) to be accessed with a single load or store instruction. Similarly, complex data types allow arithmetic instructions to reference both values by specifying a single scalar or vector register.

3.5.2 Supported Floating Point Encodings

The IEEE754 floating point standard defines the following encodings for the floating point data format.

- Signed Zero
- Denormalized Finite Numbers
- Normalized Finite Numbers
- Signed Infinites
- NaNs (Not a Number)

The following figure illustrates the use of these encodings to represent real number values.

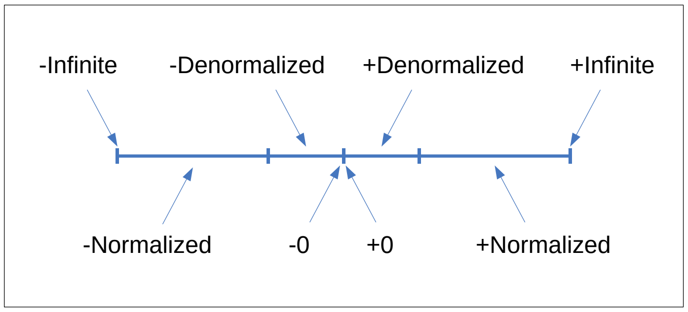


Figure 4 - Real Number Value Encodings

Real numbers are represented in the supported floating point formats as shown in the following table.

Read Number Value	Floating Point Format Encoding					
	Single Precision			Double Precision		
	Sign	Exponent	Mantissa	Sign	Exponent	Mantissa
-NAN	1	255	Not Zero	1	2047	Not Zero
-Infinite	1	255	0	1	2047	0
-Normalized	1	1-254	Any	1	1-2046	Any
-Denormalized	1	0	Not zero	1	0	Not zero
- Zero	1	0	0	1	0	0
+Zero	0	0	0	0	0	0
+Denormalized	0	0	Not zero	0	0	Not zero
+Normalized	0	1-254	Any	0	1-2046	Any
+Infinite	0	255	0	0	2047	0
+NAN	0	255	Not Zero	0	2047	Not Zero

Table 3 - Floating Point Format Encodings

The Convey coprocessor supports the most commonly used features of the IEEE754 standard. However, since the coprocessor is implemented using FPGAs, the full IEEE754 standard is not supported.

The Convey coprocessor **does not** support Denormalized Numbers. The Convey coprocessor arithmetic operations treat Denormalized Numbers as if they were zero. Note that a floating point underflow exception is detected when a floating point operation produces a value that would result in a Denormalized Number.

3.5.3 Supported Rounding Mode

The IEEE754 floating point standard defines four rounding modes:

- Round to Nearest (also known as unbiased rounding)
- Round Down (towards negative infinity)
- Round Up (towards positive infinity)
- Round to Zero (towards zero)

The standard defines the default mode as Round to Nearest. Round to Nearest is the **ONLY** mode supported by the Convey Coprocessor.

3.6 Floating Point Exceptions

Floating point exceptions are used to detect when either of two cases arise:

- The input operands for an arithmetic operation are illegal
- An arithmetic operation will produce results that cannot be represented by the specified floating point format

A status bit and a mask bit exists for each floating point exception source. For scalar instructions the status bits reside in the CPS (Coprocessor Status), and the mask bits reside in the CPC (Coprocessor Control). For application engine instructions the status bits reside in the AES (Application Engine Status), and the mask bits reside in the AEC (Application Engine Control).

The table below lists the supported floating point exceptions.

Exception Condition Name	*Status Field Name (CPS)*	*Mask Field Name (CPC)*
Scalar Floating Point Invalid Operation	SFIE	SFIM
Scalar Floating Point Denormalized Operand	SFDE	SFDM
Scalar Floating Point Divide by Zero	SFZE	SFZM
Scalar Floating Point Overflow	SFOE	SFOM
Scalar Floating Point Underflow	SFUE	SFUM
AE Floating Point Invalid Operation	AEFIE	AEFIM
AE Floating Point Denormalized Operand	AEFDE	AEFDM
AE Floating Point Divide by Zero	AEFZE	AEFZM
AE Floating Point Overflow	AEFOE	AEFOM
AE Floating Point Underflow	AEFUE	AEFUM

Table 4 - Supported Floating Point Exceptions

The following sections describe the supported floating point exceptions.

3.6.1 Invalid Floating Point Operation

An invalid operation exception occurs when the operands presented to an arithmetic unit are invalid for the operation being performed. An invalid operation exception is produced for the following conditions:

- Square root of a negative number
- NaN input operand
- Infinity divide by infinity
- Subtraction of same sign infinity operands

- Addition of different sign infinity operands

If the exception's mask bit is a zero then execution of the coprocessor is stopped and an interrupt is sent to the host processor. If the exception's mask bit is set, then the result produced by the arithmetic unit is a +NAN and coprocessor execution continues.

3.6.2 Denormalized Floating Point Operand

A denormalized operand exception occurs when an arithmetic unit input operand is denormalized (zero exponent with non-zero mantissa). If the exception's mask bit is a zero then execution of the coprocessor is stopped and an interrupt is sent to the host processor. If the exception's mask bit is set, then a result is produced by the arithmetic unit and coprocessor execution continues. The result produced is function unit dependent. The function units may:

- Perform function with the denormalized operation value
- Perform function with a zero value in place of the denormalized operand value

In some applications, substituting the value zero for the denormalized operand may produce acceptable results. If using zero instead of the denormalized operand is not acceptable, then the Denormalized Operand exception mask bit should have the value zero so that the exception terminates the application.

3.6.3 Floating Point Divide by Zero

A divide by zero exception occurs when a division instruction attempts to divide by zero.

If the exception's mask bit is a zero then execution of the coprocessor is stopped and an interrupt is sent to the host processor. If the exception's mask bit is set, then the result produced is NaN with the sign being the sign produced by the operation.

3.6.4 Floating Point Overflow

An overflow exception occurs when the value produced by an arithmetic unit cannot be represented in the selected floating point format due to the resulting exponent value being too large (greater than the largest representable exponent value minus one).

If the exception's mask bit is a zero then execution of the coprocessor is stopped and an interrupt is sent to the host processor. If the exception's mask bit is set, then the result produced is infinity with the sign being the sign produced by the operation.

3.6.5 Floating Point Underflow

An underflow exception occurs when the value produced by an arithmetic unit cannot be represented in the selected floating point format due to the resulting exponent value being too small (less than an exponent value of one). The Convey coprocessor does not support denormalized mantissa values. Allowing the application to proceed when an underflow condition is detected may result in the value zero being used for some future arithmetic operations that would not otherwise be a zero.

If the exception's mask bit is a zero then execution of the coprocessor is stopped and an interrupt is sent to the host processor. If the exception's mask bit is set, then the result produced is zero with the sign being the sign produced by the operation.

4 Addressing and Access Control

4.1 Introduction

The Convey Coprocessor implements virtual memory in a manner that creates a common linear address space between the coprocessor and the x86 host processor. This capability allows the host processor and the coprocessor to use virtual addresses to communicate the location of shared data structures or the starting address of a coprocessor routine to be executed.

The x86 host processor's architecture defines many address translation modes, privilege levels, and legacy compatibility mechanisms. The Convey coprocessor defines a single address translation mode (64-bit), minimal access protection checks, and no support for legacy compatibility with previous generations of x86 processors. All x86 applications that require legacy compatibility execute entirely on the x86 processor.

Coprocessor routines are dispatched in one of two modes, user and privileged. User mode makes all memory references through the application's virtual address space. In user mode, address translation and access protection checking are provided. Supervisor mode is used for saving and restoring coprocessor context to/from memory. In privileged mode, all address references (instruction and data) are physical.

All operating system calls are performed on the x86 processor. As such, the coprocessor does not support changing execution privilege levels. A user application is only able to dispatch a user privilege level routine. The operating system can dispatch a supervisor level routine (I.e. context save or restore). Neither a user level nor a privileged level routine can change the privilege level during execution. As a result, checking mechanisms to validate legal privilege level changes are not required.

4.2 Bit and Byte Ordering

The Convey Coprocessor architecture uses the little endian byte ordering model (the same model as is used on IA32 processors). This model has bits numbered from right to left, starting with the right most, least significant bit being bit zero. Similarly, bytes are numbered with the least significant byte of a data element being byte 0.

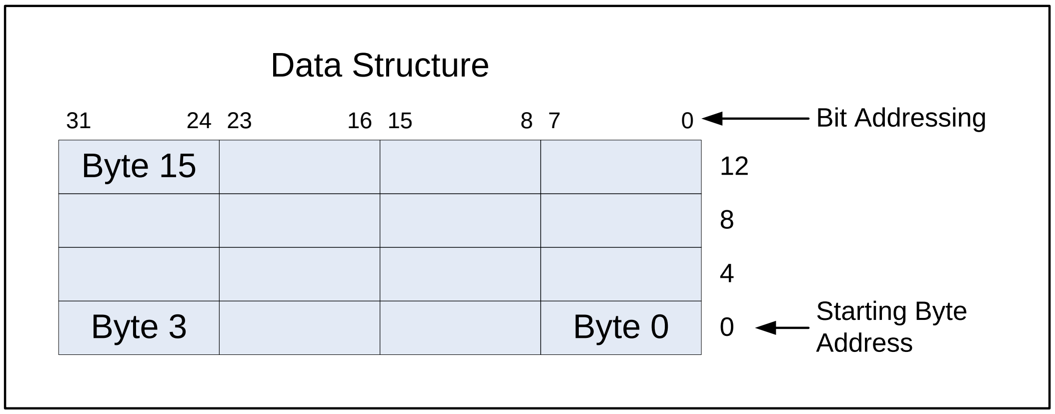


Figure 5 - Data Structure Bit and Byte Ordering

4.3 Pointers and Address Specification

All Convey coprocessor virtual addresses are 64-bits in width and must follow the Intel canonical form (see Section 3.4 Pointer Data Types).

4.4 Address Resolution and the TLB

The Convey coprocessor handles page faults to resolve address translations when the needed TLB entry is not present. The coprocessor includes a hardware based state machine that is able to walk memory based translation tables to obtain the needed TLB entry. The host processor and coprocessor share the same memory based address translation tables. The host processor manages and uses the translation tables, whereas the coprocessor only uses the translation tables to resolve pages faults.

Upon the occurrence of a page fault, the coprocessor first walks the physical memory based address translation tables. If the needed TLB entry is found then the coprocessor TLB is updated and coprocessor execution resumes. If the needed TLB entry is not found, then the coprocessor sends an interrupt to the host processor. The host processor must then handle the page fault by either determining that the page fault address is illegal for the coprocessor application (and terminating the application), or by adding the needed TLB entry to the memory based translation table (and telling the coprocessor to search the memory based translation tables again).

4.4.1 Supported Address Translation Mode

The x86 host processor supports multiple address translation modes and multiple page sizes. The coprocessor supports only one address translation mode with multiple page sizes. The supported address translation mode is referred to as IA-32e / 64-bit mode.

Applications that use the coprocessor must use IA-32e / 64-bit address translation mode. All other applications can run on the host processor without coprocessor support using any of the operating system supported address translation modes.

IA-32e / 64-bit mode is defined such that segmentation registers are not used.

IA-32e / 64-bit mode supports two pages sizes, 4K and 2M bytes. The Convey coprocessor supports all power-of-two pages sizes between 4K and 4M byte. The support of very large page sizes allows the coprocessor TLB to cover all physical memory.

4.4.2 IA-32e / 64-bit Address Translation Process

Figure 6 below shows the organization for the paging structures used for IA-32e / 64-bit mode.

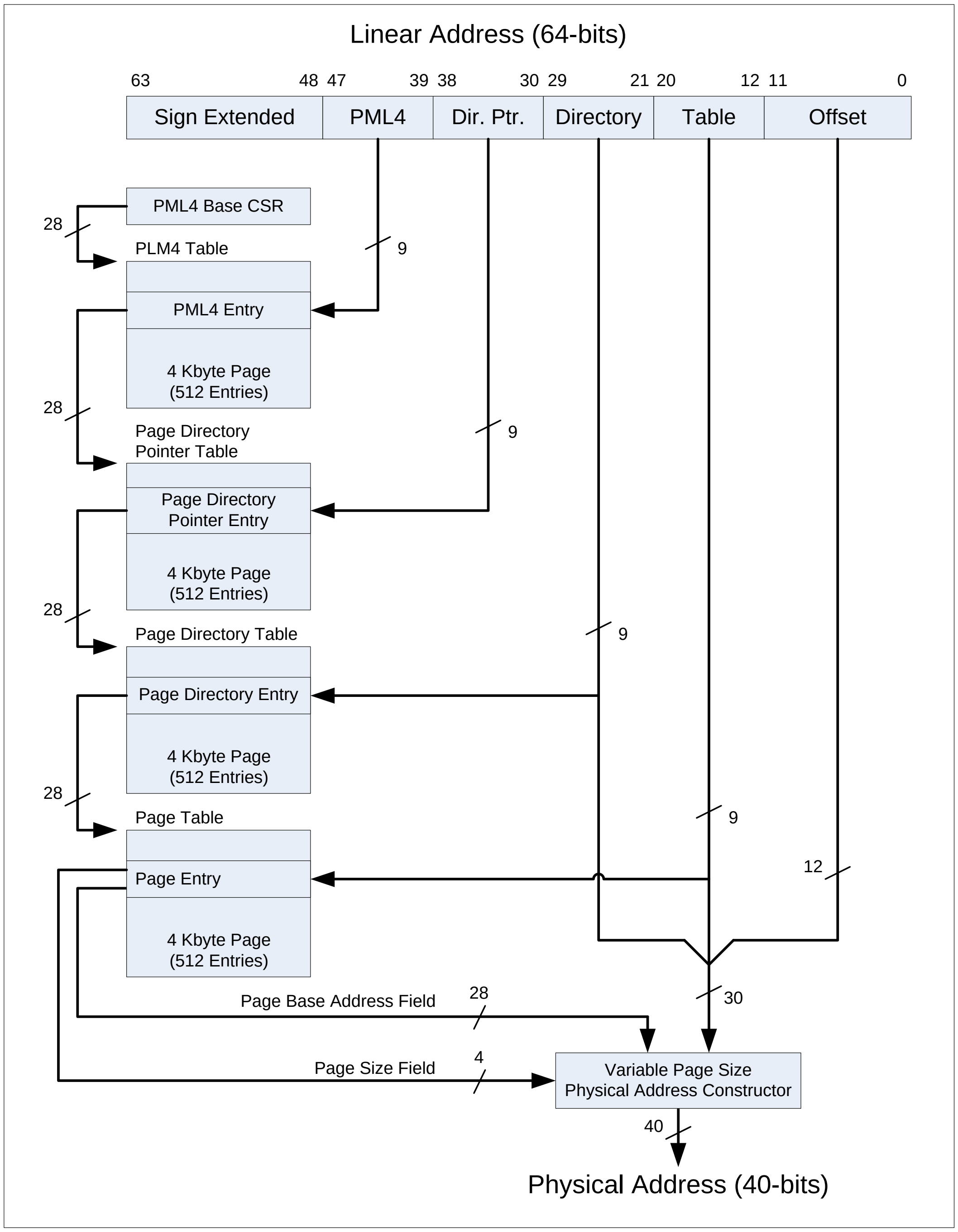


Figure 6 - IA-32e / 64-bit Mode Paging Structure Organization

The process of translating a virtual to physical address and the format of the required tables is described in the following sections.

Note that the above figure and the following sections refer to a 40-bit physical address. Future x86 processors may increase the width of the physical address to greater than 40 bits. Convey's future generation coprocessors will support the physical address width of Intel's latest generation processor. The number of memory references necessary to translate a 64-bit virtual address to a physical address is the same independent of the width of the physical address.

4.4.2.1 PML4 Base Address

Finding a TLB entry in memory for an application begins with the PML4 Base Address. This address specifies the root of the memory based address translation table for a process. Each application has a unique PML4 Base Address value. A supervisor command is used to set the PLM4 Base Address for the process currently executing on the coprocessor.

4.4.2.2 Page Map Level 4 Table

The PML4 Table provides the first level of table lookup for the virtual to physical address translation memory structure. There is a single PML4 table per application and has 512 8-byte entries. The table is indexed by bits 47-39 of the 64-bit virtual address. The format of a PML4 table entry is shown in Figure 7 below. Bits 39-12 of a PML4 entry specify the base address of the next level table (Page Directory Pointer Table).

4.4.2.3 Page Directory Pointer Table

The Page Directory Pointer Table provides the second level of table lookup for the virtual to physical address translation memory structure. There are at most 512 second level lookup tables. Each Page Directory Pointer Table has 512 8-byte entries. The table is indexed by bits 38-30 of the 64-bit virtual address. The format of the Page Directory Pointer Table entry is shown in Figure 7 below. Bits 39-12 of each entry specify the base address of the next level table (Page Directory Table).

4.4.2.4 Page Directory Table

The Page Directory Table provides the third level of table lookup for the virtual to physical address translation memory structure. There are at most 512 * 512 third level lookup tables. Each Page Directory Table has 512 8-byte entries. The table is indexed by bits 29-21 of the 64-bit virtual address. The format of the Page Directory Table entry is shown in Figure 7 below. Bits 39-12 of each entry specify the base address of a 2M Byte page of physical memory.

4.4.2.5 Page Table

The Page Table provides the last level of table lookup for the virtual to physical address translation memory structure. Each Page Table has 512 8-byte entries. The table is indexed by bits 20-12 of the 64-bit virtual address. The format of the Page Table entry is shown in Figure 7 below. Bits 39-12 of each entry specify the base address of a 4K Byte page of physical memory.

4.4.2.6 Page Table Entry Formats

The page table entry formats are defined by Intel's IA-32 architecture. This section lists the fields in each format, describes the field meaning, and indicates whether the coprocessor uses the field or ignores it. Figure 7 below shows the table entry formats.

The entries of the four levels of tables have similar formats with the only differences being the addition of three flag fields for last level page table.

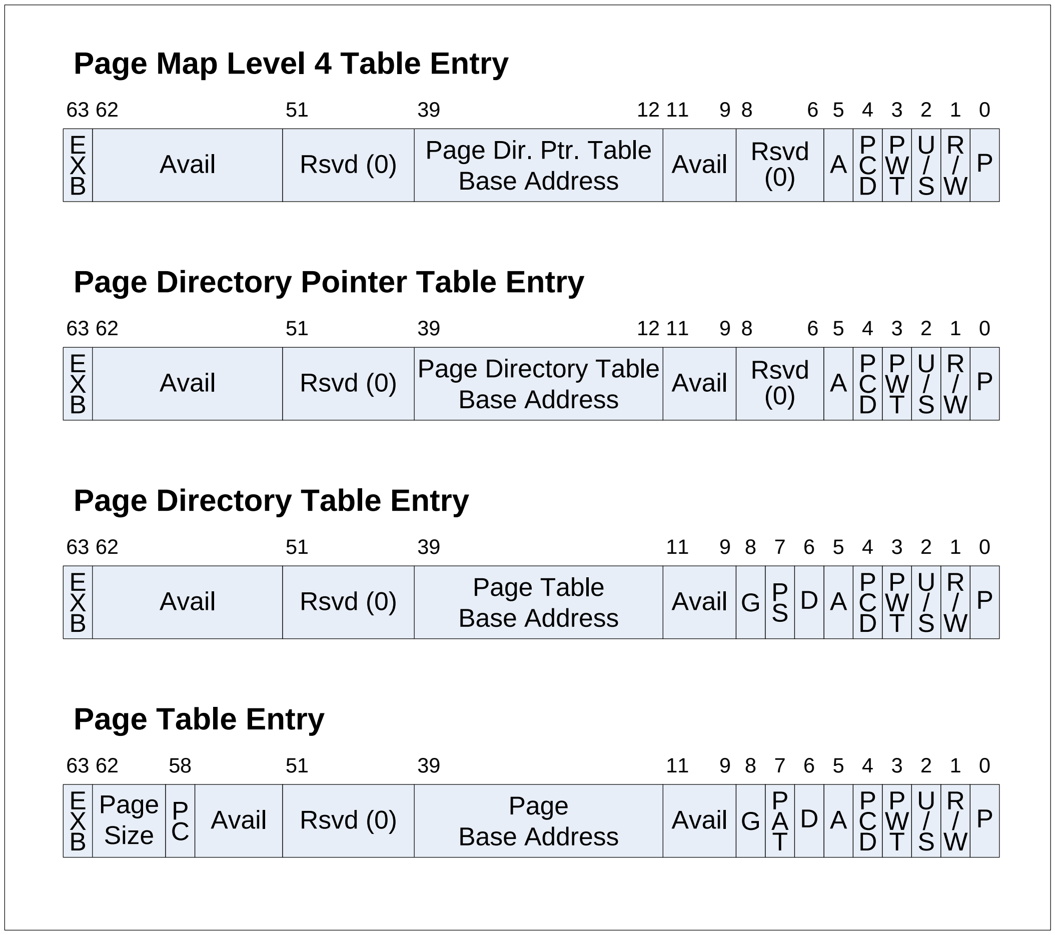


Figure 7 - Page Table Entry Formats

The various fields and their meanings are listed below:

Present (P) flag, bit 0

The P bit indicates whether a physical page exists in memory. The coprocessor sends a page fault interrupt to the host processor when an entry with the Present bit cleared. The host processor must make the page resident in physical memory and set the Present bit before instructing the coprocessor to retry accessing the page tables.

Read / Write (R/W) flag, bit 1

The R/W bit indicates the read or read & write protection mode for the page. When the bit is cleared the page is read only, otherwise the page can be read or written. The coprocessor will issue a page fault interrupt to the host processor if this bit is clear and a write is issued to the page.

User / Supervisor flag, bit 2

The U/S bit indicates whether the page can be accessed by code running at privileged level only or user and privileged levels. When the bit is cleared the page can only be accessed in supervisor privileged mode. The coprocessor only accesses virtual memory in user mode. The coprocessor will issue a page fault interrupt to the host processor when it walks the address translation tables and finds an entry with the U/S bit cleared.

Page-Level Write-Through, bit 3

The PWT bit controls whether write-through or write-back caching policy is enabled. The coprocessor only supports write-back caching policy. When the bit is cleared write-back caching policy is enabled. The coprocessor will issue a page fault interrupt to the host processor when it walks the address translation tables and finds an entry with the PWT bit set.

Page-Level cache disable (PCD) flag, bit 4

The PCD flag controls whether the page of memory is allowed to be cached. When the flag is set the page cannot be cached. The coprocessor only supports cached memory accesses. The coprocessor will issue a page fault interrupt to the host processor when it walks the address translation tables and finds an entry with the PCD flag set.

Accessed (A) flag, bit 5

The A flag indicates whether the page as been accessed by the host or coprocessor. The host processor uses this information for memory management functions. This bit is set in the last level table (Page Directory Table) by the coprocessor when it walks the address translation tables.

Dirty (D) flag, bit 6

The D flag indicates that the page of memory has been written (modified). The coprocessor sets this bit in the last level page table (Page Directory Table) when the page is modified.

Page size (PS) flag, bit 7

The PS flag indicates the host size of the memory page. When the bit is set the host page is 2M Bytes, otherwise the host page is 4K Bytes.

Page size (PAT) flag, bit 7

The PAT flag indicates that a page attribute table should be used. The coprocessor does not support page attribute tables.

Global (G) flag, bit 8

The G flag indicates that a page is global. A Global page is common to all processes. TLB entries marked Global are maintained across process context switches. Global pages are rarely if ever used for user processes. The coprocessor ignores this bit and will invalidate all TLB entries on a context switch operation.

Page Base Address, bits 39-12

The Page Base Address fields (one per format type) is used to indicate the base address of the next level table for the first (PML4 Table) and second (Page Directory Pointer Table) level tables, or the base address of the physical memory page for the last level table (Page Table). Note that the last level Page Base Address field is 28-bits wide and provides the upper bits of a 40-bit address for a 4K Byte page.

Physically Contiguous (PC), bit 58

The Physically Contiguous bit is used to indicate that the physical address space associated with the page’s virtual address range is physically contiguous. This bit must be set for all page entries that map coprocessor memory. The bit may or may not be set for all page entries that map host memory.

Note that the Physically Contiguous and Page Size fields are used by the coprocessor but ignored by the host processor. These fields allow the coprocessor to map a virtually contiguous virtual address range as a single large page in the coprocessor's TLB while the host processor uses many 4K page TLB entries. The Physically Contiguous bit works in conjunction with the Page Size field to indicate to the coprocessor that the contiguous virtual address range is also physically contiguous.

Page Size, bits 62-59

The Page Size field is only used by the coprocessor to specify the virtual address page size. The following table shows the supported page sizes, the number of base address bits, and the number of virtual address bits used to construct the 40-bit physical address.

Page Size field value	*Page Size*	*Page Base Address bits*	*Page Offset Address bits*
0	4K	39-12	11-0
1	8K	39-13	12-0
2	16K	39-14	13-0
3	32K	39-15	14-0
4	64K	39-16	15-0
5	128K	39-17	16-0
6	256K	39-18	17-0
7	512K	39-19	18-0
8	1M	39-20	19-0
9	2M	39-21	20-0
10	4M	39-22	21-0
11-15	Reserved		

Execute Disable Bit (EXB) flag, bit 63

The execute disable flag is used to specify that the host or coprocessor cannot access the memory page for code references. The coprocessor will issue a page fault interrupt when the coprocessor references a page with the EXB bit set for instruction execution.

Avail fields

These fields are available for the system software writer's use. Fields designated as "available" are ignored by host and coprocessor hardware.

Reserved fields

Reserved fields are available for future uses. Fields designated as "Reserved" are ignored by the coprocessor.

4.5 Caches and Cache Coherence

Caches are used by the host processor and coprocessor to reduce latency and increase available bandwidth to access frequently referenced memory locations.

4.5.1 Host Processor Caches

All host processor caches (L1, L2, and L3) are maintained coherent with the latest value of memory through the use of a hardware based cache coherency mechanism. Cache coherency is maintained whether the referenced memory is attached to the host processor or the coprocessor.

4.5.2 Coprocessor Caches

The coprocessor uses a data cache to reduce latency and increase bandwidth of accesses to host processor main memory. The coprocessor data cache is maintained coherent with the latest value of memory.

References by the coprocessor to coprocessor main memory (the physical memory directly attached to the coprocessor) are not cached. Hardware mechanisms will ensure that coprocessor accesses to coprocessor main memory will always operate on the latest value of memory (the coprocessor will communicate with the host processor to ensure that an access is to the latest value of memory).

The coprocessor uses an instruction cache for coprocessor instruction references. The instruction cache is not maintained coherent with memory. An instruction cache invalidate supervisor command is available for applications that need to modify the instruction space. An example application is a debugger that inserts and deletes break point instructions as the program is running.

4.6 Memory Model

The Convey system consists of two regions of physical memory, the physical memory attached to the host processor and the physical memory attached to the coprocessor.

4.6.1 Host Processor Memory References

The x86 architecture defines a strong memory ordering model. All references by the host processor will complete in the order that they were issued. The strong ordering model applies whether the host processor references the physical memory attached to itself, or to the physical memory attached to the coprocessor. Strong ordering is enforced by ensuring that all memory references to a specific bank of memory follows the same interconnect paths through the system.

4.6.2 Coprocessor Memory References

The Convey Coprocessor supports a weakly ordered memory model. All coprocessor memory reads and writes are allowed to proceed independently to memory through different interconnect paths to provide increased memory bandwidth. Requests that follow different interconnect paths may arrive at the destination memory controller in a different order than they were issued. The weak memory ordering applies between load/store instructions, as well as between elements of the same instruction. The following examples use a personality that defines vector instructions.

Example #1:

```
ST.UD       V5, V3(A4)      ; Scatter (unsigned doubleword)
```

In example #1, the elements of V5 are stored to memory using the element values of vector register V3 as the offset from A4 for each stored element. The order that the individual stores will be written to memory is not defined. If multiple elements of vector register V3 have identical values then multiple elements of vector register V5 will be written to the same location of memory. The final value of memory for locations where multiple elements are written is undefined. Note that a fence instruction cannot solve an ordering issue between elements of the same instruction. Other techniques can be used in this type of operation to ensure correct operation.

Example #2 (with memory ordering problem):

```
ST      V5,0(A5)      ; Store vector to memory
LD      0(A4),V6      ; Load vector from memory
```

In example #2, the store instruction occurs first in the executed instruction stream followed by a load instruction. The memory write operations for each vector element of the store instruction will be issued to the memory interconnect before the memory read operations for each vector element of the load instruction. The multiple memory interconnect paths may reorder the read and write prior to arriving at the memory controllers. Some of the load instruction elements may be read prior to the previous store instruction’s write to memory.

Example #3 (example #2 with fence instruction):

```
ST      V5,0(A5)      ; Store vector to memory
```

```
    FENCE                   ; Fence instruction
    LD        0(A4),V6      ; Load vector from memory
```

Example #3 adds a fence instruction to example #2 to ensure proper memory ordering. Note that the fence instruction prohibits the load instruction from beginning execution until the store instruction's write operations have been committed to the memory controller. The fence instruction ensures that all stores have passed through the memory interconnect (where reordering can occur) prior to allowing the load instruction to begin execution.

Example #4:

```
    LD        0(A4),V6      ; Load vector from memory
    FENCE                   ; Fence instruction
    ST        V5,0(A5)      ; Store vector to memory
```

Example #4 illustrates the possible ordering problem of a load followed by a store. If a fence instruction is not included then the load and store memory operations could be reordered in the memory interconnect prior to arriving at the memory controller. When the fence instruction is present the load and store instructions will operate as expected even when the two instructions reference the same memory locations.

Example #5: (example #4 with identical base addresses)

```
    LD        0(A4),V6      ; Load vector from memory
    ST        V5,0(A4)      ; Store vector to memory
```

Example #5 illustrates the case of a load followed by a store where a fence is not required. Since the base address for the load and store instructions are identical, it can safely be assumed that references to the same memory addresses follow the same interconnect path to memory. As a result, a fence is not required.

4.6.3 Fence Instruction Usage

A fence instruction can be used to control the ordering of accesses to memory. The previous section presented multiple load / store examples that require a fence instruction to ensure the desired memory access order occurs. The following table shows when a fence instruction is required to ensure that correct results are produced.

		2nd Request			
		Scalar Load	Scalar Store	AE Load	AE Store
1st Request	Scalar Load	-	-	-	Fence
	Scalar Store	-	-	Fence	Fence
	AE Load	-	Fence	-	Fence[1]
	AE Store	Fence	Fence	Fence	Fence[1]

Table 5 – Fence Instruction Usage

Note 1 – Two AE references to the same memory address that follow the same interconnect path to memory may omit the intervening fence instruction.

The table indicates when a fence is required given a 1st request type and a 2nd request type to the same memory address. Note that the table is valid for requests to both host and coprocessor memory. The 1st and 2nd requests may be separated by other load / store instructions to the same or different memory addresses. The following example illustrates the need for a fence instruction when multiple requests occur.

```
LD.UQ    0(A1),A2     // scalar load to address in A1
LD.UQ    0(A1),V2     // AE load to address in A1
ST.UQ    V3,0(A1)     // AE store to address in A1
```

A fence is not required between the first request (Scalar Load) and second request (AE Load). Similarly, a fence is not required between the second request (AE Load) and third request (AE Store). However, a fence is required between the first request (Scalar Load) and third request (AE Store). The required fence may be placed between the first and second instructions or the second and third instructions.

5 Host Interface

5.1 Introduction

This section describes the interface between the host processor and the coprocessor. The interface allows a host application to directly dispatch a coprocessor routine to begin execution. Additionally, the operating system has privileged commands available to manage the coprocessor.

All host interface command and status operations are defined to use shared memory and leverage the cache coherency protocol to minimize latency. Memory controllers minimize cache coherency latencies to provide the highest possible performance for shared memory, multi-processor applications.

Host interface commands to the coprocessor are divided into two types: supervisor and user. Supervisor commands are issued by the operating system to manage the coprocessor. User commands are issued by a user process to execute a portion of an application on the coprocessor.

Supervisor commands are issued using a supervisor command structure. The supervisor command structure is located at a fixed location in physical memory. The operating system may access the supervisor command structure using physical memory accesses, or it may map the physical memory to its virtual address space and use virtual memory accesses.

User commands are issued using a user command structure. Each user process that uses the coprocessor has its own user command structure. The operating system specifies the location of each user command structure using supervisor commands. The user command structure is mapped into the user process’s virtual address space and all references are made using virtual address accesses.

Multiple user processes can simultaneously issue commands to the coprocessor. The coprocessor monitors each process’s user command structure to determine when a user command is ready to be executed. The operating system specifies the location of each process’s user command structure to the coprocessor. The coprocessor has resources to simultaneously monitor a limited number of processes.

Communication between the host processor and the coprocessor is accomplished with the following operations:

- Host processor issues commands to the coprocessor by writing the command to a reserved memory line in main memory and then performing a write to a second reserved memory line to indicate to the coprocessor that the command is ready.
- Host processor obtains coprocessor status by performing a memory read to a reserved memory line in main memory.
- Coprocessor indicates that it needs assistance by sending an interrupt to the host processor.

5.2 Host Interface Commands and Events

The host interface provides communication between the host processor and coprocessor by means of commands from the host processor to the coprocessor, and requests for assistance from the coprocessor to the host processor. Requests for assistance from the coprocessor occur when the coprocessor detects an event that requires host processor intervention to resolve. The coprocessor requests assistance by sending an interrupt to the host processor.

The following sections list the host interface commands and events.

5.2.1 Host Interface Commands

The host processor issues commands to the coprocessor using the coprocessor host interface. Coprocessor commands are split into two types: supervisor and user. Supervisor commands are issued to the coprocessor to setup, manage, monitor and finally release the coprocessor on behalf of a user process. User commands are issued to the coprocessor to initiate execution of a section of application code on behalf of a user process. The following table lists the supervisor and user commands recognized by the coprocessor.

Name	***Type***	***Description***
No Operation	Supervisor	Command that does not perform an operation. This command can be used as a simple host interface test to verify that the coprocessor is present. Additionally, the command may be used by the operating system to acknowledge completion of the most recent host interface interrupt.
Reset	Supervisor	Reset the host interface. The reset command can be issued at any time to bring the interface to a known state.
Attach	Supervisor	Initialize the coprocessor to begin receiving user commands on behalf of a process. The attach command specifies the virtual memory table base address.
Dispatch	User	Execute a user process routine. The command specifies the virtual address of the routine’s first instruction to be executed. The command specifies a list of values to be pre-loaded into coprocessor A and S registers before execution of the routine begins and the S registers that are to be passed back to the host processor when the routine completes. The command also specifies whether the routine completes with persistent context.
Load Context	Supervisor	Load the coprocessor context from memory. The command specifies the virtual address of the load context routine and the virtual address of the context structure in memory. The load context routine address is loaded into the IP register and the context structure

		address is loaded into coprocessor register A1.
Save Context	Supervisor	Save the coprocessor context to memory. The command specifies the virtual address of the save context routine and the virtual address of the context structure in memory. The save context routine address is loaded into the IP register and the context structure address is loaded into coprocessor register A1. Note that the previous value of A1 is available in the CCX register.
Detach	Supervisor	The detach command informs the coprocessor that a process no longer requires use of the coprocessor. All virtually tagged caches associated with the process are flushed (instruction cache and TLB).
Pause	Supervisor	The pause command stops execution of the coprocessor for the specified process. If a routine is presently being executed for the specified process, then execution is stopped at the next instruction boundary. New routine dispatches are disabled for the specified process until a resume command is issued.
Resume	Supervisor	If the coprocessor is not executing instructions and the coprocessor context is from a previously paused routine for the specified process, then the resume command restarts execution. New user routines are allowed to begin execution for the specified process.
TLB Retry	Supervisor	The TLB Retry command is in response to an interrupt from the coprocessor to the host when a TLB miss occurs that can not resolved by the coprocessor. The operating system issues a TLB Retry command to inform the coprocessor that the memory based page tables have been updated and the coprocessor should reaccess the page tables to resolve the miss.
TLB Abort	Supervisor	The operating system issues a TLB Abort command to inform the coprocessor that a TLB miss cannot be resolved and the coprocessor routine must be terminated.
ICache Invalidate	Supervisor	The operating system issues an Instruction Cache invalid command to request the coprocessor to invalidate the instruction cache for a specific process.

Table 6 - Host Interface Commands

5.2.2 Host Interface Events

Host interface events are the result of coprocessor detected situations that require host processor intervention to resolve. The table below lists the host interface events, the operating system response and a description of the event. Note that each event listed below generates an interrupt to the host processor. The operating system on the host processor must resolve the event and acknowledge interrupt processing complete by issuing a command to the coprocessor. The normal acknowledge commands for each interrupt type are listed in the table below. A second event/interrupt will not occur until the acknowledge for the first event/interrupt has been received.

Name	*OS Response*	*Description*
Trap	Save Context	A trap occurs when an exception is detected during the execution of an instruction and the exception type is not masked. Trap examples include invalid input operands (i.e. divide by zero) and invalid operation result (i.e. overflow). A trap stops the execution of the coprocessor at an instruction boundary. An interrupt is sent to the host processor when a trap is detected once coprocessor execution is stopped. Context can be saved to memory by the operating system by issuing a Context Save command.
Break	Save Context	The execution of a break instruction causes coprocessor execution to stop. Once execution stops, an interrupt is sent to the host processor. Context can be saved to memory by the operating system by issuing a Context Save command.
TLB Fault	TLB Retry / TLB Abort	A TLB fault occurs when a coprocessor virtual-to-physical memory translation results in a TLB miss that can be resolved by walking the memory based page tables. An interrupt is sent to the host processor to indicate that the TLB fault occurred. The operating system resolves the TLB fault (either successfully or unsuccessfully) and then issues either a TLB Retry or TLB Abort command. The coprocessor context cannot be saved when a TLB Fault occurs until after a TLB Abort command is issued.
Personality Signature Mismatch	Resume	The Personality Signature Mismatch (PSM) event occurs when a dispatch command is received from the host processor which contains a personality signature value that does not match the currently loaded AE personality. The PSM event acts as if a Pause command was issued for the effected process. The operating system will attempt to resolve the PSM by loading a new personality using the Load Personality command. Once the new personality is loaded, the dispatch can be retried by issuing a Resume command. If the operating system is not able to resolve the PSM event, then the offending process must be detached from the coprocessor using the Detach command.
Illegal Command	Reset	The illegal command event occurs when a command is issued to the coprocessor but the coprocessor state does not allow the command to be performed. This event should not occur under normal circumstances. The only valid response from the operating system is to issue the host interface Reset command to place the host interface

	into a known state.

Table 7 - Host Interface Events

5.3 Host Interface State

A number of coprocessor host interface state bits are used to manage the interface and determine if an issued command is valid. The current value of the state bits is accessible from the supervisor status structure. The following table lists the state bits and a description of their meaning. Note that all host interface state bits a cleared when a Reset command is issued.

Name	*Association*	*Description*
Attached	One per context	The Attached state bit indicates that an Attach command as been issued for the associated process. The attached state bit is set by the Attach command, and cleared by the Detach command.
Running	One per context	The Running state bit indicates that the coprocessor is executing instructions on behalf of a process. At most one Running state bit is set at any point in time. A Running state bit is set when a Dispatch, Load Context or Save Context command is selected for execution. The Running bit is cleared when the coprocessor executes a RTN instruction with an empty call / return stack.
Paused	One per context	The Paused state bit indicates that the user context associated with the state bit is paused. Coprocessor instruction execution is disabled for a user context that is paused. The Pause state bit is set due to the detection of a Trap, the execution of a Break instruction, the issuing of a Pause or TLB Abort command, or due to the detection of a Personality Signature Mismatch event.
PState	One per context	The PState bit indicates that the coprocessor's user registers have persistent state associated with the user process. The coprocessor can only execute instructions on behalf of the associated user process when persistent state exists. Persistent state exists due to two situations. The first is when a Trap, Break instruction or Pause command occurs while the processor is executing instructions. In this case, the coprocessor contains user context associated with the process that was previously executing instructions. The second situation is due to a user Dispatch command that specifies that persistent state will remain in the coprocessor's user registers after the dispatched routine completes (i.e. executes a RTN instruction with an empty call / return stack). In this case the coprocessor will only accept a new Dispatch command from the same user process.
Resumable	One per	The Resumable state bit indicates that coprocessor instruction execution was stopped due to a Break

	context	instruction or a Pause command and instruction execution can be resumed by issuing a Resume command.
Context Save	Supervisor state	The Context Save state bit indicates that a Save Context command was issued to the coprocessor and is still executing. The Context Save state bit is used by the RTN instruction at the end of the Save Context routine to enable clearing state making the coprocessor ready to accept a new Dispatch command.
Context Load	Supervisor state	The Context Load state bit indicates that a Load Context command was issued to the coprocessor and is still executing. The Context Load state bit is used by the RTN instruction at the end of the Load Context routine to preserve state such that a Resume command can restart instruction execution.

Table 8 - Host Interface State

5.4 Host Interface State Transitions

The host interface uses a finite state machine to track which set of commands are valid for the coprocessor at any given point in time. The following table lists the commands and events, and shows the resulting state transitions.

Operation / Event	*Subtype*	*Attached*	*Running*	*Paused*	*PState*	*Resumable*	*Ctx Save*	*Ctx Load*
Reset		* → 0	* → 0	* → 0	* → 0	* → 0	* → 0	* → 0
Attach		0 → 1	0	0 → 1	0	0	0	0
Dispatch	wo/PState	1	0 → 1	0	* → 0	0	0	0
	w/PState	1	0 → 1	0	* → 1	0	0	0
Trap		1	1 → 0	0 → 1	*	0	0	0
Break		1	1 → 0	0 → 1	*	0 → 1	0	0
Pause	Running	1	1 → 0	0 → 1	*	0 → 1	0	0
	Idle	1	0	0 → 1	*	0	0	0
Save Context		1	0	*	*	*	0 → 1	0
Load Context		1	0	1 → 0	0	0	0	0 → 1
Resume	Running	1	0 → 1	1 → 0	*	1 → 0	0	0
	Idle	1	0	1 → 0	*	0	0	0
TLB Fault		1	1	0	1	0	0	0
TLB		1	1	0	1	0	0	0

Retry								
TLB Abort		1	1 → 0	0 → 1	1	0	0	0
RTN Inst.	Dispatch	1	1 → 0	0	*	0	0	0
	Save Ctx	1	0	* → 1	* → 0	* → 0	1 → 0	0
	Ctx Load, Idle wo/PState	1	0	0	0	0	0	1 → 0
	Ctx Load, Idle w/PState	1	0	0	0 → 1	0	0	1 → 0
	Ctx Load, Running, Paused, wo/PState	1	0	0 → 1	0	0	0	1 → 0
	Ctx Load, Running, Paused, w/PState	1	0	0 → 1	0 → 1	0	0	1 → 0
	Ctx Load, Running, Paused, Resumable, wo/PState	1	0	0 → 1	0	0 → 1	0	1 → 0
	Ctx Load, Running, Paused, Resumable, w/PState	1	0	0 → 1	0 → 1	0 → 1	0	1 → 0
Detach		1 → 0	* → 0	* → 0	* → 0	* → 0	* → 0	* → 0

Table 9 - Host Interface State Transitions

The above table shows the legal state transitions that are defined for the host interface. Commands that are issued that would result in illegal state transitions result in an Illegal Command event. Illegal Command events issue an interrupt to the host processor.

The entries in the table indicate the entering and resulting state values for each command / event. Each entry indicates the entering legal state values and whether the state bit changes value due to the command / event. The following table describes the meaning of each entry type.

Entry Type	*State Transition Description*
0	The state bit must be the value zero when the command is issued and does not change value when the command is performed.
1	The state bit must be the value one when the command is issued and does

	not change value when the command is performed.
*	The state bit may be either a zero or one when the command is issued and does not change value when the command is performed.
0 → 1	The state bit must be the value zero when the command is issued and changes to the value one when the command is performed.
1 → 0	The state bit must be the value one when the command is issued and changes to the value zero when the command is performed.
* → 0	The state bit may be either a zero of one when the command is issued and results in the value zero when the command is performed.
* → 1	The state bit may be either a zero of one when the command is issued and results in the value one when the command is performed.

Table 10 - State Transition Entry Description

5.5 Host Processor Interrupts

An interrupt is sent from the coprocessor to the host processor when specific events are detected on the coprocessor. The host processor can examine the Supervisor area status memory structures to determine the cause of the interrupt and obtain specific information on the event to assist the operating system in resolving the issue.

Events are classified as either synchronous or asynchronous. Synchronous events occur due to the interactions of an application that is running on the coprocessor. Synchronous events can in general be repeated by rerunning the user application. An example of a synchronous event is a TLB Fault. Asynchronous events are not tied directly to a specific running process. An example of an asynchronous event is an uncorrectable memory error (reported as a machine check).

A single synchronous event interrupt will be issued to the operating system at a time. The coprocessor is notified that interrupt processing is complete by issuing a subsequent command to the supervisor synchronous command structure. Once the coprocessor receives the command, it is free to issue another synchronous event interrupt.

The host interface is able to issue a single asynchronous event interrupt independent of any synchronous event interrupt. The host interface will not issue a synchronous interrupt if an asynchronous interrupt is outstanding.

The host processor informs the coprocessor that interrupt processing is complete by issuing a command to the supervisor command area. The type of command is specific to each interrupt type. The following table lists the interrupt types, the information available in the supervisor status structure, and the supervisor host interface command that indicates completion of the interrupt.

Interrupt	*Interrupt Type*	*Status Structure Information*	*Typical Completion Command*	*Description*
Trap	Sync.	None	Save Context	The input operands or result of an instruction operation is invalid.

Break	Sync.	None	Save Context	A break instruction was executed that stopped instruction execution.
TLB Fault	Sync.	UID and Virtual Address	TLB Retry or TLB Abort	A load or store instruction caused a TLB miss that could not be resolved by referencing the memory based address translation tables.
Personality Signature Mismatch	Sync.	Dispatch Personality Signature	Resume	A Dispatch command was issued with a personality signature that did not match the currently loaded personality.
Illegal State / Command Combination	Async.	Command Set that caused problem	Reset	An invalid command was detected based on the current host interface state.
Machine Check	Async.	Unknown at this time	Unknown at this time	An unexpected error occurred that prevents further coprocessor use. Hardware will define the machine check sources.

Table 11 - Host Processor Interrupts

5.6 Coprocessor Threading

The coprocessor's host interface is defined to allow multiple processes to simultaneously issue user commands. A light weight context switch mechanism allows switching execution on behalf of one process to another very quickly. This mechanism increases the utilization of the coprocessor.

The coprocessor provides per process state for a limited number of active coprocessor processes. An active coprocessor process is a process where its user command structure is being monitored by the coprocessor.

5.6.1 Process Context

Coprocessor process context can be very large in size and take considerable time to save and restore to memory. Performing a full context switch between coprocessor dispatch routines could result in more time spent saving and restoring process context than executing user routines.

A light weight context switch mechanism exists that allows switching coprocessor execution between multiple processes without saving/restoring any state to memory. The mechanism relies on the user process to indicate when a dispatch routine has completed without persistent user context. Persistent context is defined as user register state that must be maintained between routine dispatches by the same process.

Most user registers follow the persistent context model. A few user registers do not follow the persistent context model and a separate copy of these registers is maintained for each active user process. An example where per process state is maintained is status and control flag bits. The following table shows the process state categories and attributes.

State type	*Saved / Restored as Context*	*Initialized by all Dispatches*	*Separate copy per User Process*
Initialized State	Yes	Yes	No
Persistent State	Yes	No	Yes
Non-persistent State	Yes	No	No

Table 12 - Context State Attributes

Initialized state includes all state that is initialized as part of dispatching a coprocessor routine. Examples include the Instruction Pointer (IP) register and status register exception bits. This state is initialized as part of each routine dispatch processing. The state is saved and restored as part of a context save / restore operation.

Persistent State includes all state where a separate copy exists per user context area. This state is generally initialized when a process attaches to the coprocessor and is maintained for the life of the process. Persistent state is maintained across dispatches from different user areas (i.e. a separate copy exists per user area). Examples include control register exception mask bits.

Non-persistent state includes the majority of all coprocessor state. A single copy of this state type is shared by all user processes. A full context save or restore transfers the value of this state to/from memory. Non-persistent state is not maintained across dispatches from different user areas. Examples include the A and S registers from the Scalar ISA, V and VM registers from the Single and Double Precision Vector ISAs, and the AEG register from the User Defined Application Specific Personality ISA.

A limited amount of per process supervisor state is required in the coprocessor. An example is the virtual page table base address. A separate copy of all per process supervisor state is maintained in the coprocessor for all active coprocessor processes.

5.6.2 Dispatch Command

The Dispatch command plays a vital role in the coprocessor threading mechanism. Two variations of the dispatch command exist. The two variations indicate whether the routine will complete with or without non-persistent register state that must be maintained for the next dispatch by the same user area. Dispatching a routine with the PState indicator set ensures that all non-persistent state is maintained across coprocessor routine dispatch boundaries. Completing a routine with the User Command PState indicator forces the coprocessor to only accept new dispatch commands from the same process. Completion of a routine without the User Command PState indicator allows the next dispatched routine to originate from any attached coprocessor process.

5.7 Host Interface Memory Structures

Command and status operations between the host processor and coprocessor use the cache coherency mechanism to provide low latency communication.

Coprocessor commands are split into two types, user and supervisor. These two types of commands are discussed in the following sections.

5.7.1 Supervisor Command and Status

All supervisor command and status communication between the host processor and coprocessor occurs through a 4K byte area of physical memory. The area must be aligned on a 4K byte boundary. The communication occurs using the standard processor-to-processor cache coherency mechanisms. Figure 8 shows the structure of the area of memory used for supervisor command and status.

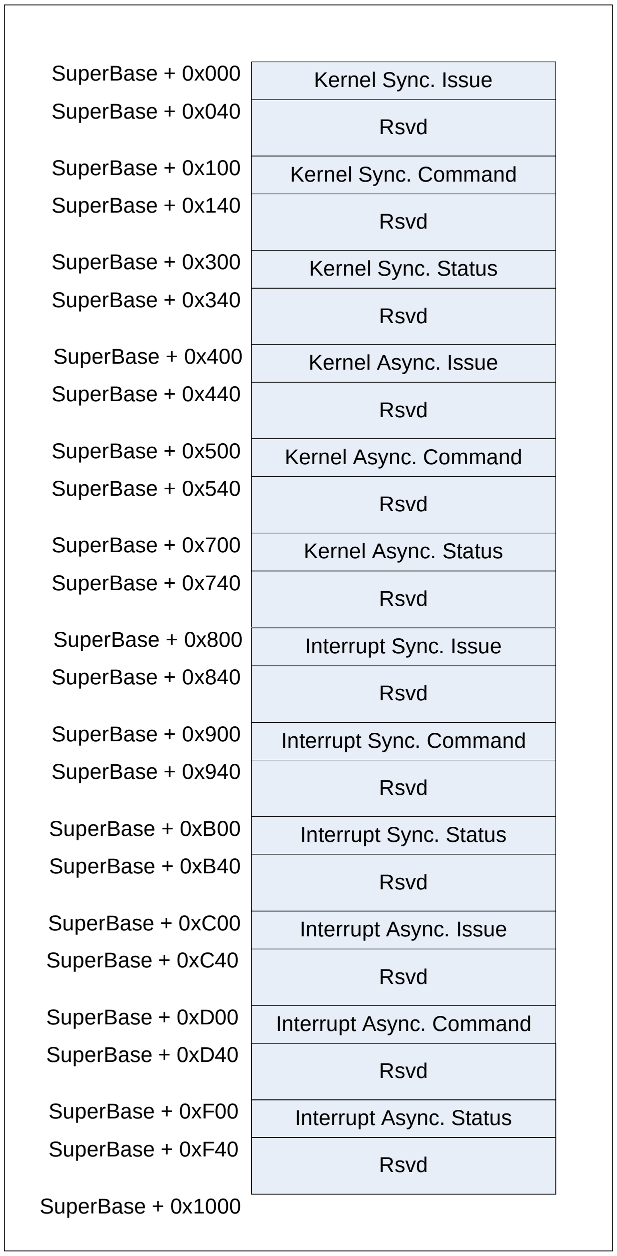


Figure 8 - Supervisor Command Memory Structure

The memory area must be accessible at boot time to both the host processor and coprocessor. The supervisor memory area is fixed to a specific physical address and is referred to as SuperBase in the figure above. The SuperBase physical address may be mapped to a virtual address within the operating system.

Four sets of Issue / Command / Status structures are provided (named Kernel Sync., Kernel Async., Interrupt Sync., and Interrupt Async.). The coprocessor checks for active commands across the four command sets in a fixed priority order. The highest priority command set is Interrupt Async., followed by Interrupt Sync, Kernel Async., and lowest priority is Kernel Sync.

5.7.1.1 Supervisor Issue Memory Structure

The format of the Issue memory structure is shown below. The IssueBase physical address is equivalent to addresses SuperBase+0x000, SuperBase+0x400, and SuperBase+0x800 as shown in Figure 8.

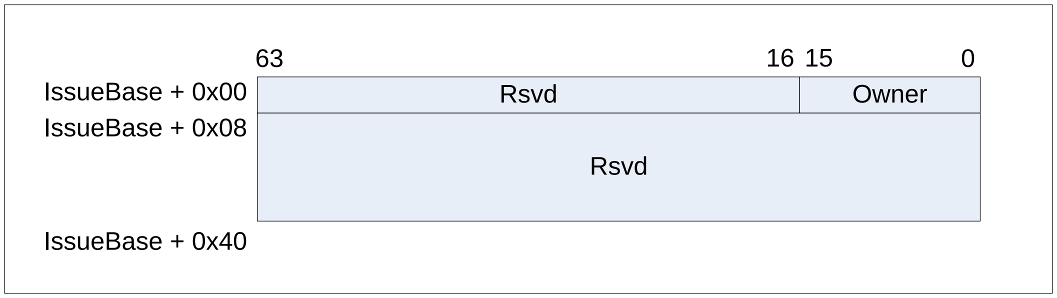


Figure 9 - Supervisor Issue Memory Structure

The *Owner* field of the Issue memory structure indicates the current owner of the command and status memory structures. When the *Owner* field has the value zero, the command set is not active and the host processor can write the command memory structure to setup a new command or read status of the previous dispatched command. When the *Owner* field has the value one, the command set is owned by the coprocessor and the current command is being performed. Modification of the command memory structure when the *Owner* field is a one may cause the previously dispatched routine to give unexpected results.

5.7.1.2 Supervisor Command Memory Structure

The format of the command memory structure is shown below. The CmdBase physical address is equivalent to addresses SuperBase+0x100, SuperBase+0x500, and SuperBase+0x900 as shown in Figure 8.

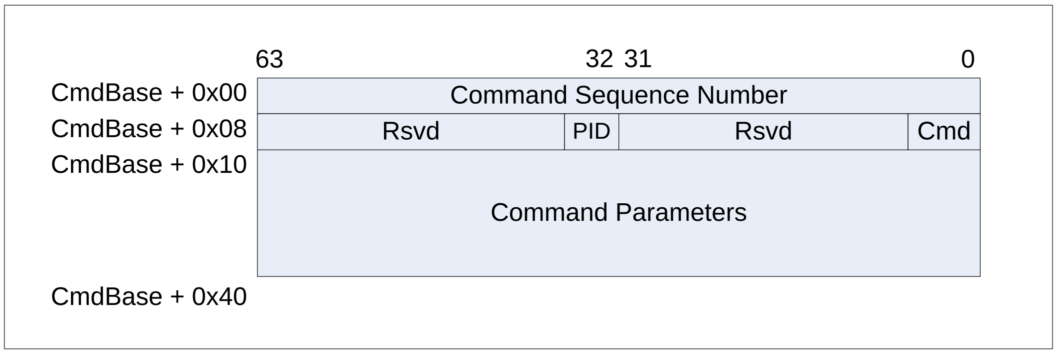


Figure 10 - Supervisor Command Memory Structure

The privileged command contains three fields plus a list of parameters: the *Command Sequence Number* field (quad word zero, bits 63-0), the *PID* (Process ID) field (quad word one, bits 33-32), and the *Cmd* (Command) field (quad word one, bits 3-0). All commands require the *Command Sequence Number* and *Command* fields. Commands that apply to a specific user process require the *Process ID* field. Additional information required by a command is specified using the *Command Parameter* fields (each parameter is 8-bytes in size). The *Command Sequence Number* field is returned in the Status memory structure when the supervisor command has completed. Table 13 lists the privilege commands.

Command Name	*Command Field*	*Process ID required*	*Description*
Nop	0	No	No operation.
Reset	1	No	Perform hardware reset on host interface state.
Attach	2	Yes	Associates a user process with a specific user command area. Two command parameters are required. The parameters are the virtual memory page table PLM4 base address (CmdBase+0x10), and the User Command Area base address (CmdBase+0x18)
Detach	3	Yes	Disassociates a user process with a specific user command area. The command also invalidates all cached entries accessed by the user processes virtual address (TLB and instruction cache entries).
Load Context	4	Yes	The Load Context command dispatches a coprocessor routine (starts instruction execution) to load a coprocessor context structure from memory to the coprocessor. Two command parameters are required: the virtual address of the load context routine (CmdBase+0x10) and the virtual address where the context structure resides in memory (CmdBase+0x18). The Load Context command completes when a RTN instruction is executed.
Save Context	5	Yes	The Save Context command dispatches a coprocessor routine (starts instruction execution) to save coprocessor context to memory. Two command parameters are required: the virtual address of the save context routine (CmdBase+0x10) and the virtual address where the context structure resides in memory (CmdBase+0x18). The Save Context command completes when a

			RTN instruction is executed.
Pause	6	Yes	The Pause command pauses the currently executing routine if the routine is executing on behalf of the provided User Process ID. The command also disables future routines to begin execution for the specified User Process ID. No parameters are required.
Resume	7	Yes	The Resume command allows new routines to begin execution, and resumes execution if a routine was paused prior to completion. No parameters are required.
TLB Retry	8	No	Informs the coprocessor that the current TLB Fault event has been resolved (I.e. the memory based TLB table has been updated with an entry for the virtual address that caused the page fault). Issuing this command causes the coprocessor’s TLB lookup engine to reaccess the memory based TLB tables for the needed entry. Once the entry is found, it is placed in the coprocessor’s TLB table and the faulting load or store is retried.
TLB Abort	9	No	Informs the coprocessor that the current TLB Fault event cannot be resolved. Issuing this command forces the coprocessor to abort all executing instructions and transition to idle with persistent state. No parameters are required.
Invalidate TLB Entry	10	Yes	Invalidate a TLB entry for the specified user process. One command parameters is required. The parameter is the virtual address of the page to be invalidated (CmdBase+0x10).
Invalidate TLB PID	10	Yes	Invalidate all TLB entries for the specified user process. One command parameters is required.
Invalidate Icache	14	Yes	Invalidate the instruction cache for a specific user area. No parameters are required.

Table 13 - Supervisor Commands

5.7.1.3 Supervisor Status Memory Structure

The format of the status memory structure is shown below. The StatusBase physical address is equivalent to address SuperBase+0x300, SuperBase+0x700, and SuperBase+0xB00 as shown in Figure 8.

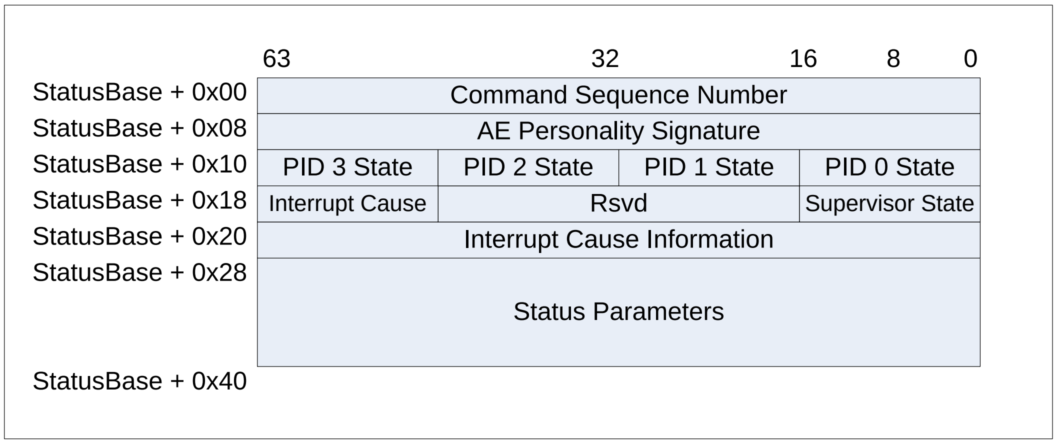


Figure 11 - Supervisor Status Memory Structure

The fields of the privileged status memory structure have the following definition:

Field	*Description*
Command Sequence Number	Value passed to coprocessor and then returned in status memory structure to determine when command has completed.
Coprocessor Personality Signature	Personality Signature of current Coprocessor FPGA image.
PID 0-3 State	The PID State fields provide visibility of the state bits for each of the four user command areas.
Supervisor State	The *Supervisor State* field provides visibility of the supervisor command area state.
Interrupt Cause	The *Interrupt Cause* field specifies the cause of the most recent interrupt sent from the coprocessor to the host processor. The cause of a synchronous event is present in the Sync. Event Status structure, and the cause of an asynchronous event is present in the Async. Event status structure. The Interrupt cause field of the Kernel Status structure is not used.
Interrupt Cause Information	The *Interrupt Cause Information* field provides information to the operating system associated with the most recent interrupt. The Interrupt Cause Information is found in the same status structure as the associated Interrupt Cause field for the specific event type (sync. or async.).

Table 14 - Supervisor Status Memory Structure Fields

Note that the PID 0-3 and Supervisor State fields are written at the time the Command Sequence Number and Interrupt Cause fields are written. The operating system can force the state fields to be updated by issuing a NOP command to the coprocessor.

5.7.1.3.1 *User area state bits*

The UID State field within the Supervisor Status memory structure contains state bits associated with a user issue / command / status interface. The field bits are shown in the following figure.

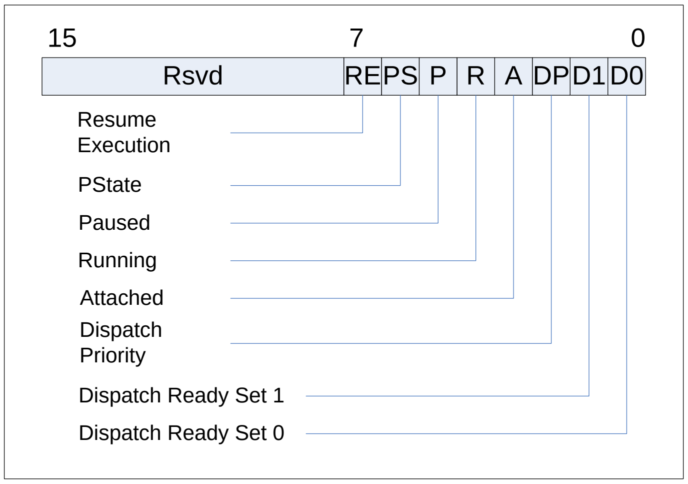


Table 15 - User Area State Bits

State Bit Name	*Description*
D0 – Dispatch Ready Set 0	The D0 bit indicates that the Set 0 Issue bit is set.
D1 – Dispatch Ready Set 1	The D1 bit indicates that the Set 1 Issue bit is set.
DP – Dispatch Priority	The DP bit indicates which command set (D0/D1) has priority if both are set.
A – Attached	The A bit indicates that the associated user command area has a user process attached.
R – Running	The R bit indicates that the coprocessor is currently executing instructions on behalf of the associated user command area.
P – Paused	The P bit indicates that the associated user command area is currently paused. When paused, the host interface will not start a new dispatch routine for the associated user command area.
PS – Pstate	The PS bit indicates that the coprocessor is currently holding persistent state for the current user command area. Other user command areas are disabled from starting a new dispatch routine when the associated user command area has persistent state.
RE – Resume Execution	The RE bit indicates that the instruction execution was paused for the associated user command area. Instruction execution can be resumed using the Resume command.

Table 16 - User Area state bit definition

5.7.1.3.2 *Supervisor area state bits*

The Supervisor State field within the Supervisor Status memory structure contains state bits associated with the Supervisor command and status area. The state bits within the field are shown in the following figure.

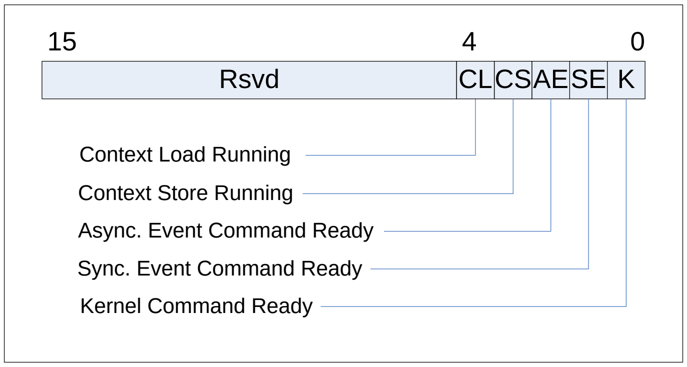


Figure 12- Supervisor area state bits

State Bit Name	*Description*
K – Kernel Command Ready	The K bit indicates that the Kernel Set Issue bit is set.
SE – Sync. Event Command Ready	The SE bit indicates that the Sync. Event Set Issue bit is set.
AE – Async. Event Command Ready	The AE bit indicates that the Async. Event Set Issue bit is set.
CS – Context Save Running	The CS bit indicates that a Save Context command is being performed (i.e. a context save routine is being executed).
CL – Context Load Running	The CL bit indicates that a Load Context command is being performed.

Table 17 - Supervisor area state bit definition

Note that for the supervisor area, commands are obtained from the three command sets in a fixed priority order (Async, Sync and Kernel).

5.7.1.4 Steps Required to Issue a Supervisor Mode Command and Obtain Status

The steps required to issue a supervisor mode command to the coprocessor are:

1. Read the value of the Issue memory structure *Owner* field. Loop until the *Owner* field is zero.
2. Setup the new command by writing to the fields of the Command memory structure. The fields can be written in any order and any operand width. Note that a variable incremented once for each supervisor command can be written to the *Command Sequence Number* field to provide a command unique identifier.

3. Issue the new command by writing a one to the *Owner* field of the Issue memory structure.

Once the coprocessor has been issued a command, the resulting status can be obtained with the following steps:

4. Read the value of the Issue memory structure Owner field. Loop until the Owner field is zero.
5. Read the value of the *Command Sequence Number* field from the Status memory structure. Loop until the obtained value equals the value specified in the Command memory structure.
6. Access the returned parameters from the Status memory structure.

The transition from one to zero of the Owner field indicates that the coprocessor can accept another supervisor command (however, the current command may not have finished). The completion of a command is determined by monitoring the *Command Sequence Number* field in the status memory structure.

Note that it is not necessary to read status of a command. A subsequent command can be issued to the coprocessor as soon as the Issue bit has transitioned back to the value zero.

The above described sequence uses standard cache coherency mechanisms and will function correctly in the presence of speculative memory accesses. Additionally, the mechanism will operate correctly for a system implements with the current FSB (Front Side Bus) technology or the forthcoming QPI (Quick Path Interconnect) technology.

5.7.2 User Mode Command and Status

All user command and status communication between the host processor and coprocessor occurs through 4K byte areas of physical memory. The areas must be aligned on a 4K byte boundary. Multiple user command and status areas exist to allow multiple processes to simultaneously issue commands to the coprocessor. The communication occurs using the standard processor-to-processor cache coherency mechanisms. The operating system may locate the 4K byte areas at any location within host’s physical memory. A 4K byte area must be mapped into an application’s virtual address space to allow an application direct access to the coprocessor.

5.7.2.1 User mode command and status area

The figure below shows the structure of each 4K byte area of memory used to issue user mode commands and obtain status.

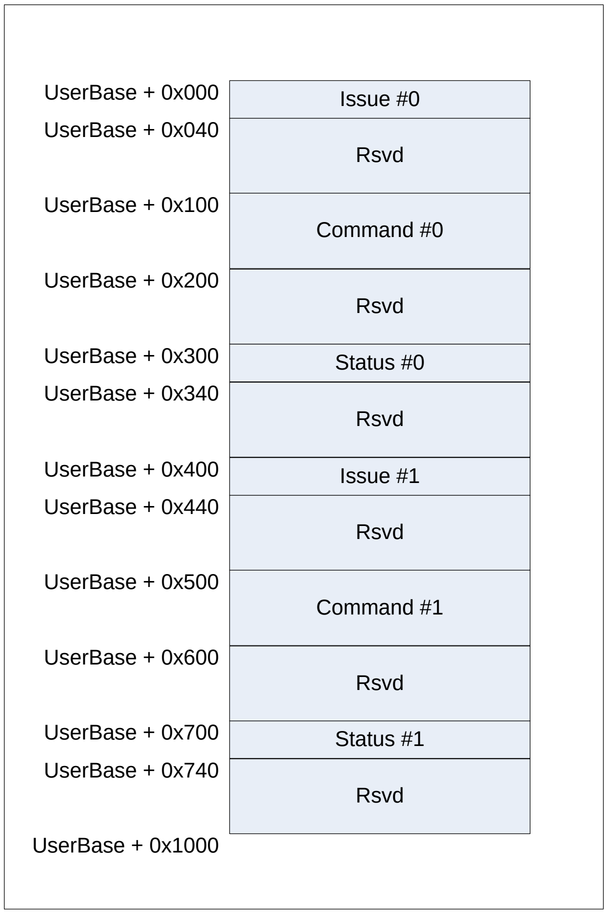


Figure 13 – User Mode Command and Status Area Structure

This figure shows two sets of issue, command and status memory structures. Two sets are provided to allow overlapped execution of one command while the second command is being setup and issued. The physical address of the user memory area can be configured by the operating system allowing each process to have its own area in physical memory. The physical address of the user area is referred to as UserBase in the figure above. The UserBase physical address is mapped to a virtual address within a user process.

5.7.2.2 User Mode Issue Memory Structure

The Issue memory structure is used to indicate whether the host processor or coprocessor owns the Command and Status memory structures. The figure below shows the format of the Issue memory structure. The IssueBase physical address is equivalent to addresses UserBase+0x000 or UserBase+0x400 as shown in Figure 13.

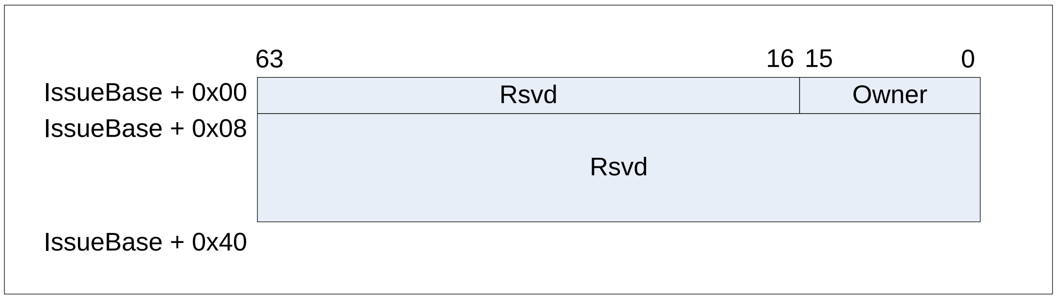


Figure 14 - User Mode Issue Memory Structure

The *Owner* field of the Issue memory structure indicates the current owner of the command and status memory structures. When the *Owner* field has the value zero, the command set is not active and the host processor can read status of the previous dispatched command, or write the command memory structures to setup a new command. When the *Owner* field has the value one, the command set is owned by the coprocessor and the current command is being performed. Modification of the command memory structures when the *Owner* field is a one may cause the previously dispatched routine to give unexpected results.

5.7.2.3 User Mode Command Memory Structure

A single user mode command is defined. The command is Routine Dispatch.

The format of the command memory structure is shown below. The CmdBase physical address is equivalent to addresses UserBase+0x100 or UserBase+0x500 as shown in Figure 13.

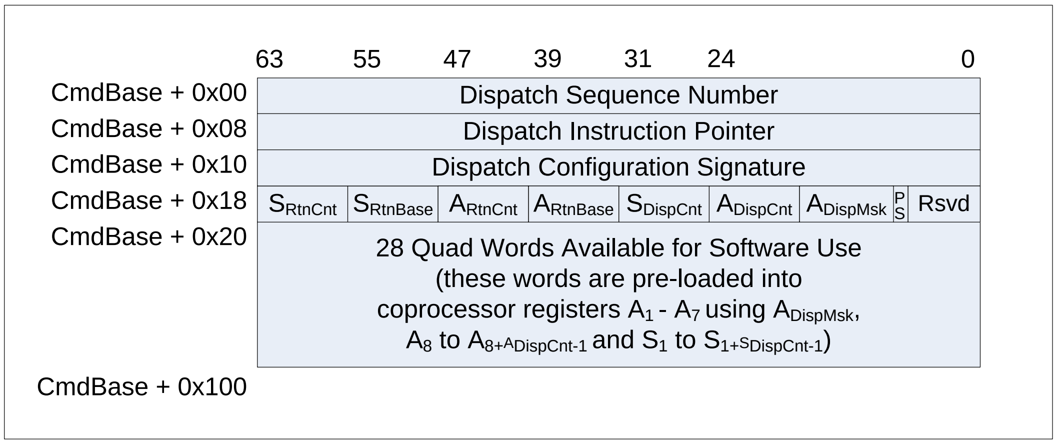

63 55 47 39 31 24 0
CmdBase + 0x00
Dispatch Sequence Number
CmdBase + 0x08
Dispatch Instruction Pointer
CmdBase + 0x10
Dispatch Configuration Signature
CmdBase + 0x18
S_{RtnCnt} $S_{RtnBase}$ A_{RtnCnt} $A_{RtnBase}$ $S_{DispCnt}$ $A_{DispCnt}$ $A_{DispMsk}$ PS Rsvd
CmdBase + 0x20
28 Quad Words Available for Software Use
(these words are pre-loaded into
coprocessor registers A_1 - A_7 using $A_{DispMsk}$,
A_8 to $A_{8+A_{DispCnt-1}}$ and S_1 to $S_{1+S_{DispCnt-1}}$)
CmdBase + 0x100

Figure 15 - User Mode Command Structure

The user mode command structure consists of the following defined fields.

Field Name	*Description*
Dispatch Sequence Number	Value passed to coprocessor and then returned in status memory structure. Used by host to determine when coprocessor routine has completed. The value is not used by the coprocessor.

Dispatch Instruction Pointer	Virtual address of first instruction of coprocessor routine.
Dispatch Personality Signature	Signature of required application engine personality. This signature is checked prior to beginning the dispatched coprocessor routine to ensure that the appropriate application engine personality is loaded. The operating system is interrupted if the loaded AE personality signature does not match the *Dispatch Personality Signature* field.
PState	The PState field (bit 8) is used to specify that the current dispatch completes with persistent state, and the next dispatch must be from the same process. The value of one indicates the current dispatch completes with persistent state.
$A_{DispMsk}$	The $A_{DispMsk}$ field (bits 15:9) specifies the non-window based A registers that are loaded when the coprocessor is dispatched. The non-window based A registers are A1 through A7. Bit 25 corresponds to A1, bit 26 to A2, through bit 31 corresponding to A7.
$A_{DispCnt}$	The $A_{DispCnt}$ field (bits 23:16) specifies the number of parameters to be preloaded into window based A-registers prior to beginning the coprocessor routine. Parameters are copied into A-registers started with register A8 through A register 8+$A_{DispCnt}$-1.
$S_{DispCnt}$	The $S_{DispCnt}$ field (bits 31:24) specifies the number of parameters to be preloaded into S-registers prior to beginning the coprocessor routine. Parameters are copied into S-registers started with register S1.
$A_{RtnBase}$	The $A_{RtnBase}$ field (bits 39:32) specifies the first A-register to be copied into status memory structure when coprocessor executes the return complete instruction.
A_{RtnCnt}	The $A_{RtnBase}$ field (bits 47:40) specifies the number of A-registers to be copied into status memory structure when coprocessor executes the return complete instruction.
$S_{RtnBase}$	The $S_{RtnBase}$ field (bits 55:48) specifies the first S-register to be copied into status memory structure when coprocessor executes the return complete instruction.
S_{RtnCnt}	The $S_{RtnBase}$ field (bits 63:56) specifies the number of S-registers to be copied into status memory structure when coprocessor executes the return complete instruction.
Quad Word Parameters	Parameters preloaded into A and S registers prior to

	beginning coprocessor routine. Note that A register parameters reside in lower order addresses followed by S-register parameters. The sum of the ADispCnt and SDispCnt fields must be less than or equal to twenty-eight.

Table 18 - User Mode Command Structure Fields

5.7.2.4 User Mode Status Memory Structure

The status memory structure is used to determine when the coprocessor has finished executing a routine and to obtain any returned values. The format of the Status Memory Structure is shown in the following figure. The StatusBase physical address is equivalent to addresses UserBase+0x300 or UserBase+0x700 as shown in Figure 13.

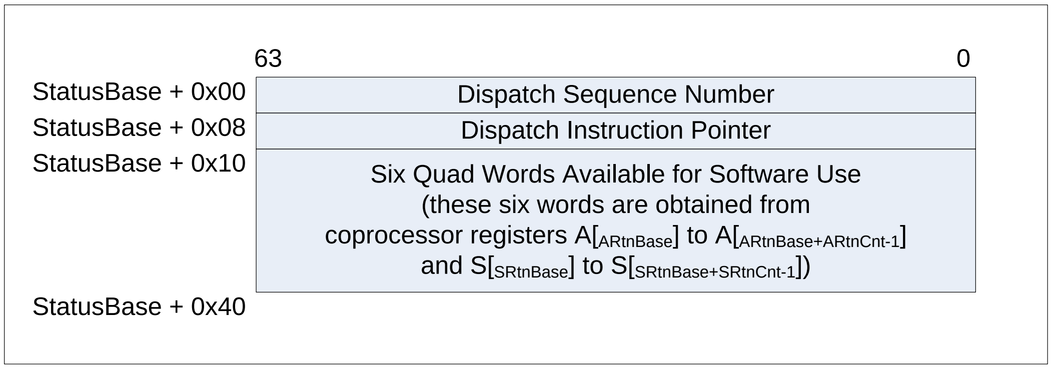

63
0
StatusBase + 0x00
Dispatch Sequence Number
StatusBase + 0x08
Dispatch Instruction Pointer
StatusBase + 0x10
Six Quad Words Available for Software Use
(these six words are obtained from
coprocessor registers A[ARtnBase] to A[ARtnBase+ARtnCnt-1]
and S[SRtnBase] to S[SRtnBase+SRtnCnt-1])
StatusBase + 0x40

Figure 16 - User Mode Status Structure

The user mode status memory structure consists of the following defined fields.

Field Name	*Description*
Dispatch Sequence Number	Value passed to coprocessor at time of dispatch and then returned in status memory structure. Used to determine when routine has completed.
Dispatch Instruction Pointer	Virtual address of first instruction of completed coprocessor routine. This field is provided for debug visibility only.
Quad Word Results	Results returned from completed coprocessor routine. The specific A and S-register values copied to the Status Memory Structure are determined by the ARtnBase, ARtnCnt, SRtnBase and SRtnCnt fields of the User Mode Command structure. Note that the A register values are copied first, followed by the S register values.

Table 19 - User Mode Status Structure Fields

5.7.2.5 Steps Required to Issue a User Mode Command and Obtain Status

Two command sets are provided as shown in Figure 13. Alternating between the two command sets will result in higher application performance by reducing coprocessor idle time. One command set can be being setup for the next command while the other set is being used by the coprocessor. Note that using a single command set to dispatch the coprocessor routines will function properly, albeit somewhat slower.

The steps required to issue a user command to coprocessor are:

1. Select the command set (0 or 1) to be used to issue the command. A host processor variable that is incremented once per issued command can be used to select which command set is to be used. If the variable is even then use set #0, otherwise use set #1.
2. Read the value of the Issue memory structure *Owner* field. Loop until the *Owner* field is zero.
3. Setup the new command by writing to the fields of the selected set's Command memory structure. The fields can be written in any order and any operand width. Note that the variable used in step #1 can be written to the *Dispatch Sequence Number* field to provide a dispatch unique identifier.
4. Issue the new command by writing a one to the *Owner* field of the Issue memory structure.

Once the coprocessor has been issued a command, the resulting status can be obtained with the following steps:

1. Read the value of the Issue memory structure *Owner* field. Loop until the *Owner* field is zero.
2. Read the value of the *Dispatch Sequence Number* field from the Status memory structure. Loop until the obtained value equals the value specified in the Command memory structure.
3. Access the returned parameters from the Status memory structure.

Note that the transition from one to zero of the *Owner* field indicates that the coprocessor can accept another user command (however, the current command may not have finished). The completion of a command is determined by monitoring the *Command Sequence Number* field in the status memory structure.

Note that it is not necessary to read status of a command. A subsequent command can be issued to the coprocessor as soon as the Issue bit has transitioned back to the value zero.

The above described sequence uses cache coherency mechanisms and will function correctly in the presence of speculative memory accesses. Additionally, the mechanism will operate correctly for a system implements with the current FSB (Front Side Bus) technology or the forthcoming QPI (Quick Path Interconnect) technology.

6 Data Mover

This section describes the data mover functionality and interface used by the host processor to issue an operation. The data mover provides high bandwidth memory copy and set operations. The copy operations are optimized for copies from host to coprocessor or coprocessor to host memory regions, but can also be used for host to host or coprocessor to coprocessor.

All data mover operations are initiated by the host processor and an operation can be performed concurrently with coprocessor operations. Note that concurrent operation of the data mover and coprocessor results in sharing of the available bandwidth of the system. The data mover is used by the operating system to perform page initialization (page zero), page migration, and I/O data copies.

The data mover can be accessed from a user process to perform memory copy or set operations. The data mover is accessed by a user process through a system call interface.

The host processor must ensure that all pages being touched by the data mover are resident in physical memory and are mapped by the page table structure. Data mover operations that fail will result in non-zero error status.

6.1 Data Mover Commands

The host processor issues commands to the data mover using the data mover host interface. The following table lists the commands recognized by the coprocessor.

Name	***Description***
Copy	The Copy command performs a copy operation from a source virtual address to a destination virtual address. The source and destination addresses can reference either host or coprocessor memory, and can be aligned to any byte boundary.
Set	The Set command performs a memory set operation with a specified 64-bit data value. The destination virtual address can reference either host or coprocessor memory, and can be aligned on any byte boundary.

6.2 Data Mover Host Interface

All data mover command and status communication between the host processor and data mover occurs through a 4K byte area of physical memory. The area must be aligned on a 4K byte boundary. The communication occurs using the standard processor-to-processor cache coherency mechanisms.

Figure 17 shows the structure of the area of memory used for data mover command and status.

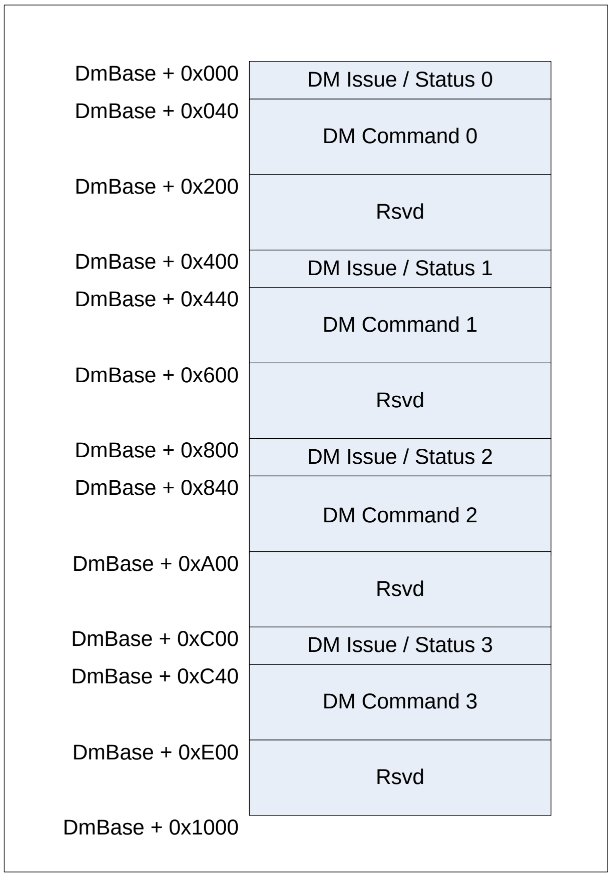


Figure 17 – Data Mover Interface Memory Structure

The data mover memory structure is fixed to a specific physical address and is referred to as DmBase in the figure above. The DmBase physical address may be mapped to a virtual address within the operating system.

Four sets of Issue / Status and Command structures are provided. The data mover checks for active commands across the four command sets in a round robin priority order.

6.2.1 Data Mover Issue / Status Memory Structure

The format of the Issue / Status memory structure is shown below. The DmIssueBase physical address is equivalent to addresses DmBase+0x000, DmBase +0x400, DmBase +0x800, and DmBase +0xC00 as shown in Figure 17.

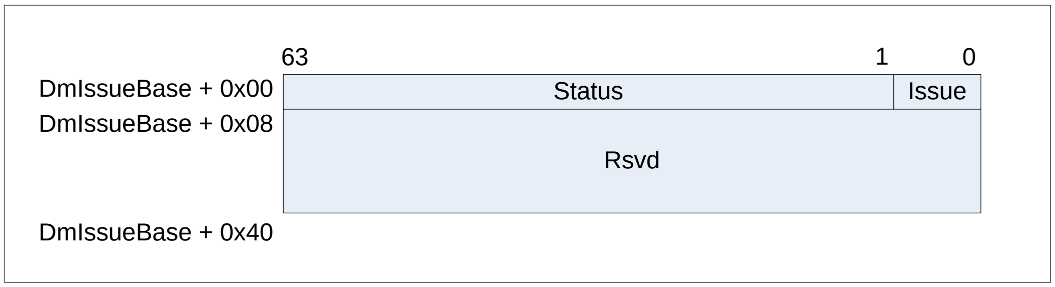


Figure 18 – Data Mover Issue Memory Structure

The *Issue* field of the Issue / Status memory structure indicates the current owner of the command memory structure. When the *Issue* field has the value zero, the command set is not active and the host processor can write the command memory structure to setup a new command or read status of the previous issued command. When the *Issue* field has the value one, the command set is owned by the data mover and the current command is being performed. Modification of the command memory structure when the *Issue* field is a one may cause the previously issued command to give unexpected results.

The *Status* field of the Issue / Status memory structure provides error status of a completed data mover operation. A status value of zero indicates the data mover operation completed successfully. The format of the *Status* field is implementation specific.

6.2.2 Data Mover Command Memory Structure

The format of the command memory structure is shown below. The DmCmdBase physical address is equivalent to addresses DmBase +0x040, DmBase +0x440, DmBase + 0x840 and DmBase +0xC40 as shown in Figure 17.

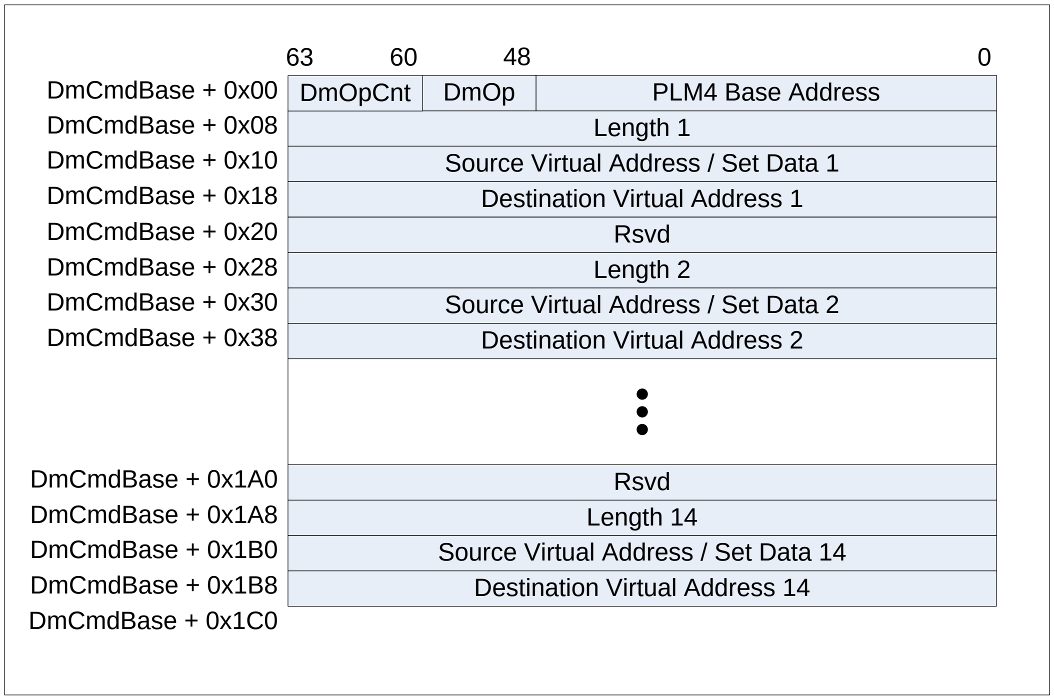


Figure 19 – Data Mover Command Memory Structure

Each data mover command memory structure allows up to 14 operations to be specified as one command to the data mover hardware. It is advantageous to gather multiple operations into a single command since a memory fence operation is performed only once after the last operation is performed. The multiple operations within a command must be the same operation type (copy or set) and have the same PML4 base address.

The *DmOpCnt* field specifies the number of operations to be performed within a single data mover command.

The *DmOp* field specifies the operation to be performed. See Table 20 for a list of operation encodings.

Command Name	*DmOp Field*	*Description*
Nop	0	No operation.
Copy	1	Perform copy operation with length *Length* from *Source Virtual Address* to *Destination Virtual Address* using virtual memory page table PLM4 base address *PLM4 Base*.
Set	2	Perform a set operation with length *Length* using data *Set Data* to *Destination Virtual Address* using virtual memory page table PLM4 base address *PLM4 Base*.

Table 20 – Data Mover Commands

Each operation includes a *Length* field which specifies the number of bytes for the operation.

The *Source Virtual Address / Set Data* field is used differently for copy and set operations. The field is used as the source virtual address for copy operations, and the set data for set operations. The *Destination Virtual Address* field specifies the destination address for both copy and set operations.

6.2.3 Steps Required to Issue a Data Mover Command and Obtain Status

The steps required to issue a data mover command are:

1. Read the value of the Issue / Status memory structure's *issue* field. Loop until the *Issue* field is zero.
2. Setup the new command by writing to the fields of the Command memory structure. The fields can be written in any order and any operand width.
3. Issue the new command by writing a one to the *Issue* field of the Issue / Status memory structure.

Once the data mover has been issued a command, the resulting status can be obtained with the following steps:

1. Read the value of the Issue / Status memory structure *Issue* field. Loop until the *Issue* field is zero.
2. Read the *Status* field from the Issue / Status memory structure.

The above described sequence uses standard cache coherency mechanisms and will function correctly in the presence of speculative memory accesses. Additionally, the mechanism will operate correctly for a system implements with the current FSB (Front Side Bus) technology or the QPI (Quick Path Interconnect) technology.

7 Exception Handling

7.1 Introduction

Exceptions are events that occur during the execution of an application that stop the sequence of instruction execution. Instruction execution may be resumed at a future point in time depending on the type of exception that occurs. The types of exception handled by the Convey coprocessor are Pause commands, Break instructions, Page Faults, and Unrecoverable Traps. These four exception types are described in the following sections.

7.2 Pause Command

A pause command is issued by the host processor when it wants the coprocessor to pause instruction execution. Pause commands may be issued due to a number of reasons. These include:

- The operating system must stop coprocessor execution to perform an application context switch.
- The operating system must stop coprocessor execution due to a Unix signal directed at the application running on the coprocessor.
- A debugger needs to gain control of the application running on the coprocessor.

Pause commands are received by the coprocessor asynchronous to the stream of instructions being executed.

The following sequence of events occurs within the coprocessor when a pause command is received:

1. The AEH instruction processor stops issuing new instructions to the coprocessor execution units.
2. The coprocessor allows all currently executing instructions to complete. Note that the currently executing instructions may cause an exception due to a page fault or an unrecoverable trap. Refer to section 7.6 below for a discussion on how multiple simultaneous exceptions are handled.
3. The coprocessor responds to the pause command once all instruction execution has completed.

Refer to section 5.7.1 for a description of how the host processor issues the Pause command and obtains Pause command status.

7.3 Break Instruction

A break instruction causes the execution of instructions to be paused at the point in the routine where the break instruction is located. Once the execution of instructions is paused, all instructions that entered execution prior to the break instruction are completed. The sequence of events for break instruction processing is:

1. The instruction processor stops issuing new instructions to the coprocessor execution units.
2. The coprocessor allows all currently executing instructions to complete. Note that the currently executing instructions may cause an exception due to a page fault, or an unrecoverable trap. Refer to section 7.6 below for a discussion on how multiple simultaneous exceptions are handled.
3. The coprocessor informs the host processor of the executed break instruction via an interrupt once all instruction execution has completed.

Refer to section 5.5 for a description of the mechanism for sending interrupts to the host processor.

7.4 Page Fault

A TLB miss occurs when a coprocessor load or store instruction or an instruction address references a virtual address that does not have an associated mapping entry in the coprocessor's TLB table. There are two outcomes for a TLB miss. The possible outcomes are 1) the referenced virtual address is a valid address for the application and the associated mapping just needs to be inserted into the TLB, or 2) the referenced virtual address is invalid and the application must be terminated. The sequence of events for TLB miss processing is:

1. The coprocessor references memory to determine if a TLB entry for the accessed virtual address exists in the memory based virtual-to-physical page table.
2. If the TLB entry is found in memory, then the coprocessor inserts the TLB entry into the coprocessor's TLB and coprocessor execution proceeds (without issuing an interrupt to the host processor).

   If the TLB entry is not found in memory, then the coprocessor allows all currently executing instructions to complete or stall. Note that the currently executing instructions may cause additional exceptions due to a break instruction or unrecoverable traps. Refer to section 7.6 below for a discussion on how multiple simultaneous exceptions are handled.
3. Once all processing has stopped, if the highest priority exception is a TLB Fault then a TLB Fault event is recognized and an interrupt is issued to the host processor.

   If the highest priority exception is an unrecoverable trap, then a Trap event is recognized and an interrupt is issued to the host processor. Note that if a Trap is recognized, then the TLB Fault will be ignored and never recognized since execution cannot be resumed from an unrecoverable trap.
4. If the OS is able to resolve the TLB fault by inserting a new virtual-to-physical mapping table entry in memory, then the OS informs the coprocessor that the page fault has been resolved using a TLB Retry command. The coprocessor then performs the hardware based lookup to obtain the needed TLB entry. Once the new entry is in the coprocessor TLB, then the referenced virtual address that caused the TLB miss is retried.

   If the OS determines that the referenced virtual address is invalid, then it terminates the offending application. The host processor informs the coprocessor to abort the currently executing routine and returns to the idle state using the TLB Abort command.

Refer to section 5.5 for a description of the mechanism for sending interrupts to the host processor.

7.5 Unrecoverable Trap

Traps are caused by the detection of an illegal condition during the execution of an instruction. The general categories for traps are:

- Instruction decode traps
- Floating point traps
- Integer overflow traps
- Memory reference traps

Unrecoverable traps are unique when compared to the other types of traps described in this section in that the only outcome for the trap is the termination of the application. However, before the application is terminated, the state of the application must be made available to the operating system to allow a user (through the use of a debugger) to determine the cause of the trap.

The Convey coprocessor supports two types of trap models: precise and imprecise. These two models provide a tradeoff between application performance and ease of determining the cause of a trap.

7.5.1 Precise Trap Model

The precise trap model limits instruction execution concurrency to ensure that the state of the coprocessor reflects the precise local address of the trap with respect to stream of instructions before and after the instruction causing the trap.

In the precise trap model, at the point in time that the trap is signaled to the host processor (via an interrupt) the state of the coprocessor will reflect all instructions initiated prior to the instruction causing the trap to be complete. Additionally, the precise trap model ensures that instructions have not started which follow the trap causing instruction.

Application performance will be degraded in precise trap mode. The degradation is caused by the limitation of instruction concurrency required to provide the precise instruction state. The precise trap mode is expected to be used as an application debugging mode only.

7.5.2 Imprecise Trap Model

The imprecise trap model allows the highest level of application performance with full instruction concurrency. When a trap occurs in this model, the user visible state will normally reflect many partially completed instructions. Determining the exact cause of the trap will be difficult if not impossible. The program counter register value will reflect the last instruction that was processed for execution by the functional units (A, S and AE). The actual instruction that caused the trap will be within the previous window of executed instructions.

7.5.3 Unrecoverable Trap Processing Steps

The following steps are used to process traps:

1. The coprocessor stops issuing new instructions to the execution units (A, S and AE).
2. The coprocessor allows all currently executing instructions to complete or stall. Note that the currently executing instructions may cause additional exceptions

due to TLB misses, or unrecoverable traps. Refer to section 7.6 for a discussion on how multiple simultaneous exceptions are handled.

3. Once all instruction execution has stopped, then the coprocessor will inform the host processor of the trap via an interrupt.

Refer to section 5.5 for a description of the mechanism for sending interrupts to the host processor.

7.6 Exception Handling Priority

Typically multiple instructions are executed concurrently on the coprocessor to achieve the highest possible application performance. One result of this concurrency is that multiple exceptions, of the same or different type, can occur at nearly the same point in time.

An example of multiple exceptions could start with the execution of a break instruction. While the coprocessor is completing execution of previously started instructions, a TLB miss could be detected on a load or store instruction. After the TLB miss is detected a floating point trap could occurs. It is possible that the host processor issues a Pause command during the period of time that the previous three exceptions occur (break instruction, TLB Fault and trap).

A priority order is used to process the multiple simultaneous exceptions. A specific processing order is required due to the different characteristics of each exception type. The following table shows the characteristics for each exception type.

Exception	*Processing Priority*	*Retry*	*Context Save*
Trap	1	No	No
Page Fault	2	Yes	No
Break Instruction	3	Yes	Yes
Pause Command	3	Yes	Yes

Table 21 - Exception Types

The priority order is defined based on the characteristics of each exception type. Traps have highest priority because the operation that caused the trap is not able to be retried. If multiple exceptions occur simultaneously and one of the exceptions is a trap, then it must be handled first.

Page faults have second highest priority because the context of the coprocessor at the time of the page fault cannot be saved to memory. Saving context at the time of a page fault would require saving the state of partially executed instructions.

Break instructions and pause commands have the lowest priority. These operations are performed in a way that allows a clean break between instructions in the instruction stream. Once a break instruction or pause command has been issued and all prior instructions have completed execution, then all application context can be saved to memory. The context saved in memory can be accessed by the operating system on behalf of a debugger to provide application state to the user. Later the memory context can be reloaded to the coprocessor and the application’s execution resumed.

8 Coprocessor Instruction Set Architecture

The Convey Coprocessor Instruction Set Architecture (ISA) is composed of multiple separate ISAs that can be layered to include additional functionality. Figure 20 below illustrates the Convey Coprocessor ISA model.

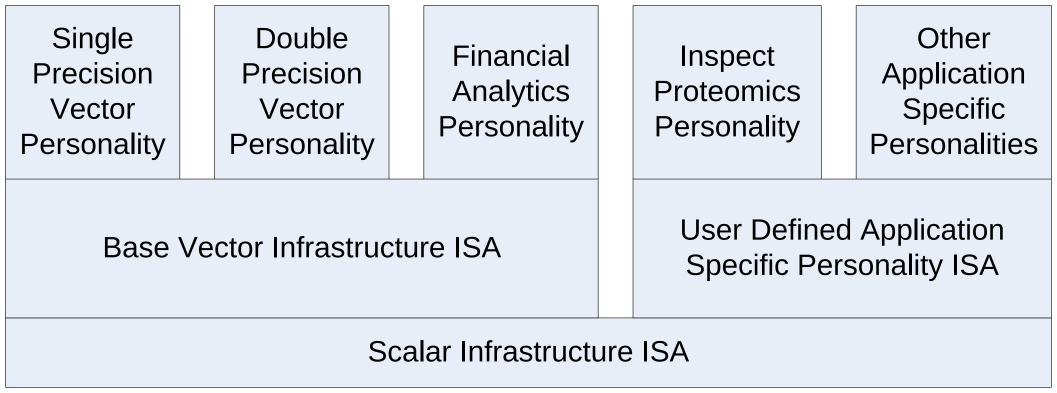

Single Precision Vector Personality
Double Precision Vector Personality
Financial Analytics Personality
Inspect Proteomics Personality
Other Application Specific Personalities
Base Vector Infrastructure ISA
User Defined Application Specific Personality ISA
Scalar Infrastructure ISA

Figure 20 - Convey Coprocessor ISA Model

The top row of blocks in the figure above are personalities. Below each personality block are the infrastructure ISAs that provide the base functionality for that personality. As an example, the Single Precision Vector Personality (SPV) relies on the Base Vector Infrastructure (BVI) and Scalar Infrastructure. The Scalar infrastructure is included in all coprocessor personalities. The Scalar infrastructure provides coprocessor program counter control, context save and resource functionality, as well as a base set of scalar instructions that modify the A and S register machine state. The Single Precision Vector personality also includes the Base Vector Infrastructure. The BVI provides the base vector machine state (vector register file and vector mask registers), memory load and store functionality, vector address calculation instructions, as well as other base vector operations. The Single Precision Vector personality defined the operations that are performed within the function units of the Base Vector Infrastructure.

The following Instruction Set Architectures are defined for the Convey Coprocessor:

- Scalar Infrastructure ISA

  The Scalar Infrastructure provides the A and S register machine state and all instructions that modify that machine state. The Scalar infrastructure is included in all personalities.

- Base Vector Infrastructure ISA

  The Base Vector Infrastructure provides the Vector and Vector Mask register machine state and instructions that modify that machine state. The BVI defines instructions to load / store the Vector and VM register machine state, defines the exception model used for vector oriented processing, and provides instructions used for vector address generation. The BVI does NOT define the non-integer data manipulation operations that are performed within the function units. The BVI infrastructure is included in all vector personalities.

- User Defined Application Specific Personality ISA

  The User Defined Application Specific Personality (ASP) defines a set of instructions that allow a user to define the complete behavior of the coprocessor application engines. The set of pre-defined instructions provide a generic way control the application engines. Personalities that are constructed using the User Defined Application Specific Personality ISA must also include the Scalar Infrastructure.

The remainder of the chapter presents common instruction formats used by all coprocessor instructions.

Subsequent chapters present the Scalar Infrastructure, Base Vector Infrastructure, User Defined Application Specific Personality Infrastructure.

8.1 Instruction Bundle Format

Convey Coprocessor instructions are either 64 or 128 bits wide. Instructions are aligned to 8-byte boundaries. The 64-bit formats include two of the three types of instructions (A, S or AE) bundled as a single instruction. The 128-bit formats always include a 64-bit constant value. The bundling of instructions allows multiple independent instructions to be processed in parallel. The figure below shows the available instruction formats.

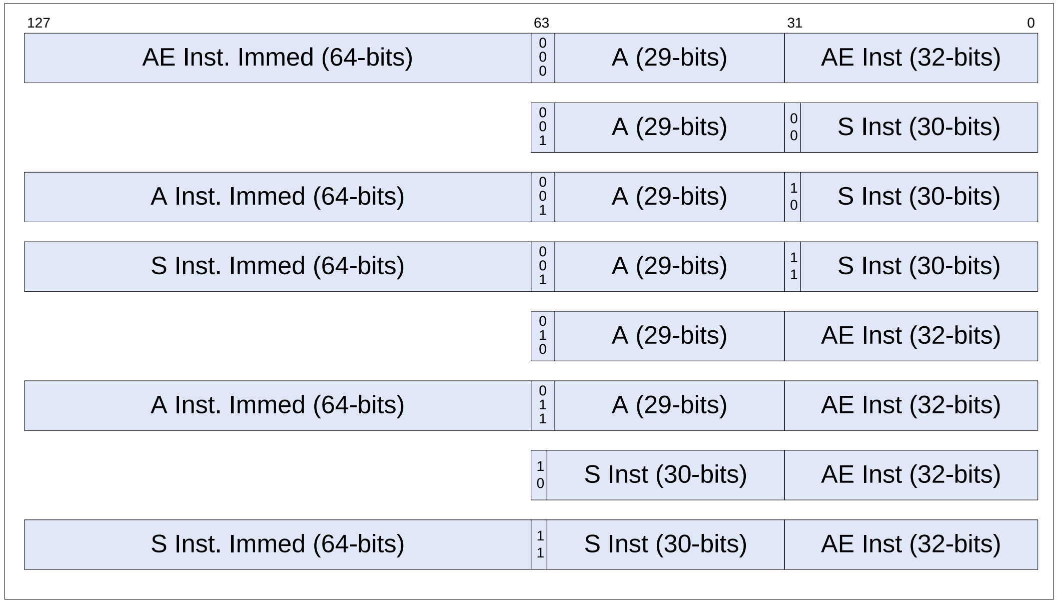


Figure 21 - Instruction Formats

As can be seen from Figure 21 above, the A, S and AE instructions have different widths (29, 30 and 32-bits respectively). However, the width is consistent for each instruction type across all instruction formats.

Some instructions require a constant value. The A, S and AE instruction formats include small constant fields (6, 10, 12 or 18 bits wide). The 64-bit instruction constants are used when larger constants are required. The 64-bit constant is used for providing a 64-bit address, large integer constants, or floating point constants.

Within an instruction bundle there are no inter-operation dependencies (I.e. the input operands of the A, S and AE instructions are produced from previous instructions in the instruction stream, not from the current instruction bundle). This allows the coprocessor to proceed in parallel with the bundled instructions without regard to register dependencies.

The order of execution of two memory reference instructions within a bundle is not defined. As an example, if a bundle contains two memory instructions with one being an A register store to address X and the other being an S register load from address X, then the S register load may complete with either the new or the previous value at address X. For the purposes of instruction bundling, the FENCE instruction is considered an A register memory reference instruction.

Execution of an undefined bundle format results in no architectural state change. Undefined bundle formats include bundles with bits <63:61> equal to one with bits <31:30> equal to one. Undefined bundle formats do not have extended immediate values.

8.2 A, S and AE Instruction Formats

The format of the A, S and AE instructions are nearly identical. The major difference is the use of the upper three bits of the formats.

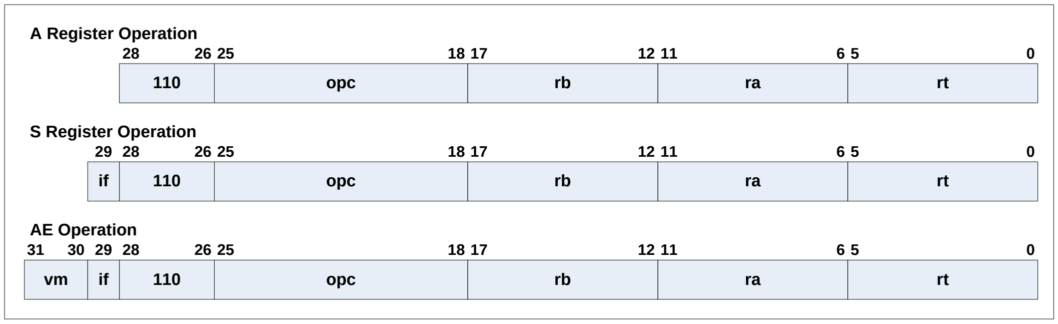


Figure 22 - Common instruction formats

The A register instruction formats are 29-bits wide (bits <28:0>). Bits <31:29> are not part of the A register instruction format, but rather encode the instruction type. See Figure 21 above for the values of the instruction type field.

The S register instructions use bit 29, the *if* field, to specify whether the data type of the instruction is integer or floating point. Bits <31:30> are not part of the S register instruction format. Bits <31:30> are used to specify the instruction type. See Figure 21 above for the values of the instruction type field.

The AE instructions use all 32-bits of the instruction format. The AE instruction format shown in Figure 22 is an instruction from a personality based on the vector infrastructure. The *vm* field specifies instructions that are to be performed conditionally based on a vector mask.

8.3 Unimplemented Instructions

The opcode space for the A, S and AE instructions is sparsely populated and contains many unimplemented instruction encodings. Execution of an unimplemented instruction is equivalent to the execution of the NOP instruction (i.e. no user defined state changes value other than the IP is advanced to the next instruction). There is one exception to an unimplemented instruction executing as a NOP. The exception is that all AE instructions within a range of opcodes has the same characteristics as to whether an A and S register value is used as a source operand or as a destination result. Execution of an AE instruction performs the A and S register accesses independent of whether the instruction is implemented or not. The result is that an SRRE exception (Scalar Register Range) can be signaled for unimplemented AE instructions. Refer to Appendix A for the defined opcode ranges and A/S register interactions.

Note that the instruction bundle format indicates if an extended immediate value exists. As a result, an unimplemented instruction may include an extended immediate.

9 Coprocessor Scalar ISA

The Scalar ISA is implemented in the Application Engine Hub and is available for all personalities. The Scalar ISA consists of a set of machine state extensions plus the instructions to manipulate the machine state.

9.1 Scalar ISA Machine State

The following figure shows the register set that is common for all application engine configurations. Each application engine configuration extends the common register set with registers specific to the configuration. The configuration specific extensions are defined in later sections.

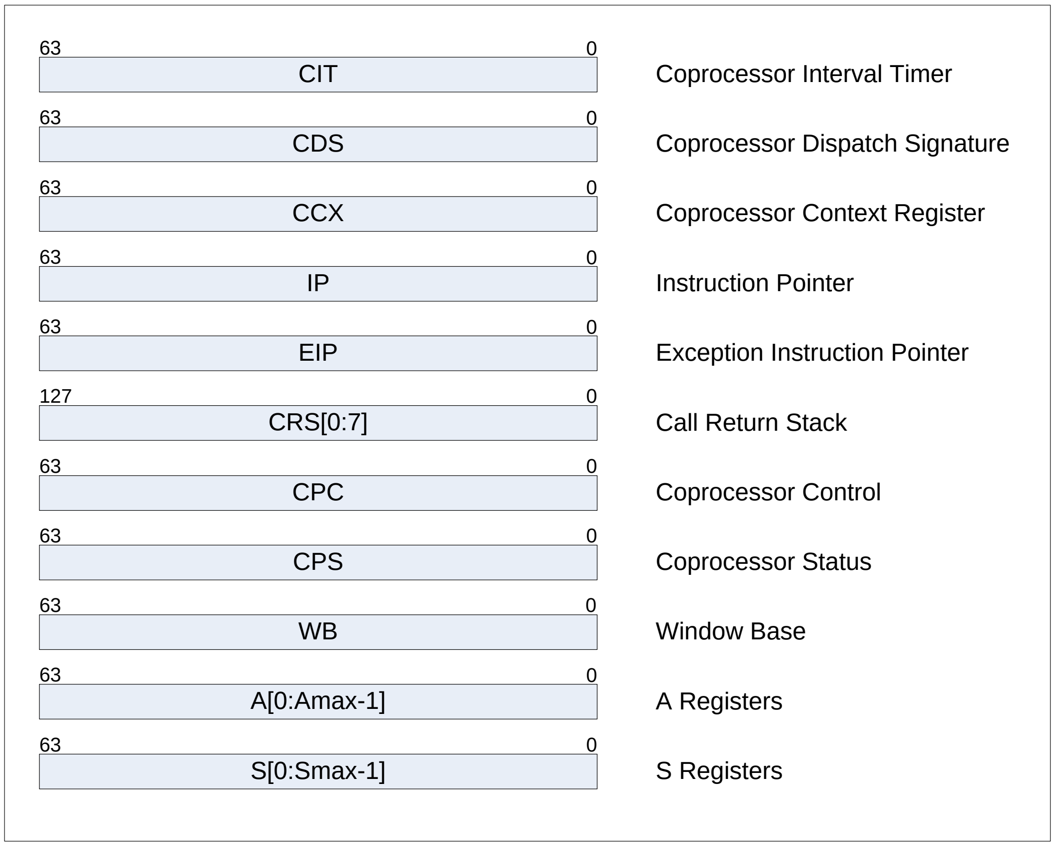


Figure 23 – Scalar ISA Machine State

9.1.1 CIT – Coprocessor Interval Timer

The CIT register is a free running counter that can be used as a time stamp or to obtain interval time within dispatched routines. The counter counts at the system clock rate of the coprocessor. The CIT register is read only.

9.1.2 CDS – Coprocessor Dispatch Signature

The CDS register is a read only register that provides access to the Dispatch Signature value within the User Command structure that initiated the coprocessor routine.

9.1.3 CCX – Coprocessor Context Register

The CCX register is used by context save and restore routines. The contents of the register is not maintained across context save / restores. Normal user code should not use this register.

9.1.4 IP – Instruction Pointer

The IP register holds the address for the next instruction to be executed. The register is initialized by the *Dispatch Instruction Pointer* field of a Command memory structure when a coprocessor routine is issued from the host processor. Each executed coprocessor instruction advances the IP register to the next instruction to be executed.

The IP register is 64-bits wide and uses the host processor's canonical address format. The host processor's canonical format defines the upper 16-bits of all virtual addresses to be a sign extension of bit 47.

9.1.5 PIP – Paused Instruction Pointer

The PIP register holds the address for the next instruction to be executed when the normal program execution flow is paused. The events that cause the PIP register to be loaded are: host processor issues a Pause command, host processor issues a TLB Abort command, execution of a Break instruction, or an instruction exception is detected. The PIP register is saved to memory as part of a coprocessor context save operation, and restored as part of coprocessor context load operation. A supervisor Resume command causes the PIP register to be copied to the IP register as the starting addresses for instruction execution.

The PIP register is 48-bits wide and sign extended to 64-bits on a read.

9.1.6 CRS – Call Return Stack

The call return stack holds a limited number of return instruction pointers and window base register values within the coprocessor. The call return stack can handle a call depth of up to eight. Execution of a call instruction results in a return instruction pointer being written to the CRS and the CRT register being incremented by one. Execution of a return instruction results in the CRT register being decremented by one and the instruction pointer being removed from the CRS. Note that the CRT (Call Return Top) value is a field of the Coprocessor Status register (CPS). The CRS registers are non-persistent state.

The format of a CRS entry is shown in the figure below.

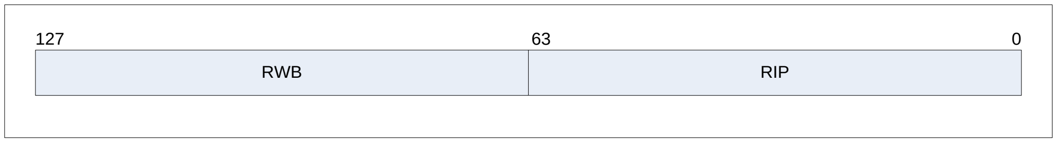


Figure 24 - CRS Entry Format

The following code shows a typical coprocessor subroutine calling sequence.

```
        WBINC.P     1,4,3,2          ; Write the WB register to CRS.RWB and set
                                     ;   CPC.WBV to 1, Increment WB.AWB by 2,
                                     ;  WB.SWB by 3, WB.VWB by 4, WB.VMWB
                                     ;  by 1 and set WB.VRRS and WB.VMA to zero.
        CALL      target      ; jump to target, write return instruction pointer
                                     ;  to CRS.RIP and increment CPS.CRT
```

The WBINC instruction increments the window base registers by the specified amount and sets the VMA and VRRS fields to zero. The WBINC instruction also optionally writes the window base register to the CRS. The CALL instruction writes the return instruction pointer to the CRS and then increments the CRT register. Eventually a RTN instruction is executed to return from the called subroutine. The RTN instruction decrements the CRT field value, restores the IP register to the CRS return instruction pointer, and restores the window base register.

9.1.7 CPC – Coprocessor Control Register

The CPC register controls various aspects of the coprocessor. The figure below shows the defined fields. The register can be read and written by coprocessor instructions.

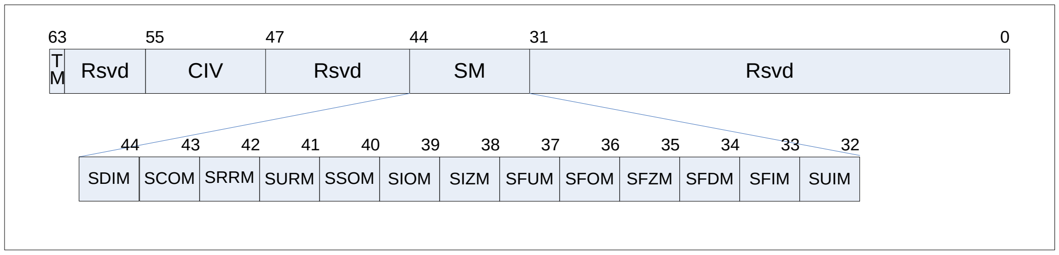


Figure 25 - Coprocessor Control Register

The fields of the CPC register are defined in the following table.

Field Name	*Bit Range*	*Description*
SM – Scalar Mask	44:32	The SM field is used to mask the exception field of the CPS register. A mask value of zero allows an exception to issue a trap to the host processor. Note that some exception types cannot be ignored. The SM field is persistent state. Refer to section 9.1.8.2 for details on each exception type.
CIV – Scalar ISA Version	55:48	The CIV field indicates the Scalar ISA version number. The initial value is one.
TM – Trap Mode	63	The TM field is used to control whether the coprocessor operates in Imprecise Trap Mode (0) or Precise Trap Mode (1). The TM field is persistent state. See section 7.5 for more details.

9.1.8 CPS – Coprocessor Status Register

The CPS register provides status of various aspects of the coprocessor. The figure below shows the defined fields.

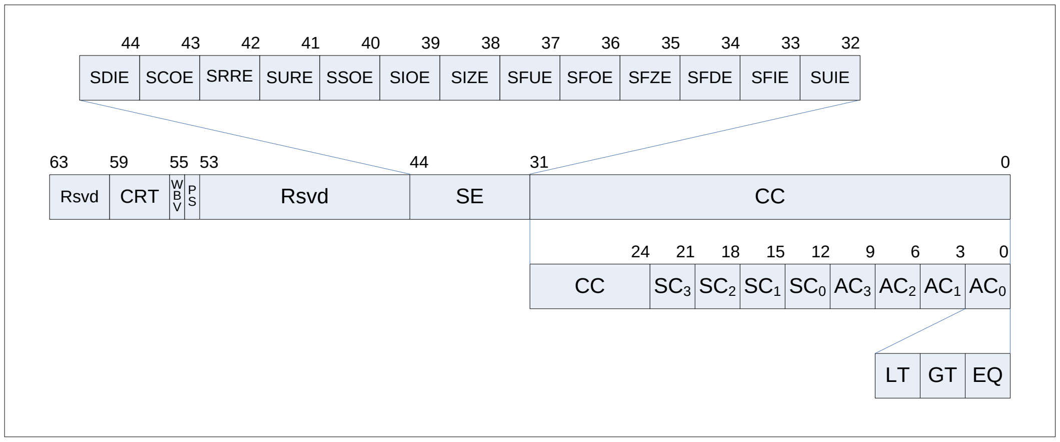


Figure 26 - Coprocessor Status Register

The fields of the CPS register are defined in the following table.

Field Name	*Bit Range*	*Description*
CC	31:0	The condition codes field used for program flow decisions. The CC field is non-persistent state. Refer to section 9.1.8.1 for further detail.
SE	44:32	The scalar exception field indicates that an exception has occurred. Exceptions that are not mask result in non-recoverable traps. The SE field is initialized to zero by a coprocessor dispatch. Refer to section 9.1.8.2 for details on each exception type.
PS	54	The PS bit indicates whether the present routine was entered with persistent state from the previous routine. The bit is a copy of the PS field within the User Command structure that initiated the previous dispatched routine.
WBV	55	The window base valid field indicates whether the CRS.RWB field is valid. The CALL instruction uses the WBV field to determine if the RWB field is valid or whether it needs to write the WB register to the CRS.RWB field. The WBV field is set to one by the WBINC.P instruction and set to zero by a CALL or RTN instruction. The WBV field is initialized to zero by a coprocessor dispatch.
CRT	59:56	The call stack top field is used as the call return stack (CRS) top pointer. The CRT field is initialized to zero by a coprocessor dispatch.

9.1.8.1 CPS Condition Code Field (CC)

The CC field holds the condition codes used for determining the direction to take for conditional program flow instructions. The format of the CC field is shown in Figure 26.

The CC field is composed of multiple 3-bit fields. Each 3-bit field is a collection of condition codes that are simultaneously set by the execution of a compare instruction. Four of the 3-bit fields are used by A-register instructions, and four by S register instructions.

A set of logical instructions are provided that allow operations to be performed on individual bits within the CC register. As an example, the instruction ‘and CC3,CC5,CC30’ would locally AND the values of CC register bit 3 with CC register bit 5 and store the result in CC register bit 30. The CC logical instructions allow high level program conditional statements to be performed without moving condition codes to general registers.

The CC logical instructions use a 5-bit field to select the input operand bits allowing all bits within the CC field of the CPS register to be selected. The result of the CC logical instructions is written to one of the 32 CC field bits.

Conditional flow control instructions (i.e. branch, call and return) use a five bit field to select one of the condition codes to be used to determine the direction taken for the conditional flow control instruction. The specific CC bit can be specified as CCn, where n is 0-31, or by specifying the bit functionally as ‘bit-set.compare-type’. Bit-set would be one of AC0, AC1, AC2, AC3, SC0, SC1, SC2 or SC3. Compare-type can be one of LT, LE, GT, GE, EQ, or NE. The compare type specifies the bit within the bit set and the polarity of the comparison. The following assembly instructions are equivalent:

```
BR.F    CC3,target
BR      AC1.NE,target
```

9.1.8.2 CPS Scalar Exception Field (SE)

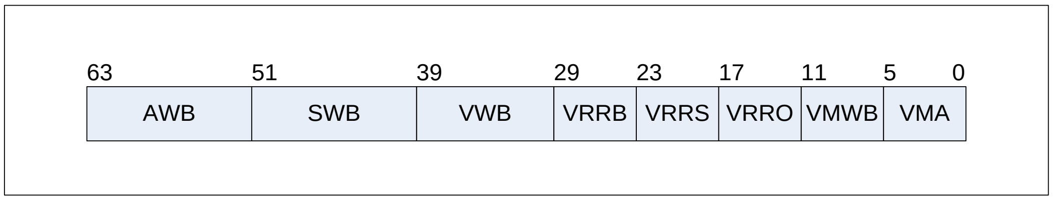
Instructions check for exceptions during their execution. Detected exceptions are recorded in the Scalar Exception field. The Scalar Exception field is masked with the CPC SM field to determine if a recorded exception should interrupt the host processor. The result of the following exception checks is recorded.

Exception Name	*Bit within SE field*	*Description*
SUIE	32	Scalar Unimplemented Instruction Exception. An attempt was made to execute an unimplemented instruction. Unimplemented instructions are treated as a NOP.
SFIE	33	Scalar Floating Point Invalid Operand Exception. A floating point instruction detected invalid input operands for the operation to be performed.
SFDE	34	Scalar Floating Point Denormalized Exception. A floating point instruction detected a denormalized input operand.
SFZE	35	Scalar Floating Point Divide By Zero Exception. A floating point divide instruction detected a zero denominator operand.

SFOE	36	Scalar Floating Point Overflow Exception. A floating point instruction produced a result with an exponent value that was too large for floating point data format. The result of a floating point operation that overflows is infinite.
SFUE	37	Scalar Floating Point Underflow Exception. A floating point instruction produced a result with an exponent value that was too small for floating point data format. The result of a floating point operation that underflows is zero.
SIZE	38	Scalar Integer Divide By Zero Exception. An integer divide instruction detected a zero denominator operand.
SIOE	39	Scalar Integer Overflow Exception. An integer instruction produced a result that exceeded the range of the data format.
SSOE	40	Scalar Store Overflow Exception. An A or S-register integer store was performed where the value to be stored exceeded the range of the memory data size. The lower bits of the source register are stored to memory when the exception is detected.
SURE	41	Scalar Unaligned Memory Reference Exception. An instruction referenced memory using an address that was not aligned for the size of the access. An unaligned memory reference is performed by forcing the least significant 3 bits of the virtual address are forced to zero.
SRRE	42	Scalar Register Range Exception. An instruction referenced an A or S register where the sum of the instruction's register index field and the A or S window base register value exceeded the count of A or S registers. The modulo function is used to generate the register index when the register range is exceeded.
SCOE	43	Scalar Call Return Stack Overflow Exception. A CALL instruction was executed when the Call Return Stack was full. The call instruction that caused the overflow is completed. When the SCOE bit is set then execution of a return instruction causes the dispatched routine to complete.
SDIE	44	Scalar Dispatch Invalid Exception. An invalid dispatch was detected. Errors that would cause an SDIE exception are: greater than 28 quad word parameters specified as input parameters or greater than 6 quad words specified as returned results.

9.1.9 WB – Window Base Register

The fields of the Window Base register are shown in the following figure.



The window base register fields are used to map instruction register fields to the index of the actual register to be accessed. Four types of registers are mapped by the Window Base register: A, S, V and VM. The V and VM registers are defined by the Single Precision and Double Precision Application Engine personalities. Note that future AE personalities may use the V and VM Window Base register fields for other purposes.

The entire Window Base register is pushed on the Call / Return Stack (CRS) when a ***WBINC.P*** instruction is executed, and the Window Base register is restored when a ***RTN*** instruction is executed.

All fields of the WB register are initialized to zero by a coprocessor dispatch.

Field Name	*Bit Range*	*Description*
AWB	63:52	The A register window base field is used to specify the base of the accessible A registers. The A register index mapping function is (AWB + Ax), where Ax is an instruction’s A register field. If the resulting value is greater than the actual number of A registers then a Scalar Register Range Exception (SRRE) is signaled.
SWB	51:40	The S register window base field is used to specify the base of the accessible S registers. The S register index mapping function is (SWB + Sx), where Sx is an instruction’s S register field. If the resulting value is greater than the actual number of S registers then a Scalar Register Range Exception (SRRE) is signaled.
VWB	39:30	The V register window base field is used to specify the base of the accessible V registers. The V register index mapping function is (VWB + Vx), where Vx is an instruction’s V register field. If the resulting value is greater than the actual number of V registers then an AE Register Range Exception (AERRE) is signaled. Note that the V register index mapping function (VWB + Vx) applies when VRRS field is zero. Refer to the VRRB field description for the mapping function when the VRRS field is non-zero.
VRRB	29:24	The V Register Rotation Base field is used to specify the base of the V registers that are indexed with a rotation function. The V register index mapping function is: If (Vx >= VRRB && Vx < VRRB + VRRS) // within rotation range

		Vidx = VWB + VRRB + (Vx – VRRB + VRRO) % VRRS;
		else
		Vidx = VWB + Vx;
		Where Vx is an instruction’s V register field. If the resulting value is greater than the actual number of V registers then an AE Register Range Exception (AERRE) is signaled.

VRRS	23:18	The V register rotation size field specifies the number of V registers that are included in the rotation function. A value of zero indicates that register rotation is disabled. The maximum rotating register size is 63.
VRRO	17:12	The V register rotation offset field specifies the offset to be used within the V register rotation region. The VRRO field can be incremented with the ***VRRINC*** instruction. Note that if the VRRO field becomes greater than or equal to the VRRS field then the VRRO field is set to zero.
VMWB	11:6	The VM register window base field is used to specify the base of the accessible VM registers. The VM register index mapping function is (VMWB + VMx), where VMx is an instruction’s VM register field. If the resulting value is greater than the actual number of VM registers then an AE Register Range Exception (AERRE) is signaled.
VMA	5:0	The VM active field is used to specify the active VM register. The active VM register is used as the source of the vector mask for under mask operations. The index mapping function for accesses using the VMA field is (VMWB + VMA). If the resulting value is greater than the actual number of VM registers then an AE Register Range Exception (AERRE) is signaled.

9.1.10 A Registers

The A registers are a set of general purpose registers intended to be used to manipulate addresses for S and AE register loads and stores. Additionally, the A registers are used to handle loop counts and calculating vector length, vector stride, vector partition length and vector partition stride.

The number of A registers is not architecturally defined, but rather is implementation dependent. The name Amax is used to refer to the number of implemented A registers.

The A registers are non-persistent state.

9.1.11 S Registers

The S registers are a set of general purpose registers intended to be used to manipulate scalar data for S and V operations.

The number of S registers is not architecturally defined, but rather is implementation dependent. The name Smax is used to refer to the number of implemented S registers.

The S registers are non-persistent state.

9.2 Operand Register Selection

The formats for the A, S and AE operations use 6-bit register fields. This allows an operation to directly specify one of 64 registers to be used as the source or destination for an operation. However, the number of A, S and AE registers can be greater than 64. For A, S and AE operations, the 6-bit register field value is used as an offset for the A, S and AE window base registers. The window base register plus the register field value is used to select one of the A, S or AE registers. The figure below shows the mechanism for S register selection. The mechanism for A and AE register selection is similar.

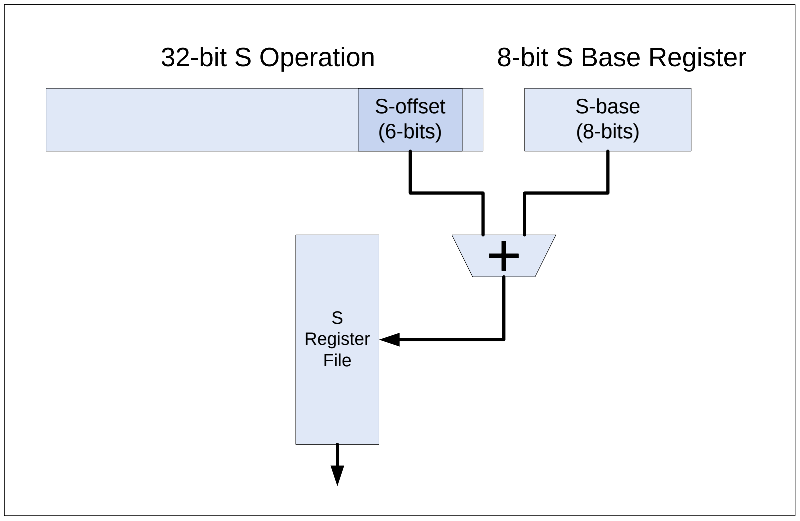


Figure 27 – S Register Selection Mechanism

The mechanism provides a window of registers that are directly accessible by A, S and AE operations. The mechanism allows all operation formats to use 6-bit register fields and provides a convenient means to allow assembly code reuse.

9.2.1 A and S register zero

Specifying a zero in an instruction's A or S register field results in the value zero being used for the input to the operation. This is particularly useful for load or store instructions where the memory address is specified as offset plus base register and the register value needs to be zero.

Instruction writes to registers A0 and S0 are ignored but all other instruction effects occur (such as traps or exceptions).

9.2.2 Global A-register Selection

The first eight A-registers (A0-A7) are directly accessible independent of the A window base value. These registers hold values that are needed across subroutine calls. As an example, the stack pointer register must be accessible across subroutine.

Register A0 is defined to hold the value zero (see section 9.2.1). Registers A1-A7 are general purpose registers allowing software to define their usage.

The figure below shows illustrates the A-register selection mechanism. The AWB register has the value 24 in the example. The accessible A-registers are the global register A0-A7 and registers A32-A88 (accessed as A8-A63).

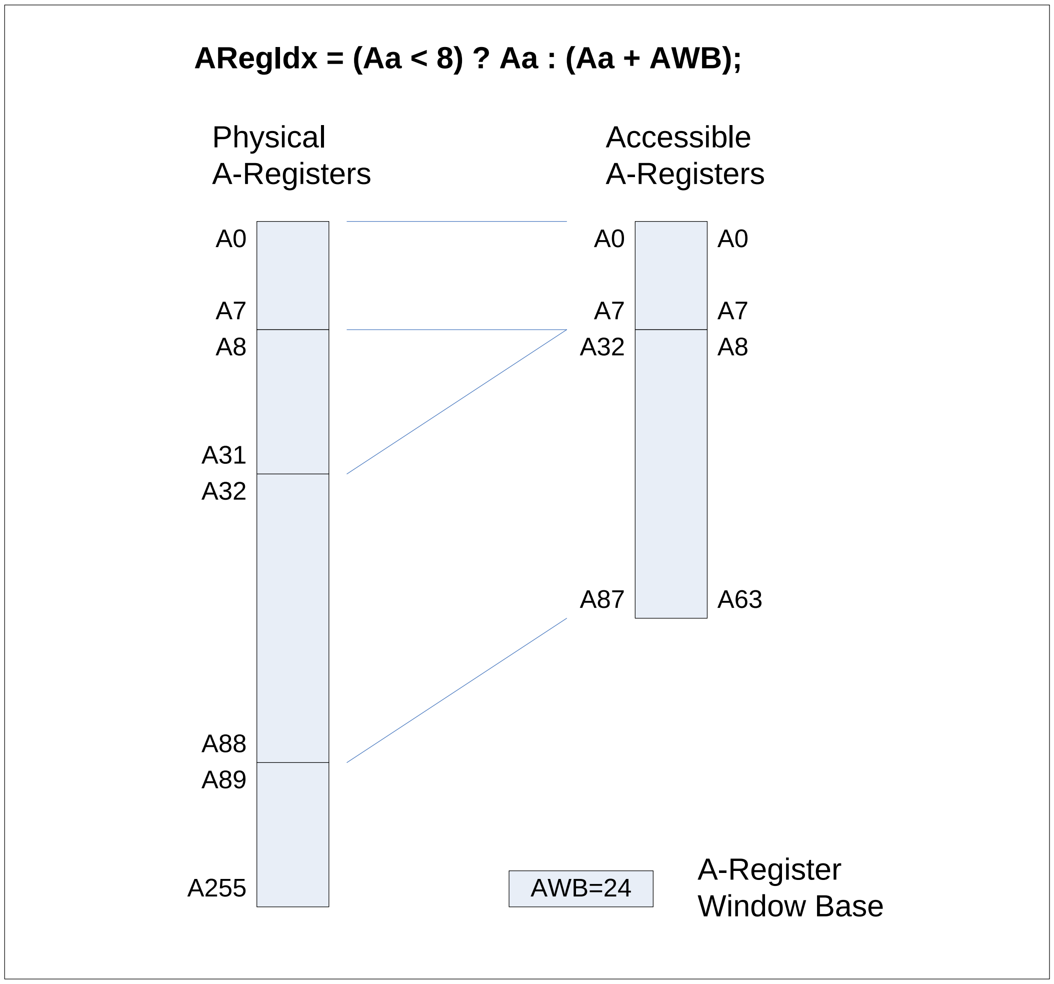


Figure 28 - Global A-Register Selection

9.2.3 Example Usage of Window Base Registers

This section presents two examples for usage of the window base registers. The first example shows how the window base registers can eliminate the need for a memory based stack when making subroutine calls.

9.2.3.1 Call / Return stack using Window Base Registers

The example code below shows the entry point for a dispatch routine, ‘Dispatch Entry’. The dispatch passes one A-register parameter to the coprocessor in register A8. The first instruction of the dispatch routine is setting up a parameter for the call to ‘Routine’. The first instruction adds eight to the dispatch parameter in A8 and places the result in A9. The value in A9 is used as the first parameter for the call to ‘Routine’. The WBINC instruction advances the AWB register by one.

Dispatch Entry:

```
            ADD.UQ        A8,8,A9          ; Initialize input parameter to Routine
            WBINC.P      1,0,0,0           ; inc A window base register
            CALL          Routine          ; push return IP on internal call/return stack

                                           ;  and jump to first instruction of Routine
Routine:   . . .

            LD.FD         0(A8),V0
            RTN                            ; return to instruction after call
```

The example uses the WBINC instruction to increment the AWB register by 1. The WBINC instruction advances the window base register to provide the called routine with input parameters at known register locations and unused registers beyond the parameter registers.

Note that the example above uses a vector register load operation. This operation is available in the double precision floating point application engine configuration.

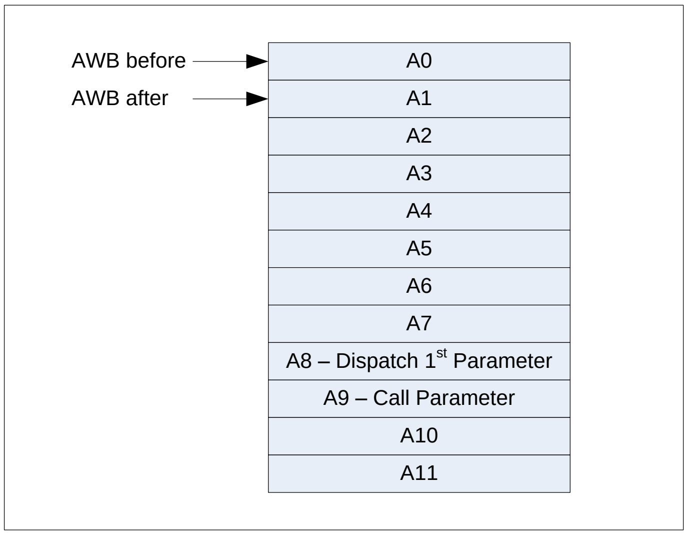


Figure 29 - Window Base Call / Return Example

The above figure shows the A registers and AWB before and after the above call sequence. Initially the AWB register is set to A0 at entry to a coprocessor dispatch routine. The first eight A-registers are reserved for global values. The AWB register is ignored when an instruction references one of the first eight A-registers. Dispatch A-register values are copied into A registers starting at A8. Incrementing the AWB register allows the called routine to reference its input parameter as A8 independent of the value of register AWB.

The example code uses register A8 to load vector register V0. The routine then returns using the RTN instruction. The RTN instruction restores the AWB, SWB and AEWB registers to the values prior to the execution of the WBINC instruction and sets the instruction pointer (IP) to the instruction after the CALL instruction.

9.2.3.2 Indirect access to the general purpose registers

The coprocessor has a large number of general purpose registers. There are situations where it may be necessary to initialize a significant number of these register. One example is when an algorithm requires using a large number of scalar register values, such as matrix multiple. Since an instruction’s register field can only reference up to 64 registers, it is necessary to modify the window base register to initialize greater than 64 registers.

A simple example that initializes 4 S-register values is shown below.

```
        LDI     0, A8
        LDI     address, A9
Loop:   LD.FD   0(A9), S1             ; Load S-Reg with double from memory
        LD.FD   8(A9), S2
        LD.FD   16(A9), S3
        LD.FD   24(A9), S4
        WBINC   0,4,0,0               ; Increment SWB by 4
        ADD     A9, 32, A9            ; Increment address by 4 doubles
        ADD     A8, 4, A8             ; Increment address by 4 doubles
        CMP     A8, 64, AC0           ; Check for loop complete
        BR.F    AC0.GE,Loop           ; Branch if not done
        WBDEC   0,64,0,0              ; Decrement SWB by 64
```

The code in the example is a simple loop. However, it allows 64 S registers to be loaded without having to lay down 64 individual S register load instructions (each specifying a different S register).

The windowing mechanism allows a routine to directly access up to 64 registers without having to modify the window base registers. Routines that need more than 64 registers will typically be using the S or AE registers as arrays of registers due to inner loop unrolling.

9.3 Scalar ISA Instruction Set

This section presents the instruction set that is common for all application engine configurations. The instructions are listed in functional groups.

9.3.1 Instruction Set Organized by Function

9.3.1.1 Program Flow Instructions

Operation		*Type*	*Operation Description*	*Encoding **
NOP		A,S	No Operation	F3, 1, 00
BRK		A	Break	F2, 02
BR	target(At)	A	Branch	F2, 04
BR	CCt, target	A	Conditional branch (no hint)	F2, 05
BR.BL	CCt, target	A	Conditional branch (likely)	F2, 06
BR.BU	CCt, target	A	Conditional branch (unlikely)	F2, 07
CALL	target(At)	A	Call	F2, 08
CALL	CCt, target	A	Conditional call (no hint)	F2, 09
CALL.BL	CCt, target	A	Conditional call (likely)	F2, 0A
CALL.BU	CCt, target	A	Conditional call (unlikely)	F2, 0B
RTN		A	Return	F2, 0C
RTN	CCt	A	Conditional return (no hint)	F2, 0D
RTN.BL	CCt	A	Conditional return (likely)	F2, 0E
RTN.BU	CCt	A	Conditional Return (unlikely)	F2, 0F

* The **Encoding** column specifies the instruction format type and *opc* field value for A type instructions, and the format, if, opc field values for S and AE type instructions . The symbol # indicates that the instruction is an alias provided by the assembler. Refer to Appendix A for a list of instruction format types.

9.3.1.2 Compare and Condition Code Logical Instructions

Operation		*Type*	*Operation Description*	*Encoding*
CMP.UQ	Aa,Immed,ACt	A	Compare unsigned immediate	F1, 3A
CMP.SQ	Aa,Immed,ACt	A	Compare signed immediate	F1, 3B
CMP.UQ	Sa,Immed,SCt	S	Compare unsigned immediate	F1, 1, 3A
CMP.SQ	Sa,Immed,SCt	S	Compare signed immediate	F1, 1, 3B
CMP.FS	Sa,Immed,SCt	S	Compare float single immed.	F3, 0, 3A
CMP.FD	Sa,Immed,SCt	S	Compare float double immed.	F3, 0, 3B
CMP.UQ	Aa,Ab,ACt	A	Compare unsigned	F4, 3A
CMP.SQ	Aa,Ab,ACt	A	Compare signed	F4, 3B
CMP.UQ	Sa,Sb,SCt	S	Compare unsigned	F4, 1, 3A
CMP.SQ	Sa,Sb,SCt	S	Compare signed	F4, 1, 3B
CMP.FS	Sa,Sb,SCt	S	Compare float single	F4, 0, 3A
CMP.FD	Sa,Sb,SCt	S	Compare float double	F4, 0, 3B
AND	CCa,CCb,CCt	A	Logical AND condition code	F4, 10

OR	CCa,CCb,CCt	A	Logical OR condition code	F4, 11
NAND	CCa,CCb,CCt	A	Logical NAND condition code	F4, 12
NOR	CCa,CCb,CCt	A	Logical NOR condition code	F4, 13
XOR	CCa,CCb,CCt	A	Logical XOR condition code	F4, 14
XNOR	CCa,CCb,CCt	A	Logical XNOR condition code	F4, 15
ANDC	CCa,CCb,CCt	A	Logical ANDC condition code	F4, 16
ORC	CCa,CCb,CCt	A	Logical ORC condition code	F4, 17

9.3.1.3 Load / Store Instructions

Operation		*Type*	*Operation Description*	*Encoding*
LD.UB	offset(Aa),At	A	Load unsigned byte	F1, 00
LD.UW	offset(Aa),At	A	Load unsigned word	F1, 01
LD.UD	offset(Aa),At	A	Load unsigned double word	F1, 02
LD.UQ	offset(Aa),At	A	Load unsigned quad word	F1, 03
LD.SB	offset(Aa),At	A	Load signed byte	F1, 04
LD.SW	offset(Aa),At	A	Load signed word	F1, 05
LD.SD	offset(Aa),At	A	Load signed double word	F1, 06
LD.SQ	offset(Aa),At	A	Load signed quad word	F1, 03 #
LD.UB	offset(Aa),St	S	Load unsigned byte	F1, 1, 00
LD.UW	offset(Aa),St	S	Load unsigned word	F1, 1, 01
LD.UD	offset(Aa),St	S	Load unsigned double word	F1, 1, 02
LD.UQ	offset(Aa),St	S	Load unsigned quad word	F1, 1, 03
LD.SB	offset(Aa),St	S	Load signed byte	F1, 1, 04
LD.SW	offset(Aa),St	S	Load signed word	F1, 1, 05
LD.SD	offset(Aa),St	S	Load signed double word	F1, 1, 06
LD.SQ	offset(Aa),St	S	Load signed quad word	F1, 1, 03 #
LD.FS	offset(Aa),St	S	Load float single	F1, 1, 02 #
LD.FD	offset(Aa),St	S	Load float double	F1, 1, 03 #
LD.UB	Ab(Aa),At	A	Load unsigned byte	F4, 00
LD.UW	Ab(Aa),At	A	Load unsigned word	F4, 01
LD.UD	Ab(Aa),At	A	Load unsigned double word	F4, 02
LD.UQ	Ab(Aa),At	A	Load unsigned quad word	F4, 03
LD.SB	Ab(Aa),At	A	Load signed byte	F4, 04
LD.SW	Ab(Aa),At	A	Load signed word	F4, 05
LD.SD	Ab(Aa),At	A	Load signed double word	F4, 06
LD.SQ	Ab(Aa),At	A	Load signed quad word	F4, 03 #
LD.UB	Ab (Aa),St	S	Load unsigned byte	F4, 1, 00
LD.UW	Ab (Aa),St	S	Load unsigned word	F4, 1, 01
LD.UD	Ab (Aa),St	S	Load unsigned double word	F4, 1, 02
LD.UQ	Ab (Aa),St	S	Load unsigned quad word	F4, 1, 03
LD.SB	Ab (Aa),St	S	Load signed byte	F4, 1, 04
LD.SW	Ab (Aa),St	S	Load signed word	F4, 1, 05
LD.SD	Ab (Aa),St	S	Load signed double word	F4, 1, 06

LD.SQ	Ab(Aa),St	S	Load signed quad word	F4, 1, 03 #
LD.FS	Ab(Aa),St	S	Load float single	F4, 1, 02 #
LD.FD	Ab(Aa),St	S	Load float double	F4, 1, 03 #
ST.UB	At,offset(Aa)	A	Store unsigned byte	F1, 08
ST.UW	At,offset(Aa)	A	Store unsigned word	F1, 09
ST.UD	At,offset(Aa)	A	Store unsigned double word	F1, 0A
ST.UQ	At,offset(Aa)	A	Store unsigned quad word	F1, 0B
ST.SB	At,offset(Aa)	A	Store signed byte	F1, 0C
ST.SW	At,offset(Aa)	A	Store signed word	F1, 0D
ST.SD	At,offset(Aa)	A	Store signed double word	F1, 0E
ST.SQ	At,offset(Aa)	A	Store signed quad word	F1, 0B #
ST.UB	St,offset(Aa)	S	Store unsigned byte	F1, 1, 08
ST.UW	St,offset(Aa)	S	Store unsigned word	F1, 1, 09
ST.UD	St,offset(Aa)	S	Store unsigned double word	F1, 1, 0A
ST.UQ	St,offset(Aa)	S	Store unsigned quad word	F1, 1, 0B
ST.SB	St,offset(Aa)	S	Store signed byte	F1, 1, 0C
ST.SW	St,offset(Aa)	S	Store signed word	F1, 1, 0D
ST.SD	St,offset(Aa)	S	Store signed double word	F1, 1, 0E
ST.SQ	St,offset(Aa)	S	Store signed quad word	F1, 1, 0B #
ST.FS	St,offset(Aa)	S	Store float single	F1, 0, 0A
ST.FD	St,offset(Aa)	S	Store float double	F1, 1, 0B #
ST.UB	At,Ab(Aa)	A	Store unsigned byte	F4, 08
ST.UW	At,Ab(Aa)	A	Store unsigned word	F4, 09
ST.UD	At,Ab(Aa)	A	Store unsigned double word	F4, 0A
ST.UQ	At,Ab(Aa)	A	Store unsigned quad word	F4, 0B
ST.SB	At,Ab(Aa)	A	Store signed byte	F4, 0C
ST.SW	At,Ab(Aa)	A	Store signed word	F4, 0D
ST.SD	At,Ab(Aa)	A	Store signed double word	F4, 0E
ST.SQ	At,Ab(Aa)	A	Store signed quad word	F4, 0B #
ST.UB	St,Ab(Aa)	S	Store unsigned byte	F4, 1, 08
ST.UW	St,Ab(Aa)	S	Store unsigned word	F4, 1, 09
ST.UD	St,Ab(Aa)	S	Store unsigned double word	F4, 1, 0A
ST.UQ	St,Ab(Aa)	S	Store unsigned quad word	F4, 1, 0B
ST.SB	St,Ab(Aa)	S	Store signed byte	F4, 1, 0C
ST.SW	St,Ab(Aa)	S	Store signed word	F4, 1, 0D
ST.SD	St,Ab(Aa)	S	Store signed double word	F4, 1, 0E
ST.SQ	St,Ab(Aa)	S	Store signed quad word	F4, 1, 0B #
ST.FS	St,Ab(Aa)	S	Store float single	F4, 0, 0A
ST.FD	St,Ab(Aa)	S	Store float double	F4, 1, 0B #

9.3.1.4 Arithmetic Operations

Operation	*Type*	*Operation Description*	*Encoding*

LDI.UQ	Immed,At	A	Load unsigned immediate	F1, 30 #
LDI.SQ	Immed,At	A	Load signed immediate	F1, 31 #
LDI.UQ	Immed,St	S	Load unsigned immediate	F1, 1, 30 #
LDI.SQ	Immed,St	S	Load signed immediate	F1, 1, 31 #
LDI.FS	Immed,St	S	Load float single immediate	F3, 0, 30 #
LDI.FD	Immed,St	S	Load float double immediate	F1, 1, 31 #
ABS.SQ	Aa,At	A	Absolute signed	F3, 0B
ABS.SQ	Sa,St	S	Absolute signed	F3, 1, 0B
ABS.FS	Sa,St	S	Absolute float single	F3, 0, 0A
ABS.FD	Sa,St	S	Absolute float double	F3, 0, 0B
NEG.SQ	Aa,At	A	Negate signed	F3, 0D
NEG.SQ	Sa,St	S	Negate signed	F3, 1, 0D
NEG.FS	Sa,St	S	Negate float single	F3, 0, 0C
NEG.FD	Sa,St	S	Negate float double	F3, 0, 0D
SQRT.FS	Sa,St	S	Square root float single	F3, 0, 0E
SQRT.FD	Sa,St	S	Square root float double	F3, 0, 0F
ADD.UQ	Aa,Immed,At	A	Add unsigned immediate	F1, 30
ADD.SQ	Aa,Immed,At	A	Add signed immediate	F1, 31
ADD.UQ	Sa,Immed,St	S	Add unsigned immediate	F1, 1, 30
ADD.SQ	Sa,Immed,St	S	Add signed immediate	F1, 1, 31
ADD.FS	Sa,Immed,St	S	Add float single immediate	F3, 0, 30
ADD.FD	Sa,Immed,St	S	Add float double immediate	F3, 0, 31
ADD.UQ	Aa,Ab,At	A	Add unsigned	F4, 30
ADD.SQ	Aa,Ab,At	A	Add signed	F4, 31
ADD.UQ	Sa,Sb,St	S	Add unsigned	F4, 1, 30
ADD.SQ	Sa,Sb,St	S	Add signed	F4, 1, 31
ADD.FS	Sa,Sb,St	S	Add float single	F4, 0, 30
ADD.FD	Sa,Sb,St	S	Add float double	F4, 0, 31
SUB.UQ	Aa,Immed,At	A	Subtract unsigned immediate	F1, 32
SUB.SQ	Aa,Immed,At	A	Subtract signed immediate	F1, 33
SUB.UQ	Sa,Immed,St	S	Subtract unsigned immediate	F1, 1, 32
SUB.SQ	Sa,Immed,St	S	Subtract signed immediate	F1, 1, 33
SUB.FS	Sa,Immed,St	S	Subtract float single immediate	F3, 0, 32
SUB.FS	Immed,Sa,St	S	Subtract float single immediate	F3, 0, 12
SUB.FD	Sa,Immed,St	S	Subtract float double immediate	F3, 0, 33
SUB.FD	Immed,Sa,St	S	Subtract float double immediate	F3, 0, 13
SUB.UQ	Aa,Ab,At	A	Subtract unsigned	F4, 32
SUB.SQ	Aa,Ab,At	A	Subtract signed	F4, 33
SUB.UQ	Sa,Sb,St	S	Subtract unsigned	F4, 1, 32

SUB.SQ	Sa,Sb,St	S	Subtract signed	F4, 1, 33
SUB.FS	Sa,Sb,St	S	Subtract float single	F4, 0, 32
SUB.FD	Sa,Sb,St	S	Subtract float double	F4, 0, 33
MUL.UQ	Aa,Immed,At	A	Multiply unsigned immediate	F1, 34
MUL.SQ	Aa,Immed,At	A	Multiply signed immediate	F1, 35
MUL.UQ	Sa,Immed,St	S	Multiply unsigned immediate	F1, 1, 34
MUL.SQ	Sa,Immed,St	S	Multiply signed immediate	F1, 1, 35
MUL.FS	Sa,Immed,St	S	Multiply float single	F3, 0, 34
MUL.FD	Sa,Immed,St	S	Multiply float double	F3, 0, 35
MUL.UQ	Aa,Ab,At	A	Multiply unsigned	F4, 34
MUL.SQ	Aa,Ab,At	A	Multiply signed	F4, 35
MUL.UQ	Sa,Sb,St	S	Multiply unsigned	F4, 1, 34
MUL.SQ	Sa,Sb,St	S	Multiply signed	F4, 1, 35
MUL.FS	Sa,Sb,St	S	Multiply float single	F4, 0, 34
MUL.FD	Sa,Sb,St	S	Multiply float double	F4, 0, 35
DIV.UQ	Aa,Immed,At	A	Divide unsigned immediate	F3, 36
DIV.SQ	Aa,Immed,At	A	Divide signed immediate	F3, 37
DIV.UQ	Sa,Immed,St	S	Divide unsigned immediate	F3, 1, 36
DIV.SQ	Sa,Immed,St	S	Divide signed immediate	F3, 1, 37
DIV.FS	Sa,Immed,St	S	Divide float single immediate	F3, 0, 36
DIV.FD	Sa,Immed,St	S	Divide float double immediate	F3, 0, 37
DIV.UQ	Aa,Ab,At	A	Divide unsigned	F4, 36
DIV.SQ	Aa,Ab,At	A	Divide signed	F4, 37
DIV.UQ	Sa,Sb,St	S	Divide unsigned	F4, 1, 36
DIV.SQ	Sa,Sb,St	S	Divide signed	F4, 1, 37
DIV.FS	Sa,Sb,St	S	Divide float single	F4, 0, 36
DIV.FD	Sa,Sb,St	S	Divide float double	F4, 0, 37

9.3.1.5 Logical and Shift Operations

Operation		*Type*	*Operation Description*	*Encoding*
AND	Aa,Immed,At	A	Logical AND immediate	F1, 20
OR	Aa,Immed,At	A	Logical OR immediate	F1, 21
NAND	Aa,Immed,At	A	Logical NAND immediate	F1, 22
NOR	Aa,Immed,At	A	Logical NOR immediate	F1, 23
XOR	Aa,Immed,At	A	Logical XOR immediate	F1, 24
XNOR	Aa,Immed,At	A	Logical XNOR immediate	F1, 25
ANDC	Aa,Immed,At	A	Logical ANDC immediate	F1, 26
ORC	Aa,Immed,At	A	Logical ORC immediate	F1, 27
AND	Sa,Immed,St	S	Logical AND immediate	F1, 1, 20

OR	Sa,Immed,St	S	Logical OR immediate	F1, 1, 21
NAND	Sa,Immed,St	S	Logical NAND immediate	F1, 1, 22
NOR	Sa,Immed,St	S	Logical NOR immediate	F1, 1, 23
XOR	Sa,Immed,St	S	Logical XOR immediate	F1, 1, 24
XNOR	Sa,Immed,St	S	Logical XNOR immediate	F1, 1, 25
ANDC	Sa,Immed,St	S	Logical ANDC immediate	F1, 1, 26
ORC	Sa,Immed,St	S	Logical ORC immediate	F1, 1, 27
AND	Aa,Ab,At	A	Logical AND	F4, 20
OR	Aa,Ab,At	A	Logical OR	F4, 21
NAND	Aa,Ab,At	A	Logical NAND	F4, 22
NOR	Aa,Ab,At	A	Logical NOR	F4, 23
XOR	Aa,Ab,At	A	Logical XOR	F4, 24
XNOR	Aa,Ab,At	A	Logical XNOR	F4, 25
ANDC	Aa,Ab,At	A	Logical ANDC	F4, 26
ORC	Aa,Ab,At	A	Logical ORC	F4, 27
AND	Sa,Sb,St	S	Logical AND	F4, 1, 20
OR	Sa,Sb,St	S	Logical OR	F4, 1, 21
NAND	Sa,Sb,St	S	Logical NAND	F4, 1, 22
NOR	Sa,Sb,St	S	Logical NOR	F4, 1, 23
XOR	Sa,Sb,St	S	Logical XOR	F4, 1, 24
XNOR	Sa,Sb,St	S	Logical XNOR	F4, 1, 25
ANDC	Sa,Sb,St	S	Logical ANDC	F4, 1, 26
ORC	Sa,Sb,St	S	Logical ORC	F4, 1, 27
SHFL.UQ	Aa,Immed,At	A	Shift left unsigned immediate	F3, 2C
SHFL.SQ	Aa, Immed,At	A	Shift left signed immediate	F3, 2D
SHFL.UQ	Sa, Immed,St	S	Shift left unsigned immediate	F3, 1, 2C
SHFL.SQ	Sa, Immed,St	S	Shift left signed immediate	F3, 1, 2D
SHFR.UQ	Aa, Immed,At	A	Shift right unsigned immediate	F3, 2E
SHFR.SQ	Aa, Immed,At	A	Shift right signed immediate	F3, 2F
SHFR.UQ	Sa, Immed,St	S	Shift right unsigned immediate	F3, 1, 2E
SHFR.SQ	Sa, Immed,St	S	Shift right signed immediate	F3, 1, 2F
SHFL.UQ	Aa,Ab,At	A	Shift left unsigned	F4, 2C
SHFL.SQ	Aa,Ab,At	A	Shift left signed	F4, 2D
SHFL.UQ	Sa,Sb,St	S	Shift left unsigned	F4, 1, 2C
SHFL.SQ	Sa,Sb,St	S	Shift left signed	F4, 1, 2D
SHFR.UQ	Aa,Ab,At	A	Shift right unsigned	F4, 2E
SHFR.SQ	Aa,Ab,At	A	Shift right signed	F4, 2F
SHFR.UQ	Sa,Sb,St	S	Shift right unsigned	F4, 1, 2E
SHFR.SQ	Sa,Sb,St	S	Shift right signed	F4, 1, 2F

9.3.1.6 Convert Operations

Operation		*Type*	*Operation Description*	*Encoding*
CVT.UQ.SQ	Aa,At	A	Convert from UQ to SQ	F3, 01
CVT.SQ.UQ	Aa,At	A	Convert from SQ to UQ	F3, 04
CVT.UQ.SQ	Sa,St	S	Convert from UQ to SQ	F3, 1, 01
CVT.SQ.UQ	Sa,St	S	Convert from SQ to UQ	F3, 1, 04
CVT.SQ.FS	Sa,St	S	Convert from SQ to FS	F3, 1, 06
CVT.SQ.FD	Sa,St	S	Convert from SQ to FD	F3, 1, 07
CVT.FS.SQ	Sa,St	S	Convert from FS to SQ	F3, 0, 01
CVT.FS.FD	Sa,St	S	Convert from FS to FD	F3, 0, 03
CVT.FD.SQ	Sa,St	S	Convert from FD to SQ	F3, 0, 05
CVT.FD.FS	Sa,St	S	Convert from FD to FS	F3, 0, 06

9.3.1.7 Move and Misc. Operations

Operation		*Type*	*Operation Description*	*Fmt, if, opc*
MOV	Aa, At	A	Move A-reg to A-reg	F1, 21 #
MOV	Sa,St	S	Move S-reg to S-reg	F1, 1, 21 #
MOV	Aa,St	S	Move A-reg to S-reg	F3, 1, 1C
MOV	Sa,At	A	Move S-reg to A-reg	F3, 1C
MOV	AAa,At	A	Move A-reg absolute to A-reg	F3, 1E
MOV	Aa,AAt	A	Move A-reg to A-reg absolute	F3, 1F
MOV	SAa,St	S	Move S-reg absolute to S-reg	F3, 1, 1E
MOV	Sa,SAt	S	Move S-reg to S-reg absolute	F3, 1, 1F
MOV	CIT,At	A	Move CIT to A-reg	F6, 16
MOV	CDS,At	A	Move CDS to A-reg	F6, 17
MOV	Aa,CCX	A	Move A-reg to CCX	F5, 18
MOV	CCX,At	A	Move CCX to A-reg	F6, 18
MOV	Aa,PIP	A	Move A-reg to PIP	F5, 10
MOV	PIP,At	A	Move PIP to A-reg	F6, 10
MOV	Aa,CPC	A	Move A-reg to CPC	F5, 11
MOV	CPC,At	A	Move CPC to A-reg	F6, 11
MOV	Aa,CPS	A	Move A-reg to CPS	F5, 12
MOV	CPS,At	A	Move CPS to A-reg	F6, 12
MOV	Aa,Immed,CRSL	A	Move A-reg to CRSL[Immed]	F5, 13
MOV	CRSL,Immed,At	A	Move CRSL[Immed] to A-reg	F6, 13
MOV	Aa,Immed,CRSU	A	Move A-reg to CRSU[Immed]	F5, 14
MOV	CRSU,Immed,At	A	Move CRSU[Immed] to A-reg	F6, 14
MOV	Aa,WB	A	Move A-reg to WB	F5, 15
MOV	WB,At	A	Move WB to A-reg	F6, 15
WBINC	AWB,SWB,VWB,VMWB	A	Inc. window base fields	F7, 00

WBINC.P	AWB,SWB,VWB,VMWB	A	Inc. window base & write CRS	F7, 01
WBDEC	AWB,SWB,VWB,VMWB	A	Dec. window base fields	F7, 02
WBSET	AWB,SWB,VWB,VMWB	A	Set window base fields	F7, 03
VRRINC		A	Increment VRRO field	F7, 08
VRRSET	Base,Size,Offset	A	Set vector register rotation fields	F7, 09
MOV	Immed,VMA	A	Move Immediate to VMA	F7, 15
SEL	AC3.EQ,Aa,Ab,At	A	Select using AC3.EQ	F4, 18
SEL	AC3.GT,Aa,Ab,At	A	Select using AC3.GT	F4, 19
SEL	AC3.LT,Aa,Ab,At	A	Select using AC3.LT	F4, 1A
SEL	SC3.EQ,Aa,Ab,At	A	Select using SC3.EQ	F4, 1C
SEL	SC3.GT,Aa,Ab,At	A	Select using SC3.GT	F4, 1D
SEL	SC3.LT,Aa,Ab,At	A	Select using SC3.LT	F4, 1E
SEL	AC3.EQ,Sa,Sb,St	S	Select using AC3.EQ	F4, 1, 18
SEL	AC3.GT,Sa,Sb,St	S	Select using AC3.GT	F4, 1, 19
SEL	AC3.LT,Sa,Sb,St	S	Select using AC3.LT	F4, 1, 1A
SEL	SC3.EQ,Sa,Sb,St	S	Select using SC3.EQ	F4, 1, 1C
SEL	SC3.GT,Sa,Sb,St	S	Select using SC3.GT	F4, 1, 1D
SEL	SC3.LT,Sa,Sb,St	S	Select using SC3.LT	F4, 1, 1E
MOV	Acnt,At	A	Move Acnt to A-reg	F6, 1C
MOV	Scnt,At	A	Move Scnt to A-reg	F6, 1D
MOV	CRScnt,At	A	Move CRScnt to A-reg	F6, 1E
FENCE		A	Memory Fence	F7, 0F

10 Base Vector Infrastructure

Convey supports vector personalities based on a common vector infrastructure. These personalities provide extensions to the Scalar ISA that support vector oriented operations. The personalities are appropriate for workloads with data organized as single or multi-dimensional arrays.

10.1 Vector Machine State Extensions

The following figure shows the machine state defined by the base vector infrastructure.

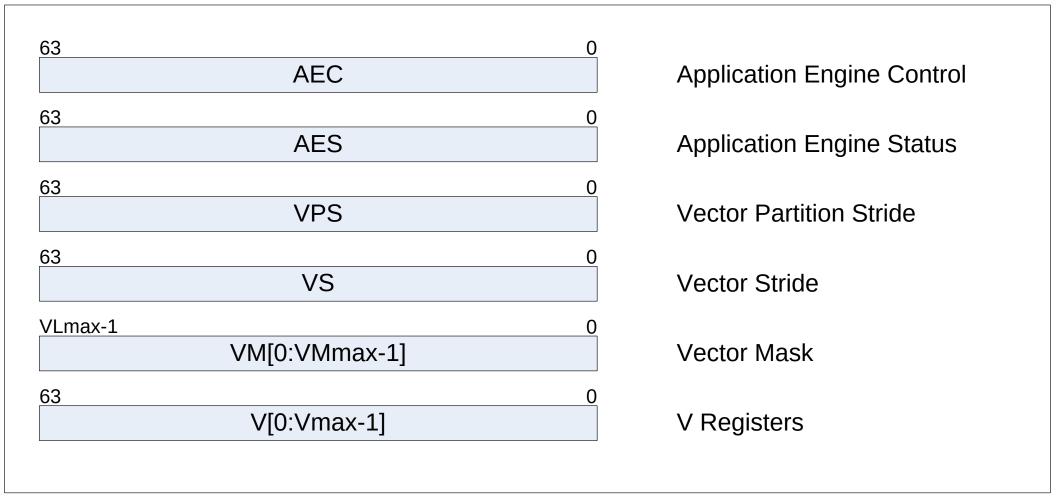


Figure 30 – Single and Double Precision Vector Machine State Extensions

10.1.1 AEC – Application Engine Control Register

The AEC register is composed of a number of fields that control various aspects of the application engine. The value of the AEC register is the same for each AE. A separate value of the AEM field is maintained per user command area. The AEM field is persistent state. The VL, VPA, VPL and VPM fields are non-persistent state. The following figure shows the fields of the AEC register.

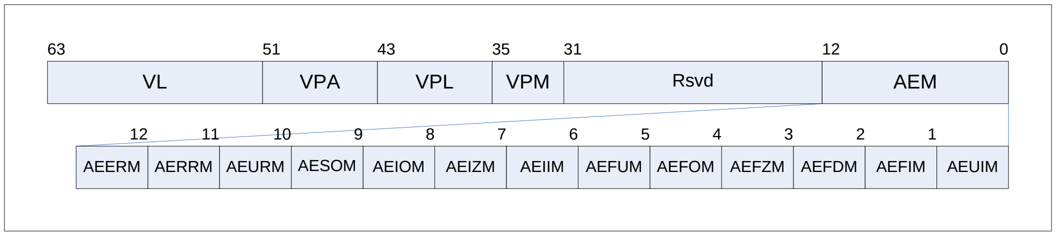


Figure 31 - Application Engine Control Register

The following fields exist within the AEC register:

AEM – Application Engine Mask

The application engine mask specifies which exceptions are to be masked (i.e. ignored by the coprocessor). Exceptions with their mask set to one are ignored. Refer to section 10.1.2 for a description of each exception type.

VPM – Vector Partition Mode

The VPM register is used to set the vector partition configuration. The vector partition configuration sets the number of function pipes per partition. The following table lists the supported modes.

Field Value	*Mode*	*Vector Partitions*
0	Classical Vector	1
1	Physical Partition	4 (one per AE)
2	Short Vector	One partition per function pipe

Table 22 - Vector Partition Modes

Three modes are supported: all function pipes in a single partition (Classic Vector mode), all function pipes within each AE in separate partitions (Physical Partition mode), and multiple partitions within each AE (Short Vector mode). Refer to 10.3 for a discussion of vector partitions.

VPL – Vector Partition Length Field

The VPL field is used to specify the number of vector partitions that are to participate in a vector operation. Refer to section 10.3 for a discussion of vector partitions.

VPA – Active Vector Partition Field

Instructions that operate on a single partition use the VPA field to determine the active partition for the operation. An example instruction that uses the VPA field is move S-register to a Vector register element. The instruction uses the VPA field to determine which partition the operation is to be applied.

VL – Vector Length Field

The Vector Length field specifies the number of vector elements in each vector partition.

10.1.2 AES – Application Engine Status

The application engine status register holds the various status fields for an application engine. Each application engine may have different AES register values. The AEE field is set to zero at the entry to each dispatched routine.

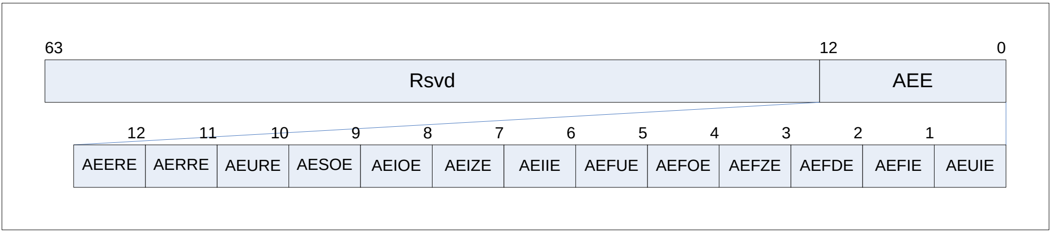


Table 23 - Application Engine Status Register

Instructions check for exceptions during their execution. Detected exceptions are recorded in the AEE field. The AEE field is masked with the AEC AEM field to determine if a recorded exception should interrupt the host processor. The results of the following exception checks are recorded.

Exception Name	*AEE Bit*	*Description*
AEUIE	0	AE Unimplemented Instruction Exception. An attempt was made to execute an unimplemented instruction. The operation of the unimplemented instruction depends on the value of the instruction's opcode. If the opcode is in a range of opcodes that is defined to return a value to the scalar processor then the operation is equivalent to the instruction MOV VL,Aa; otherwise the instruction is equivalent to a NOP.
AEFIE	1	AE Floating Point Invalid Operand Exception. A floating point instruction detected invalid input operands for the operation to be performed.
AEFDE	2	AE Floating Point Denormalized Exception. A floating point instruction detected a denormalized input operand.
AEFZE	3	AE Floating Point Divide By Zero Exception. A floating point divide instruction detected a zero denominator operand.
AEFOE	4	AE Floating Point Overflow Exception. A floating point instruction produced a result with an exponent value that was too large for floating point data format. The result of a floating point operation that overflows is infinite.
AEFUE	5	AE Floating Point Underflow Exception. A floating point instruction produced a result with an exponent value that was too small for floating point data format. The result of a floating point operation that underflows is zero.
AEIIE	6	AE Integer Invalid Operand Exception. An integer instruction detected invalid input operands for the operation being performed.
AEIZE	7	AE Integer Divide By Zero Exception. An integer divide instruction detected a zero denominator operand.
AEIOE	8	AE Integer Overflow Exception. An integer instruction produced a result that exceeded the range of the data format.
AESOE	9	AE Store Overflow Exception. A vector register integer store was performed where the value to be stored exceeded the range of the memory data size. The lower bits of the source register are stored to memory when the exception is detected.
AEURE	10	AE Unaligned Memory Reference Exception. An instruction referenced memory using an address that was not aligned

		for the size of the access. An unaligned memory reference is performed by forcing the least significant 3 bits of the virtual address to zero.
AERRE	11	AE Register Range Exception. An instruction referenced a V or VM register where the sum of the instruction's register index field and the V or VM window base register value exceeded the count of V or VM registers. The modulo function is used to generate the register index when the register range is exceeded.
AEERE	12	AE Element Range Exception. An instruction referenced a V, VM or AEG register where the element index value exceeded the valid range. The element index is forced to zero when this exception occurs.

10.1.3 VPS – Vector Partition Stride Register

The VPS register is used to specify the stride between consecutive vector partitions. The VPS register is 64-bits wide using the canonical address format of the host processor. The upper 16 bits of the VPS register is reserved for future use. Reading the VPS register results in the lower 48 bits being sign extended to form a 64-bit value. The VPS register is non-persistent state.

10.1.4 VS – Vector Stride Register

The VS register is used to specify the distance in bytes between consecutive data elements loaded or stored to memory. The VS register is 64-bits wide using the canonical address format of the host processor. The upper 16 bits of the VS register is reserved for future use. Reading the VS register results in the lower 48 bits being sign extended to form a 64-bit value. The VS register is non-persistent state.

10.1.5 VM – Vector Mask Register

The vector mask registers are used to allow conditional execution per element for a vector instruction. The width of the VM register is equal to the number of elements in each vector register. The VM register width is referred to as Vmax.

Multiple VM registers exist to allow complex under mask operations without requiring the movement of VM register values to memory. Instructions that are executed under mask use the active VM register (VMA) to specify the VM register to be used. Vector compare instructions set a VM register's value. The vector compare instruction specifies which VM register is to be written.

The number of VM registers is not architecturally defined, but rather is implementation dependent. The name VMcnt is used to refer to the number of implemented VM registers.

The VM registers are non-persistent state.

10.1.6 Vector Register Set

The Vector registers are a set of general purpose registers where each register contains multiple elements (VLmax). Vector register elements are 64 bits wide. Operations on Vector registers are performed on each element within the vector.

The number of Vector registers is not architecturally defined, but rather is implementation dependent. The name Vcnt is used to refer to the number of implemented Vector registers.

The Vector registers are non-persistent state.

10.2 Vector Register Access

10.2.1 Vector Register Left and Right 32-bit Accesses

Vector registers for the single precision vector personality are defined to be 64-bits wide with direct access to the left and right 32-bit components. The direct 32-bit accessibility is provided to allow efficient use of the 64-bit registers for 32-bit float single data and to allow direct access to the real and imaginary components of single precision complex data.

AE instructions that specify a data type of Float Single read or write 32-bits of a 64-bit vector register. The access can be to the left or right 32-bit half of the 64-bit vector register. AE instructions that specify a data type of unsigned quad word integer, signed quad word integer or complex single precision access the entire 64-bit vector register.

Coprocessor instructions provide 6-bit register specification fields (ra, rb, rc and rt). The 6-bit field is able to specify one of 64 registers to be used by the instruction. The 6-bit field is used differently depending on whether 32-bits are being accessed (float single) or 64-bits are being accessed (integer, double precision or complex single).

An instruction that accesses 64-bit data types uses the entire 6-bit register field to select the vector register (V0 through V63). Instructions that operate on floating point single type data use the 6-bit register field to specify one of 32 vector registers (V0 through V31) and whether the access is to the left or right 32-bit component. The figure below shows how the 6-bit register field is encoded for 32-bit vector register accesses. Note that the upper bit of the 6-bit field (l/r bit) is used to specify the left or right 32-bit register halves.

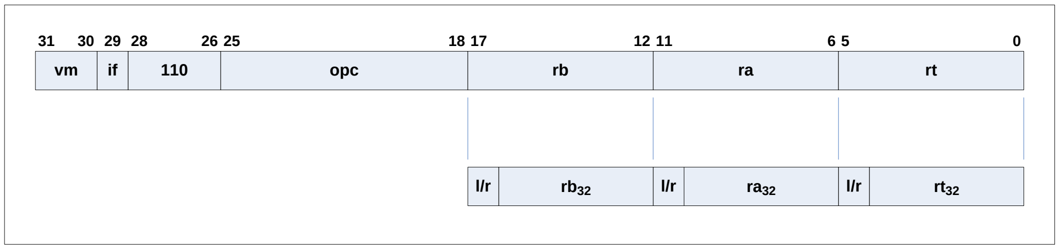


Figure 32 - 32-bit register component specification

An instruction that operates on integer or complex single data specifies a 64-bit register using the names V0 through V63. An instruction that operates on floating single data specifies the register using the names V0L through V31L and V0R through V31R. Note that V0L is the left (upper) 32-bit component of the register V0, and V0R is the right (lower) 32-bit component. The following are example instructions.

```
ADD.FS     V3R,V4R,V7R          ; Add right 32-bit components
ADD.FS     V3L,V4L,V7L          ; Add left 32-bit components
ADD.CS     V4,V3,V7             ; Equivalent to previous two instructions
LD.FS      0(A3),V5R            ; Load float single from memory,
                                          ;   leave upper 32-bits unmodified
LD.UD        0(A3),V5                     ; Load float single from memory
                                          ;   zero fill upper 32-bits
MOV.FS       V4R,V4L                      ; move from right to left 32-bits
```

The vector register window base register (VWB) is used to specify the base for all vector register fields. As an example, if the register VWB has the value 3, then specifying register V4 in an instruction would access vector register 7. Specifying register V4R would access the right 32-bits of register 7.

10.2.2 Vector Register Rotation

The vector personalities support vector register rotation (VRR). The VRR mechanism is very efficient for implementing array template functions. The following example illustrates usage of the mechanism.

C-code:

```
for (i = 0; i < rows; i += 1)
    for (j = 1; j < cols-1; j += 1)
        B[i][j] = A[i][j-1] * W1 + A[i][j] * W2 + A[i][j+1] * W3;
```

Assemble:

```
        MOV        ROWS,VL         // ELEMENTS IN ROW
        MOV        4,VS            // STRIDE OF FLOAT SINGLE DATA
        LDI.SQ     COLS*4,A2       // STRIDE OF WHOLE ROW
        LDI.SQ     0,A3            // A ROW BASE
        LDI.SQ     0,A4            // B ROW BASE
        LD.FS      A(A3),V1        // LOAD A[I][J-1]
        ADD.SQ     A3,A2,A3        // INCREMENT A ROW BASE
        LD.FS      A(A3),V2        // LOAD A[I][J]
        ADD.SQ     A3,A2,A3        // INCREMENT A ROW BASE
        LDI.SQ     COLS-2,A5       // LOOP COUNT
        VRRSET     1,3,0           // INITIALIZE VRR VALUES
                                   //   (BASE=1,SIZE=3,OFFSET=0)
LOOP:   CMP.SQ     A5,0,AC0        // CHECK FOR END OF LOOP
        BR         AC0.LT,0,END    // BRANCH IF END OF LOOP
        LD.FS      A(A3),V3        // LOAD A[I][J+1]
        MUL.FS     V1,S1,V4        // A[I][J-1] * W1
        FMA.FS     V2,S2,V4,V4     // A[I][J] * W2 + V4
        FMA.FS     V3,S3,V4,V4     // A[I][J+1] * W3 + V4
        ST.FS      V4,B(A4)        // STORE B[I][J]
        ADD.SQ      A3,A2,A3        // INCREMENT A ROW BASE
        ADD.SQ      A4,A2,A4        // INCREMENT B ROW BASE
        SUB.SQ      A5,1,A5         // DECREMENT LOOP COUNT
        VRRINC                      // ROTATE REGISTERS
        BR          LOOP            // BRANCH TO TOP OF LOOP
END:
```

The preceding example shows how the vector register rotation mechanism allows assembly code to be written that eliminates redundant memory loads.

10.3 Vector Partitioning

Vector Partitioning is used to partition the parallel function units of the application engines to eliminate communication between application engines, or provide increased efficiency on short vector lengths. All partitions participate in each vector operation (vector partitioning is an enhancement that maintains SIMD execution).

An example where eliminating communication between application engines is required is the FFT algorithm. FFTs require complex data shuffle networks when accessing data elements from the vector register file. With one partition per application engine, “physical partition mode”, an FFT is performed entirely within a single application engine. By partitioning the parallel function units into one partition per application engine, communication between application engines is eliminated.

A second usage of vector partitioning is for increasing the performance on short vectors. The following code performs addition between two matrices with the result going to a third.

```
long long A[64][33], B[64][33], C[64][33];          // 64-bit signed integer
For (int I = 0; I < 64; I += 1)
        For (int j = 0; j < 32; j += 1)
            A[i][j] = B[i][j] + C[i][j];
```

The declared matrices are 64 by 33 in size. A compiler’s only option is to perform operations one row at a time since the addition is performed on 32 of the 33 elements in each row. In “classical vector mode” (I.e. without vector partitions), a vector register would use only 32 of a vector register’s data elements. With vector partitioning, a vector register can be partitioned for “short vector operations”. If the vector register has 1024 data elements, then the partitioning would result in thirty-two partitions with 32 data elements each. A single vector load operation would load all thirty-two partitions with 32 data elements each. Similarly, a vector add would perform the addition for all thirty-two partitions. Using vector partitions turns a vector operation where 32 data elements are valid within each vector register to an operation with all 1024 data elements being valid. A vector operation with only 32 data elements is likely to run at less than peak performance for the coprocessor, whereas peak performance is likely when using all data elements within a vector register.

10.3.1 Vector Partitioning Configurations

Vector partitioning is limited to specific configurations. The configurations ensure that all Vector Partitions have the same number of function pipes. The following table shows the allowed configurations.

Vector Partition Mode (VPM)	*Partition Count*	*Vector Register Data Elements Per Partition*	*Mode Description*
0	1	VLmax	Classical Vector
1	4	VLmax / 4	Physical Partitioning
2	32	VLmax / 32	Short Vector

Table 24 - Vector Partition Configurations

As an example, assume that the coprocessor has 32 function pipes with a vector register having 1024 elements. If the vector partition mode (VPM) register has the value of 2, then there are 32 partitions with 32 data elements per partition.

10.3.2 Function Pipe Usage

This section discusses the mapping of partitions to function pipes (FP). The figure below illustrates the mapping of FPs to partitions for the modes discussed in the previous section. The figure represents each FP as a small rectangle with eight FPs per AE.

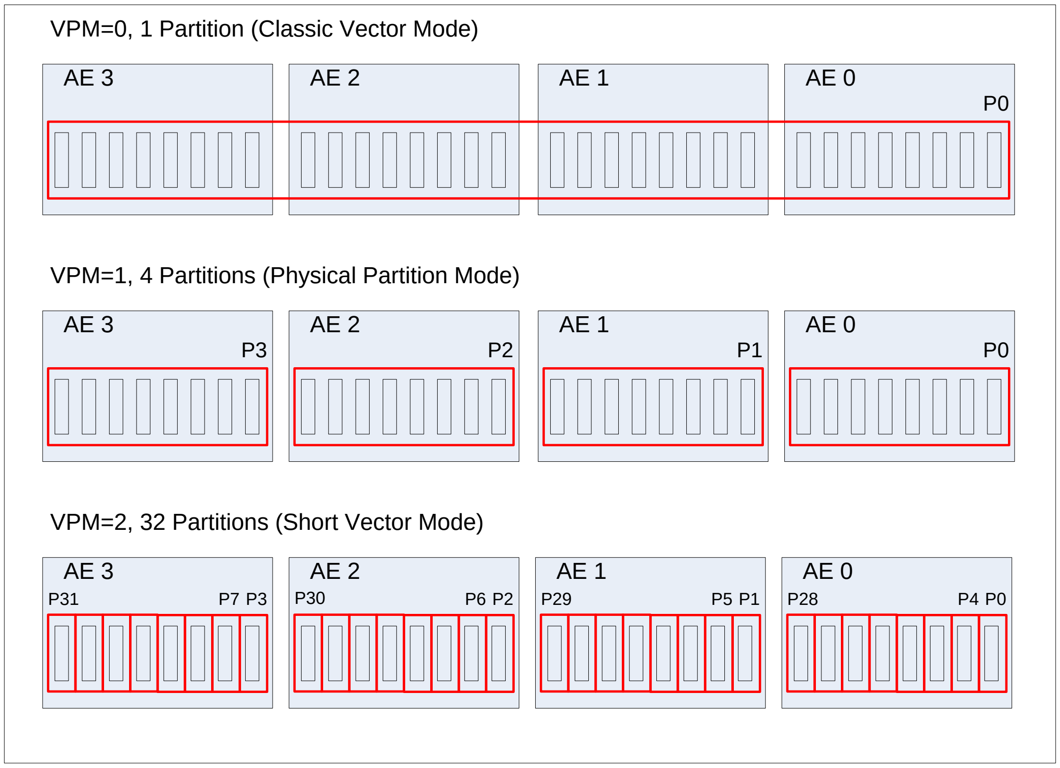

VPM=0, 1 Partition (Classic Vector Mode)
AE 3
AE 2
AE 1
AE 0
P0
VPM=1, 4 Partitions (Physical Partition Mode)
AE 3
P3
AE 2
P2
AE 1
P1
AE 0
P0
VPM=2, 32 Partitions (Short Vector Mode)
AE 3
P31
P7 P3
AE 2
P30
P6 P2
AE 1
P29
P5 P1
AE 0
P28
P4 P0

Figure 33 - Mapping of Vector Partitions to Function Pipes

10.3.3 Data Organization

Data is mapped to function pipes within a partition based on the following criteria:

- Each function pipe has the same number of data elements (+/- 1). The execution time of an operation within a partition is minimized by uniformly spreading the data elements across the function pipes.
- Consecutive vector elements are mapped to the same FP before transitioning to the next function pipe.

10.3.3.1 Classical Vector Mode

The mapping of data elements to function pipes in classical vector mode follows the guide lines listed above. The result is that depending on the total number of vector elements (I.e. the value of VL), a specific data element will be mapped to a different application engine / function pipe. The following figures show how data elements are mapped in classical vector mode for VL=10, and VL=90.

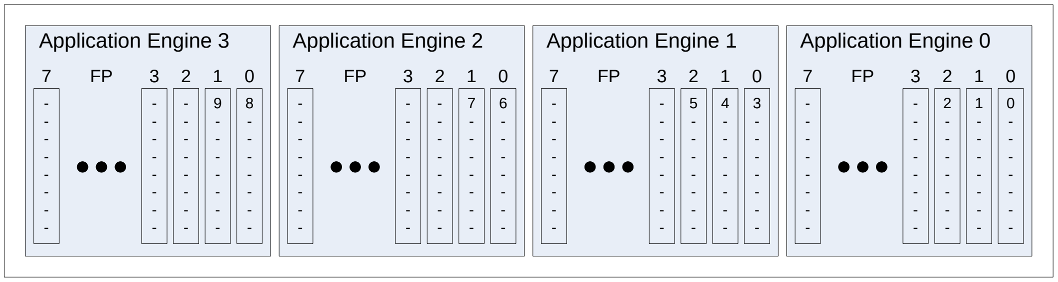


Figure 34 - Example Classical Mode Data Mapping, VL=10

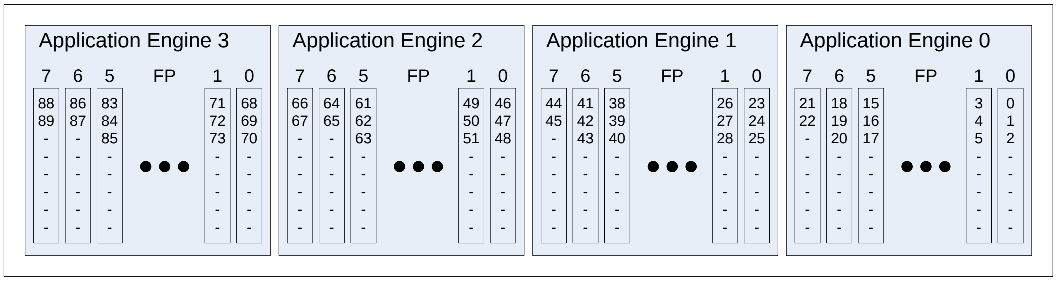


Figure 35 - Example Classical Mode Data Mapping, VL=90

As shown in the above figures, the vector register elements are uniformly distributed across the function pipes, and the elements are contiguous within each application engine.

10.3.3.2 Physical Partition Mode

In Physical Partition Mode (VPM=1), the elements are mapped to the function pipes within an application engine in a striped manner with all function pipes having the same number of elements (+/- 1). The figure below shows how data elements are mapped in physical partition mode for VL=23.

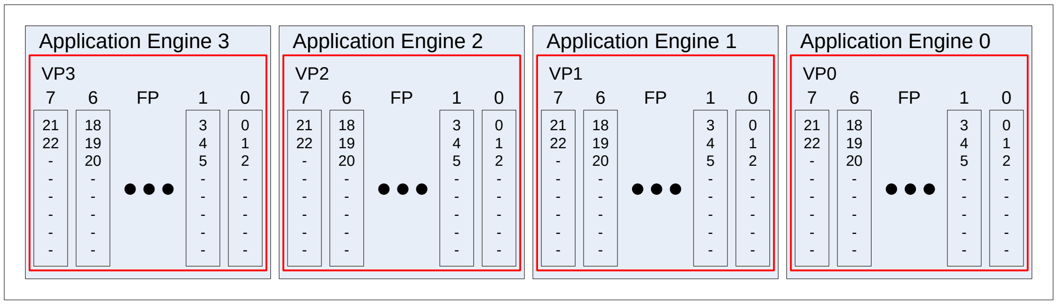


Figure 36 - Example Physical Partition Mode Data Mapping, VL=23

The physical partition mode has the same vector length (VL) value per partition.

10.3.3.3 Short Vector Mode

In Short Vector Mode (VPM=2), the elements are mapped to a single function pipe within each partition. The figure below shows how data elements are mapped for VL=3.

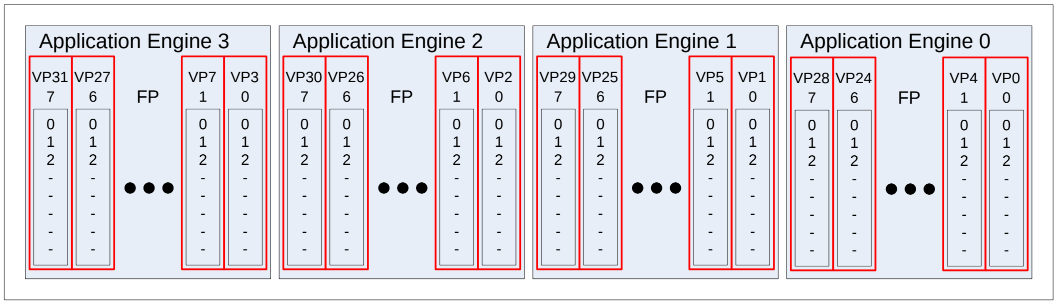


Figure 37 - Example Short Vector Mode Data Mapping, VL=3

The Short Vector mode has a common vector length (VL) value for all partitions. Note that partitions are interleaved across the Application Engines to provide balanced processing when not all partitions are being used (I.e. VPL is less than 32).

10.3.4 User Registers Associated with Vector Partitions

Three registers exist to control vector partitions. These registers are the Vector Partition Mode (VPM), Vector Partition Length (VPL) and Vector Partition Stride (VPS).

The Vector Partition Length register indicates the number of vector partitions that are to participate in the vector operation. As an example, if VPM=2 (32 partitions) and VPL=12, then vector partitions 0-11 will participate in vector operations and partitions 12-31 will not participate.

The Vector Partition Stride register (VPS) indicate the stride in bytes between the first data element of consecutive partitions for vector load and store operations.

Note that the Vector Length register indicates the number of data elements that participates in a vector operation within each vector partition. Similarly, the Vector Stride register indicates the stride in bytes between consecutive data elements within a vector partition. The use of these registers (VL and VS) is consistent whether operating in “Classical Vector mode” with a single partition, or with multiple partitions.

10.3.5 Operations with Vector Partitions

10.3.5.1 Load / Store Operations

Vector loads and stores use the VL and VPL registers to determine which data elements within each vector partition are to be loaded or stored to memory. The VL value indicates how many data elements are to be loaded/stored within each partition. The VPL value indicates how many of the vector partitions are to participate in the vector load/store operation.

The VS and VPS registers are used to determine the address for each data element memory access. The pseudo code below shows the algorithm used to calculate the address for each data element of a vector load/store.

Instruction:

```
ld.sq       offset(A4), V0
```

Equivalent Pseudo Code:

```
for (int vp = 0; vp < VPL; vp += 1)          ; vp is the vector partition index
    for (int ve = 0; ve < VL; ve += 1)       ; ve is the vector register element index
        V0[vp][ ve] = offset + A[4] + ve * VS + vp * VPS
```

Note that setting VS and/or VPS to zero results in the same location of memory being accessed multiple times for a load or store instruction. The following special cases can be created:

Value of VPS and VS	*Operation Description*
VPS == 0, VS != 0	All partitions receive the same values. (I.e. data element zero of all partitions access the same location in memory, data element one of all partitions access the next location in memory).
VPS != 0, VS == 0	Each partition accesses a different location in memory, but all data elements within a partition access the same location in memory.
VPS == 0, VS == 0	All elements in all partitions access the same location in memory.

Table 25 - Special Memory Access Patterns Created by Setting VPS or VS to Zero

10.3.5.2 Scalar / Vector Operations

Scalar / Vector operations are operations where a scalar value is applied to all elements of a vector. When considering vector partitions, vector / scalar operations take on two forms. The first form is when all elements of all partitions use the same scalar value. Operations of this form are performed using the defined scalar / vector instructions. An example instruction would be:

```
ADD.SQ      V1, S3, V2
```

The addition operation adds S3 plus elements of V1 and puts the result in V2. The values of VPM, VPL and VL determine which elements of the vector operation are to participate in the addition. The key in this example is that all elements that participate in the operation use the same scalar value.

The second scalar / vector form is when all elements of a partition use the same scalar value, but different partitions use different scalar values. In this case, there is a vector of scalar values, one value for each partition. This form is handled as a vector operation. The multiple scalars (one per partition) are loaded into a vector register using a vector load instruction with VS equal zero, and VPS non-zero. Setting VS equal to zero has the effect of loading the same scalar value to all elements of a partition. Setting VPS to a non-zero value results in a different value being loaded into each partition.

10.3.6 Sample code using Vector Partitioning

The following example shows how vector partitioning can be used to efficiently perform the following sample code.

```
long long A[16][32], B[16][32], C[16];      // 64-bit signed integer
For (int I = 0; I < 16; I += 1)
    For (int j = 0; j < 32; j += 1)
        A[I][j] = B[i][j] + C[i];
```

Coprocessor instructions:

```
MOV       4, VPM          ; 16 partitions
MOV       32, VL          ; 32 elements per partition

MOV       16, VPL         ; all 16 partitions participate
MOV       0, VS           ; stride of zero within partition
MOV       1, VPS          ; stride of one between partitions
LD.SQ     addr_C, V0      ; replicate C for all elements of a partition
MOV       1, VS           ; stride of one within partition
MOV       32, VPS         ; stride of 32 between partitions
LD.SQ     addr_B, V1
ADD.SQ    V0, V1, V2
ST.SQ     V2, addr_A
```

The above code sequence illustrates techniques that could be used on the inner loop of a matrix multiple routine.

10.4 Vector Reduction Operations

Vector reduction operations are accomplished by executing a sequence of instructions. The first step is to execute an AE reduction instruction. The AE reduction instruction reduces a vector of values to one or more partial results per partition. The number of partial results per partition is implementation dependent. The base vector infrastructure is expected to produce four partial results in Classic Vector mode, and one result per partition in Physical and Short Vector modes. An instruction is provided that provides the number of partial reduction results per partition (*MOV RedCnt, At*). An instruction is provided that allows a partial result to be moved to an S register (*MOVR Va,Ab,St*). The reduction operation can be completed using scalar instructions. The following instruction sequence can be used to perform a 64-bit unsigned integer summation reduction.

```
        SUMR.UQ    V0,V1
        MOV        RedCnt,A1
        MOVR       V1,A0,S1
        LDI.SQ     1,A2
LOOP:   CMP.SQ     A1,A2,AC0
        BR         AC0.EQ,END
        MOVR       V1,A2,S2
        ADD.UQ     S2,S1,S1
        ADD.SQ     A2,1,A2
        BR         LOOP
END:
```

The above instruction sequence can be optimized for the expected case of four partial results by performing the scalar unsigned integer adds using a tree height reduction approach.

10.5 Base Vector Instructions (BVI)

This section presents a set of instructions provided by the base vector infrastructure. All personalities that include the base vector infrastructure support the following instructions. One exception is that personalities that do not require 32-bit access to the vector registers may choose to not support base vector instructions that facilitate that capability.

The base vector instruction set contains the following classes of instructions:

- Vector quad word integer compares
- Vector double word and quad word integer loads and stores
- Vector integer logical operations
- Vector integer negate, add, subtract and multiply operations
- Vector partitioning
- Vector mask boolean operations
- Vector index generation
- Vector integer reduction operations (min, max, sum)

10.5.1 Instruction Set Organized by Function

Instructions in the following table marked with a ‘#’ are aliased to the same instruction with another name. Instructions that are included to support 32-bit vector register accesses are marked with a ‘ * ‘. These instructions are optional in personalities that do not need 32-bit vector register access.

10.5.1.1 Compare Instructions

Operation		*Type*	*Operation Description*	*Fmt, if, opc*
EQ.UQ	Va,Sb,VMt	AE	Equal unsigned scalar	F3, 1, 3A
EQ.UQ	Va,Vb,VMt	AE	Equal unsigned	F4, 1, 3A
EQ.SQ	Va,Sb,VMt	AE	Equal signed scalar	F3, 1, 3B
EQ.SQ	Va,Vb,VMt	AE	Equal signed	F4, 1, 3B
LT.UQ	Va,Sb,VMt	AE	Less than unsigned scalar	F3, 1, 3C
LT.UQ	Va,Vb,VMt	AE	Less than unsigned	F4, 1, 3C
LT.SQ	Va,Sb,VMt	AE	Less than signed scalar	F3, 1, 3D
LT.SQ	Va,Vb,VMt	AE	Less than signed	F4, 1, 3D
GT.UQ	Va,Sb,VMt	AE	Greater than unsigned scalar	F3, 1, 3E
GT.UQ	Va,Vb,VMt	AE	Greater than unsigned	F4, 1, 3E
GT.SQ	Va,Sb,VMt	AE	Greater than signed scalar	F3, 1, 3F
GT.SQ	Va,Vb,VMt	AE	Greater than signed	F4, 1, 3F
AND	VMa,VMb,VMt	AE	Logical AND	F4, 1, 10
OR	VMa,VMb,VMt	AE	Logical OR	F4, 1, 11
NAND	VMa,VMb,VMt	AE	Logical NAND	F4, 1, 12

NOR	VMa,VMb,VMt	AE	Logical NOR	F4, 1, 13
XOR	VMa,VMb,VMt	AE	Logical XOR	F4, 1, 14
XNOR	VMa,VMb,VMt	AE	Logical XNOR	F4, 1, 15
ANDC	VMa,VMb,VMt	AE	Logical ANDC	F4, 1, 16
ORC	VMa,VMb,VMt	AE	Logical ORC	F4, 1, 17

10.5.1.2 Load / Store Instructions

Operation		*Type*	*Operation Description*	*Fmt, if, opc*
LD.UD	offset(Aa),Vt	AE	Load unsigned double word	F1, 1, 02
LD.UQ	offset(Aa),Vt	AE	Load unsigned quad word	F1, 1, 03
LD.SD	offset(Aa),Vt	AE	Load signed double word	F1, 1, 06
LD.SQ	offset(Aa),Vt	AE	Load signed quad word	F1, 1, 03 #
LD.DW	offset(Aa),Vt	AE	Load double word to 32-bit vector register	F1, 0, 02 *
LD.DDW	offset(Aa),Vt	AE	Load dual double words with 32-bit alignment	F1, 0, 06 *
LD.UD	Vb(Aa),Vt	AE	Load unsigned double word	F4, 1, 02
LD.UQ	Vb(Aa),Vt	AE	Load unsigned quad word	F4, 1, 03
LD.SD	Vb(Aa),Vt	AE	Load signed double word	F4, 1, 06
LD.SQ	Vb(Aa),Vt	AE	Load signed quad word	F4, 1, 03 #
LD.DW	Vb(Aa),Vt	AE	Load double word to 32-bit vector register	F4, 0, 02 *
ST.UD	Vt,offset(Aa)	AE	Store unsigned double word	F1, 1, 0A
ST.UQ	Vt,offset(Aa)	AE	Store unsigned quad word	F1, 1, 0B
ST.SD	Vt,offset(Aa)	AE	Store signed double word	F1, 1, 0E
ST.SQ	Vt,offset(Aa)	AE	Store signed quad word	F1, 1, 0B #
ST.DW	Vt,offset(Aa)	AE	Store double word from 32-bit vector register	F1, 0, 0A *
ST.DDW	Vt,offset(Aa)	AE	Store dual double words with 32-bit alignment	F1, 0, 0E *
ST.UD	Vt,Vb(Aa)	AE	Store unsigned double word	F4, 1, 0A
ST.UQ	Vt,Vb(Aa)	AE	Store unsigned quad word	F4, 1, 0B
ST.SD	Vt,Vb(Aa)	AE	Store signed double word	F4, 1, 0E
ST.SQ	Vt,Vb(Aa)	AE	Store signed quad word	F4, 1, 0B #
ST.DW	Vt,Vb(Aa)	AE	Store double word from 32-bit vector register	F4, 0, 0A *
LD	offset(Aa),VMt	AE	Load VM from memory	F1, 1, 13
ST	VMt,offset(Aa)	AE	Store VM to memory	F1, 1, 1B

10.5.1.3 Arithmetic Operations

Operation		*Type*	*Operation Description*	*Fmt, if, opc*
ABS.SQ	Va,Vt	AE	Absolute signed	F3, 1, 0B
NEG.SQ	Va,Vt	AE	Negate signed	F3, 1, 0D
ADD.UQ	Va,Sb,Vt	AE	Add unsigned scalar	F3, 1, 30
ADD.UQ	Va,Vb,Vt	AE	Add unsigned	F4, 1, 30
ADD.SQ	Va,Sb,Vt	AE	Add signed scalar	F3, 1, 31
ADD.SQ	Va,Vb,Vt	AE	Add signed	F4, 1, 31
SUB.UQ	Va,Sb,Vt	AE	Subtract unsigned scalar	F3, 1, 32
SUB.UQ	Va,Vb,Vt	AE	Subtract unsigned	F4, 1, 32
SUB.SQ	Va,Sb,Vt	AE	Subtract signed scalar	F3, 1, 33
SUB.SQ	Va,Vb,Vt	AE	Subtract signed	F4, 1, 33
MUL.UQ	Va,Sb,Vt	AE	Multiply 48x48 unsigned scalar	F3, 1, 34
MUL.UQ	Va,Vb,Vt	AE	Multiply 48x48 unsigned	F4, 1, 34
MUL.SQ	Va,Sb,Vt	AE	Multiply 48x48 signed scalar	F3, 1, 35
MUL.SQ	Va,Vb,Vt	AE	Multiply 48x48 signed	F4, 1, 35
MIN.UQ	Va,Vb,Vt	AE	Minimum unsigned	F4, 1, 28
MIN.UQ	Va,Sb,Vt	AE	Minimum unsigned scalar	F3, 1, 28
MIN.SQ	Va,Vb,Vt	AE	Minimum signed	F4, 1, 29
MIN.SQ	Va,Sb,Vt	AE	Minimum signed scalar	F3, 1, 29
MAX.UQ	Va,Vb,Vt	AE	Maximum unsigned	F4, 1, 2A
MAX.UQ	Va,Sb,Vt	AE	Maximum unsigned scalar	F3, 1, 2A
MAX.SQ	Va,Vb,Vt	AE	Maximum signed	F4, 1, 2B
MAX.SQ	Va,Sb,Vt	AE	Maximum signed scalar	F3, 1, 2B
SUMR.UQ	Va,Vt	AE	Sum reduction step	F3, 1, 48
SUMR.SQ	Va,Vt	AE	Sum reduction step	F3, 1, 49
MINR.UQ	Va,Vt	AE	Minimum reduction step	F3, 1, 4C
MINR.SQ	Va,Vt	AE	Minimum reduction step	F3, 1, 4D
MAXR.UQ	Va,Vt	AE	Maximum reduction step	F3, 1, 4E
MAXR.SQ	Va,Vt	AE	Maximum reduction step	F3, 1, 4F

10.5.1.4 Logical and Shift Operations

Operation		*Type*	*Operation Description*	*Fmt, if, opc*
AND	Va,Sb,Vt	AE	Logical AND scalar	F3, 1, 20
OR	Va,Sb,Vt	AE	Logical OR scalar	F3, 1, 21
NAND	Va,Sb,Vt	AE	Logical NAND scalar	F3, 1, 22
NOR	Va,Sb,Vt	AE	Logical NOR scalar	F3, 1, 23
XOR	Va,Sb,Vt	AE	Logical XOR scalar	F3, 1, 24
XNOR	Va,Sb,Vt	AE	Logical XNOR scalar	F3, 1, 25
ANDC	Va,Sb,Vt	AE	Logical ANDC scalar	F3, 1, 26
ORC	Va,Sb,Vt	AE	Logical ORC scalar	F3, 1, 27

AND	Va,Vb,Vt	AE	Logical AND	F4, 1, 20
OR	Va,Vb,Vt	AE	Logical OR	F4, 1, 21
NAND	Va,Vb,Vt	AE	Logical NAND	F4, 1, 22
NOR	Va,Vb,Vt	AE	Logical NOR	F4, 1, 23
XOR	Va,Vb,Vt	AE	Logical XOR	F4, 1, 24
XNOR	Va,Vb,Vt	AE	Logical XNOR	F4, 1, 25
ANDC	Va,Vb,Vt	AE	Logical ANDC	F4, 1, 26
ORC	Va,Vb,Vt	AE	Logical ORC	F4, 1, 27
SHFL.UQ	Va,Sb,Vt	AE	Shift left unsigned scalar	F3, 1, 2C
SHFL.UQ	Va,Vb,Vt	AE	Shift left unsigned	F4, 1, 2C
SHFL.SQ	Va,Sb,Vt	AE	Shift left signed scalar	F3, 1, 2D
SHFL.SQ	Va,Vb,Vt	AE	Shift left signed	F4, 1, 2D
SHFR.UQ	Va,Sb,Vt	AE	Shift right unsigned scalar	F3, 1, 2E
SHFR.UQ	Va,Vb,Vt	AE	Shift right unsigned	F4, 1, 2E
SHFR.SQ	Va,Sb,Vt	AE	Shift right signed scalar	F3, 1, 2F
SHFR.SQ	Va,Vb,Vt	AE	Shift right signed	F4, 1, 2F

10.5.1.5 Move and Misc. Operations

Operation		*Type*	*Operation Description*	*Fmt, if, opc*
NOP		AE	No Operation	F3, 1, 00
MOV	Aa,AEC	AE	Move A-reg to AEC	F5, 1, 16
MOV	AEC,At	AE	Move AEC to A-reg	F6, 1, 16
MOV	Aa,AES	AE	Move A-reg to AES	F5, 1, 17
MOV	AES,At	AE	Move AES to A-reg	F6, 1, 17
MOV.DW	Va,Vt	AE	Move 32-bit V-reg to V-reg	F3, 1, 08 *
MOV	Va,Vt	AE	Move V-reg to V-reg	F3, 1, 09
MOV	Vcnt,At	AE	Move Vcnt to A-reg	F6, 1, 1D
MOV	VMcnt,At	AE	Move VMcnt to A-reg	F6, 1, 1E
MOV	Immed,VPM	AE	Move Immediate to VPM	F7, 1, 10
MOV	Aa,VPM	AE	Move A-reg to VPM	F5, 1, 10
MOV	VPM,At	AE	Move VPM to A-reg	F6, 1, 10
MOV	Immed,VPA	AE	Move Immediate to VPA	F7, 1, 1A
MOV	Aa,VPA	AE	Move A-reg to VPA	F5, 1, 1A
MOV	VPA,At	AE	Move VPA to A-reg	F6, 1, 1A
MOV	VLmax,At	AE	Move VLmax to A-reg	F6, 1, 18
MOV	Immed,VL	AE	Move Immediate to VL	F7, 1, 11

MOV	Aa,VL	AE	Move A-reg to VL	F5, 1, 11
MOV	VL,At	AE	Move VL to A-reg	F6, 1, 11
MOV	Immed,VS	AE	Move Immediate to VS	F7, 1, 12
MOV	Aa,VS	AE	Move A-reg to VS	F5, 1, 12
MOV	VS,At	AE	Move VS to A-reg	F6, 1, 12
MOV	VPLmax,At	AE	Move VPLmax to A-reg	F6, 1, 19
MOV	Immed,VPL	AE	Move Immediate to VPL	F7, 1, 13
MOV	Aa,VPL	AE	Move A-reg to VPL	F5, 1, 13
MOV	VPL,At	AE	Move VPL to A-reg	F6, 1, 13
MOV	Immed,VPS	AE	Move Immediate to VPS	F7, 1, 14
MOV	Aa,VPS	AE	Move A-reg to VPS	F5, 1, 14
MOV	VPS,At	AE	Move VPS to A-reg	F6, 1, 14
MOV	Sa,Ab,Vt	AE	Move to 64-bit V-reg element	F4, 1, 51
MOV	Va,Ab,St	AE	Move from 64-bit V-reg element	F4, 1, 61
MOV.DW	Sa,Ab,Vt	AE	Move to 32-bit V-reg element	F4, 1, 50 *
MOV.DW	Va,Ab,St	AE	Move from 32-bit V-reg element	F4, 1, 60 *
MOVR	Va,Ab,St	AE	Move reduction partial result	F4, 1, 63
MOVR.DW	Va,Ab,St	AE	Move reduction partial result float single	F4, 1, 62 *
MOV	REDcnt,At	AE	Move reduction partial result count	F6, 1, 1B
TZC	VMa,At	AE	Trailing Zero count VMa to At	F6, 1, 0E
PLC	VMa,At	AE	Population count VMa to At	F6, 1, 0F
VIDX	Aa,SH,Vt	AE	Load index vector plus scalar	F5, 1, 00
VSHF	Va,Sb,Vt	AE	Vector Element Shift	F3, 1, 57
FILL	Sb,Vt	AE	Fill vector with scalar	F3, 0, 2E
SEL	Va,Sb,Vt	AE	Select vector / scalar w/VM	F3, 0, 2F
SEL	Va,Vb,Vt	AE	Select vector / vector w/VM	F4, 0, 2F

11 User Defined Application Specific Personalities

User defined application specific personalities are used to replace a performance critical code section from an application with a single coprocessor instruction. A user defined application specific personality is appropriate when a vector personality does not meet the requirements of the application. Typically, an application specific personality is appropriate when the amount of data required to perform an operation, or the operation itself varies from one operation to the next.

The User Defined Application Specific Infrastructure defines machine state and a set of instructions to access the machine state. A group of instructions are provided that allow the user to define their functionality. The user can access memory with these instructions as well as modify the personality machine state.

11.1 User Defined Application Specific Machine State

The following figure shows the machine state extensions defined for user defined application specific personalities.

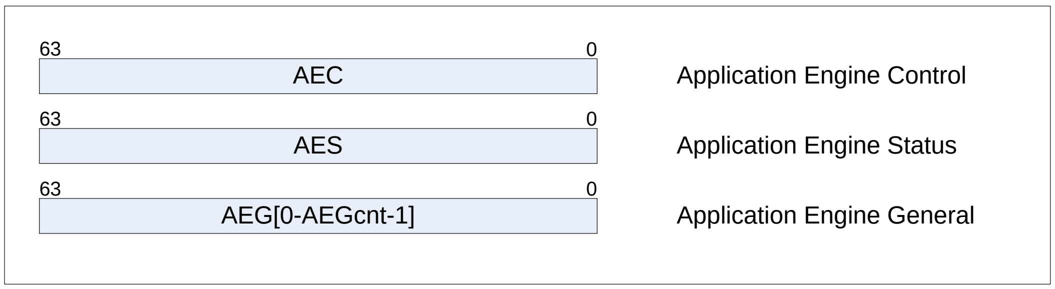

63
0
AEC
Application Engine Control
63
0
AES
Application Engine Status
63
0
AEG[0-AEGcnt-1]
Application Engine General

Figure 38 – User Defined Application Specific Personality Machine State

11.1.1 AEC – Application Engine Control Register

The AEC register consists of the read, write and execute instruction mask fields and exception mask field. The meanings of all bits other than those within the defined fields are reserved. A separate copy of the AEC register is maintained per user command area. All fields are persistent state. The following figure shows the fields of the AEC register.

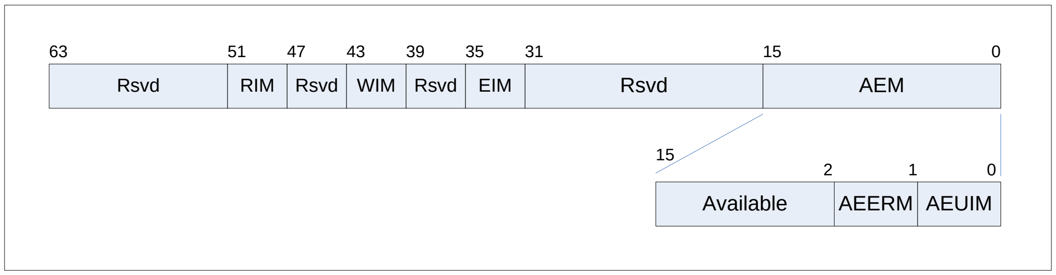


Figure 39 - Application Engine Control Register

The following fields exist within the AEC register:

AEM – Application Engine Mask

The application engine mask (bits 15:0) specifies which exceptions are to be masked (i.e. ignored by the coprocessor). Exceptions with their mask set to one are ignored. Refer to section 11.1.2 for a description of each exception type. The exception types marked as available can be defined by the user defined AE personality.

EIM – Execute Instruction Mask

The execute instruction mask (bits 35:32) specifies which application engines are to participate in a CAEP instruction. A one in bit 32 indicates that AE0 is to execute the instruction; a zero implies that AE0 should ignore the instruction.

Note that two types of CAEP instructions exist, those that use the CIM mask, and those that specify the single AE that is to execute the instruction. The CAEP instructions that specify the single AE that is to execute the instruction do not use the CIM mask.

WIM – Write Instruction Mask

The write instruction mask (bits 43:40) specifies which application engines are to participate in the move data to AE register instructions. A one in bit 40 indicates that AE0 should perform the register write instruction; a zero implies that AE0 should ignore the instruction.

Note that two sets of move instructions exist for user defined application engine personalities. One set uses the WIM mask to specify which AEs participate in a write register instruction. The other set directly specifies a single AE to perform the operation.

RIM – Read Instruction Mask

The read instruction mask (bits 51:48) specifies which application engines are to participate in the move data from AE register instructions. A one in bit 48 indicates that AE0 should perform the register read instruction; a zero implies that AE0 should ignore the instruction.

Note that two sets of move instructions exist for user defined application engine personalities. One set uses the RIM mask to specify which AEs participate in a read register instruction. The other set directly specifies a single AE to provide the result.

The data from all application engines that participate in a register read instruction are OR'ed together and written to the destination register (an A or S register). As an example, reading the AES register with a RIM value of 0xF results in the combine exception status from all four AEs. Performing a register read instruction with a RIM value of 0x0 (I.e. no AEs participate) results in a zero in the destination A or S register.

11.1.2 AES – Application Engine Status

The application engine status register holds the status fields for an application engine. Each application engine may have different AES register values. The entire register is initialized to zero by a coprocessor dispatch. The meaning of all bits other than those within the AEE field is reserved and read as zero.

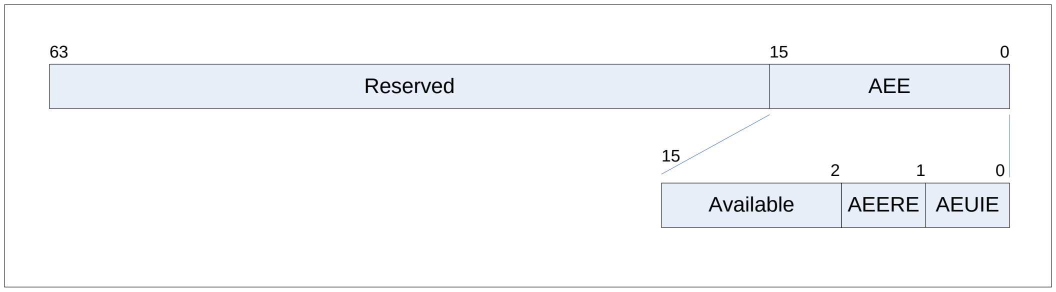


Table 26 - Application Engine Status Register

Instructions check for exceptions during their execution. Detected exceptions are recorded in the AEE field. The AEE field is masked with the AEC AEM field to determine if a recorded exception should interrupt the host processor. The results of the following exception checks are recorded.

Exception Name	*AEE Bit*	*Description*
AEUIE	0	AE Unimplemented Instruction Exception. An attempt was made to execute an unimplemented instruction. The operation of the unimplemented instruction depends on the value of the instruction's opcode. If the opcode is in a range of opcodes that is defined to return a value to the scalar processor then the value zero is returned; otherwise the instruction is equivalent to a NOP.
AEERE	1	AE Element Range Exception. An instruction referenced an AEG register where the element index value exceeded the valid range.

11.1.3 AEG Register

The AEG register is a single register with multiple elements (AEGcnt). Each element is 64-bits in size. The User Defined Application Specific ISA defines instructions that access the AEG register as a single one-dimensional structure. The user defined personality may subdivide the AEG register elements as needed, with each subset organized as multi-dimensional structures as appropriate for the customer algorithm being defined.

The AEG register is non-persistent state.

11.2 User Defined Application Specific Personality Context Save and Restore

The User Defined Application Specific ISA defines the mechanism that context is saved / restored from memory. User defined application specific personality context is saved / restored when a break point is reached when using a debugger. Additionally, context is saved in the form of a process core file when an unmasked exception is detected.

The state saved / restored for user defined application specific personalities includes the AEC, AES and AEG registers. The instruction 'MOV AEGcnt,At' is used to obtain the number of elements contained within the AEG register for each user defined application specific personality.

11.3 User Defined Application Specific ISA

The user defined application specific personality instruction set extensions contain the following classes of instructions:

- Moves to/from Scalar ISA machine state to User Defined Application Specific ISA machine state
- User defined instructions

11.3.1 Instruction Set Organized by Function

11.3.1.1 Move Instructions

The FENCE instruction is a Scalar ISA instruction that when executed is passed on to the application engines.

Operation		*Type*	*Operation Description*	*Fmt, if, opc*
NOP		AE	No Operation	F3, 1, 00
FENCE		A	Memory Fence	F7, 1, 0F
MOV.AEx	Aa,AEC	AE	Move A-reg to AEC, AEx	F5, 0, 16
MOV	Aa,AEC	AE	Move A-reg to AEC, Masked	F5, 1, 16
MOV.AEx	AEC,At	AE	Move AEC to A-reg, AEx	F6, 0, 16
MOV	AEC,At	AE	Move AEC to A-reg, Masked	F6, 1, 16
MOV	Immed,RIM	AE	Move Immed. to AEC.RIM field	F7, 1, 1C
MOV	Immed,WIM	AE	Move Immed. to AEC.WIM field	F7, 1, 1D
MOV	Immed,CIM	AE	Move Immed. to AEC.CIM field	F7, 1, 1E
MOV	Aa,RIM	AE	Move A-reg. to AEC.RIM field	F5, 1, 1C
MOV	Aa,WIM	AE	Move A-reg. to AEC.WIM field	F5, 1, 1D
MOV	Aa,CIM	AE	Move A-reg. to AEC.CIM field	F5, 1, 1E
MOV.AEx	Aa,AES	AE	Move A-reg to AES, AEx	F5, 0, 17
MOV	Aa,AES	AE	Move A-reg to AES, Masked	F5, 1, 17
MOV.AEx	AES,At	AE	Move AES to A-reg, AEx	F6, 0, 17
MOV	AES,At	AE	Move AES to A-reg, Masked	F6, 1, 17
MOV.AEx	AEGcnt,At	AE	Move AEGcnt to A-reg, AEx	F6, 0, 1D
MOV.AEx	Aa,Imm12,AEG	AE	Move A-reg to AEG[Imm12], AEx	F5, 0, 18
MOV	Aa,Imm12,AEG	AE	Move A-reg to AEG[Imm12], Masked	F5, 1, 18
MOV.AEx	AEG,Imm12,At	AE	Move AEG[Imm12] to A-reg, AEx	F6, 0, 1C
MOV	AEG,Imm12,At	AE	Move AEG[Imm12] to A-reg, Masked	F6, 1, 1C

MOV.AEx	Sa,Imm12,AEG	AE	Move S-reg to AEG[Imm12], AEx	F5, 0, 20
MOV	Sa,Imm12,AEG	AE	Move S-reg to AEG[Imm12], Masked	F5, 1, 20
MOV.AEx	AEG,Imm12,St	AE	Move AEG[Imm12] to S-reg, AEx	F4, 0, 70
MOV	AEG,Imm12,St	AE	Move AEG[Imm12] to S-reg, Masked	F4, 1, 70
MOV.AEx	Sa,Ab,AEG	AE	Move S-reg to AEG[A[Ab]] , AEx	F4, 0, 40
MOV	Sa,Ab,AEG	AE	Move S-reg to AEG[A[Ab]] , Masked	F4, 1, 40
MOV.AEx	AEG,Ab,St	AE	Move AEG[A[Ab]] to S-reg, AEx	F4, 0, 68
MOV	AEG,Ab,St	AE	Move AEG[A[Ab]] to S-reg, Masked	F4, 1, 68

11.3.1.2 User Defined Instructions

Operation		*Type*	*Operation Description*	*Fmt, if, opc*
CAEP00.AEx	Imm18	AE	User defined instruction 0x00, AEx	F7, 0, 20
CAEP01.AEx	Imm18	AE	User defined instruction 0x01, AEx	F7, 0, 21
CAEP02.AEx	Imm18	AE	User defined instruction 0x02, AEx	F7, 0, 22
CAEP03.AEx	Imm18	AE	User defined instruction 0x03, AEx	F7, 0, 23
CAEP04.AEx	Imm18	AE	User defined instruction 0x04, AEx	F7, 0, 24
CAEP05.AEx	Imm18	AE	User defined instruction 0x05, AEx	F7, 0, 25
CAEP06.AEx	Imm18	AE	User defined instruction 0x06, AEx	F7, 0, 26
CAEP07.AEx	Imm18	AE	User defined instruction 0x07, AEx	F7, 0, 27
CAEP08.AEx	Imm18	AE	User defined instruction 0x08, AEx	F7, 0, 28
CAEP09.AEx	Imm18	AE	User defined instruction 0x09, AEx	F7, 0, 29
CAEP0A.AEx	Imm18	AE	User defined instruction 0x0A, AEx	F7, 0, 2A
CAEP0B.AEx	Imm18	AE	User defined instruction 0x0B, AEx	F7, 0, 2B
CAEP0C.AEx	Imm18	AE	User defined instruction 0x0C, AEx	F7, 0, 2C
CAEP0D.AEx	Imm18	AE	User defined instruction 0x0D, AEx	F7, 0, 2D
CAEP0E.AEx	Imm18	AE	User defined instruction 0x0E, AEx	F7, 0, 2E
CAEP0F.AEx	Imm18	AE	User defined instruction 0x0F, AEx	F7, 0, 2F
CAEP10.AEx	Imm18	AE	User defined instruction 0x10, AEx	F7, 0, 30
CAEP11.AEx	Imm18	AE	User defined instruction 0x11, AEx	F7, 0, 31
CAEP12.AEx	Imm18	AE	User defined instruction 0x12, AEx	F7, 0, 32
CAEP13.AEx	Imm18	AE	User defined instruction 0x13, AEx	F7, 0, 33
CAEP14.AEx	Imm18	AE	User defined instruction 0x14, AEx	F7, 0, 34
CAEP15.AEx	Imm18	AE	User defined instruction 0x15, AEx	F7, 0, 35
CAEP16.AEx	Imm18	AE	User defined instruction 0x16, AEx	F7, 0, 36
CAEP17.AEx	Imm18	AE	User defined instruction 0x17, AEx	F7, 0, 37
CAEP18.AEx	Imm18	AE	User defined instruction 0x18, AEx	F7, 0, 38
CAEP19.AEx	Imm18	AE	User defined instruction 0x19, AEx	F7, 0, 39
CAEP1A.AEx	Imm18	AE	User defined instruction 0x1A, AEx	F7, 0, 3A
CAEP1B.AEx	Imm18	AE	User defined instruction 0x1B, AEx	F7, 0, 3B
CAEP1C.AEx	Imm18	AE	User defined instruction 0x1C, AEx	F7, 0, 3C
CAEP1D.AEx	Imm18	AE	User defined instruction 0x1D, AEx	F7, 0, 3D
CAEP1E.AEx	Imm18	AE	User defined instruction 0x1E, AEx	F7, 0, 3E
CAEP1F.AEx	Imm18	AE	User defined instruction 0x1F, AEx	F7, 0, 3F

CAEP00	Imm18	AE	User defined instruction 0x00, Masked	F7, 1, 20
CAEP01	Imm18	AE	User defined instruction 0x01, Masked	F7, 1, 21
CAEP02	Imm18	AE	User defined instruction 0x02, Masked	F7, 1, 22
CAEP03	Imm18	AE	User defined instruction 0x03, Masked	F7, 1, 23
CAEP04	Imm18	AE	User defined instruction 0x04, Masked	F7, 1, 24
CAEP05	Imm18	AE	User defined instruction 0x05, Masked	F7, 1, 25
CAEP06	Imm18	AE	User defined instruction 0x06, Masked	F7, 1, 26
CAEP07	Imm18	AE	User defined instruction 0x07, Masked	F7, 1, 27
CAEP08	Imm18	AE	User defined instruction 0x08, Masked	F7, 1, 28
CAEP09	Imm18	AE	User defined instruction 0x09, Masked	F7, 1, 29
CAEP0A	Imm18	AE	User defined instruction 0x0A, Masked	F7, 1, 2A
CAEP0B	Imm18	AE	User defined instruction 0x0B, Masked	F7, 1, 2B
CAEP0C	Imm18	AE	User defined instruction 0x0C, Masked	F7, 1, 2C
CAEP0D	Imm18	AE	User defined instruction 0x0D, Masked	F7, 1, 2D
CAEP0E	Imm18	AE	User defined instruction 0x0E, Masked	F7, 1, 2E
CAEP0F	Imm18	AE	User defined instruction 0x0F, Masked	F7, 1, 2F
CAEP10	Imm18	AE	User defined instruction 0x10, Masked	F7, 1, 30
CAEP11	Imm18	AE	User defined instruction 0x11, Masked	F7, 1, 31
CAEP12	Imm18	AE	User defined instruction 0x12, Masked	F7, 1, 32
CAEP13	Imm18	AE	User defined instruction 0x13, Masked	F7, 1, 33
CAEP14	Imm18	AE	User defined instruction 0x14, Masked	F7, 1, 34
CAEP15	Imm18	AE	User defined instruction 0x15, Masked	F7, 1, 35
CAEP16	Imm18	AE	User defined instruction 0x16, Masked	F7, 1, 36
CAEP17	Imm18	AE	User defined instruction 0x17, Masked	F7, 1, 37
CAEP18	Imm18	AE	User defined instruction 0x18, Masked	F7, 1, 38
CAEP19	Imm18	AE	User defined instruction 0x19, Masked	F7, 1, 39
CAEP1A	Imm18	AE	User defined instruction 0x1A, Masked	F7, 1, 3A
CAEP1B	Imm18	AE	User defined instruction 0x1B, Masked	F7, 1, 3B
CAEP1C	Imm18	AE	User defined instruction 0x1C, Masked	F7, 1, 3C
CAEP1D	Imm18	AE	User defined instruction 0x1D, Masked	F7, 1, 3D
CAEP1E	Imm18	AE	User defined instruction 0x1E, Masked	F7, 1, 3E
CAEP1F	Imm18	AE	User defined instruction 0x1F, Masked	F7, 1, 3F

Note that the user defined instructions can be further subdivided by using the 18-bit immediate field to differentiate the operation to be performed.

12 Debugging Support

12.1 Introduction

The ability to effectively determine why an application is not operating as expected is a significant aspect of programmer productivity. The coprocessor provides the generally expected debugging capabilities: single stepping, running to a break point, and viewing/modifying machine state.

12.2 Break Instruction

The coprocessor supports a break instruction (BRK). The execution of the break instruction causes instruction execution to stop cleanly such that coprocessor context can be saved to memory. All currently executing instructions are allowed to complete and no new instructions are started. Execution can be resumed by issuing the supervisor resume command.

12.3 Single Stepping

Single stepping is accomplished by using break instructions to allow only one user application instruction to be executed. Non conditional program flow instructions require a single break instruction to be inserted. Conditional program flow instructions require two break instructions to be inserted.

12.4 Machine State Visibility

Machine state can be viewed by saving user context to memory and then using normal host processor memory loads. Coprocessor register state can be modified by changing the values in a process's memory context and then loading the context back into the coprocessor registers.

12.5 Precise Trap Mode

The coprocessor executes many instructions in parallel. When a trap or page fault occurs it will be difficult to determine, by looking at the current user register state, how and why the exception occurred. The coprocessor supports an execution mode that allows only one instruction to be executed at a time. This mode significantly slows down the execution of the program, but will allow an exception to be isolated to a single instruction. Precise Trap mode can be enabled by writing to the Coprocessor Control register (CPC). See section 9.1.7 for more information on setting Precise Trap mode.

A Instruction Formats

The A, S and AE instruction formats are described in this appendix. Instruction formats consist of a number of fields. Three types of fields exist: instruction bundle specifier, instruction format specifier, and instruction specific fields. The instruction bundle specifier field, **it**, is used to determine the format of the instruction bundle. The instruction format specifier, **fmt**, is a variable width field with its most significant bit being bit 28 of each instruction. The instruction opcode field, **opc**, is used to specify the specific operation to be performed when the instruction is executed. All other fields within the instruction are instruction specific depending on the needs of each instruction.

The eight defined formats are listed below with an example for each of the three instruction types (A, S and AE). Note that for the example instructions shown below the **vm**, **it**, and all fields to the right of the opcode field (**opc**) are instruction specific and will change for other instructions within the same format.

Format F1

A Register Instruction Example: LD.UQ Immed(Aa),At

31 29	28	27 22	21 12	11 6	5 0
it	0	opc	immed10	Aa	At

S Register Instruction Example: LD.UQ Immed(Aa),St

31 30	29	28	27 22	21 12	11 6	5 0
it	if	0	opc	immed10	Aa	St

AE Instruction Example: LD.UQ Immed(Aa),Vt

31 30	29	28	27 22	21 12	11 6	5 0
vm	if	0	opc	immed10	Aa	Vt

Format F2

A Register Instruction Example: BR.BU CCt,Target

31 29	28 27	26 22	21	20 6	5	4 0
it	10	opc	ar	immed15	tf	cc

S Register Instruction – No S instructions reside in format F2

31 30	29	28 27	26 24	23 0
it	if	10	opc	Instruction Specific

AE Instruction Example: FMA.FS Va,Sb,Vc,Vt

31	29	28 27	26 24	23 18	17 12	11 6	5 0
vm	if	10	opc	Vc	Sb	Va	Vt

Format F3

A Register Instruction Example: SHFL.UQ Aa,SH,At

31 29	28 25	24 18	17 12	11 6	5 0
it	1100	opc	Immed6	Aa	At

S Register Instruction Example: SHFL.UQ Sa,SH,St

31	30	29 28	25 24 18	17 12	11 6	5 0
it	if	1100	opc	Immed6	Sa	St

AE Instruction Example: SHFL.UQ Va,Sb,Vt

31	30	29 28	25 24 18	17 12	11 6	5 0
vm	if	1100	opc	Sb	Va	Vt

Format F4

A Register Instruction Example: LD.UQ Ab(Aa),At

31 29	28 25	24 18	17 12	11 6	5 0
it	1101	opc	Ab	Aa	At

S Register Instruction Example: LD.UQ Ab(Aa),St

31	30	29 28	25 24 18	17 12	11 6	5 0
it	if	1101	opc	Sb	Sa	St

AE Instruction Example: LD.UQ Vb(Aa),Vt

31	30	29 28	25 24 18	17 12	11 6	5 0
vm	if	1101	opc	Vb	Va	Vt

Format F5

A Register Instructions Example: MOV Aa,PIP

31 29	28 24	23 18	17 12	11 6	5 0
it	11100	opc	000000	Aa	000000

S Register Instructions – No S instructions reside in format F5

31	30	29 28 24	23 18	17 0
it	if	11100	opc	Instruction Specific

AE Instruction Example: MOV Aa,AEC

31	30	29 28 24	23 18	17 12	11 6	5 0
vm	if	11100	opc	000000	Aa	000000

Format F6

A Register Instruction Example: MOV PIP,At

31 29	28 24	23 18	17 12	11 6	5 0
it	11101	opc	000000	000000	At

S Register Instructions – No S instructions reside in format F6

31	30 29	28 24	23 18	17 0
it	if	11101	opc	Instruction Specific

AE Instruction Example: MOV VPM,At

31	30 29	28 24	23 18	17 12	11 6	5 0
vm	if	11101	opc	000000	000000	At

Format F7

A Register Instructions Example: MOV Imm,VMA

31 29	28 24	23 18	17 0
it	11110	opc	Immed18

S Register Instructions – No S instructions reside in format F7

31	30 29	28 24	23 18	17 0
it	if	11110	opc	Instruction Specific

AE Instruction Example: MOV Imm,VPM

31	30 29	28 24	23 18	17 0
vm	if	11110	opc	Immed18

Format F8

A Register Instructions – No A instructions reside in format F8

31 29	28 24	23 18	17 0
it	11111	opc	Instruction Specific

S Register Instructions – No S instructions reside in format F8

31	30 29	28 24	23 18	17 0
it	if	11111	opc	Instruction Specific

AE Instructions – No AE instructions reside in format F8

31	30 29	28 24	23 18	17 0
vm	if	11111	opc	Instruction Specific

B Opcode Space Usage

Appendix B lists the instructions ordered numerically by opcode. Blocks of AE opcode space are reserved for instructions that require specific interactions with the A and S registers. Future AE instructions may be added to current or future AE personalities provided that they reside in the appropriate opcode blocks. The following table lists the opcode blocks and defined A/S register interactions.

Instruction bundle type zero contains an A instruction, an AE instruction, and a 64-bit AE constant. The 64-bit AE constant is used instead of an A or S register reference for AE instructions. The table below indicates using bold text where the 64-bit constant is used.

Format Range	*Example Instruction*		*A/S Register Interactions*
F1,00 – F1,1F	LD.UQ	**Offset**(Aa),Vt	Send A[Aa]+Offset to AE
F1,20 – F1,3F	-		No interaction
F2,00 – F2,03	FMA.FS	Va,**Sb**,Vc,Vt	Send S[Sb] to AE
F2,04 – F2,07	FMA.FS	Va,Vb,Vc,Vt	No interaction
F3,00 – F3,0F	NEG.FS	Va,Vt	No interaction
F3,10 – F3,3F	ADD.FS	Va,**Sb**,Vt	Send S[Sb] to AE
F3,40 – F3,4F	SUM.FS	Va,Vt	No interaction
F3,50 – F3,6F	VSHF	Va,**Sb**,Vt	Send S[Sb] to AE
F3,70 – F3,7F	RCPTLA.FS	Va,Vt	No interaction
F4,00 – F4,0F	LD.UQ	Vb(**Aa**),Vt	Send A[Aa] to AE
F4,10 – F4,3F	ADD.FS	Va,Vb,Vt	No interaction
F4,40 – F4,4F	MOV	**Sa**,Ab,AEG	Send S[Sa], Send A[Ab]<17:0> in Ra, Rb and Rt fields
F4,50 – F4,5F	MOV	**Sa**,Ab,Vt	Send S[Sa], Send A[Ab]<11:0> in Ra and Rb fields
F4,60 – F4,6F	MOV	Va,**Ab**,St	Send A[Ab] to AE, Receive S[St] from AE
F4,70 – F4,7F	MOV	AEG,**Imm18**,St	Send Immed18 to AE, Receive S[St] from AE
F5,00 – F5,1F	VIDX	**Aa**,SH,Vt	Send A[Aa] to AE
F5,20 – F5,3F	MOV	**Sa**,Imm12,AEG	Send S[Sa] to AE
F6,00 – F6,1F	MOV	AEC,At	Receive A[At] from AE
F6,20 – F6,2F	-		Send S[Sb] to AE
F6,30 – F6,3F	-		No Interaction
F7,00 – F7,1F	MOV	**Imm18**,VS	Send Immed18 to AE
F7,20 – F7,3F	CAEP	**Imm18**	Send Immed18 to AE
F8,00 – F7,3F			Reserved

Format F1

Format	*A Instruction*		*S Instruction (if=0)*		*S Instruction (if=1)*		*AE Instruction (if=0)*			*AE Instruction (if=1)*		
F1,00	LD.UB	Off(Aa),At	-		LD.UB	Off(Aa),St	-			LD.UB	Off(Aa),Vt	BVI
F1,01	LD.UW	Off(Aa),At	-		LD.UW	Off(Aa),St	-			LD.UW	Off(Aa),Vt	BVI
F1,02	LD.UD	Off(Aa),At	-		LD.UD	Off(Aa),St	LD.DW	Off(Aa),Vt	BVI	LD.UD	Off(Aa),Vt	BVI
F1,03	LD.UQ	Off(Aa),At	-		LD.UQ	Off(Aa),St	-			LD.UQ	Off(Aa),Vt	BVI
F1,04	LD.SB	Off(Aa),At	-		LD.SB	Off(Aa),St	-			LD.SB	Off(Aa),Vt	BVI
F1,05	LD.SW	Off(Aa),At	-		LD.SW	Off(Aa),St	-			LD.SW	Off(Aa),Vt	BVI
F1,06	LD.SD	Off(Aa),At	-		LD.SD	Off(Aa),St	LD.DDW	Off(Aa),Vt	BVI	LD.SD	Off(Aa),Vt	BVI
F1,07	-		-		-		-			-		
F1,08	ST.UB	At,Off(Aa)	-		ST.UB	St,Off(Aa)	-			ST.UB	Vt,Off(Aa)	BVI
F1,09	ST.UW	At,Off(Aa)	-		ST.UW	St,Off(Aa)	-			ST.UW	Vt,Off(Aa)	BVI
F1,0A	ST.UD	At,Off(Aa)	ST.FS	St,Off(Aa)	ST.UD	St,Off(Aa)	ST.DW	Vt,Off(Aa)	BVI	ST.UD	Vt,Off(Aa)	BVI
F1,0B	ST.UQ	At,Off(Aa)	-		ST.UQ	St,Off(Aa)	-			ST.UQ	Vt,Off(Aa)	BVI
F1,0C	ST.SB	At,Off(Aa)	-		ST.SB	St,Off(Aa)	-			ST.SB	Vt,Off(Aa)	BVI
F1,0D	ST.SW	At,Off(Aa)	-		ST.SW	St,Off(Aa)	-			ST.SW	Vt,Off(Aa)	BVI
F1,0E	ST.SD	At,Off(Aa)	-		ST.SD	St,Off(Aa)	ST.DDW	Vt,Off(Aa)	BVI	ST.SD	Vt,Off(Aa)	BVI
F1,0F	-		-		-		-			-		
F1,10	-		-		-		-			-		
F1,11	-		-		-		-			-		
F1,12	-		-		-		-			-		
F1,13	-		-		-		-			LD	Off(Aa),VMt	BVI
F1,14	-		-		-		-			-		
F1,15	-		-		-		-			-		
F1,16	-		-		-		-			-		
F1,17	-		-		-		-			-		
F1,18	-		-		-		-			-		
F1,19	-		-		-		-			-		
F1,1A	-		-		-		-			-		
F1,1B	-		-		-		-			ST	VMt,Off(Aa)	BVI
F1,1C	-		-		-		-			-		
F1,1D	-		-		-		-			-		
F1,1E	-		-		-		-			-		
F1,1F	-		-		-		-			-		
F1,20	AND	Aa,Imm,At	-		AND	Sa,Imm,St	-			-		
F1,21	OR	Aa,Imm,At	-		OR	Sa,Imm,St	-			-		
F1,22	NAND	Aa,Imm,At	-		NAND	Sa,Imm,St	-			-		
F1,23	NOR	Aa,Imm,At	-		NOR	Sa,Imm,St	-			-		
F1,24	XOR	Aa,Imm,At	-		XOR	Sa,Imm,St	-			-		
F1,25	XNOR	Aa,Imm,At	-		XNOR	Sa,Imm,St	-			-		
F1,26	ANDC	Aa,Imm,At	-		ANDC	Sa,Imm,St	-			-		
F1,27	ORC	Aa,Imm,At	-		ORC	Sa,Imm,St	-			-		
F1,28	-		-		-		-			-		
F1,29	-		-		-		-			-		
F1,2A	-		-		-		-			-		
F1,2B	-		-		-		-			-		
F1,2C	-		-		-		-			-		
F1,2D	-		-		-		-			-		
F1,2E	-		-		-		-			-		
F1,2F	-		-		-		-			-		
F1,30	ADD.UQ	Aa,Imm,At	-		ADD.UQ	Sa,Imm,St	-			-		
F1,31	ADD.SQ	Aa,Imm,At	-		ADD.SQ	Sa,Imm,St	-			-		
F1,32	SUB.UQ	Aa,Imm,At	-		SUB.UQ	Sa,Imm,St	-			-		
F1,33	SUB.SQ	Aa,Imm,At	-		SUB.SQ	Sa,Imm,St	-			-		
F1,34	MUL.UQ	Aa,Imm,At	-		MUL.UQ	Sa,Imm,St	-			-		
F1,35	MUL.SQ	Aa,Imm,At	-		MUL.SQ	Sa,Imm,St	-			-		
F1,36	-		-		-		-			-		
F1,37	-		-		-		-			-		
F1,38	-		-		-		-			-		
F1,39	-		-		-		-			-		
F1,3A	CMP.UQ	Aa,Im,ACt	-		CMP.UQ	Sa,Im,SCt	-			-		
F1,3B	CMP.SQ	Aa,Im,ACt	-		CMP.US	Sa,Im,SCt	-			-		
F1,3C	-		-		-		-			-		
F1,3D	-		-		-		-			-		
F1,3E	-		-		-		-			-		
F1,3F	-		-		-		-			-		

Format F2

Format	*A Instruction*		*S Instruction (if=0)*	*S Instruction (if=1)*	*AE Instruction (if=0)*	*AE Instruction (if=1)*
F2,00	-		-	-	-	-
F2,01	-		-	-	-	-
F2,02	BRK		-	-	-	-
F2,03	-		-	-	-	-
F2,04	BR	Target(At)	-	-	-	-
F2,05	BR	CCt,Target	-	-	-	-
F2,06	BR.BL	CCt,Target	-	-	-	-
F2,07	BR.BU	CCt,Target	-	-	-	-
F2,08	CALL	Target(At)				
F2,09	CALL	CCt,Target				
F2,0A	CALL.BL	CCt,Target				
F2,0B	CALL.BU	CCt,Target				
F2,0C	RTN					
F2,0D	RTN	CCt				
F2,0E	RTN.BL	CCt				
F2,0F	RTN.BU	CCt				
F2,10	-					
F2,11	-					
F2,12	-					
F2,13	-					
F2,14	-					
F2,15	-					
F2,16	-					
F2,17	-					
F2,18	-					
F2,19	-					
F2,1A	-					
F2,1B	-					
F2,1C	-					
F2,1D	-					
F2,1E	-					
F2,1F	-					

Format F3

Format	A Instruction		S Instruction (if=0)		S Instruction (if=1)		AE Instruction (if=0)			AE Instruction (if=1)		
F3,00	NOP		-		NOP		-			NOP		
F3,01	CVT.UQ.SQ	Aa,At	CVT.FS.SQ	Sa,St	CVT.UQ.SQ	Sa,St	-			-		
F3,02	-		-		-		-			CVT.UQ.FS	Va,Vt	BVI
F3,03	-		CVT.FS.FD	Sa,St	-		-			-		
F3,04	CVT.SQ.UQ	Aa,At	-		CVT.SQ.UQ	Sa,St	-			-		
F3,05	-		CVT.FD.SQ	Sa,St	-		-			-		
F3,06	-		CVT.FD.FS	Sa,St	CVT.SQ.FS	Sa,St	-			CVT.SQ.FS	Va,Vt	BVI
F3,07	-		-		CVT.SQ.FD	Sa,St	-			-		
F3,08	-		-		-		-			MOV.FS	Va,Vt	BVI
F3,09	-		-		-		-			MOV	Va,Vt	BVI
F3,0A	-		ABS.FS	Sa,St	-		-			-		
F3,0B	ABS.SQ	Aa,At	ABS.FD	Sa,St	ABS.SQ	Sa,St	-			ABS.SQ	Va,Vt	BVI
F3,0C	-		NEG.FS	Sa,St	-		-			-		
F3,0D	NEG.SQ	Aa,At	NEG.FD	Sa,St	NEG.SQ	Sa,St	-			NEG.SQ	Va,Vt	BVI
F3,0E	-		SQRT.FS	Sa,St	-		-			-		
F3,0F	-		SQRT.FD	Sa,St	-		-			-		
F3,10	-		-		-		-			-		
F3,11	-		-		-		-			-		
F3,12	-		SUB.FS	Imm,Sa,St	-		-			SUB.UQ	Sb,Va,Vt	BVI
F3,13	-		SUB.FD	Imm,Sa,St	-		-			SUB.SQ	Sb,Va,Vt	BVI
F3,14	-		-		-		-			-		
F3,15	-		-		-		-			-		
F3,16	-		-		-		-			-		
F3,17	-		-		-		-			-		
F3,18	-		-		-		-			-		
F3,19	-		-		-		-			-		
F3,1A	-		-		-		-			-		
F3,1B	-		-		-		-			-		
F3,1C	MOV	Sa,At	-		MOV	Aa,St	-			-		
F3,1D	-		-		-		-			-		
F3,1E	MOV	AAa,At	-		MOV	SAa,St	-			-		
F3,1F	MOV	Aa,AAt	-		MOV	Sa,SAt	-			-		
F3,20	-		-		-		-			AND	Va,Sb,Vt	BVI
F3,21	-		-		-		-			OR	Va,Sb,Vt	BVI
F3,22	-		-		-		-			NAND	Va,Sb,Vt	BVI
F3,23	-		-		-		-			NOR	Va,Sb,Vt	BVI
F3,24	-		-		-		-			XOR	Va,Sb,Vt	BVI
F3,25	-		-		-		-			XNOR	Va,Sb,Vt	BVI
F3,26	-		-		-		-			ANDC	Va,Sb,Vt	BVI
F3,27	-		-		-		-			ORC	Va,Sb,Vt	BVI
F3,28	-		-		-		-			MIN.UQ	Va,Sb,Vt	BVI
F3,29	-		-		-		-			MIN.SQ	Va,Sb,Vt	BVI
F3,2A	-		-		-		-			MAX.UQ	Va,Sb,Vt	BVI
F3,2B	-		-		-		-			MAX.SQ	Va,Sb,Vt	BVI
F3,2C	SHFL.UQ	Aa,SH,At	-		SHFL.UQ	Sa,SH,St	-			SHFL.UQ	Va,Sb,Vt	BVI
F3,2D	SHFL.SQ	Aa,SH,At	-		SHFL.SQ	Sa,SH,St	-			SHFL.SQ	Va,Sb,Vt	BVI
F3,2E	SHFR.UQ	Aa,SH,At	-		SHFR.UQ	Sa,SH,St	FILL	Sb,Vt	BVI	SHFR.UQ	Va,Sb,Vt	BVI
F3,2F	SHFR.SQ	Aa,SH,At	-		SHFR.SQ	Sa,SH,St	SEL	Va,Sb,Vt	BVI	SHFR.SQ	Va,Sb,Vt	BVI
F3,30	-		ADD.FS	Sa,Imm,St	-		-			ADD.UQ	Va,Sb,Vt	BVI
F3,31	-		ADD.FD	Sa,Imm,St	-		-			ADD.SQ	Va,Sb,Vt	BVI
F3,32	-		SUB.FS	Sa,Imm,St	-		-			SUB.UQ	Va,Sb,Vt	BVI
F3,33	-		SUB.FD	Sa,Imm,St	-		-			SUB.SQ	Va,Sb,Vt	BVI
F3,34	-		MUL.FS	Sa,Imm,St	-		-			MUL.UQ	Va,Sb,Vt	BVI
F3,35	-		MUL.FD	Sa,Imm,St	-		-			MUL.SQ	Va,Sb,Vt	BVI
F3,36	DIV.UQ	Aa,Imm,At	DIV.FS	Sa,Imm,St	DIV.UQ	Sa,Imm,St	-			-		
F3,37	DIV.SQ	Aa,Imm,At	DIV.FD	Sa,Imm,St	DIV.SQ	Sa,Imm,St	-			-		
F3,38	-		-		-		-			-		
F3,39	-		-		-		-			DOTB	Va,Sb,VMt	BVI
F3,3A	-		CMP.FS	Sa,Imm,SCt	-		-			EQ.UQ	Va,Sb,VMt	BVI
F3,3B	-		CMP.FD	Sa,Imm,SCt	-		-			EQ.SQ	Va,Sb,VMt	BVI
F3,3C	-		-		-		-			LT.UQ	Va,Sb,VMt	BVI
F3,3D	-		-		-		-			LT.SQ	Va,Sb,VMt	BVI
F3,3E	-		-		-		-			GT.UQ	Va,Sb,VMt	BVI
F3,3F	-		-		-		-			GT.SQ	Va,Sb,VMt	BVI

Format	*A Instruction*	*S Instruction (if=0)*	*S Instruction (if=1)*	*AE Instruction (if=0)*	*AE Instruction (if=1)*
F3,40	-	-	-	-	-
F3,41	-	-	-	-	-
F3,42	-	-	-	-	-
F3,43	-	-	-	-	-
F3,44	-	-	-	-	-
F3,45	-	-	-	-	-
F3,46	-	-	-	-	-
F3,47	-	-	-	-	-
F3,48	-	-	-	-	SUMR.UQ Va,Vt BVI
F3,49	-	-	-	-	SUMR.SQ Va,Vt BVI
F3,4A	-	-	-	-	-
F3,4B	-	-	-	-	-
F3,4C	-	-	-	-	MINR.UQ Va,Vt BVI
F3,4D	-	-	-	-	MINR.SQ Va,Vt BVI
F3,4E	-	-	-	-	MAXR.UQ Va.Vt BVI
F3,4F	-	-	-	-	MAXR.SQ Va,Vt BVI
F3,50	-	-	-	-	-
F3,51	-	-	-	-	-
F3,52	-	-	-	-	-
F3,53	-	-	-	-	-
F3,54	-	-	-	-	-
F3,55	-	-	-	-	-
F3,56	-	-	-	-	-
F3,57	-	-	-	-	VSHF Va,Sb,Vt BVI
F3,58	-	-	-	-	-
F3,59	-	-	-	-	-
F3,5A	-	-	-	-	-
F3,5B	-	-	-	-	-
F3,5C	-	-	-	-	-
F3,5D	-	-	-	-	-
F3,5E	-	-	-	-	-
F3,5F	-	-	-	-	-
F3,60	-	-	-	-	-
F3,61	-	-	-	-	-
F3,62	-	-	-	-	-
F3,63	-	-	-	-	-
F3,64	-	-	-	-	-
F3,65	-	-	-	-	-
F3,66	-	-	-	-	-
F3,67	-	-	-	-	-
F3,68	-	-	-	-	-
F3,69	-	-	-	-	-
F3,6A	-	-	-	-	-
F3,6B	-	-	-	-	-
F3,6C	-	-	-	-	-
F3,6D	-	-	-	-	-
F3,6E	-	-	-	-	-
F3,6F	-	-	-	-	-
F3,70	-	-	-	-	-
F3,71	-	-	-	-	-
F3,72	-	-	-	-	-
F3,73	-	-	-	-	-
F3,74	-	-	-	-	-
F3,75	-	-	-	-	-
F3,76	-	-	-	-	-
F3,77	-	-	-	-	-
F3,78	-	-	-	-	-
F3,79	-	-	-	-	-
F3,7A	-	-	-	-	-
F3,7B	-	-	-	-	-
F3,7C	-	-	-	-	-
F3,7D	-	-	-	-	-
F3,7E	-	-	-	-	-
F3,7F	-	-	-	-	-

Format F4

Format	*A Instruction*	*S Instruction (if=0)*	*S Instruction (if=1)*	*AE Instruction (if=0)*	*AE Instruction (if=1)*
F4,00	LD.UB Ab(Aa),At	-	LD.UB Ab(Aa),St	-	-
F4,01	LD.UW Ab(Aa),At	-	LD.UW Ab(Aa),St	-	-
F4,02	LD.UD Ab(Aa),At	-	LD.UD Ab(Aa),St	LD.DW Vb(Aa),Vt BVI	LD.UD Vb(Aa),Vt BVI
F4,03	LD.UQ Ab(Aa),At	-	LD.UQ Ab(Aa),St	-	LD.UQ Vb(Aa),Vt BVI
F4,04	LD.SB Ab(Aa),At	-	LD.SB Ab(Aa),St	-	-
F4,05	LD.SW Ab(Aa),At	-	LD.SW Ab(Aa),St	-	-
F4,06	LD.SD Ab(Aa),At	-	LD.SD Ab(Aa),St	-	LD.SD Vb(Aa),Vt BVI
F4,07	-	-	-	-	-
F4,08	ST.UB At,Ab(Aa)	-	ST.UB St,Ab(Aa)	-	-
F4,09	ST.UW At,Ab(Aa)	-	ST.UW St,Ab(Aa)	-	-
F4,0A	ST.UD At,Ab(Aa)	ST.FS St,Ab(Aa)	ST.UD St,Ab(Aa)	ST.DW Vt,Vb(Aa) BVI	ST.UD Vt,Vb(Aa) BVI
F4,0B	ST.UW At,Ab(Aa)	-	ST.UW St,Ab(Aa)	-	ST.UQ Vt,Vb(Aa) BVI
F4,0C	ST.SB At,Ab(Aa)	-	ST.SB St,Ab(Aa)	-	-
F4,0D	ST.SW At,Ab(Aa)	-	ST.SW St,Ab(Aa)	-	-
F4,0E	ST.SD At,Ab(Aa)	-	ST.SD St,Ab(Aa)	-	ST.SD Vt,Vb(Aa) BVI
F4,0F	-	-	-	-	-
F4,10	AND CCa,CCb,CCt	-	-	-	AND VMa,VMb,VMt BVI
F4,11	OR CCa,CCb,CCt	-	-	-	OR VMa,VMb,VMt BVI
F4,12	NAND CCa,CCb,CCt	-	-	-	NAND VMa,VMb,VMt BVI
F4,13	NOR CCa,CCb,CCt	-	-	-	NOR VMa,VMb,VMt BVI
F4,14	XOR CCa,CCb,CCt	-	-	-	XOR VMa,VMb,VMt BVI
F4,15	XNOR CCa,CCb,CCt	-	-	-	XNOR VMa,VMb,VMt BVI
F4,16	ANDC CCa,CCb,CCt	-	-	-	ANDC VMa,VMb,VMt BVI
F4,17	ORC CCa,CCb,CCt	-	-	-	ORC VMa,VMb,VMt BVI
F4,18	SEL AC3.EQ,Aa,Ab,At	-	SEL AC3.EQ,Sa,Sb,St	-	-
F4,19	SEL AC3.GT,Aa,Ab,At	-	SEL AC3.GT,Sa,Sb,St	-	-
F4,1A	SEL AC3.LT,Aa,Ab,At	-	SEL AC3.LT,Sa,Sb,St	-	-
F4,1B	-	-	-	-	-
F4,1C	SEL SC3.EQ,Aa,Ab,At	-	SEL SC3.EQ,Sa,Sb,St	-	-
F4,1D	SEL SC3.GT,Aa,Ab,At	-	SEL SC3.GT,Sa,Sb,St	-	-
F4,1E	SEL SC3.LT,Aa,Ab,At	-	SEL SC3.LT,Sa,Sb,St	-	-
F4,1F	-	-	-	-	-
F4,20	AND Aa,Ab,At	-	AND Sa,Sb,St	-	AND Va,Vb,Vt BVI
F4,21	OR Aa,Ab,At	-	OR Sa,Sb,St	-	OR Va,Vb,Vt BVI
F4,22	NAND Aa,Ab,At	-	NAND Sa,Sb,St	-	NAND Va,Vb,Vt BVI
F4,23	NOR Aa,Ab,At	-	NOR Sa,Sb,St	-	NOR Va,Vb,Vt BVI
F4,24	XOR Aa,Ab,At	-	XOR Sa,Sb,St	-	XOR Va,Vb,Vt BVI
F4,25	XNOR Aa,Ab,At	-	XNOR Sa,Sb,St	-	XNOR Va,Vb,Vt BVI
F4,26	ANDC Aa,Ab,At	-	ANDC Sa,Sb,St	-	ANDC Va,Vb,Vt BVI
F4,27	ORC Aa,Ab,At	-	ORC Sa,Sb,St	-	ORC Va,Vb,Vt BVI
F4,28	-	-	-	-	MIN.UQ Va,Vb,Vt BVI
F4,29	-	-	-	-	MIN.SQ Va,Vb,Vt BVI
F4,2A	-	-	-	-	MAX.UQ Va,Vb,Vt BVI
F4,2B	-	-	-	-	MAX.SQ Va,Vb,Vt BVI
F4,2C	SHFL.UQ Aa,Ab,At	-	SHFL.UQ Sa,Sb,St	-	SHFL.UQ Va,Vb,Vt BVI
F4,2D	SHFL.SQ Aa,Ab,At	-	SHFL.SQ Sa,Sb,St	-	SHFL.SQ Va,Vb,Vt BVI
F4,2E	SHFR.UQ Aa,Ab,At	-	SHFR.UQ Sa,Sb,St	-	SHFR.UQ Va,Vb,Vt BVI
F4,2F	SHFR.SQ Aa,Ab,At	-	SHFR.SQ Sa,Sb,St	SEL Va,Vb,Vt BVI	SHFR.SQ Va,Vb,Vt BVI
F4,30	ADD.UQ Aa,Ab,At	ADD.FS Sa,Sb,St	ADD.UQ Sa,Sb,St	-	ADD.UQ Va,Vb,Vt BVI
F4,31	ADD.SQ Aa,Ab,At	ADD.FD Sa,Sb,St	ADD.SQ Sa,Sb,St	-	ADD.SQ Va,Vb,Vt BVI
F4,32	SUB.UQ Aa,Ab,At	SUB.FS Sa,Sb,St	SUB.UQ Sa,Sb,St	-	SUB.UQ Va,Vb,Vt BVI
F4,33	SUB.SQ Aa,Ab,At	SUB.FD Sa,Sb,St	SUB.SQ Sa,Sb,St	-	SUB.SQ Va,Vb,Vt BVI
F4,34	MUL.UQ Aa,Ab,At	MUL.FS Sa,Sb,St	MUL.UQ Sa,Sb,St	-	MUL.UQ Va,Vb,Vt BVI
F4,35	MUL.SQ Aa,Ab,At	MUL.FD Sa,Sb,St	MUL.SQ Sa,Sb,St	-	MUL.SQ Va,Vb,Vt BVI
F4,36	DIV.UQ Aa,Ab,At	DIV.FS Sa,Sb,St	DIV.UQ Sa,Sb,St	-	-
F4,37	DIV.SQ Aa,Ab,At	DIV.FD Sa,Sb,St	DIV.SQ Sa,Sb,St	-	-
F4,38	-	-	-	-	-
F4,39	-	-	-	-	-
F4,3A	CMP.UQ Aa,Ab,ACt	CMP.FS Sa,Sb,SCt	CMP.UQ Sa,Sb,SCt	-	EQ.UQ Va,Vb,VMt BVI
F4,3B	CMP.SQ Aa,Ab,ACt	CMP.FD Sa,Sb,SCt	CMP.SQ Sa,Sb,SCt	-	EQ.SQ Va,Vb,VMt BVI
F4,3C	-	-	-	-	LT.UQ Va,Vb,VMt BVI
F4,3D	-	-	-	-	LT.SQ Va,Vb,VMt BVI
F4,3E	-	-	-	-	GT.UQ Va,Vb,VMt BVI
F4,3F	-	-	-	-	GT.SQ Va,Vb,VMt BVI

Format	*A Instruction*	*S Instruction (if=0)*	*S Instruction (if=1)*	*AE Instruction (if=0)*	*AE Instruction (if=1)*
F4,40	-	-	-	MOV.AEx Sa,Ab,AEG ASP	MOV Sa,Ab,AEG ASP
F4,41	-	-	-	-	-
F4,42	-	-	-	-	-
F4,43	-	-	-	-	-
F4,44	-	-	-	-	-
F4,45	-	-	-	-	-
F4,46	-	-	-	-	-
F4,47	-	-	-	-	-
F4,48	-	-	-	-	-
F4,49	-	-	-	-	-
F4,4A	-	-	-	-	-
F4,4B	-	-	-	-	-
F4,4C	-	-	-	-	-
F4,4D	-	-	-	-	-
F4,4E	-	-	-	-	-
F4,4F	-	-	-	-	-
F4,50	-	-	-	-	MOV.FS Sa,Ab,Vt BVI
F4,51	-	-	-	-	MOV Sa,Ab,Vt BVI
F4,52	-	-	-	-	-
F4,53	-	-	-	-	-
F4,54	-	-	-	-	-
F4,55	-	-	-	-	-
F4,56	-	-	-	-	-
F4,57	-	-	-	-	-
F4,58	-	-	-	-	-
F4,59	-	-	-	-	-
F4,5A	-	-	-	-	-
F4,5B	-	-	-	-	-
F4,5C	-	-	-	-	-
F4,5D	-	-	-	-	-
F4,5E	-	-	-	-	-
F4,5F	-	-	-	-	-
F4,60	-	-	-	-	MOV.FS Va,Ab,St BVI
F4,61	-	-	-	-	MOV Va,Ab,St BVI
F4,62	-	-	-	-	MOVR.FS Va,Ab,St BVI
F4,63	-	-	-	-	MOVR Va,Ab,St BVI
F4,64	-	-	-	-	-
F4,65	-	-	-	-	-
F4,66	-	-	-	-	-
F4,67	-	-	-	-	-
F4,68	-	-	-	MOV.AEx AEG,Ab,St ASP	MOV AEG,Ab,St ASP
F4,69	-	-	-	-	-
F4,6A	-	-	-	-	-
F4,6B	-	-	-	-	-
F4,6C	-	-	-	-	-
F4,6D	-	-	-	-	-
F4,6E	-	-	-	-	-
F4,6F	-	-	-	-	-
F4,70	-	-	-	MOV.AEx AEG,Imm,St ASP	MOV AEG,Imm,St ASP
F4,71	-	-	-	-	-
F4,72	-	-	-	-	-
F4,73	-	-	-	-	-
F4,74	-	-	-	-	-
F4,75	-	-	-	-	-
F4,76	-	-	-	-	-
F4,77	-	-	-	-	-
F4,78	-	-	-	-	-
F4,79	-	-	-	-	-
F4,7A	-	-	-	-	-
F4,7B	-	-	-	-	-
F4,7C	-	-	-	-	-
F4,7D	-	-	-	-	-
F4,7E	-	-	-	-	-
F4,7F	-	-	-	-	-

Format F5

Format	*A Instruction*	*S Instruction (if=0)*	*S Instruction (if=1)*	*AE Instruction (if=0)*	*AE Instruction (if=1)*
F5,00	-	-	-	-	VIDX Aa,SH,Vt BVI
F5,01	-	-	-	-	-
F5,02	-	-	-	-	-
F5,03	-	-	-	-	-
F5,04	-	-	-	-	-
F5,05	-	-	-	-	-
F5,06	-	-	-	-	-
F5,07	-	-	-	-	-
F5,08	-	-	-	-	-
F5,09	-	-	-	-	-
F5,0A	-	-	-	-	-
F5,0B	-	-	-	-	-
F5,0C	-	-	-	-	-
F5,0D	-	-	-	-	-
F5,0E	-	-	-	-	-
F5,0F	-	-	-	-	-
F5,10	MOV Aa,PIP	-	-	-	MOV Aa,VPM BVI
F5,11	MOV Aa,CPC	-	-	-	MOV Aa,VL BVI
F5,12	MOV Aa,CPS	-	-	-	MOV Aa,VS BVI
F5,13	MOV Aa,Imm,CRSL	-	-	-	MOV Aa,VPL BVI
F5,14	MOV Aa,Imm,CRSU	-	-	-	MOV Aa,VPS BVI
F5,15	MOV Aa,WB	-	-	Reserved for AEUIE	Reserved WB → AE
F5,16	-	-	-	MOV.AEx Aa,AEC ASP	MOV Aa,AEC All
F5,17	-	-	-	MOV.AEx Aa,AES ASP	MOV Aa,AES All
F5,18	MOV Aa,CCX	-	-	MOV.AEx Aa,Imm,AEG ASP	MOV Aa,Imm,AEG ASP
F5,19	-	-	-	-	-
F5,1A	-	-	-	-	MOV Aa,VPA BVI
F5,1B	-	-	-	-	-
F5,1C	-	-	-	-	MOV Aa,RIM ASP
F5,1D	-	-	-	-	MOV Aa,WIM ASP
F5,1E	-	-	-	-	MOV Aa,CIM ASP
F5,1F	-	-	-	-	-
F5,20	-	-	-	MOV.AEx Sa,Imm,AEG ASP	MOV Sa,Imm,AEG ASP
F5,21	-	-	-	-	-
F5,22	-	-	-	-	-
F5,23	-	-	-	-	-
F5,24	-	-	-	-	-
F5,25	-	-	-	-	-
F5,26	-	-	-	-	-
F5,27	-	-	-	-	-
F5,28	-	-	-	-	-
F5,29	-	-	-	-	-
F5,2A	-	-	-	-	-
F5,2B	-	-	-	-	-
F5,2C	-	-	-	-	-
F5,2D	-	-	-	-	-
F5,2E	-	-	-	-	-
F5,2F	-	-	-	-	-
F5,30	-	-	-	-	-
F5,31	-	-	-	-	-
F5,32	-	-	-	-	-
F5,33	-	-	-	-	-
F5,34	-	-	-	-	-
F5,35	-	-	-	-	-
F5,36	-	-	-	-	-
F5,37	-	-	-	-	-
F5,38	-	-	-	-	-
F5,39	-	-	-	-	-
F5,3A	-	-	-	-	-
F5,3B	-	-	-	-	-
F5,3C	-	-	-	-	-
F5,3D	-	-	-	-	-
F5,3E	-	-	-	-	-
F5,3F	-	-	-	-	-

Format F6

Format	A Instruction	S Instruction (if=0)	S Instruction (if=1)	AE Instruction (if=0)	AE Instruction (if=1)
F6,00	-	-	-	-	-
F6,01	-	-	-	-	-
F6,02	-	-	-	-	-
F6,03	-	-	-	-	-
F6,04	-	-	-	-	-
F6,05	-	-	-	-	-
F6,06	-	-	-	-	-
F6,07	-	-	-	-	-
F6,08	-	-	-	-	-
F6,09	-	-	-	-	-
F6,0A	-	-	-	-	-
F6,0B	-	-	-	-	-
F6,0C	-	-	-	-	-
F6,0D	-	-	-	-	-
F6,0E	-	-	-	-	TZC VMa,At BVI
F6,0F	-	-	-	-	PLC VMa,At BVI
F6,10	MOV PIP,At	-	-	-	MOV VPM,At BVI
F6,11	MOV CPC,At	-	-	-	MOV VL,At BVI
F6,12	MOV CPS,At	-	-	-	MOV VS,At BVI
F6,13	MOV CRSL,Imm,At	-	-	-	MOV VPL,At BVI
F6,14	MOV CRSU,Imm,At	-	-	-	MOV VPS,At BVI
F6,15	MOV WB,At	-	-	-	-
F6,16	MOV CIT,At	-	-	MOV.AEx AEC,At ASP	MOV AEC,At All
F6,17	MOV CDS,At	-	-	MOV.AEx AES,At ASP	MOV AES,At All
F6,18	MOV CCX,At	-	-	-	MOV VLmax,At BVI
F6,19	-	-	-	-	MOV VPLmax,At BVI
F6,1A	-	-	-	-	MOV VPA,At BVI
F6,1B	-	-	-	-	MOV REDcnt,At BVI
F6,1C	MOV Acnt,At	-	-	MOV.AEx AEG,Imm,At ASP	MOV AEG,Imm,At ASP
F6,1D	MOV Scnt,At	-	-	MOV.AEx AEGcnt,At ASP	MOV Vcnt,At BVI
F6,1E	MOV CRScnt,At	-	-	-	MOV VMcnt,At BVI
F6,1F	-	-	-	-	-
F6,20	-	-	-	-	-
F6,21	-	-	-	-	-
F6,22	-	-	-	-	-
F6,23	-	-	-	-	-
F6,24	-	-	-	-	-
F6,25	-	-	-	-	-
F6,26	-	-	-	-	-
F6,27	-	-	-	-	-
F6,28	-	-	-	-	-
F6,29	-	-	-	-	-
F6,2A	-	-	-	-	-
F6,2B	-	-	-	-	-
F6,2C	-	-	-	-	-
F6,2D	-	-	-	-	-
F6,2E	-	-	-	-	-
F6,2F	-	-	-	-	-
F6,30	-	-	-	-	-
F6,31	-	-	-	-	-
F6,32	-	-	-	-	-
F6,33	-	-	-	-	-
F6,34	-	-	-	-	-
F6,35	-	-	-	-	-
F6,36	-	-	-	-	-
F6,37	-	-	-	-	-
F6,38	-	-	-	-	-
F6,39	-	-	-	-	-
F6,3A	-	-	-	-	-
F6,3B	-	-	-	-	-
F6,3C	-	-	-	-	-
F6,3D	-	-	-	-	-
F6,3E	-	-	-	-	-
F6,3F	-	-	-	-	-

Format F7

Format	*A Instruction*		*S Instruction (if=0)*	*S Instruction (if=1)*	*AE Instruction (if=0)*			*AE Instruction (if=1)*		
F7,00	WBINC	A,S,V,VM	-	-	-			-		
F7,01	WBINC.P	A,S,V,VM	-	-	-			-		
F7,02	WBDEC	A,S,V,VM	-	-	-			-		
F7,03	WBSET	A,S,V,VM	-	-	-			-		
F7,04	-		-	-	-			-		
F7,05	-		-	-	-			-		
F7,06	-		-	-	-			-		
F7,07	-		-	-	-			-		
F7,08	VRRINC		-	-	-			-		
F7,09	VRRSET	B,S,O	-	-	-			-		
F7,0A	-		-	-	-			-		
F7,0B	-		-	-	-			-		
F7,0C	-		-	-	-			-		
F7,0D	-		-	-	-			-		
F7,0E	-		-	-	-			-		
F7,0F	FENCE		-	-	Rsvd for AEUIE			Rsvd	Fence→AE	
F7,10	-		-	-	-			MOV	Imm,VPM	BVI
F7,11	-		-	-	-			MOV	Imm,VL	BVI
F7,12	-		-	-	-			MOV	Imm,VS	BVI
F7,13	-		-	-	-			MOV	Imm,VPL	BVI
F7,14	-		-	-	-			MOV	Imm,VPS	BVI
F7,15	MOV	Imm,VMA	-	-	-			-		
F7,16	-		-	-	-			-		
F7,17	-		-	-	-			-		
F7,18	-		-	-	-			-		
F7,19	-		-	-	-			-		
F7,1A	-		-	-	-			MOV	Imm,VPA	BVI
F7,1B	-		-	-	-			-		
F7,1C	-		-	-	-			MOV	Imm,RIM	ASP
F7,1D	-		-	-	-			MOV	Imm,WIM	ASP
F7,1E	-		-	-	-			MOV	Imm,CIM	ASP
F7,1F	-		-	-	-			-		
F7,20	-		-	-	CAEP00.AEx	Imm18	ASP	CAEP00	Imm18	ASP
F7,21	-		-	-	CAEP01.AEx	Imm18	ASP	CAEP01	Imm18	ASP
F7,22	-		-	-	CAEP02.AEx	Imm18	ASP	CAEP02	Imm18	ASP
F7,23	-		-	-	CAEP03.AEx	Imm18	ASP	CAEP03	Imm18	ASP
F7,24	-		-	-	CAEP04.AEx	Imm18	ASP	CAEP04	Imm18	ASP
F7,25	-		-	-	CAEP05.AEx	Imm18	ASP	CAEP05	Imm18	ASP
F7,26	-		-	-	CAEP06.AEx	Imm18	ASP	CAEP06	Imm18	ASP
F7,27	-		-	-	CAEP07.AEx	Imm18	ASP	CAEP07	Imm18	ASP
F7,28	-		-	-	CAEP08.AEx	Imm18	ASP	CAEP08	Imm18	ASP
F7,29	-		-	-	CAEP09.AEx	Imm18	ASP	CAEP09	Imm18	ASP
F7,2A	-		-	-	CAEP0A.AEx	Imm18	ASP	CAEP0A	Imm18	ASP
F7,2B	-		-	-	CAEP0B.AEx	Imm18	ASP	CAEP0B	Imm18	ASP
F7,2C	-		-	-	CAEP0C.AEx	Imm18	ASP	CAEP0C	Imm18	ASP
F7,2D	-		-	-	CAEP0D.AEx	Imm18	ASP	CAEP0D	Imm18	ASP
F7,2E	-		-	-	CAEP0E.AEx	Imm18	ASP	CAEP0E	Imm18	ASP
F7,2F	-		-	-	CAEP0F.AEx	Imm18	ASP	CAEP0F	Imm18	ASP
F7,30	-		-	-	CAEP10.AEx	Imm18	ASP	CAEP10	Imm18	ASP
F7,31	-		-	-	CAEP11.AEx	Imm18	ASP	CAEP11	Imm18	ASP
F7,32	-		-	-	CAEP12.AEx	Imm18	ASP	CAEP12	Imm18	ASP
F7,33	-		-	-	CAEP13.AEx	Imm18	ASP	CAEP13	Imm18	ASP
F7,34	-		-	-	CAEP14.AEx	Imm18	ASP	CAEP14	Imm18	ASP
F7,35	-		-	-	CAEP15.AEx	Imm18	ASP	CAEP15	Imm18	ASP
F7,36	-		-	-	CAEP16.AEx	Imm18	ASP	CAEP16	Imm18	ASP
F7,37	-		-	-	CAEP17.AEx	Imm18	ASP	CAEP17	Imm18	ASP
F7,38	-		-	-	CAEP18.AEx	Imm18	ASP	CAEP18	Imm18	ASP
F7,39	-		-	-	CAEP19.AEx	Imm18	ASP	CAEP19	Imm18	ASP
F7,3A	-		-	-	CAEP1A.AEx	Imm18	ASP	CAEP1A	Imm18	ASP
F7,3B	-		-	-	CAEP1B.AEx	Imm18	ASP	CAEP1B	Imm18	ASP
F7,3C	-		-	-	CAEP1C.AEx	Imm18	ASP	CAEP1C	Imm18	ASP
F7,3D	-		-	-	CAEP1D.AEx	Imm18	ASP	CAEP1D	Imm18	ASP
F7,3E	-		-	-	CAEP1E.AEx	Imm18	ASP	CAEP1E	Imm18	ASP
F7,3F	-		-	-	CAEP1F.AEx	Imm18	ASP	CAEP1F	Imm18	ASP

C Instruction Set Reference

Appendix C presents a detailed description of each instruction. Section C.1 describes the language used to describe the operation performed by an instruction. Section C.2 lists all pseudo code functions referenced by the instruction description pages. Section C.3 list all defined instructions for the Convey Coprocessor.

C.1 Instruction Pseudo Code Notation

The instruction descriptions include pseudo code to precisely describe each instruction's functionality. The pseudo code is based on the C++ programming language with a few additional operators to specify bit ranges. The extended operators include:

Operator	Description
<x>	Bit selector operator where 'x' indicates the index of the desired bit
<x:y>	Bit range selector operator where 'x' specifies the high bit of the range, and 'y' specifies the low bit

C.2 Instruction Pseudo Code Functions

C.2.1 Abs(a)

// return the absolute value of the input argument

Abs(a) { return (a < 0) ? –a : a; }

C.2.2 AregIdx(rt)

// Access A-register set

AregIdx(rt) { return (rx < 8) ? rx : (rx + AWB); }

C.2.3 Converts

// Convert from one data type to another

CvtFdFs(rt) { return converted value from FD to FS }

CvtFdSq(rt) { return converted value from FD to SQ }

CvtFsFd(rt) { return converted value from FS to FD }

CvtFsSq(rt) { return converted value from FS to SQ }

CvtSqFd(rt) { return converted value from SQ to FD }

CvtSqFs(rt) { return converted value from SQ to FS }

CvtSqUq(rt) { return converted value from SQ to UQ }

CvtUqSq(rt) { return converted value from UQ to SQ }

C.2.4 ExecuteCustomInstruction()

```
ExecuteUserDefinedInstruction() {
        // Perform the operation of the user defined instruction
}
```

C.2.5 InstLength()

```
// return the size in bytes of the current instruction bundle
InstLength() { return IsExtImmed() ? 16 : 8; }
```

C.2.6 InterruptHostProcessor()

```
InterruptHostProcessor() {
        // Send an interrupt to the host processor
}
```

C.2.7 Max(a)

```
// return the maximum of two input arguments
Max(a, b) { return (a > b) ? a : b; }
```

C.2.8 MaximumFloatDoubleValue()

```
// return the maximum Float Double value
MaximumFloatDoubleValue() { return 0x7fef ffff ffff ffff; }
```

C.2.9 MaximumFloatSingleValue()

```
// return the maximum Float Single value
MaximumFloatSingleValue() { return 0x7f7f ffff; }
```

C.2.10 MaximumSignedQuadValue()

```
// return the maximum Signed Quad word value
MaximumSignedQuadValue() { return 0x7fff ffff ffff ffff; }
```

C.2.11 MemFence()

```
MemFence() {
        // perform a memory fence operation to enforce memory request ordering
}
```

C.2.12 MemLoad(effa, width)

```
MemLoad(effa, width) {
        // read data from memory at address ‘effa’ and size ‘width’
}
```

C.2.13 MemStore(effa, width, data)

```
MemStore(effa, width, data) {
    // store 'data' to memory at address 'effa' and size 'width'
}
```

C.2.14 MinimumFloatDoubleValue()

```
// return the minimum Float Double value
MinimumFloatDoubleValue() { return 0xffef ffff ffff ffff; }
```

C.2.15 MinimumFloatSingleValue()

```
// return the minimum Float Single value
MinimumFloatSingleValue() { return 0xff7f ffff; }
```

C.2.16 MinimumSignedQuadValue()

```
// return the minimum Signed Quad word value
MinimumSignedQuadValue() { return 0x8000 0000 0000 0000; }
```

C.2.17 Min(a)

```
// return the minimum of two input arguments
Min(a, b) { return (a < b) ? a : b; }
```

C.2.18 Nop()

```
// Do nothing
Nop() { }
```

C.2.19 Offset(shftAmt)

```
// return the offset value for a load or store instruction
Offset(shftAmt) {
    return IsExtImmed() ? ExtImmed64 : Sext64(Immed10 << shftAmt, 10 + shftAmt);
}
```

C.2.20 PopulationCount(value)

```
PopulationCount(value) {
    // return the number of one bits in the input value
 }
```

C.2.21 RecipricalInputAdjust(FDvalue)

```
RecipricalInputAdjust(FDvalue) {
    // The input value is adjusted to produce the expected result when needed
    // otherwise the input is return unmodified.  Cases where the input value must
```

```
        // be modified are when the input value is zero, infinite, and NaN.
}
```

C.2.22 RecipricalTableLookup(FDvalue)

```
RecipricalTableLookup(FDvalue) {
        // The starting value for a Newton Raphson iterative approximation approach
}
```

C.2.23 RecipricalTableLookup(FSvalue)

```
RecipricalTableLookup(FSvalue) {
        // The result is a 64-bit value where the upper 32-bits is the input adjusted value
        // and the lower 32-bits is the starting value for a Newton Raphson iterative
        // approximation.
}
```

C.2.24 ReductionMapFunction(index)

```
ReductionMapFunction(index) {
        // return the vector register element for the partial reduction result
        //    identified by index
}
```

C.2.25 Sext64(value, width)

```
Sext64(value, width) {
        // sign extend the input value from 'width' bits to 64-bits
}
```

C.2.26 Simmed10()

```
// return an immediate value selecting either the instructions immed10 field or the
//    extended immediate
Simmed10() { return IsExtImmed() ? ExtImmed64 : Sext64(Immed10, 10); }
```

C.2.27 Simmed15()

```
// return an immediate value selecting either the instructions immed15 field or the
//    extended immediate
Simmed15() { return IsExtImmed() ? ExtImmed64 : Sext64(Immed15 << 3, 18); }
```

C.2.28 SovflChk(value, width)

```
SovflChk(value, width) {
        // range check 'value' to fit within a signed integer of 'width' bits
}
```

C.2.29 SregIdx(rx)

```
// Access S-register set
SregIdx(rt) { return (rx < 1) ? rx : (rx + SWB); }
```

C.2.30 Sqrt(value)

```
Sqrt(value) {
    // return the IEEE compliant square root of the input value
}
```

C.2.31 SquareRootInputAdjust(FDvalue)

```
SquareRootInputAdjust(FDvalue) {
    // The input value is adjusted to produce the expected result when needed
    // otherwise the input is return unmodified.  Cases where the input value must
    // be modified are when the input value is zero, infinite, and NaN.
}
```

C.2.32 SquareRootTableLookup(FDvalue)

```
SquareRootTableLookup(FDvalue) {
    // The starting value for a Newton Raphson iterative approximation approach
}
```

C.2.33 SquareRootTableLookup(FSvalue)

```
SquareRootTableLookup(FSvalue) {
    // The result is a 64-bit value where the upper 32-bits is the input adjusted value
    // and the lower 32-bits is the starting value for a Newton Raphson iterative
    // approximation.
}
```

C.2.34 StopExecution()

```
StopExecution() {
    // Stop coprocessor instruction execution
}
```

C.2.35 Uimmed10()

```
// return an immediate value selecting either the instructions immed10 field or the
//     extended immediate
Uimmed10() { return IsExtImmed() ? ExtImmed64 : Zext64(Immed10, 10); }
```

C.2.36 Uimmed6()

```
// return an immediate value selecting either the instructions immed6 field or the
//     extended immediate
Uimmed6() { return IsExtImmed() ? ExtImmed64 : Zext64(Immed6, 6); }
```

C.2.37 UndefinedValue()

```
UndefinedValue() {
        // The result of the operation is undefined.
}
```

C.2.38 UpdateUserModeStatus()

```
UpdateUserModeStatus() {
        // Update the status to reflect the completion of the current dispatch
}
```

C.2.39 VMregIdx(rx)

```
// Access VM-register set
VMregIdx(rx) { return rx + WB.VMWB; }
```

C.2.40 VregIdx(rx)

```
// Access V-register set
VregIdx(rx) {
        If (rx  >= WB.VRRB && rx < WB.VRRB + WB.VRRS)      // within rotation range
            return WB.VWB + WB.VRRB + (rx – WB.VRRB + WB.VRRO) % WB.VRRS;
        else
            return WB.VWB + rx;
}
```

C.2.41 Zext64(value, width)

```
// Zero extend the input value with ‘width’ bits to 64-bits
Zext64(value, width) { return ‘value’ with ‘width’ bits zero extended to 64-bits; }
```

C.2.42 ZovflChk(value, width)

```
ZovflChk(value, width) {
        // range check ‘value’ to fit within an unsigned integer of ‘width’ bits
 }
```

C.3 Alphabetized list of Instructions

ABS – Scalar Absolute Value Float Single

ABS.FS Sa,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	0	1100	opc	000000	Sa	St

Description

The instruction performs the absolute value operation on the source scalar register using single precision floating point data format. The value of the operation is written to the destination scalar register.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx]<63:32> = 0;
S[stx]<31:0> = Abs( S[sax]<31:0> );
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
ABS.FS	Sa,St	Scalar	S	F3,0,0A

Exceptions

Scalar Float Invalid Operand (SFIE) Scalar Register Range (SRRE)

ABS – Scalar Absolute Value Float Double

ABS.FD Sa,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	**0**	**1100**	**opc**	**000000**	**Sa**	**St**

Description

The instruction performs the absolute value operation on the source scalar register using double precision floating point data format. The value of the operation is written to the destination scalar register.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx] = Abs( S[sax] );
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
ABS.FD	Sa,St	Scalar	S	F3,0,0B

Exceptions

Scalar Float Invalid Operand (SFIE) Scalar Register Range (SRRE)

ABS – Address Absolute Value Signed Quad Word

ABS.SQ Aa,At

31 29	28 25	24 18	17 12	11 6	5 0
it	1100	opc	000000	Aa	At

Description

The instruction performs the absolute value operation on the source scalar register using signed quad word integer data format. The value of the operation is written to the destination scalar register.

Pseudo Code

```
aax = AregIdx(Aa);
atx = AregIdx(At);
A[atx] = Abs( A[aax] );
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*
ABS.SQ	Aa,At	Scalar	A	F3, 0B

Exceptions

Scalar Register Range (SRRE)

ABS – Scalar Absolute Value Signed Quad Word

ABS.SQ Sa,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	**1**	**1100**	**opc**	**000000**	**Sa**	**St**

Description

The instruction performs the absolute value operation on the source scalar register using signed quad word integer data format. The value of the operation is written to the destination scalar register.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx] = Abs( S[sax] );
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
ABS.SQ	Sa,St	Scalar	S	F3,1,0B

Exceptions

Scalar Register Range (SRRE) Scalar Integer Overflow (SIOE)

ABS – Vector Absolute Value Signed Quad Word

ABS.SQ Va,Vt
ABS.SQ.T Va,Vt
ABS.SQ.F Va,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	**0**	**1100**	**opc**	**000000**	**Va**	**Vt**

Description

The instruction performs the absolute value operation on the input vector register using signed quad word data format. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of the operation is written to the destination vector register. The masked element results are undefined and exceptions are not signaled.

Pseudo Code

```
vax = VregIdx(Va);
vtx = VregIdx(Vt);
vmax = VMregIdx(VMA);
for (j = 0; j < VPL; j += 1) {
    for (i = 0; i < VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j]. V[vtx][i] = Abs( VP[j].V[vax][i] );
        else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
ABS.SQ	Va,Vt	BVI	AE	F3,1,0B	0
ABS.SQ.T	Va,Vt	BVI	AE	F3,1,0B	2
ABS.SQ.F	Va,Vt	BVI	AE	F3,1,0B	3

Exceptions

AE Register Range (AERRE)   AE Integer Overflow (AEIOE)

ADD – Address Addition Integer with Immediate

ADD.UQ	Aa,Immed,At
ADD.SQ	Aa,Immed,At

31 29	28	27 22	21 12	11 6	5 0
it	0	opc	immed10	Aa	At

Description

The instruction performs addition between an address register and an immediate value using unsigned or signed integer arithmetic. The immediate value is either the zero or signed extended 10-bit Immed10 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code (example for ADD.SQ)

```
aax = AregIdx(Aa);
atx = AregIdx(At);
A[atx] = A[aax] + Aimmed10();
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
ADD.UQ	Aa,Immed,At	Scalar	A	F1,30
ADD.SQ	Aa,Immed,At	Scalar	A	F1,31

Exceptions

Scalar Register Range (SRRE) Scalar Integer Overflow (SIOE)

ADD – Scalar Addition Integer with Immediate

ADD.UQ	Sa,Immed,St
ADD.SQ	Sa,Immed,St

31 30	29	28	27 22	21 12	11 6	5 0
it	1	0	opc	immed10	Sa	St

Description

The instruction performs addition between a scalar register and an immediate value using unsigned or signed integer arithmetic. The immediate value is either the zero or signed extended 10-bit Immed10 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code (example for ADD.SQ)

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx] = S[sax] + Simmed10();
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
ADD.UQ	Sa,Immed,St	Scalar	S	F1,1,30
ADD.SQ	Sa,Immed,St	Scalar	S	F1,1,31

Exceptions

Scalar Register Range (SRRE)	Scalar Integer Overflow (SIOE)

ADD – Scalar Addition Float Double with Immediate

ADD.FD Sa,Immed,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	**0**	**1100**	**opc**	**Immed6**	**Sa**	**St**

Description

The instruction performs addition between a scalar register and an immediate value using float double arithmetic. The immediate value is either the zero extended 6-bit Immed6 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx] = S[sax] + Uimmed6();
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
ADD.FD	Sa,Immed,St	Scalar	S	F3,0,31

Exceptions

Scalar Register Range (SRRE)	Scalar Float Invalid Operand (SFIE)
Scalar Float Overflow (SFOE)	Scalar Float Underflow (SFUE)

ADD – Scalar Addition Float Single with Immediate

ADD.FS Sa,Immed,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	0	1100	opc	Immed6	Sa	St

Description

The instruction performs addition between a scalar register and an immediate value using float single arithmetic. The immediate value is either the zero extended 6-bit Immed10 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx]<63:0> = 0;
S[stx]<31:0> = S[sax]<31:0> + Uimmed6()<31:0>;
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
ADD.FS	Sa,Immed,St	Scalar	S	F3,0,30

Exceptions

Scalar Register Range (SRRE)	Scalar Float Invalid Operand (SFIE)
Scalar Float Overflow (SFOE)	Scalar Float Underflow (SFUE)

ADD – Address Addition Integer

ADD.UQ	Aa,Ab,At
ADD.SQ	Aa,Ab,At

31 29	28 25	24 18	17 12	11 6	5 0
it	1101	opc	Ab	Aa	At

Description

The instruction performs addition on two address register values using signed or unsigned integer arithmetic.

Pseudo Code

```
aax = AregIdx(Aa);
abx = AregIdx(Ab);
atx = AregIdx(At);
A[atx] = A[aax] + A[abx];
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
ADD.UQ	Aa,Ab,At	Scalar	A	F4,30
ADD.SQ	Aa,Ab,At	Scalar	A	F4,31

Exceptions

Scalar Register Range (SRRE)

Scalar Integer Overflow (SIOE)

ADD – Scalar Addition Integer

ADD.UQ	Sa,Sb,St
ADD.SQ	Sa,Sb,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	**1**	**1101**	**opc**	**Sb**	**Sa**	**St**

Description

The instruction performs addition on two scalar register values using signed or unsigned integer arithmetic.

Pseudo Code

```
sax = SregIdx(Sa);
sbx = SregIdx(Sb);
stx = SregIdx(St);
S[stx] = S[sax] + S[sbx];
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
ADD.UQ	Sa,Sb,St	Scalar	S	F4,1,30
ADD.SQ	Sa,Sb,St	Scalar	S	F4,1,31

Exceptions

Scalar Register Range (SRRE)

Scalar Integer Overflow (SIOE)

ADD – Scalar Addition Float Double

ADD.FD Sa,Sb,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	0	1101	opc	Sb	Sa	St

Description

The instruction performs addition on two scalar register values using float double arithmetic.

Pseudo Code

```
sax = SregIdx(Sa);
sbx = SregIdx(Sb);
stx = SregIdx(St);
S[stx] = S[sax] + S[sbx];
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
ADD.FD	Sa,Sb,St	Scalar	S	F4,0,31

Exceptions

Scalar Register Range (SRRE)	Scalar Float Invalid Operand (SFIE)
Scalar Float Overflow (SFOE)	Scalar Float Underflow (SFUE)

ADD – Scalar Addition Float Single

ADD.FS Sa,Sb,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	0	1101	opc	Sb	Sa	St

Description

The instruction performs addition on two scalar register values using float single arithmetic.

Pseudo Code

```
sax = SregIdx(Sa);
sbx = SregIdx(Sb);
stx = SregIdx(St);
S[stx]<63:32> = 0;
S[stx]<31:0> = S[sax]<31:0> + S[sbx]<31:0>;
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
ADD.FS	Sa,Sb,St	Scalar	S	F4,0,30

Exceptions

Scalar Register Range (SRRE)	Scalar Float Invalid Operand (SFIE)
Scalar Float Overflow (SFOE)	Scalar Float Underflow (SFUE)

ADD – Vector Add Signed Quad Word Scalar

ADD.SQ	Va,Sb,Vt
ADD.SQ.T	Va,Sb,Vt
ADD.SQ.F	Va,Sb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	if	1100	opc	Sb	Va	Vt

Description

The instruction adds a 64-bit vector register to a 64-bit scalar value using signed quad word integer data format. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of the output vector register for masked elements is undefined and exceptions are not recorded for masked elements.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j]. V[vtx] [i] = VP[j]. V [vax][i] + S[sbx];
        else
            VP[j]. V[vtx] [i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
ADD.SQ	Va,Sb,Vt	BVI	AE	F3,1,31	0
ADD.SQ.T	Va,Sb,Vt	BVI	AE	F3,1,31	2
ADD.SQ.F	Va,Sb,Vt	BVI	AE	F3,1,31	3

Exceptions

AE Integer Overflow (AEIOE)

AE Register Range (AERRE)

Scalar Register Range (SRRE)

ADD – Vector Add Signed Quad Word

ADD.SQ	Va,Vb,Vt
ADD.SQ.T	Va,Vb,Vt
ADD.SQ.F	Va,Vb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1101	opc	Vb	Va	Vt

Description

The instruction adds two 64-bit vector register using signed quad word integer data format. The 'vm' field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of the output vector register for masked elements is undefined and exceptions are not recorded for masked elements.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j].V[vtx][i] = VP[j]. V[vax][i] + VP[j].V[vbx][i];
        else
            VP[j].V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
ADD.SQ	Va,Vb,Vt	BVI	AE	F4,1,31	0
ADD.SQ.T	Va,Vb,Vt	BVI	AE	F4,1,31	2
ADD.SQ.F	Va,Vb,Vt	BVI	AE	F4,1,31	3

Exceptions

AE Integer Overflow (AEIOE) AE Register Range (AERRE)

ADD – Vector Add Unsigned Quad Word Scalar

ADD.UQ	Va,Sb,Vt
ADD.UQ.T	Va,Sb,Vt
ADD.UQ.F	Va,Sb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	**if**	**1100**	**opc**	**Sb**	**Va**	**Vt**

Description

The instruction adds a 64-bit vector register to a 64-bit scalar value using unsigned quad word integer data format. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of the output vector register for masked elements is undefined and exceptions are not recorded for masked elements.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j]. V[vtx] [i] = VP[j]. V [vax][i] + S[sbx];
        else
            VP[j]. V[vtx] [i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
ADD.UQ	Va,Sb,Vt	BVI	AE	F3,1,30	0
ADD.UQ.T	Va,Sb,Vt	BVI	AE	F3,1,30	2
ADD.UQ.F	Va,Sb,Vt	BVI	AE	F3,1,30	3

Exceptions

AE Integer Overflow (AEIOE)	AE Register Range (AERRE)
Scalar Register Range (SRRE)	

ADD – Vector Add Unsigned Quad Word

ADD.UQ	Va,Vb,Vt
ADD.UQ.T	Va,Vb,Vt
ADD.UQ.F	Va,Vb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	**1**	**1101**	**opc**	**Vb**	**Va**	**Vt**

Description

The instruction adds two 64-bit vector register using unsigned quad word integer data format. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of the output vector register for masked elements is undefined and exceptions are not recorded for masked elements.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j].V[vtx][i] = VP[j]. V[vax][i] + VP[j].V[vbx][i];
        else
            VP[j].V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
ADD.UQ	Va,Vb,Vt	BVI	AE	F4,1,30	0
ADD.UQ.T	Va,Vb,Vt	BVI	AE	F4,1,30	2
ADD.UQ.F	Va,Vb,Vt	BVI	AE	F4,1,30	3

Exceptions

AE Integer Overflow (AEIOE)	AE Register Range (AERRE)

AND, OR, NAND, NOR, XOR, XNOR, ANDC, ORC – Address Logical w/Immed

AND	Aa,Immed,At	XOR	Aa,Immed,At
OR	Aa,Immed,At	XNOR	Aa,Immed,At
NAND	Aa,Immed,At	ANDC	Aa,Immed,At
NOR	Aa,Immed,At	ORC	Aa,Immed,At

31 29	28	27 22	21 12	11 6	5 0
it	0	opc	immed10	Aa	At

Description

Perform a logical operation between each element in an address register with an immediate value. The immediate value is either the zero extended 10-bit Immed field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code (AND instruction example)

```
aax = AregIdx(Aa);
atx = AregIdx(At);
A[atx] = A[aax] & Uimmed10();
```

Opc Field Encoding

Instruction		*ISA*	*Type*	*Encoding*	*Operation*
AND	Aa,Immed,At	Scalar	A	F1,20	Aa & Immed
OR	Aa,Immed,At	Scalar	A	F1,21	Aa | Immed
NAND	Aa,Immed,At	Scalar	A	F1,22	~(Aa & Immed)
NOR	Aa,Immed,At	Scalar	A	F1,23	~(Aa | Immed)
XOR	Aa,Immed,At	Scalar	A	F1,24	Aa ^ Immed
XNOR	Aa,Immed,At	Scalar	A	F1,25	~(Aa ^ Immed)
ANDC	Aa,Immed,At	Scalar	A	F1,26	~Aa & Immed
ORC	Aa,Immed,At	Scalar	A	F1,27	~Aa | Immed

Exceptions

Scalar Register Range (SRRE)

AND, OR, NAND, NOR, XOR, XNOR, ANDC, ORC – Address Logical

AND	Aa,Ab,At	XOR	Aa,Ab,At
OR	Aa,Ab,At	XNOR	Aa,Ab,At
NAND	Aa,Ab,At	ANDC	Aa,Ab,At
NOR	Aa,Ab,At	ORC	Aa,Ab,At

31 29	28 25	24 18	17 12	11 6	5 0
it	1100	opc	Ab	Aa	At

Description

Perform a logical operation between two address registers.

Pseudo Code (AND instruction example)

```
aax = AregIdx(Aa);
abx = AregIdx(Ab);
atx = AregIdx(At);
A[atx] = A[aax] & A[abx];
```

Opc Field Encoding

Instruction		***ISA***	***Type***	***Encoding***	***Operation***
AND	Aa,Ab,At	Scalar	A	F4,20	Aa & Ab
OR	Aa,Ab,At	Scalar	A	F4,21	Aa | Ab
NAND	Aa,Ab,At	Scalar	A	F4,22	~(Aa & Ab)
NOR	Aa,Ab,At	Scalar	A	F4,23	~(Aa | Ab)
XOR	Aa,Ab,At	Scalar	A	F4,24	Aa ^ Ab
XNOR	Aa,Ab,At	Scalar	A	F4,25	~(Aa ^ Ab)
ANDC	Aa,Ab,At	Scalar	A	F4,26	~Aa & Ab
ORC	Aa,Ab,At	Scalar	A	F4,27	~Aa | Ab

Exceptions

Scalar Register Range (SRRE)

AND, OR, NAND, NOR, XOR, XNOR, ANDC, ORC – Condition Code Logical

AND	CCa,CCb,CCt	XOR	CCa,CCb,CCt
OR	CCa,CCb,CCt	XNOR	CCa,CCb,CCt
NAND	CCa,CCb,CCt	ANDC	CCa,CCb,CCt
NOR	CCa,CCb,CCt	ORC	CCa,CCb,CCt

31 29	28 25	24 18	17 12	11 6	5 0
it	1101	opc	CCb	CCa	CCt

Description

Perform a logical operation between two condition code bits.

Pseudo Code (AND instruction example)

```
cca = CCa<4:0>;
ccb = CCb<4:0>;
cct = CCt<4:0>;
CPS.CC<cct> = CPS.CC<cca> & CPS.CC<ccb>;
```

Opc Field Encoding

Instruction		**ISA**	*Type*	*Encoding*	*Operation*
AND	CCa,CCb,CCt	Scalar	A	F4,10	CCa & CCb
OR	CCa,CCb,CCt	Scalar	A	F4,11	CCa | CCb
NAND	CCa,CCb,CCt	Scalar	A	F4,12	~(CCa & CCb)
NOR	CCa,CCb,CCt	Scalar	A	F4,13	~(CCa | CCb)
XOR	CCa,CCb,CCt	Scalar	A	F4,14	CCa ^ CCb
XNOR	CCa,CCb,CCt	Scalar	A	F4,15	~(CCa ^ CCb)
ANDC	CCa,CCb,CCt	Scalar	A	F4,16	~CCa & CCb
ORC	CCa,CCb,CCt	Scalar	A	F4,17	~ CCa | CCb

Exceptions

None

AND, OR, NAND, NOR, XOR, XNOR, ANDC, ORC – Scalar Logical w/Immed.

AND	Sa,Immed,St	XOR	Sa,Immed,St
OR	Sa,Immed,St	XNOR	Sa,Immed,St
NAND	Sa,Immed,St	ANDC	Sa,Immed,St
NOR	Sa,Immed,St	ORC	Sa,Immed,St

31 29	28	27 22	21 12	11 6	5 0
it	0	opc	immed10	Sa	St

Description

Perform a logical operation between a scalar register with an immediate value. The immediate value is either the zero extended 6-bit Immed6 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code (AND instruction example)

```
sax = SregIdx(Sa);
stx = SregIdx(At);
S[stx] = S[sax] & Uimmed6();
```

Opc Field Encoding

Instruction		***ISA***	***Type***	***Encoding***	***Operation***
AND	Sa,Immed,St	Scalar	S	F1,1,20	Sa & Immed
OR	Sa,Immed,St	Scalar	S	F1,1,21	Sa | Immed
NAND	Sa,Immed,St	Scalar	S	F1,1,22	~(Sa & Immed)
NOR	Sa,Immed,St	Scalar	S	F1,1,23	~(Sa | Immed)
XOR	Sa,Immed,St	Scalar	S	F1,1,24	Sa ^ Immed
XNOR	Sa,Immed,St	Scalar	S	F1,1,25	~(Sa ^ Immed)
ANDC	Sa,Immed,St	Scalar	S	F1,1,26	~Sa & Immed
ORC	Sa,Immed,St	Scalar	S	F1,1,27	~Sa | Immed

Exceptions

Scalar Register Range (SRRE)

AND, OR, NAND, NOR, XOR, XNOR, ANDC, ORC – Scalar Logical

AND	Sa,Sb,St	XOR	Sa,Sb,St
OR	Sa,Sb,St	XNOR	Sa,Sb,St
NAND	Sa,Sb,St	ANDC	Sa,Sb,St
NOR	Sa,Sb,St	ORC	Sa,Sb,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	if	1101	opc	Sb	Sa	St

Description

Perform a logical operation between two scalar registers.

Pseudo Code (AND instruction example)

```
sax = SregIdx(Sa);
sbx = SregIdx(Sb);
stx = SregIdx(St);
S[stx] = S[sax] & S[sbx];
```

Opc Field Encoding

Instruction		*ISA*	*Type*	*Encoding*	*Operation*
AND	Sa,Sb,St	Scalar	S	F4, 1, 20	Sa & Sb
OR	Sa,Sb,St	Scalar	S	F4, 1, 21	Sa | Sb
NAND	Sa,Sb,St	Scalar	S	F4, 1, 22	~(Sa & Sb)
NOR	Sa,Sb,St	Scalar	S	F4, 1, 23	~(Sa | Sb)
XOR	Sa,Sb,St	Scalar	S	F4, 1, 24	Sa ^ Sb
XNOR	Sa,Sb,St	Scalar	S	F4, 1, 25	~(Sa ^ Sb)
ANDC	Sa,Sb,St	Scalar	S	F4, 1, 26	~Sa & Sb
ORC	Sa,Sb,St	Scalar	S	F4, 1, 27	~Sa | Sb

Exceptions

Scalar Register Range (SRRE)

AND – Vector Logical And

AND	Va,Vb,Vt
AND.T	Va,Vb,Vt
AND.F	Va,Vb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	**if**	**1100**	**opc**	**Sb**	**Va**	**Vt**

Description

The instruction performs a logical ‘and’ operation between the two 64-bit source vector registers. The result of the operation is written to a 64-bit destination vector register. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of masked elements is undefined.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(VMA);
for (j = 0; j < VPL; j += 1) {
    for (i = 0; i < VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j].V[vtx][i] = VP[j].V[vax][i]  & VP[j].V[vbx][i];
        else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
AND	Va,Vb,Vt	BVI	AE	F4,1,20	0
AND.T	Va,Vb,Vt	BVI	AE	F4,1,20	2
AND.F	Va,Vb,Vt	BVI	AE	F4,1,20	3

Exceptions

AE Register Range (AERRE)

AND – Vector Logical And with Scalar

AND	Va,Sb,Vt
AND.T	Va,Sb,Vt
AND.F	Va,Sb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	**if**	**1100**	**opc**	**Sb**	**Va**	**Vt**

Description

The instruction performs a logical ‘and’ operation between a 64-bit source vector register and an S register. The result of the operation is written to a 64-bit destination vector register. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of masked elements is undefined.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vtx = VregIdx(Vt);
vmax = VMregIdx(VMA);
for (j = 0; j < VPL; j += 1) {
    for (i = 0; i < VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j].V[vtx][i] = VP[j].V[vax][i]  & S[sbx];
        else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
AND	Va,Sb,Vt	BVI	AE	F3,1,20	0
AND.T	Va,Sb,Vt	BVI	AE	F3,1,20	2
AND.F	Va,Sb,Vt	BVI	AE	F3,1,20	3

Exceptions

AE Register Range (AERRE)

Scalar Register Range (SRRE)

ANDC – Vector Logical And Compliment

ANDC	Va,Vb,Vt
ANDC.T	Va,Vb,Vt
ANDC.F	Va,Vb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	if	1100	opc	Sb	Va	Vt

Description

The instruction performs a logical ‘and’ compliment operation between the two 64-bit source vector registers. The result of the operation is written to a 64-bit destination vector register. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of masked elements is undefined.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(VMA);
for (j = 0; j < VPL; j += 1) {
    for (i = 0; i < VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j].V[vtx][i] = ~VP[j].V[vax][i]  & VP[j].V[vbx][i];
        else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
ANDC	Va,Vb,Vt	BVI	AE	F4,1,26	0
ANDC.T	Va,Vb,Vt	BVI	AE	F4,1,26	2
ANDC.F	Va,Vb,Vt	BVI	AE	F4,1,26	3

Exceptions

AE Register Range (AERRE)

ANDC – Vector Logical And Compliment with Scalar

ANDC	Va,Sb,Vt
ANDC.T	Va,Sb,Vt
ANDC.F	Va,Sb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	if	1100	opc	Sb	Va	Vt

Description

The instruction performs a logical ‘and’ compliment operation between a 64-bit source vector register and an S register. The result of the operation is written to a 64-bit destination vector register. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of masked elements is undefined.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vtx = VregIdx(Vt);
vmax = VMregIdx(VMA);
for (j = 0; j < VPL; j += 1) {
    for (i = 0; i < VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j].V[vtx][i] = ~VP[j].V[vax][i]  & S[sbx];
        else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
ANDC	Va,Sb,Vt	BVI	AE	F3,1,26	0
ANDC.T	Va,Sb,Vt	BVI	AE	F3,1,26	2
ANDC.F	Va,Sb,Vt	BVI	AE	F3,1,26	3

Exceptions

AE Register Range (AERRE)   Scalar Register Range (SRRE)

AND, OR, NAND, NOR, XOR, XNOR, ANDC, ORC – Vector Mask Logical

AND	VMa,VMb,VMt	XOR	VMa,VMb,VMt
OR	VMa,VMb,VMt	XNOR	VMa,VMb,VMt
NAND	VMa,VMb,VMt	ANDC	VMa,VMb,VMt
NOR	VMa,VMb,VMt	ORC	VMa,VMb,VMt

31 30	29	28 25	24 18	17 12	11 6	5 0
00	if	1101	opc	VMb	VMa	VMt

Description

Perform a logical operation between each bit of two vector mask registers and write the result to a vector mask register.

Pseudo Code (AND instruction example)

```
vmax = VMregIdx(VMa);
vmbx = VMregIdx(VMb);
vmtx = VMregIdx(VMt);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1)
        VP[j].VM[vmtx]<i> = VP[j].VM[vmax]<i>  & VP[j].VM[vmbx]<i>;
}
```

Opc Field Encoding

Instruction		***ISA***	***Type***	***Encoding***	***Operation***
AND	VMa,VMb,VMt	BVI	AE	F4,1,10	VMa & VMb
OR	VMa,VMb,VMt	BVI	AE	F4,1,11	VMa | VMb
NAND	VMa,VMb,VMt	BVI	AE	F4,1,12	~(VMa & VMb)
NOR	VMa,VMb,VMt	BVI	AE	F4,1,13	~(VMa | VMb)
XOR	VMa,VMb,VMt	BVI	AE	F4,1,14	VMa ^ VMb
XNOR	VMa,VMb,VMt	BVI	AE	F4,1,15	~(VMa ^ VMb)
ANDC	VMa,VMb,VMt	BVI	AE	F4,1,16	~VMa & VMb
ORC	VMa,VMb,VMt	BVI	AE	F4,1,17	~VMa | VMb

Exceptions

AE Register Range (AERRE)

BR – Branch

br target(At) ; branch unconditional

31 29	28 27	26 22	21	20 6	5 0
it	**10**	**opc**	**ar**	**immed15**	**rt**

br(.bl|.bu).(t|f) CCt, target ; branch conditional

31 29	28 27	26 22	21	20 6	5	4 0
it	**10**	**opc**	**ar**	**immed15**	**tf**	**cc**

Description

Program flow is transferred to the target address of the branch instruction. Conditional branches can select from any of the condition code bits within the CC register. Conditional branches can use the value of the CC register or the complemented value. Additionally, conditional branches can be hinted as whether the branch instruction is likely or unlikely to branch.

The target address can be specified with either absolute or relative addressing. The target address is formed by using the 15-bit immediate field within the instruction or optionally the 64-bit extended immediate field value. Note that the 15-bit immediate field within the instruction is shifted up by three bits, whereas the 64-bit extended immediate value is not shifted. Unconditional branches use the *At* field to specify an A register to be used as the base of the target address.

Pseudo Code

```
atx = AregIdx(At);
Target = Simmed15() + ((opc == 0x4 && atx != 0) ? A[atx] : 0);
Target &= ~7;                          // force instruction boundary alignment
if ( opc == 4 || CC<cc> == tf )
    IP = ar ? Target : IP + Target;
else
    IP = IP + InstLength();
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
BR	target(At)	Scalar	A	F2,04
BR	CC,target	Scalar	A	F2,05
BR.BL	CC,target	Scalar	A	F2,06
BR.BU	CC,target	Scalar	A	F2,07

Exceptions

Scalar Register Range (SRRE)

BRK – Break

brk ; break

31 29	28 27	26 22	21	20 6	5 0
it	**10**	**opc**	**-**	**-**	**-**

Description

The coprocessor routine is stopped and the host processor is sent an interrupt indicating that assistance is required. The instruction pointer register is not modified by the break instruction.

Pseudo Code

```
InterruptHostProcessor();
StopExecution();
```

Instruction Encoding

Instruction	*ISA*	*Type*	*Encoding*
BRK	Scalar	A	F2,02

Exceptions

None

CAEP00 – CAEP1F – Custom AE Instruction

CAEPyy.AEx	Immed18 ; where yy=00-1F and x=0,1,2,3
CAEPyy	Immed18

31 30	29	28 24	23 18	17 0
ae	m	11110	opc	Immed18

Description

These instructions are used by user defined AE personalities to initiate an operation. Thirty-two instructions are defined (00-1F). Additional CAEP instructions can be defined by using part or the entire 18-bit immediate field as additional opcode bits.

The instruction's m bit is used to determine if the instruction uses the AEC.CIM mask field to select which AEs participate (m=1), or the instruction directly specifies the single AE that participates in the operation (m=0).

Pseudo Code

```
if (m == 1) {
    for (i = 0; i < 4; i += 1)
        if (AEC.CIM[i])
            AE[i].ExecuteCustomInstruction();
} else
    AE[ae].ExecuteCustomInstruction();
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
CAEP00.AEx	Immed18	ASP	AE	F7,0,20
:		:	:	:
CAEP1F.AEx	Immed18	ASP	AE	F7,0,3F
CAEP00	Immed18	ASP	AE	F7,1,20
:		:	:	:
CAEP1F	Immed18	ASP	AE	F7,1,3F

Exceptions

Each CAEP instruction may generate Custom AE ISA defined exceptions.

CALL – Subroutine Call

call target(At) ; call unconditional

31 29	28 27	26 22	21	20 6	5 0
it	**10**	**opc**	**ar**	**immed15**	**rt**

call(.bl\|.bu).(t\|f) CC,target ; call conditional

31 29	28 27	26 22	21	20 6	5	4 0
it	**10**	**opc**	**ar**	**immed15**	**tf**	**cc**

Description

Program flow is transferred to the target address of the call instruction. The next sequential instruction address is pushed onto the call return address stack. Conditional calls can select from any of the condition code bits within the CC register. Conditional calls can use the value of the CC register or the complemented value. Additionally, conditional calls can be hinted whether the call instruction is likely or unlikely to branch.

The target address can be specified with either absolute or relative addressing. The target address is formed by using the 15-bit immediate field within the instruction or optionally the 64-bit extended immediate field value. Unconditional branches use the *At* field to specify an A register to be used as the base of the target address.

The next sequential instruction pointer and the window base register are pushed onto the call / return stack for unconditional calls as well as conditional calls that are taken.

Pseudo Code

```
atx = AregIdx(At)
if ( opc == 0x08 || CC<cc> == tf ) {
        If (!CPS.WBV)
            CRS[CPS.CRT  8].RWB = WB;
        CRS[CPS.CRT  8].RIP = IP + InstLength();
        If (CPS.CRT >= 8)
            CPS.SCOE = 1;
        else
            CPS.CRT += 1;
        CPS.WBV = 0;
        Target = Simmed15() + ((opc == 0x08 && atx != 0) ? A[atx] : 0);
        Target &= ~7;                          // force instruction boundary alignment
        IP = ar ? Target : IP + Target;
} else
        IP = IP + InstLength();
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*
CALL	target(At)	Scalar	A	F2,08
CALL	CC,target	Scalar	A	F2,09
CALL.BL	CC,target	Scalar	A	F2,0A
CALL.BU	CC,target	Scalar	A	F2,0B

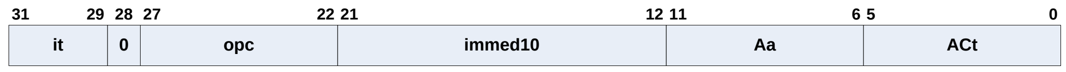
Exceptions

Scalar Register Range (SRRE)

Scalar CRS Overflow (SCOE)

CMP – Address Compare Integer with Immediate

CMP.UQ	Aa,Immed,ACt
CMP.SQ	Aa,Immed,ACt



Description

The instruction compares an address register to an immediate value for less than, greater than, and equal. The three comparison results are written to one of the four sets of address condition codes. The immediate value is either the zero or signed extended 10-bit Immed10 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code (example for CMP.SQ)

```
aax = AregIdx(Aa);
CPS.AC[ACt].EQ = A[aax] == Simmed10();
CPS.AC[ACt].GT = A[aax] > Simmed10();
CPS.AC[ACt].LT = A[aax] < Simmed10();
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
CMP.UQ	Aa,Immed,ACt	Scalar	A	F1,3A
CMP.SQ	Aa,Immed,ACt	Scalar	A	F1,3B

Exceptions

Scalar Register Range (SRRE)

CMP – Scalar Compare Integer with Immediate

CMP.UQ	Sa,Immed,SCt
CMP.SQ	Sa,Immed,SCt

31 30	29	28	27 22	21 12	11 6	5 0
it	if	0	opc	immed10	Sa	SCt

Description

The instruction compares a scalar register to an immediate value for less than, greater than, and equal. The three comparison results are written to one of the four sets of scalar condition codes. The immediate value is either the zero or signed extended 10-bit Immed10 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code (example for CMP.SQ)

```
sax = SregIdx(Sa);
CPS.SC[SCt].EQ = S[sax] == Simmed10();
CPS.SC[SCt].GT = S[sax] > Simmed10();
CPS.SC[SCt].LT = S[sax] < Simmed10();
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
CMP.UQ	Sa,Immed,SCt	Scalar	S	F1,1,3A
CMP.SQ	Sa,Immed,SCt	Scalar	S	F1,1,3B

Exceptions

Scalar Register Range (SRRE)

CMP – Scalar Compare Float Double with Immediate

CMP.FD Sa,Immed,SCt

31	30	29 28	25 24	18 17	12 11	6 5	0
it	0	1100	opc	Immed6	Sa	SCt	

Description

The instruction compares a scalar register to an immediate value for less than, greater than, and equal using float double data format. The three comparison results are written to one of the four sets of scalar condition codes. The immediate value is either the extended 6-bit Immed6 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
sax = SregIdx(Sa);
CPS.SC[SCt].EQ = S[sax] == Uimmed6();
CPS.SC[SCt].GT = S[sax]  >  Uimmed6();
CPS.SC[SCt].LT = S[sax]   <  Uimmed6();
```

Instruction Encoding

Instruction	***ISA***	***Type***	***Encoding***
CMP.FD Sa,Immed,SCt	Scalar	S	F3,0,3B

Exceptions

Scalar Register Range (SRRE) Scalar Float Invalid Operand (SFIE)

CMP – Scalar Compare Float Single with Immediate

CMP.FS Sa,Immed,SCt

31 30	29	28 25	24 18	17 12	11 6	5 0
it	0	1100	opc	Immed6	Sa	SCt

Description

The instruction compares a scalar register to an immediate value for less than, greater than, and equal using float single data format. The three comparison results are written to one of the four sets of scalar condition codes. The immediate value is either the extended 6-bit Immed6 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
sax = SregIdx(Sa);
CPS.SC[SCt].EQ = S[sax]<31:0> == Uimmed6()<31:0>;
CPS.SC[SCt].GT = S[sax]<31:0>  >  Uimmed6()<31:0>;
CPS.SC[SCt].LT = S[sax]<31:0>   <  Uimmed6()<31:0>;
```

Instruction Encoding

Instruction	***ISA***	***Type***	***Encoding***
CMP.FS Sa,Immed,SCt	Scalar	S	F3,0,3A

Exceptions

Scalar Register Range (SRRE)   Scalar Float Invalid Operand (SFIE)

CMP – Address Compare Integer

CMP.UQ	Aa,Ab,ACt
CMP.SQ	Aa,Ab,ACt

31 29	28 25	24 18	17 12	11 6	5 0
it	1101	opc	Ab	Aa	ACt

Description

The instruction compares two address register values for less than, greater than, and equal. The three comparison results are written to one of the four sets of address condition codes.

Pseudo Code

```
aax = AregIdx(Aa);
abx = AregIdx(Ab);
CPS.AC[ACt].EQ = A[aax] == A[abx];
CPS.AC[ACt].GT = A[aax] > A[abx];
CPS.AC[ACt].LT = A[aax] < A[abx];
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
CMP.UQ	Aa,Ab,ACt	Scalar	A	F4,3A
CMP.SQ	Aa,Ab,ACt	Scalar	A	F4,3B

Exceptions

Scalar Register Range (SRRE)

CMP – Scalar Compare Integer

CMP.UQ	Sa,Sb,SCt
CMP.SQ	Sa,Sb,SCt

31 30	29	28 25	24 18	17 12	11 6	5 0
it	1	1101	opc	Sb	Sa	SCt

Description

The instruction compares two scalar register values for less than, greater than, and equal. The three comparison results are written to one of the four sets of scalar condition codes.

Pseudo Code

```
sax = SregIdx(Sa);
sbx = SregIdx(Sb);
CPS.SC[SCt].EQ = S[sax] == S[sbx];
CPS.SC[SCt].GT = S[sax] > S[sbx];
CPS.SC[SCt].LT = S[sax] < S[sbx];
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
CMP.UQ	Sa,Sb,SCt	Scalar	S	F4,1,3A
CMP.SQ	Sa,Sb,SCt	Scalar	S	F4,1,3B

Exceptions

Scalar Register Range (SRRE)

CMP – Scalar Compare Float Double

CMP.FD Sa,Sb,SCt

31 30	29	28 25	24 18	17 12	11 6	5 0
it	0	1101	opc	Sb	Sa	SCt

Description

The instruction compares two scalar register values for less than, greater than, and equal using float double data format. The three comparison results are written to one of the four sets of scalar condition codes.

Pseudo Code

```
sax = SregIdx(Sa);
sbx = SregIdx(Sb);
CPS.SC[SCt].EQ = S[sax] == S[sbx];
CPS.SC[SCt].GT = S[sax] > S[sbx];
CPS.SC[SCt].LT = S[sax] < S[sbx];
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
CMP.FD	Sa,Sb,SCt	Scalar	S	F4,0,3B

Exceptions

Scalar Register Range (SRRE)   Scalar Float Invalid Operand (SFIE)

CMP – Scalar Compare Float Single

CMP.FS Sa,Sb,SCt

31 30	29	28 25	24 18	17 12	11 6	5 0
it	0	1101	opc	Sb	Sa	SCt

Description

The instruction compares two scalar register values for less than, greater than, and equal using float single data format. The three comparison results are written to one of the four sets of scalar condition codes.

Pseudo Code

```
sax = SregIdx(Sa);
sbx = SregIdx(Sb);
CPS.SC[SCt].EQ = S[sax]<31:0> == S[sbx]<31:0>;
CPS.SC[SCt].GT = S[sax]<31:0>  >  S[sbx]<31:0>;
CPS.SC[SCt].LT = S[sax]<31:0>   <  S[sbx]<31:0>;
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
CMP.FS	Sa,Sb,SCt	Scalar	S	F4,0,3A

Exceptions

Scalar Register Range (SRRE)

Scalar Float Invalid Operand (SFIE)

CVT – Scalar Convert Float Double to Float Single

CVT.FD.FS Sa,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	if	1100	opc	000000	Sa	St

Description

The instruction copies the source register to the destination register converting from float double to float single.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx] = CvtFdFs( S[sax] );
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
CVT.FD.FS	Sa,St	Scalar	S	F3,0,06

Exceptions

Scalar Register Range (SRRE)	Scalar Float Invalid Operand (SFIE)
Scalar Float Overflow (SFOE)	Scalar Float Underflow (SFUE)

CVT – Scalar Convert Float Double to Signed Quad Word

CVT.FD.SQ Sa,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	if	1100	opc	000000	Sa	St

Description

The instruction copies the source register to the destination register converting from float double to signed quad word.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx] = CvtFdSq( S[sax] );
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
CVT.FD.SQ	Sa,St	Scalar	S	F3,0,05

Exceptions

Scalar Register Range (SRRE)

Scalar Float Invalid Operand (SFIE)

Scalar Integer Overflow (SIOE)

CVT – Scalar Convert Float Single to Float Double

CVT.FS.FD Sa,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	if	1100	opc	000000	Sa	St

Description

The instruction copies the source register to the destination register converting from float single to float double.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx] = CvtFsFd( S[sax] );
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
CVT.FS.FD	Sa,St	Scalar	S	F3,0,03

Exceptions

Scalar Register Range (SRRE)

Scalar Float Invalid Operand (SFIE)

CVT – Scalar Convert Float Single to Signed Quad Word

CVT.FS.SQ Sa,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	if	1100	opc	000000	Sa	St

Description

The instruction copies the source register to the destination register converting from float single to signed quad word.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx] = CvtFsSq( S[sax] );
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
CVT.FS.SQ	Sa,St	Scalar	S	F3,0,01

Exceptions

Scalar Register Range (SRRE)

Scalar Integer Overflow (SIOE)

Scalar Float Invalid Operand (SFIE)

CVT – Scalar Convert Signed Quad Word to Float Double

CVT.SQ.FD Sa,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	if	1100	opc	000000	Sa	St

Description

The instruction copies the source register to the destination register converting from signed quad word to float double.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx] = CvtSqFd( S[sax] );
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
CVT.SQ.FD	Sa,St	Scalar	S	F3,1,07

Exceptions

Scalar Register Range (SRRE)

CVT – Scalar Convert Signed Quad Word to Float Single

CVT.SQ.FS Sa,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	if	1100	opc	000000	Sa	St

Description

The instruction copies the source register to the destination register converting from signed quad word to float single.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx] = CvtSqFs( S[sax] );
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
CVT.SQ.FS	Sa,St	Scalar	S	F3,1,06

Exceptions

Scalar Register Range (SRRE)

CVT – Address Convert Signed to Unsigned Quad Word

CVT.SQ.UQ Aa,At

31 29	28 25	24 18	17 12	11 6	5 0
it	1100	opc	000000	Aa	At

Description

The instruction copies the source register to the destination register checking if the signed quad word source data type can be represented as an unsigned quad word destination data type.

Pseudo Code

```
aax = AregIdx(Aa);
atx = AregIdx(At);
A[atx] = CvtSqUq( A[aax] );
```

Instruction Encoding

Instruction	***ISA***	***Type***	***Encoding***
CVT.SQ.UQ Aa,At	Scalar	A	F3,04

Exceptions

Scalar Register Range (SRRE)

Scalar Integer Overflow (SIOE)

CVT – Scalar Convert Signed to Unsigned Quad Word

CVT.SQ.UQ Sa,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	if	1100	opc	000000	Sa	St

Description

The instruction copies the source register to the destination register checking if the signed quad word source data type can be represented as an unsigned quad word destination data type.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx] = CvtSqUq( S[sax] );
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
CVT.SQ.UQ	Sa,St	Scalar	S	F3,1,04

Exceptions

Scalar Register Range (SRRE) Scalar Integer Overflow (SIOE)

CVT – Address Convert Unsigned to Signed Quad Word

CVT.UQ.SQ Aa,At

31 29	28 25	24 18	17 12	11 6	5 0
it	1100	opc	000000	Aa	At

Description

The instruction copies the source register to the destination register checking if the unsigned quad word source data type can be represented as a signed quad word destination data type.

Pseudo Code

```
aax = AregIdx(Aa);
atx = AregIdx(At);
A[atx] = CvtUqSq( A[aax] );
```

Instruction Encoding

Instruction	***ISA***	***Type***	***Encoding***
CVT.UQ.SQ Aa,At	Scalar	A	F3,01

Exceptions

Scalar Register Range (SRRE) Scalar Integer Overflow (SIOE)

CVT – Scalar Convert Unsigned to Signed Quad Word

CVT.UQ.SQ Sa,St

31	30	29	28 25	24 18	17 12	11 6	5 0
it		if	1100	opc	000000	Sa	St

Description

The instruction copies the source register to the destination register checking if the unsigned quad word source data type can be represented as a signed quad word destination data type.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx] = CvtUqSq( S[sax] );
```

Instruction Encoding

Instruction	ISA	Type	Encoding
CVT.UQ.SQ Sa,St	Scalar	S	F3,1,01

Exceptions

Scalar Register Range (SRRE) Scalar Integer Overflow (SIOE)

DIV – Address Divide Integer with Immediate

DIV.UQ	Aa,Immed,At
DIV.SQ	Aa,Immed,At

31 29	28	27 22	21 12	11 6	5 0
it	0	opc	immed10	Aa	At

Description

The instruction performs division between an address register and an immediate value using unsigned or signed integer arithmetic. The immediate value is either the zero or signed extended 10-bit Immed10 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code (example for DIV.SQ)

```
aax = AregIdx(Aa);
atx = AregIdx(At);
A[atx] = A[aax] / Simmed10();
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*
DIV.UQ	Aa,Immed,At	Scalar	A	F3,36
DIV.SQ	Aa,Immed,At	Scalar	A	F3,37

Exceptions

Scalar Register Range (SRRE)

Scalar Integer Overflow (SIOE)

Scalar Integer Divide by Zero (SIZE)

DIV – Address Divide Integer

DIV.UQ Aa,Ab,At
DIV.SQ Aa,Ab,At

31 29	28 25	24 18	17 12	11 6	5 0
it	1101	opc	Ab	Aa	At

Description

The instruction performs division on two address register values using signed or unsigned integer arithmetic.

Pseudo Code

```
aax = AregIdx(Aa);
abx = AregIdx(Ab);
atx = AregIdx(At);
A[atx] = A[aax] / A[abx];
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
DIV.UQ	Aa,Ab,At	Scalar	A	F4,36
DIV.SQ	Aa,Ab,At	Scalar	A	F4,37

Exceptions

Scalar Register Range (SRRE)

Scalar Integer Overflow (SIOE)

Scalar Integer Divide by Zero (SIZE)

DIV – Scalar Divide Integer with Immediate

DIV.UQ	Sa,Immed,St
DIV.SQ	Sa,Immed,St

31 30	29	28	27 22	21 12	11 6	5 0
it	1	0	opc	immed10	Sa	St

Description

The instruction performs division between a scalar register and an immediate value using unsigned or signed integer arithmetic. The immediate value is either the zero or signed extended 10-bit Immed10 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx] = S[sax] / Simmed10();
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*
DIV.UQ	Sa,Immed,St	Scalar	S	F3,1,36
DIV.SQ	Sa,Immed,St	Scalar	S	F3,1,37

Exceptions

Scalar Register Range (SRRE)

Scalar Integer Overflow (SIOE)

Scalar Integer Divide by Zero (SIZE)

DIV – Scalar Divide Integer

DIV.UQ	Sa,Sb,St
DIV.SQ	Sa,Sb,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	1	1101	opc	Sb	Sa	St

Description

The instruction performs division on two scalar register values using signed or unsigned integer arithmetic.

Pseudo Code

```
sax = SregIdx(Sa);
sbx = SregIdx(Sb);
stx = SregIdx(St);
S[stx] = S[sax] / S[sbx];
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
DIV.UQ	Sa,Sb,St	Scalar	S	F4,1,36
DIV.SQ	Sa,Sb,St	Scalar	S	F4,1,37

Exceptions

Scalar Register Range (SRRE)

Scalar Integer Overflow (SIOE)

Scalar Integer Divide by Zero (SIZE)

DIV – Scalar Division Float Double with Immediate

DIV.FD Sa,Immed,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	0	1100	opc	Immed6	Sa	St

Description

The instruction performs division between a scalar register and an immediate value using float double arithmetic. The immediate value is either the zero extended 6-bit Immed6 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx] = S[sax] / Uimmed6();
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
DIV.FD	Sa,Immed,St	Scalar	S	F3,0,37

Exceptions

Scalar Register Range (SRRE)

Scalar Float Invalid Operand (SFIE)

Scalar Float Overflow (SFOE)

Scalar Float Underflow (SFUE)

Scalar Float Divide by Zero (SFZE)

DIV – Scalar Division Float Single with Immediate

DIV.FS Sa,Immed,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	0	1100	opc	Immed6	Sa	St

Description

The instruction performs division between a scalar register and an immediate value using float single arithmetic. The immediate value is either the zero extended 6-bit Immed6 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx]<63:0> = 0;
S[stx]<31:0> = S[sax]<31:0> / Uimmed6()<31:0>;
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
DIV.FS	Sa,Immed,St	Scalar	S	F3,0,36

Exceptions

Scalar Register Range (SRRE)

Scalar Float Invalid Operand (SFIE)

Scalar Float Overflow (SFOE)

Scalar Float Underflow (SFUE)

Scalar Float Divide by Zero (SFZE)

DIV – Scalar Division Float Double

DIV.FD Sa,Sb,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	0	1101	opc	Sb	Sa	St

Description

The instruction performs division on two scalar register values using float double arithmetic.

Pseudo Code

```
sax = SregIdx(Sa);
sbx = SregIdx(Sb);
stx = SregIdx(St);
S[stx] = S[sax] / S[sbx];
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*
DIV.FD	Sa,Sb,St	Scalar	S	F4,0,37

Exceptions

Scalar Register Range (SRRE)

Scalar Float Invalid Operand (SFIE)

Scalar Float Overflow (SFOE)

Scalar Float Underflow (SFUE)

Scalar Float Divide by Zero (SFZE)

DIV – Scalar Division Float Single

DIV.FS Sa,Sb,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	0	1101	opc	Sb	Sa	St

Description

The instruction performs division on two scalar register values using float single arithmetic.

Pseudo Code

```
sax = SregIdx(Sa);
sbx = SregIdx(Sb);
stx = SregIdx(St);
S[stx]<63:32> = 0;
S[stx]<31:0> = S[sax]<31:0> / S[sbx]<31:0>;
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
DIV.FS	Sa,Sb,St	Scalar	S	F4,0,36

Exceptions

Scalar Register Range (SRRE)

Scalar Float Invalid Operand (SFIE)

Scalar Float Overflow (SFOE)

Scalar Float Underflow (SFUE)

Scalar Float Divide by Zero (SFZE)

EQ – Vector Compare for Equal Signed Quad Word Scalar

EQ.SQ	Va,Sb,VMt
EQ.SQ.T	Va,Sb,VMt
EQ.SQ.F	Va,Sb,VMt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1100	opc	Sb	Va	VMt

Description

The instruction compares for equal a vector register and an S register using signed quad word data format. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The VMt instruction field specifies the VM register written by the instruction. Note that the VMt register cannot be equal to the VMA register.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vmtx = VMregIdx(VMt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>){
            VP[j]. VM[vmtx]<i> = VP[j]. V[vax][i] == S[sbx];
        } else
            VP[j]. VM[vmtx]<i> = false;
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
EQ.SQ	Va,Sb,VMt	BVI	AE	F3,1,3B	0
EQ.SQ.T	Va,Sb,VMt	BVI	AE	F3,1,3B	2
EQ.SQ.F	Va,Sb,VMt	BVI	AE	F3,1,3B	3

Exceptions

AE Register Range (AERRE) Scalar Register Range (SRRE)

EQ – Vector Compare for Equal Signed Quad Word

EQ.SQ	Va,Vb,VMt
EQ.SQ.T	Va,Vb,VMt
EQ.SQ.F	Va,Vb,VMt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1101	opc	Vb	Va	VMt

Description

The instruction compares for equal two vector registers using signed quad word data format. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The VMt instruction field specifies the VM register written by the instruction.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vmtx = VMregIdx(VMt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>){
            VP[j]. VM[vmtx]<i> = VP[j]. V[vax][i] == VP[j].V[vbx][i];
        } else
            VP[j]. VM[vmtx]<i> = false;
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
EQ.SQ	Va,Vb,VMt	BVI	AE	F4,1,3B	0
EQ.SQ.T	Va,Vb,VMt	BVI	AE	F4,1,3B	2
EQ.SQ.F	Va,Vb,VMt	BVI	AE	F4,1,3B	3

Exceptions

AE Register Range (AERRE)

EQ – Vector Compare for Equal Unsigned Quad Word Scalar

EQ.UQ	Va,Sb,VMt
EQ.UQ.T	Va,Sb,VMt
EQ.UQ.F	Va,Sb,VMt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1100	opc	Sb	Va	VMt

Description

The instruction compares for equal a vector register and an S register using unsigned quad word data format. The 'vm' field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The VMt instruction field specifies the VM register written by the instruction. Note that the VMt register cannot be equal to the VMA register.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vmtx = VMregIdx(VMt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>){
            VP[j]. VM[vmtx]<i> = VP[j]. V[vax][i] == S[sbx];
        } else
            VP[j]. VM[vmtx]<i> = false;
    }
}
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*	*VM*
EQ.UQ	Va,Sb,VMt	BVI	AE	F3,1,3A	0
EQ.UQ.T	Va,Sb,VMt	BVI	AE	F3,1,3A	2
EQ.UQ.F	Va,Sb,VMt	BVI	AE	F3,1,3A	3

Exceptions

AE Register Range (AERRE)   Scalar Register Range (SRRE)

EQ – Vector Compare for Equal Unsigned Quad Word

EQ.UQ	Va,Vb,VMt
EQ.UQ.T	Va,Vb,VMt
EQ.UQ.F	Va,Vb,VMt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1101	opc	Vb	Va	VMt

Description

The instruction compares for equal two vector registers using unsigned quad word data format. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The VMt instruction field specifies the VM register written by the instruction.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vmtx = VMregIdx(VMt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>){
            VP[j]. VM[vmtx]<i> = VP[j]. V[vax][i] == VP[j].V[vbx][i];
        } else
            VP[j]. VM[vmtx]<i> = false;
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
EQ.UQ	Va,Vb,VMt	BVI	AE	F4,1,3A	0
EQ.UQ.T	Va,Vb,VMt	BVI	AE	F4,1,3A	2
EQ.UQ.F	Va,Vb,VMt	BVI	AE	F4,1,3A	3

Exceptions

AE Register Range (AERRE)

FENCE – Memory Fence

FENCE

31 29	28 24	23 18	17 0
it	11110	opc	00 0000 0000 0000 0000

Description

The instruction performs a memory fence operation. The fence ensures that all memory operations issued before the fence are completed before any memory operations executed after the fence.

Pseudo Code

```
MemFence();
```

Instruction Encoding

Instruction	***ISA***	***Type***	***Encoding***
FENCE	Scalar	A	F7,0F

Exceptions

None

FILL – Fill Vector with Scalar

FILL	Sb,Vt
FILL.T	Sb,Vt
FILL.F	Sb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	0	1100	opc	Sb	000000	Vt

Description

The instruction fills a vector register with a scalar value. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

Pseudo Code

```
sbx = SregIdx(Sb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>){
            VP[j]. V[vtx]<i> = S[sbx];
        } else
            VP[j]. V[vtx]<i> = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***vm***
FILL	Sb,Vt	BVI	AE	F3,0,2E	0
FILL.T	Sb,Vt	BVI	AE	F3,0,2E	2
FILL.F	Sb,Vt	BVI	AE	F3,0,2E	3

Exceptions

AE Register Range (AERRE)   Scalar Register Range (SRRE)

GT – Vector Compare for Greater Signed Quad Word Scalar

GT.SQ	Va,Sb,VMt
GT.SQ.T	Va,Sb,VMt
GT.SQ.F	Va,Sb,VMt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1100	opc	Sb	Va	VMt

Description

The instruction compares for greater than a vector register and an S register using signed quad word data format. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The VMt instruction field specifies the VM register written by the instruction.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vmtx = VMregIdx(VMt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>){
            VP[j]. VM[vmtx]<i> = VP[j]. V[vax][i] > S[sbx];
        } else
            VP[j]. VM[vmtx]<i> = false;
    }
}
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*	*VM*
GT.SQ	Va,Sb,VMt	BVI	AE	F3,1,3F	0
GT.SQ.T	Va,Sb,VMt	BVI	AE	F3,1,3F	2
GT.SQ.F	Va,Sb,VMt	BVI	AE	F3,1,3F	3

Exceptions

AE Register Range (AERRE)

Scalar Register Range (SRRE)

GT – Vector Compare for Greater Signed Quad Word

GT.SQ	Va,Vb,VMt
GT.SQ.T	Va,Vb,VMt
GT.SQ.F	Va,Vb,VMt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	**1**	**1101**	**opc**	**Vb**	**Va**	**VMt**

Description

The instruction compares for greater than two vector registers using signed quad word data format. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The VMt instruction field specifies the VM register written by the instruction.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vmtx = VMregIdx(VMt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>){
            VP[j]. VM[vmtx]<i> = VP[j]. V[vax][i] > VP[j].V[vbx][i];
        } else
            VP[j]. VM[vmtx]<i> = false;
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
GT.SQ	Va,Vb,VMt	BVI	AE	F4,1,3F	0
GT.SQ.T	Va,Vb,VMt	BVI	AE	F4,1,3F	2
GT.SQ.F	Va,Vb,VMt	BVI	AE	F4,1,3F	3

Exceptions

AE Register Range (AERRE)

GT – Vector Compare for Greater Unsigned Quad Word Scalar

GT.UQ	Va,Sb,VMt
GT.UQ.T	Va,Sb,VMt
GT.UQ.F	Va,Sb,VMt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1100	opc	Sb	Va	VMt

Description

The instruction compares for greater than a vector register and an S register using unsigned quad word data format. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The VMt instruction field specifies the VM register written by the instruction.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vmtx = VMregIdx(VMt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>){
            VP[j]. VM[vmtx]<i> = VP[j]. V[vax][i] > S[sbx];
        } else
            VP[j]. VM[vmtx]<i> = false;
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
GT.UQ	Va,Sb,VMt	BVI	AE	F3,1,3E	0
GT.UQ.T	Va,Sb,VMt	BVI	AE	F3,1,3E	2
GT.UQ.F	Va,Sb,VMt	BVI	AE	F3,1,3E	3

Exceptions

AE Register Range (AERRE)

Scalar Register Range (SRRE)

GT – Vector Compare for Greater Unsigned Quad Word

GT.UQ	Va,Vb,VMt
GT.UQ.T	Va,Vb,VMt
GT.UQ.F	Va,Vb,VMt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1101	opc	Vb	Va	VMt

Description

The instruction compares for greater than two vector registers using unsigned quad word data format. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The VMt instruction field specifies the VM register written by the instruction.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vmtx = VMregIdx(VMt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>){
            VP[j]. VM[vmtx]<i> = VP[j]. V[vax][i] > VP[j].V[vbx][i];
        } else
            VP[j]. VM[vmtx]<i> = false;
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
GT.UQ	Va,Vb,VMt	BVI	AE	F4,1,3E	0
GT.UQ.T	Va,Vb,VMt	BVI	AE	F4,1,3E	2
GT.UQ.F	Va,Vb,VMt	BVI	AE	F4,1,3E	3

Exceptions

AE Register Range (AERRE)

LD – Load with offset (A Register)

ld.(ub|uw|ud|uq|sb|sw|sd|sq)   offset(Aa),At   ; load with offset A-reg

31 30	29	28	27 22	21 12	11 6	5 0
it	if	0	opc	immed10	ra	rt

Description

Load an A register with a value from memory. The load can be Byte, Word, Double Word or Quad Word in size and the value can be either zero or signed extended to the 64-bit A register width. The effective address is formed by adding an offset to an A register value. The offset is either the signed extended 10-bit Immed10 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
aax = AregIdx(Aa);
atx = AregIdx(At);
effa = A[aax] + Offset(opc<1:0>);
switch ( opc<2:0> ) {
case 0: A[atx] = Zext64 ( MemLoad( effa, 8 ) ); break;      // unsigned byte
case 1: A[atx] = Zext64 ( MemLoad( effa, 16 ) ); break;     // unsigned word
case 2: A[atx] = Zext64 ( MemLoad( effa, 32 ) ); break;     // unsigned double word
case 3: A[atx] = MemLoad( effa, 64 ); break;                // quad word
case 4: A[atx] = Sext64 ( MemLoad( effa, 8 ) ); break;      // signed byte
case 5: A[atx] = Sext64 ( MemLoad( effa, 16 ) ); break;     // signed word
case 6: A[atx] = Sext64 ( MemLoad( effa, 32 ) ); break;     // signed double word
}
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*
LD.UB	offset (Aa),At	Scalar	A	F1,00
LD.UW	offset (Aa),At	Scalar	A	F1,01
LD.UD	offset (Aa),At	Scalar	A	F1,02
LD.UQ	offset (Aa),At	Scalar	A	F1,03
LD.SB	offset (Aa),At	Scalar	A	F1,04
LD.SW	offset (Aa),At	Scalar	A	F1,05
LD.SD	offset (Aa),At	Scalar	A	F1,06

Exceptions

Scalar Unaligned Reference (SURE)   Scalar Register Range (SRRE)

LD – Load with offset (S Register)

ld.(ub|uw|ud|uq|sb|sw|sd|sq) offset(Aa),St

31 30	29	28	27 22	21 12	11 6	5 0
it	1	0	opc	immed10	Aa	St

Description

Load an S register with a value from memory. The load can be Byte, Word, Double Word or Quad Word in size and the value can be either zero or signed extended to the 64-bit A register width. The effective address is formed by adding an offset to an A register value. The offset is either the signed extended 10-bit Immed10 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
aax = AregIdx(Aa);
stx = SregIdx(St);
effa = A[aax] + Offset(opc<1:0>);
switch ( opc<2:0> ) {
case 0: S[stx] = Zext64 ( MemLoad( effa, 8 ) ); break;      // unsigned byte
case 1: S[stx] = Zext64 ( MemLoad( effa, 16 ) ); break;     // unsigned word
case 2: S[stx] = Zext64 ( MemLoad( effa, 32 ) ); break;     // unsigned double word
case 3: S[stx] = MemLoad( effa, 64 ); break;                // quad word
case 4: S[stx] = Sext64 ( MemLoad( effa, 8 ) ); break;      // signed byte
case 5: S[stx] = Sext64 ( MemLoad( effa, 16 ) ); break;     // signed word
case 6: S[stx] = Sext64 ( MemLoad( effa, 32 ) ); break;     // signed double word
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
LD.UB	offset (Aa),St	Scalar	S	F1,1,00
LD.UW	offset (Aa),St	Scalar	S	F1,1,01
LD.UD	offset (Aa),St	Scalar	S	F1,1,02
LD.UQ	offset (Aa),St	Scalar	S	F1,1,03
LD.SB	offset (Aa),St	Scalar	S	F1,1,04
LD.SW	offset (Aa),St	Scalar	S	F1,1,05
LD.SD	offset (Aa),St	Scalar	S	F1,1,06

Exceptions

Scalar Unaligned Reference (SURE) Scalar Register Range (SRRE)

LD – Load indexed (A Register)

ld.(ub|uw|ud|uq|sb|sw|sd|sq)   Ab(Aa),At

31 30	29	28 26	25 18	17 12	11 6	5 0
it	**if**	**110**	**opc**	**rb**	**ra**	**rt**

Description

Load an A register with a value from memory. The load can be Byte, Word, Double Word or Quad Word in size and the value can be either zero or signed extended to the 64-bit A register width. The effective address is formed by adding two A register values.

Pseudo Code

```
aax = AregIdx(Aa);
abx = AregIdx(Ab);
atx = AregIdx(At);
effa = A[aax] + A[abx];
switch ( opc<2:0> ) {
case 0: A[atx] = Zext64 ( MemLoad( effa, 8 ) ); break;          // unsigned byte
case 1: A[atx] = Zext64 ( MemLoad( effa, 16 ) ); break;         // unsigned word
case 2: A[atx] = Zext64 ( MemLoad( effa, 32 ) ); break;         // unsigned double word
case 3: A[atx] = MemLoad( Effa, 64 ); break;                    // quad word
case 4: A[atx] = Sext64 ( MemLoad( effa, 8 ) ); break;          // signed byte
case 5: A[atx] = Sext64 ( MemLoad( effa, 16 ) ); break;         // signed word
case 6: A[atx] = Sext64 ( MemLoad( effa, 32 ) ); break;         // signed double word
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
LD.UB	Ab(Aa),At	Scalar	A	F4,00
LD.UW	Ab(Aa),At	Scalar	A	F4,01
LD.UD	Ab(Aa),At	Scalar	A	F4,02
LD.UQ	Ab(Aa),At	Scalar	A	F4,03
LD.SB	Ab(Aa),At	Scalar	A	F4,04
LD.SW	Ab(Aa),At	Scalar	A	F4,05
LD.SD	Ab(Aa),At	Scalar	A	F4,06

Exceptions

Scalar Unaligned Reference (SURE)   Scalar Register Range (SRRE)

LD – Load indexed (S Register)

ld.(ub|uw|ud|uq|sb|sw|sd|sq)   Ab(Aa),St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	**1**	**1101**	**opc**	**Ab**	**Aa**	**St**

Description

Load an S register with a value from memory. The load can be Byte, Word, Double Word or Quad Word in size and the value can be either zero or signed extended to the 64-bit S register width. The effective address is formed by adding two A register values.

Pseudo Code

```
aax = AregIdx(Aa);
abx = AregIdx(Ab);
stx = SregIdx(St);
effa = A[aax] + A[abx];
switch ( opc<2:0> ) {
case 0: S[stx] = Zext64 ( MemLoad( effa, 8 ) ); break;        // unsigned byte
case 1: S[stx] = Zext64 ( MemLoad( effa, 16 ) ); break;       // unsigned word
case 2: S[stx] = Zext64 ( MemLoad( effa, 32 ) ); break;       // unsigned double word
case 3: S[stx] = MemLoad( Effa, 64 ); break;                  // quad word
case 4: S[stx] = Sext64 ( MemLoad( effa, 8 ) ); break;        // signed byte
case 5: S[stx] = Sext64 ( MemLoad( effa, 16 ) ); break;       // signed word
case 6: S[stx] = Sext64 ( MemLoad( effa, 32 ) ); break;       // signed double word
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
LD.UB	Ab(Aa),St	Scalar	S	F4,1,00
LD.UW	Ab(Aa),St	Scalar	S	F4,1,01
LD.UD	Ab(Aa),St	Scalar	S	F4,1,02
LD.UQ	Ab(Aa),St	Scalar	S	F4,1,03
LD.SB	Ab(Aa),St	Scalar	S	F4,1,04
LD.SW	Ab(Aa),St	Scalar	S	F4,1,05
LD.SD	Ab(Aa),St	Scalar	S	F4,1,06

Exceptions

Scalar Unaligned Reference (SURE)   Scalar Register Range (SRRE)

LD – Load Vector Dual Double Word

LD.DDW	offset(Aa),Vt
LD.DDW.T	offset(Aa),Vt
LD.DDW.F	offset(Aa),Vt

31 30	29	28	27 22	21 12	11 6	5 0
vm	0	0	opc	immed10	Aa	Vt

Description

The instruction loads eight bytes from memory to a 64-bit destination vector register. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. Note that the data can be aligned on four byte memory boundaries. The offset is either the signed extended 10-bit Immed10 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
aax = AregIdx(Aa);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
effaBase = A[aax] + Offset(2);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            effa = effaBase + VPS * j + VS * i;
            VP[j].[vtx][i] = MemLoad( effa<47:0>, 64 );
        } else
            VP[j].[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
LD.DDW	offset(Aa),Vt	BVI	AE	F1,0,06	0
LD.DDW.T	offset(Aa),Vt	BVI	AE	F1,0,06	2
LD.DDW.F	offset(Aa),Vt	BVI	AE	F1,0,06	3

Exceptions

AE Register Range (AERRE)

AE Unaligned Reference (AEURE)

Scalar Register Range (SRRE)

LD – Load Vector Double Word

LD.DW	offset(Aa),Vt
LD.DW.T	offset(Aa),Vt
LD.DW.F	offset(Aa),Vt

31 30	29	28	27 22	21 12	11 6	5 0
vm	0	0	opc	immed10	Aa	Vt

Description

The instruction loads four bytes from memory to a 32-bit destination vector register. The 'vm' field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The offset is either the signed extended 10-bit Immed10 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
aax = AregIdx(Aa);
vtx = VregIdx(Vt<4:0>);
vmax = VMregIdx(WB.VMA);
effaBase = A[aax] + Offset(opc<1:0>);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            effa = effaBase + VPS * j + VS * i;
            VP[j]. V32[Vt<5>][vtx][i] = MemLoad( effa<47:0>, 32 );
        } else
            VP[j]. V32[Vt<5>] [vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
LD.DW	offset(Aa),Vt	BVI	AE	F1,0,02	0
LD.DW.T	offset(Aa),Vt	BVI	AE	F1,0,02	2
LD.DW.F	offset(Aa),Vt	BVI	AE	F1,0,02	3

Exceptions

AE Register Range (AERRE)	Scalar Register Range (SRRE)
AE Unaligned Reference (AEURE)	

LD – Load Vector Indexed Double Word

LD.DW	Vb(Aa),Vt
LD.DW.T	Vb(Aa),Vt
LD.DW.F	Vb(Aa),Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	0	1101	opc	Vb	Aa	Vt

Description

The instruction loads four bytes from memory to a 32-bit destination vector register using a vector of indexes. The 'vm' field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

Pseudo Code

```
aax = AregIdx(Aa);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt<4:0>);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            effa = A[aax] + VP[j].V[vbx][i];
            VP[j]. V32[Vt<5>][vtx]<i> = MemLoad( effa<47:0>, 32 );
        } else
            VP[j]. V32[Vt<5>] [vtx]<i> = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*	*VM*
LD.DW	Vb(Aa),Vt	BVI	AE	F4,0,02	0
LD.DW.T	Vb(Aa),Vt	BVI	AE	F4,0,02	2
LD.DW.F	Vb(Aa),Vt	BVI	AE	F4,0,02	3

Exceptions

AE Register Range (AERRE)	Scalar Register Range (SRRE)
AE Unaligned Reference (AEURE)	

LD – Load Vector Unsigned Double Word

LD.UD	offset(Aa),Vt
LD.UD.T	offset(Aa),Vt
LD.UD.F	offset(Aa),Vt

31 30	29	28	27 22	21 12	11 6	5 0
vm	1	0	opc	immed10	Aa	Vt

Description

The instruction loads four bytes from memory to a destination vector register. The four bytes returned from memory is zero extended to eight bytes. The 'vm' field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The offset is either the signed extended 10-bit Immed10 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
aax = AregIdx(Aa);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
effaBase = A[aax] + Offset(opc<1:0>);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            effa = effaBase + VPS * j + VS * i;
            VP[j]. V[vtx][i] = ZeroExt64 ( MemLoad( effa<47:0>, 32 ) );
        } else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
LD.UD	offset(Aa),Vt	BVI	AE	F1,1,02	0
LD.UD.T	offset(Aa),Vt	BVI	AE	F1,1,02	2
LD.UD.F	offset(Aa),Vt	BVI	AE	F1,1,02	3

Exceptions

AE Register Range (AERRE)

AE Unaligned Reference (AEURE)

Scalar Register Range (SRRE)

LD – Load Vector Unsigned Quad Word

LD.UQ	offset(Aa),Vt
LD.UQ.T	offset(Aa),Vt
LD.UQ.F	offset(Aa),Vt

31 30	29	28	27 22	21 12	11 6	5 0
vm	1	0	opc	immed10	Aa	Vt

Description

The instruction loads eight bytes from memory to a destination vector register. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The offset is either the signed extended 10-bit Immed10 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
aax = AregIdx(Aa);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
effaBase = A[aax] + Offset(opc<1:0>);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            effa = effaBase + VPS * j + VS * i;
            VP[j]. V[vtx][i] = MemLoad( effa<47:0>, 64 );
        } else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
LD.UQ	offset(Aa),Vt	BVI	AE	F1,1,03	0
LD.UQ.T	offset(Aa),Vt	BVI	AE	F1,1,03	2
LD.UQ.F	offset(Aa),Vt	BVI	AE	F1,1,03	3

Exceptions

AE Register Range (AERRE)

Scalar Register Range (SRRE)

AE Unaligned Reference (AEURE)

LD – Load Vector Signed Double Word

LD.SD	offset(Aa),Vt
LD.SD.T	offset(Aa),Vt
LD.SD.F	offset(Aa),Vt

31 30	29	28	27 22	21 12	11 6	5 0
vm	1	0	opc	immed10	Aa	Vt

Description

The instruction loads four bytes from memory to a destination vector register. The four bytes returned from memory are sign extended to eight bytes. The 'vm' field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The offset is either the signed extended 10-bit Immed10 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
aax = AregIdx(Aa);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
effaBase = A[aax] + Offset(opc<1:0>);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            effa = effaBase + VPS * j + VS * i;
            VP[j]. V[vtx][i] = SignExt64 ( MemLoad( effa<47:0>, 32 ) );
        } else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*	*VM*
LD.SD	offset(Aa),Vt	BVI	AE	F1,1,06	0
LD.SD.T	offset(Aa),Vt	BVI	AE	F1,1,06	2
LD.SD.F	offset(Aa),Vt	BVI	AE	F1,1,06	3

Exceptions

AE Register Range (AERRE)

Scalar Register Range (SRRE)

AE Unaligned Reference (AEURE)

LD – Load Vector Indexed Unsigned Double Word

LD.UD	Vb(Aa),Vt
LD.UD.T	Vb(Aa),Vt
LD.UD.F	Vb(Aa),Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1101	opc	Vb	Aa	Vt

Description

The instruction loads four bytes from memory to a destination vector register using a vector of indexes. The four bytes returned from memory is zero extended to eight bytes. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

Pseudo Code

```
aax = AregIdx(Aa);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            effa = A[aax] + VP[j].V[vbx][i];
            VP[j]. V[vtx][i] = ZeroExt64 ( MemLoad( effa<47:0>, 32 ) );
        } else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
LD.UD	Vb(Aa),Vt	BVI	AE	F4,1,02	0
LD.UD.T	Vb(Aa),Vt	BVI	AE	F4,1,02	2
LD.UD.F	Vb(Aa),Vt	BVI	AE	F4,1,02	3

Exceptions

AE Register Range (AERRE)

Scalar Register Range (SRRE)

AE Unaligned Reference (AEURE)

LD – Load Vector Indexed Unsigned Quad Word

LD.UQ	Vb(Aa),Vt
LD.UQ.T	Vb(Aa),Vt
LD.UQ.F	Vb(Aa),Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1101	opc	Vb	Aa	Vt

Description

The instruction loads eight bytes from memory to a destination vector register using a vector of indexes. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

Pseudo Code

```
aax = AregIdx(Aa);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            effa = A[aax] + VP[j].V[vbx][i];
            VP[j]. V[vtx][i] = MemLoad( effa<47:0>, 64 );
        } else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*	*VM*
LD.UQ	Vb(Aa),Vt	BVI	AE	F4,1,03	0
LD.UQ.T	Vb(Aa),Vt	BVI	AE	F4,1,03	2
LD.UQ.F	Vb(Aa),Vt	BVI	AE	F4,1,03	3

Exceptions

AE Register Range (AERRE)

Scalar Register Range (SRRE)

AE Unaligned Reference (AEURE)

LD – Load Vector Indexed Signed Double Word

LD.SD	Vb(Aa),Vt
LD.SD.T	Vb(Aa),Vt
LD.SD.F	Vb(Aa),Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1101	opc	Vb	Aa	Vt

Description

The instruction loads four bytes from memory to a destination vector register using a vector of indexes. The four bytes returned from memory are sign extended to eight bytes. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

Pseudo Code

```
aax = AregIdx(Aa);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            effa = A[aax] + VP[j].V[vbx][i];
            VP[j]. V[vtx][i] = SignExt64 ( MemLoad( effa<47:0>, 32 ) );
        } else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
LD.SD	Vb(Aa),Vt	BVI	AE	F4,1,06	0
LD.SD.T	Vb(Aa),Vt	BVI	AE	F4,1,06	2
LD.SD.F	Vb(Aa),Vt	BVI	AE	F4,1,06	3

Exceptions

AE Register Range (AERRE)

Scalar Register Range (SRRE)

AE Unaligned Reference (AEURE)

LD – Load Vector Mask Register

LD offset(Aa),VMt

31 30	29	28	27 22	21 12	11 6	5 0
00	0	0	opc	immed10	Aa	Vt

Description

The instruction loads the specified VM register from memory. The effective address must be aligned on an 8-byte boundary. The format of the VM bits within memory is implementation specific.

HC-1 Implementation Details

The instruction writes four bytes per function pipe to memory (a total of 128 bytes are written). The VPM, VL and VPL must be set such that 32 4-byte writes occur. The following table specifies the legal values of VL and VPL for the defined values of VPM. An AE Element Range Exception (AEERE) is signaled if VPM, VL and VPL have values other than those in the table. The table also shows values of VS and VPS that will result in the VM register being written to contiguous locations in memory (no exception is signaled for values of VS and VPS).

VPM	*VL*	*VPL*	*VS*	*VPS*
0 (Classical)	32	ignored	4	ignored
1 (Physical)	8	4	4	32
2 (Short Vector)	1	32	Ignored	4

Future product generations may change the number of function pipes, and or elements per function pipe. These changes would result in a different number of bytes being written to memory.

Pseudo Code

```
aax = AregIdx(Aa);
vmtx = VregIdx(VMt<5:0>);
effaBase = A[aax] + Offset(opc<1:0>);
VM[vmtx] = MemLoad( effa<47:0>, 1024 );
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
LD	offset(Aa),VMt	BVI	AE	F1,1,13	0

Exceptions

AE Register Range (AERRE)

AE Unaligned Reference (AEURE)

Scalar Register Range (SRRE)

AE Element Range (AEERE)

LT – Vector Compare Less Than Signed Quad Word Scalar

LT.SQ	Va,Sb,VMt
LT.SQ.T	Va,Sb,VMt
LT.SQ.F	Va,Sb,VMt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1100	opc	Sb	Va	VMt

Description

The instruction compares for less than a vector register and an S register using signed quad word data format. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The VMt instruction field specifies the VM register written by the instruction.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vmtx = VMregIdx(VMt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>){
            VP[j]. VM[vmtx]<i> = VP[j]. V[vax][i] < S[sbx];
        } else
            VP[j]. VM[vmtx]<i> = false;
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
LT.SQ	Va,Sb,VMt	BVI	AE	F3,1,3D	0
LT.SQ.T	Va,Sb,VMt	BVI	AE	F3,1,3D	2
LT.SQ.F	Va,Sb,VMt	BVI	AE	F3,1,3D	3

Exceptions

AE Register Range (AERRE)

Scalar Register Range (SRRE)

LT – Vector Compare Less Than Signed Quad Word

LT.SQ	Va,Vb,VMt
LT.SQ.T	Va,Vb,VMt
LT.SQ.F	Va,Vb,VMt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1101	opc	Vb	Va	VMt

Description

The instruction compares for less than two vector registers using signed quad word data format. The 'vm' field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The VMt instruction field specifies the VM register written by the instruction.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vmtx = VMregIdx(VMt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>){
            VP[j]. VM[vmtx]<i> = VP[j]. V[vax][i] < VP[j].V[vbx][i];
        } else
            VP[j]. VM[vmtx]<i> = false;
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
LT.SQ	Va,Vb,VMt	BVI	AE	F4,1,3D	0
LT.SQ.T	Va,Vb,VMt	BVI	AE	F4,1,3D	2
LT.SQ.F	Va,Vb,VMt	BVI	AE	F4,1,3D	3

Exceptions

AE Register Range (AERRE)

LT – Vector Compare Less Than Unsigned Quad Word Scalar

LT.UQ	Va,Sb,VMt
LT.UQ.T	Va,Sb,VMt
LT.UQ.F	Va,Sb,VMt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1100	opc	Sb	Va	VMt

Description

The instruction compares for less than a vector register and an S register using unsigned quad word data format. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The VMt instruction field specifies the VM register written by the instruction.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vmtx = VMregIdx(VMt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>){
            VP[j]. VM[vmtx]<i> = VP[j]. V[vax][i] < S[sbx];
        } else
            VP[j]. VM[vmtx]<i> = false;
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
LT.UQ	Va,Sb,VMt	BVI	AE	F3,1,3C	0
LT.UQ.T	Va,Sb,VMt	BVI	AE	F3,1,3C	2
LT.UQ.F	Va,Sb,VMt	BVI	AE	F3,1,3C	3

Exceptions

AE Register Range (AERRE)   Scalar Register Range (SRRE)

LT – Vector Compare Less Than Unsigned Quad Word

LT.UQ	Va,Vb,VMt
LT.UQ.T	Va,Vb,VMt
LT.UQ.F	Va,Vb,VMt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1101	opc	Vb	Va	VMt

Description

The instruction compares for less than two vector registers using unsigned quad word data format. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The VMt instruction field specifies the VM register written by the instruction.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vmtx = VMregIdx(VMt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>){
            VP[j]. VM[vmtx]<i> = VP[j]. V[vax][i] < VP[j].V[vbx][i];
        } else
            VP[j]. VM[vmtx]<i> = false;
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
LT.UQ	Va,Vb,VMt	BVI	AE	F4,1,3C	0
LT.UQ.T	Va,Vb,VMt	BVI	AE	F4,1,3C	2
LT.UQ.F	Va,Vb,VMt	BVI	AE	F4,1,3C	3

Exceptions

AE Register Range (AERRE)

MAX – Vector Maximum Integer

MAX.UQ	Va,Vb,Vt	MAX.SQ	Va,Vb,Vt
MAX. UQ.T	Va,Vb,Vt	MAX. SQ.T	Va,Vb,Vt
MAX. UQ.F	Va,Vb,Vt	MAX. SQ.F	Va,Vb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1101	opc	Vb	Va	Vt

Description

The instruction compares two vector registers using quad word integer arithmetic. The destination vector register contains the maximum of the source vector registers. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of the output vector register for masked elements is undefined.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j]. V[vtx] [i] = Max(VP[j]. V[vax][i], VP[j]. V[vbx][i]);
        else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
MAX.UQ	Va,Vb,Vt	BVI	AE	F4,1,2A	0
MAX. UQ.T	Va,Vb,Vt	BVI	AE	F4,1,2A	2
MAX. UQ.F	Va,Vb,Vt	BVI	AE	F4,1,2A	3
MAX.SQ	Va,Vb,Vt	BVI	AE	F4,1,2B	0
MAX. SQ.T	Va,Vb,Vt	BVI	AE	F4,1,2B	2
MAX. SQ.F	Va,Vb,Vt	BVI	AE	F4,1,2B	3

Exceptions

AE Register Range (AERRE)

Scalar Register Range (SRRE)

MAX – Vector Maximum Integer with Scalar

MAX.UQ	Va,Sb,Vt	MAX.SQ	Va,Sb,Vt
MAX. UQ.T	Va,Sb,Vt	MAX. SQ.T	Va,Sb,Vt
MAX. UQ.F	Va,Sb,Vt	MAX. SQ.F	Va,Sb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1100	opc	Sb	Va	Vt

Description

The instruction compares each element of a vector register to a scalar value using quad word integer arithmetic. The destination vector register contains the maximum of the source vector register and the scalar value. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of the output vector register for masked elements is undefined.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j]. V[vtx] [i] = Max(VP[j]. V[vax][i], S[sbx]);
        else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
MAX.UQ	Va,Sb,Vt	BVI	AE	F3,1,2A	0
MAX. UQ.T	Va,Sb,Vt	BVI	AE	F3,1,2A	2
MAX. UQ.F	Va,Sb,Vt	BVI	AE	F3,1,2A	3
MAX.SQ	Va,Sb,Vt	BVI	AE	F3,1,2B	0
MAX. SQ.T	Va,Sb,Vt	BVI	AE	F3,1,2B	2
MAX. SQ.F	Va,Sb,Vt	BVI	AE	F3,1,2B	3

Exceptions

AE Register Range (AERRE)

Scalar Register Range (SRRE)

MAXR – Vector Maximum Reduction Integer

MAXR.UQ	Va,Vt	MAXR.SQ	Va,Vt
MAXR.UQ.T	Va,Vt	MAXR.SQ.T	Va,Vt
MAXR.UQ.F	Va,Vt	MAXR.SQ.T	Va,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1100	opc	000000	Va	Vt

Description

The instruction performs a minimum reduction on a vector register. The result is one or more partial results per partition (implementation dependent). If more than one result is produced per partition then scalar instructions must be used to complete the reduction operation to a single value. The ***MOV REDcnt,At*** instruction can be used to obtain the number of partial reduction results produced per partition. The ***MOVR Va,Ab,St*** instruction must be used to obtain the reduction results with the A register value indicating the index of the partial result being accessed. The order that the vector elements are combined is implementation dependent. The 'vm' field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

If a load instruction with VS=0 was previously used to load the MAXR source vector register (Va), then the result of the reduction is undefined and an AEERE exception is issued. Note that the exception is issued independent of the number of instructions that separate the load with VS=0 and the reduction instruction.

Pseudo Code (example code produces one result per partition)

```
vax = VregIdx(Va);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    tmp = MinimumUnsignedQuadValue();
    for (i = 0; i < AEC.VL; i += 1)
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            tmp = Max(tmp, VP[j].V[vax][i]);
    VP[j]. V[vtx][0] = tmp;
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
MAXR.UQ	Va,Vt	BVI	AE	F3,1,4E	0
MAXR.UQ.T	Va,Vt	BVI	AE	F3,1,4E	2
MAXR.UQ.F	Va,Vt	BVI	AE	F3,1,4E	3
MAXR.SQ	Va,Vt	BVI	AE	F3,1,4F	0

MAXR.SQ.T	Va,Vt	BVI	AE	F3,1,4F	2
MAXR.SQ.F	Va,Vt	BVI	AE	F3,1,4F	3

Exceptions

AE Register Range (AERRE)

AE Element Range (AEERE)

MIN – Vector Minimum Integer

MIN.UQ	Va,Vb,Vt	MIN.SQ	Va,Vb,Vt
MIN.UQ.T	Va,Vb,Vt	MIN.SQ.T	Va,Vb,Vt
MIN.UQ.F	Va,Vb,Vt	MIN.SQ.F	Va,Vb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1101	opc	Vb	Va	Vt

Description

The instruction compares two vector registers using quad word integer arithmetic. The destination vector register contains the minimum of the source vector registers. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

The value of the output vector register for masked elements is undefined.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j]. V[vtx] [i] = Min(VP[j].V[vax][i], VP[j].V[vbx][i]);
        else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
MIN.UQ	Va,Vb,Vt	BVI	AE	F4,1,28	0
MIN. UQ.T	Va,Vb,Vt	BVI	AE	F4,1,28	2
MIN. UQ.F	Va,Vb,Vt	BVI	AE	F4,1,28	3
MIN.SQ	Va,Vb,Vt	BVI	AE	F4,1,29	0
MIN. SQ.T	Va,Vb,Vt	BVI	AE	F4,1,29	2
MIN. SQ.F	Va,Vb,Vt	BVI	AE	F4,1,29	3

Exceptions

AE Register Range (AERRE)

Scalar Register Range (SRRE)

MIN – Vector Minimum Integer with Scalar

MIN.UQ	Va,Sb,Vt	MIN.SQ	Va,Sb,Vt
MIN.UQ.T	Va,Sb,Vt	MIN.SQ.T	Va,Sb,Vt
MIN.UQ.F	Va,Sb,Vt	MIN.SQ.F	Va,Sb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1100	opc	Sb	Va	Vt

Description

The instruction compares each element of a vector register to a scalar value using quad word integer arithmetic. The destination vector register contains the minimum of the source vector register and the scalar value. The 'vm' field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

The value of the output vector register for masked elements is undefined.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1)
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j]. V[vtx] [i] = Min(VP[j].V[vax][i], S[sbx]);
        else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
MIN.UQ	Va,Sb,Vt	BVI	AE	F3,1,28	0
MIN. UQ.T	Va,Sb,Vt	BVI	AE	F3,1,28	2
MIN. UQ.F	Va,Sb,Vt	BVI	AE	F3,1,28	3
MIN.SQ	Va,Sb,Vt	BVI	AE	F3,1,29	0
MIN. SQ.T	Va,Sb,Vt	BVI	AE	F3,1,29	2
MIN. SQ.F	Va,Sb,Vt	BVI	AE	F3,1,29	3

Exceptions

AE Register Range (AERRE)

AE Integer Overflow (AEIOE)

Scalar Register Range (SRRE)

MINR – Vector Minimum Reduction Integer

MINR.UQ	Va,Vt	MINR.SQ	Va,Vt
MINR.UQ.T	Va,Vt	MINR.SQ.T	Va,Vt
MINR.UQ.F	Va,Vt	MINR.SQ.T	Va,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1100	opc	000000	Va	Vt

Description

The instruction performs a minimum reduction on a vector register. The result is one or more partial results per partition (implementation dependent). If more than one result is produced per partition then scalar instructions must be used to complete the reduction operation to a single value. The ***MOV REDcnt,At*** instruction can be used to obtain the number of partial reduction results produced per partition. The ***MOVR Va,Ab,St*** instruction must be used to obtain the reduction results with the A register value indicating the index of the partial result being accessed. The order that the vector elements are combined is implementation dependent. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

If a load instruction with VS=0 was previously used to load the MAXR source vector register (Va), then the result of the reduction is undefined and an AEERE exception is issued. Note that the exception is issued independent of the number of instructions that separate the load with VS=0 and the reduction instruction.

Pseudo Code (example code produces one result per partition)

```
vax = VregIdx(Va);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    tmp = MaximumUnsignedQuadValue();
    for (i = 0; i < AEC.VL; i += 1)
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            tmp = Min(tmp, VP[j].V[vax][i]);
    VP[j]. V[vtx][0] = tmp;
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
MINR.UQ	Va,Vt	BVI	AE	F3,1,4C	0
MINR.UQ.T	Va,Vt	BVI	AE	F3,1,4C	2
MINR.UQ.F	Va,Vt	BVI	AE	F3,1,4C	3
MINR.SQ	Va,Vt	BVI	AE	F3,1,4D	0

MINR.SQ.T	Va,Vt	BVI	AE	F3,1,4D	2
MINR.SQ.F	Va,Vt	BVI	AE	F3,1,4D	3

Exceptions

AE Register Range (AERRE)	AE Element Range (AEERE)

MOV – Absolute Address Register to Address Register

MOV AAa,At
MOV Aa,AAt

31 30	29	28 25	24 18	17 12	11 6	5 0
it	1	1100	opc	AAa / AAt	Aa	At

Description

The instruction copies the contents of a source address register to a destination address register. The instructions allow absolute addressing (I.e. ignores the window base) for either the source or destination address register.

Pseudo Code (example for MOV AAa,At)

```
aax<11:6> = AAa<5:0>;
aax<5:0> = Aa<5:0>;
atx = AregIdx(At);
A[atx] = A[aax];
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*
MOV	AAa,At	Scalar	A	F3,1E
MOV	Aa,AAt	Scalar	A	F3,1F

Exceptions

Scalar Register Range (SRRE)

MOV – Absolute Scalar Register to Scalar Register

MOV	SAa,St
MOV	Sa,SAt

31 30	29	28 25	24 18	17 12	11 6	5 0
it	**1**	**1100**	**opc**	**SAa / SAt**	**Sa**	**St**

Description

The instruction copies the contents of a source scalar register to a destination scalar register. The instructions allow absolute addressing (I.e. ignores the window base) for either the source or destination scalar register.

Pseudo Code (example for MOV SAa,St)

```
sax<11:6> = SAa<5:0>;
sax<5:0> = Sa<5:0>;
stx = SregIdx(St);
S[stx] = S[sax];
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
MOV	SAa,St	Scalar	S	F3,1,1E
MOV	Sa,SAt	Scalar	S	F3,1,1F

Exceptions

Scalar Register Range (SRRE)

MOV – Address Register to Scalar Register

MOV Aa,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	1	1100	opc	000000	Aa	St

Description

The instruction copies the contents of an address register to a scalar register.

Pseudo Code

```
aax = AregIdx(Aa);
stx = SregIdx(St);
S[stx] = A[aax];
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
MOV	Aa,St	Scalar	S	F3,1,1C

Exceptions

Scalar Register Range (SRRE)

MOV – Scalar Register to Address Register

MOV Sa,At

31 29	28 25	24 18	17 12	11 6	5 0
it	1100	opc	000000	Sa	At

Description

The instruction copies the contents of a scalar register to an address register.

Pseudo Code

```
sax = SregIdx(Sa);
atx = AregIdx(At);
A[atx] = S[sax];
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
MOV	Sa,At	Scalar	A	F3,1C

Exceptions

Scalar Register Range (SRRE)

MOV – Vector Register Move

MOV	Va,Vt
MOV.T	Va,Vt
MOV.F	Va,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1100	opc	000000	Va	Vt

Description

The instruction copies the values of the 64-bit source register to the 64-bit destination register. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

Pseudo Code

```
vmax = VMregIdx(WB.VMA);
vax = VregIdx(Va);
vtx = VregIdx(Vt);
for (j = 0; j < AEC.VPL; j += 1)
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j]. V[vtx][i] = VP[j].V [vax][i];
        else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
MOV	Va,Vt	BVI	AE	F3,1,09	0
MOV.T	Va,Vt	BVI	AE	F3,1,09	2
MOV.T	Va,Vt	BVI	AE	F3,1,09	3

Exceptions

AE Register Range (AERRE)

MOV – Move from AE Field or Register to A Register

MOV	VPM,At	MOV	Vcnt,At
MOV	VL,At	MOV	VMcnt,At
MOV	VS,At	MOV	VLmax,At
MOV	VPL,At	MOV	VPLmax,At
MOV	VPS,At	MOV	VPA,At
MOV	REDcnt,At		

31 30	29	28 24	23 18	17 12	11 6	5 0
00	if	11101	opc	000000	000000	At

Description

The instructions move a value from an A register to an AE register or field of a register.

The move VS and VPS instructions sign extend the lower 48-bit value to 64-bits before writing the value to the destination A register.

Pseudo Code

```
atx = AregIdx(At);
A[atx] = AE Register/Field;
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*
MOV	VPM,At	BVI	AE	F6,1,10
MOV	VL,At	BVI	AE	F6,1,11
MOV	VS,At	BVI	AE	F6,1,12
MOV	VPL,At	BVI	AE	F6,1,13
MOV	VPS,At	BVI	AE	F6,1,14
MOV	VLmax,At	BVI	AE	F6,1,18
MOV	VPLmax,At	BVI	AE	F6,1,19
MOV	VPA,At	BVI	AE	F6,1,1A
MOV	REDcnt,At	BVI	AE	F6,1,1B
MOV	Vcnt,At	BVI	AE	F6,1,1C
MOV	VMcnt,At	BVI	AE	F6,1,1D

Exceptions

Scalar Register Range (SRRE)

MOV – Move from AEC or AES Register to A Register

MOV.AEx	AEC,At	; where x=0,1,2,3
MOV	AEC,At	
MOV.AEx	AES,At	
MOV	AES,At	

31 30	29	28 24	23 18	17 12	11 6	5 0
ae	m	11101	opc	000000	000000	At

Description

The instruction moves a value from an AEC or AES register on one or more application engines to an A register. The instruction's *m* bit is used to determine if the move instruction uses the AEC.RIM mask field to select which AEs participate (m=1), or the instruction directly specifies the single AE that participates in the operation (m=0).

Note that the SPV and DPV personalities only support the masked read version of the instruction and the mask value is fixed at 0xF (all AEs perform the read and the responses are OR'ed together).

Pseudo Code (for AEC instruction)

```
atx = AregIdx(At);
if (m == 1) {
    A[atx] = 0;
    for (i = 0; i < 4; i += 1)
        if (AEC.RIM<i>)                 // RIM mask specifies zero or more AEs
            A[atx] |= AE[i].AEC;
} else
    A[atx] = AE[ae].AEC;                // ae field specifies single AE
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
MOV.AEx	AEC,At	ASP	AE	F6,0,16
MOV	AEC,At	BVI, ASP	AE	F6,1,16
MOV.AEx	AES,At	ASP	AE	F6,0,17
MOV	AES,At	BVI, ASP	AE	F6,1,17

Exceptions

Scalar Register Range (SRRE)

MOV – Move A Register to AE Field or Register

MOV	Aa,VPM	MOV	Aa,VPS
MOV	Aa,VL	MOV	Aa,VPL
MOV	Aa,VS	MOV	Aa,VPA

31 30	29	28 24	23 18	17 12	11 6	5 0
00	1	11100	opc	000000	Aa	000000

Description

The instruction moves a value from an A register to an AE register or field of a register.

A move to VPM range checks the value written. A value outside the range of 0-2 is forced to two and the AES.AEIIE (Integer Invalid Operand) exception is set. Additionally, a move to VPM will range check the values in the AEC fields VL, VPL and VPA. Values outside the valid range will be reduced to the larges valid value.

Pseudo Code

```
aax = AregIdx(Aa);
AE Register/Field = A[aax];
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
MOV	Aa,VPM	BVI	AE	F5,1,10
MOV	Aa,VL	BVI	AE	F5,1,11
MOV	Aa,VS	BVI	AE	F5,1,12
MOV	Aa,VPL	BVI	AE	F5,1,13
MOV	Aa,VPS	BVI	AE	F5,1,14
MOV	Aa,VPA	BVI	AE	F5,1,1A

Exceptions

Scalar Register Range (SRRE)

MOV – Move A Register to ASP AEC Field

MOV	Aa,RIM
MOV	Aa,WIM
MOV	Aa,CIM

31 30	29	28 24	23 18	17 12	11 6	5 0
00	1	11100	opc	000000	Aa	000000

Description

The instruction moves the low order 4-bits of the specified A register to the target AEC register mask field on all application engines.

Pseudo Code (RIM example)

```
aax = AregIdx(Aa);
for (i=0; i<4; i+=1)
    AE[i].AEC.RIM = A[aax]<3:0>;
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*
MOV	Aa,RIM	ASP	AE	F5,1,1C
MOV	Aa,WIM	ASP	AE	F5,1,1D
MOV	Aa,CIM	ASP	AE	F5,1,1E

Exceptions

Scalar Register Range (SRRE)

MOV – Move A Register to AEC or AES Register

```
MOV.AEx    Aa,AEC   ; where x=0,1,2,3
MOV        Aa,AEC
MOV.AEx    Aa,AES
MOV        Aa,AES
```

31 30	29	28 24	23 18	17 12	11 6	5 0
ae	m	11100	opc	000000	Aa	000000

Description

The instruction moves an A register value to an AEC or AES register on one or more application engines. The instruction's *m* bit is used to determine if the move instruction uses the AEC.WIM mask field to select which AEs participate (m=1), or the instruction directly specifies the single AE that participates in the operation (m=0).

Note that the SPV and DPV personalities only support the masked write version of the instruction and the mask value is fixed at 0xF (all AEs perform the write).

For SPV and DPV personalities, a move to AEC performs range checking on the value written. A VPM value outside the range of 0-2 is forced to two and the AES.AEIIE (Integer Invalid Operand) exception is set. Additionally, the values in the AEC fields VL, VPL and VPA are range checked. Values outside the valid range will be reduced to the largest valid value.

Pseudo Code (for AEC instruction)

```
aax = AregIdx(Aa);
if (m == 1) {
    for (i = 0; i < 4; i += 1)
        if (WIM<i>)              // WIM mask specifies zero or more AEs
            AE[i].AEC = A[aax];
} else
    AE[ae].AEC = A[aax];         // ae field specifies single AE
```

Instruction Encoding

Instruction		ISA	Type	Encoding
MOV.AEx	Aa,AEC	ASP	AE	F5,0,16
MOV	Aa,AEC	BVI, ASP	AE	F5,1,16
MOV.AEx	Aa,AES	ASP	AE	F5,0,17
MOV	Aa,AES	BVI, ASP	AE	F5,1,17

Exceptions

Scalar Register Range (SRRE)

MOV – Move A Register to CCX, PIP, CPC, CPS or WB Register

MOV	Aa,CCX
MOV	Aa,PIP
MOV	Aa,CPC
MOV	Aa,CPS
MOV	Aa,WB

31 29	28 24	23 18	17 12	11 6	5 0
it	11100	opc	000000	Aa	000000

Description

The instruction moves a value from an A register to one of the registers CCX, PIP, CPC, CPS or WB.

The MOV Aa,CPS instruction moves an A register to the CPS register. The CPS register includes the Exception Status field, *SE*. The *SE* field value loaded from the A register includes exceptions generated from either instruction in the instruction bundle that includes the MOV Aa,CPS instruction.

Pseudo Code (example for PIP register)

```
aax = AregIdx(Aa);
PIP = A[aax];
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
MOV	Aa,PIP	Scalar	A	F5,10
MOV	Aa,CPC	Scalar	A	F5,11
MOV	Aa,CPS	Scalar	A	F5,12
MOV	Aa,WB	Scalar	A	F5,15
MOV	Aa,CCX	Scalar	A	F5,18

Exceptions

Scalar Register Range (SRRE)

MOV – Move CCX, PIP, CPC, CPS, CIT, CDS or WB to an A Reg.

MOV	CCX,At
MOV	PIP,At
MOV	CPC,At
MOV	CPS,At
MOV	CIT,At
MOV	CDS,At
MOV	WB,At

31 29	28 24	23 18	17 12	11 6	5 0
it	11101	opc	000000	000000	At

Description

The instructions move a value from one of the registers CCX, CIT, CDS, PIP, CPC, CPS or WB to an A register.

The MOV CPS, At instruction moves the CPS register to an A register. The CPS register includes the Exception Status field, *SE*. The *SE* field value moved to the A register does not include exceptions generated from either instruction in the instruction bundle that includes the MOV CPS, At instruction.

Pseudo Code (example for PIP register)

```
atx = AregIdx(At);
A[atx] = PIP;
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
MOV	PIP,At	Scalar	A	F6,10
MOV	CPC,At	Scalar	A	F6,11
MOV	CPS,At	Scalar	A	F6,12
MOV	WB,At	Scalar	A	F6,15
MOV	CIT,At	Scalar	A	F6,16
MOV	CDS,At	Scalar	A	F6,17
MOV	CCX,At	Scalar	A	F6,18

Exceptions

Scalar Register Range (SRRE)

MOV – Move A Register to CRSL or CRSU Register

MOV Aa,Immed,CRSL

MOV Aa,Immed,CRSU

31 29	28 24	23 18	17 12	11 6	5 0
it	11100	opc	Immed	Aa	000000

Description

The instruction moves a value from an A register to the lower or upper quad word of a call return stack entry. The specific entry is selected using the instruction's immediate field.

Pseudo Code (example for CRSL register)

```
aax = AregIdx(Aa);
CRS[Immed<2:0>].RIP = A[aax];
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
MOV	Aa,Immed,CRSL	Scalar	A	F5,13
MOV	Aa,Immed,CRSU	Scalar	A	F5,14

Exceptions

Scalar Register Range (SRRE)

MOV – Move CRSL or CRSU Register to A Register

MOV CRSL,Immed,At

MOV CRSU,Immed,At

31 29	28 24	23 18	17 12	11 6	5 0
it	11101	opc	Immed	000000	At

Description

The instruction moves a value from the lower or upper quad word of a call return stack entry to an A register. The specific entry is selected using the instruction’s immediate field.

Pseudo Code (example for CRSL register)

```
atx = AregIdx(At);
A[atx] = CRS[Immed<2:0>].RIP;
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*
MOV	CRSL,Immed,At	Scalar	A	F6,13
MOV	CRSU,Immed,At	Scalar	A	F6,14

Exceptions

Scalar Register Range (SRRE)

MOV – Move Immediate to AE Field or Register

MOV	Immed,VPM	MOV	Immed,VPS
MOV	Immed,VL	MOV	Immed,VPL
MOV	Immed,VS	MOV	Immed,VPA

31 30	29	28 24	23 18	17 0
00	if	11110	opc	Immed18

Description

The instruction moves an 18-bit immediate value to an AE register or field of a register. The low order bits of the immediate value are used for fields that are less than 18-bits wide. The immediate is sign extended for registers that are wider than 18-bits. No exceptions are generated.

A move to VPM range checks the value written. A value outside the range of 0-2 is forced to two and the AES.AEIIE (Integer Invalid Operand) exception is set. Additionally, a move to VPM will range check the values in the AEC fields VL, VPL and VPA. Values outside the valid range will be reduced to the largest valid value.

Pseudo Code

```
AE Register/Field = Immed18;
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
MOV	Immed,VPM	BVI	AE	F7,1,10
MOV	Immed,VL	BVI	AE	F7,1,11
MOV	Immed,VS	BVI	AE	F7,1,12
MOV	Immed,VPL	BVI	AE	F7,1,13
MOV	Immed,VPS	BVI	AE	F7,1,14
MOV	Immed,VPA	BVI	AE	F7,1,1A

Exceptions

None

MOV – Move Immediate to ASP AEC Field

MOV Immed,RIM

MOV Immed,WIM

MOV Immed,CIM

31 30	29	28 24	23 18	17 0
00	1	11110	opc	Immed18

Description

The instruction moves the low order 4-bits of the 18-bit immediate value to the target AEC register mask field on all application engines. No exceptions are generated.

Pseudo Code (RIM example)

```
for (i=0; i<4; i+=1)
    AE[i].AEC.RIM = Immed18<3:0>;
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*
MOV	Immed,RIM	ASP	AE	F7,1,1C
MOV	Immed,WIM	ASP	AE	F7,1,1D
MOV	Immed,CIM	ASP	AE	F7,1,1E

Exceptions

None

MOV – Move Scalar to Vector Register Element

MOV Sa,Ab,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
00	1	1101	opc	Ab	Sa	Vt

Description

The instruction moves a scalar register value to a vector register element. The vector specific register element is specified by the contents of an A register.

The element index is checked to be within the valid range specified by the AEC field values VPM, VPL, and VL. An invalid element index sets the AES AEERE exception bit and no vector register element is modified.

Pseudo Code

```
sax = SregIdx(Sa);
abx = AregIdx(Ab);
vtx = VregIdx(Vt);
VP[VPA].V[vtx][ A[abx] ] = S[sax];
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*
MOV	Sa,Ab,Vt	BVI	AE	F4,1,51

Exceptions

AE Register Range (AERRE)

AE Element Range (AEERE)

Scalar Register Range (SRRE)

MOV – Move Vector Register Element to Scalar Register

MOV Va,Ab,St

31 30	29	28 25	24 18	17 12	11 6	5 0
00	1	1101	opc	Ab	Va	St

Description

The instruction moves a vector register element to a scalar register. The vector element is specified by the contents of an A register and the partition is specified by the contents of the AEC register's VPA field.

The element index is checked to be within the valid range specified by the AEC field values VPM, VPL, and VL. An invalid element index sets the AES AEERE exception bit and stores the value zero to the destination S register.

Pseudo Code

```
vax = VregIdx(Va);
abx = AregIdx(Ab);
stx = SregIdx(St);
S[stx] = VP[VPA].V[vax][ A[abx]<11:0> ];
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
MOV	Va,Ab,St	BVI	AE	F4,1,61

Exceptions

AE Register Range (AERRE) AE Element Range (AEERE)

Scalar Register Range (SRRE)

MOV – Move Acnt, Scnt or CRScnt to an A Register

MOV	Acnt,At
MOV	Scnt,At
MOV	CRScnt,At

31 30	29	28 24	23 18	17 12	11 6	5 0
it	0	11101	opc	000000	000000	At

Description

The instruction moves the Acnt, Scnt, or CRScnt value to an A register. The Acnt is the number implementation defined A registers. The Scnt is the number of S registers. The CRScnt is the number of call / return stack entries.

Pseudo Code (MOV Scnt,At instruction)

```
atx = AregIdx(At);
A[atx] = Scnt;
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*
MOV	Acnt,At	Scalar	A	F6,1C
MOV	Scnt,At	Scalar	A	F6,1D
MOV	CRScnt,At	Scalar	A	F6,1E

Exceptions

Scalar Register Range (SRRE)

MOV – Move AEGcnt Value to A Register

MOV.AEx AEGcnt,At ; where x=0,1,2,3

31 30	29	28 24	23 18	17 12	11 6	5 0
ae	0	11101	opc	000000	000000	At

Description

The instruction moves an AEGcnt value to an A register. The instruction uses the *ae* field to specify which AE is to respond.

Pseudo Code

```
atx = AregIdx(At);
A[atx] = AE[ae].AEGcnt;
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***ae***
MOV.AE0	AEGcnt,At	ASP	AE	F6,0,1D	0
MOV.AE1	AEGcnt,At	ASP	AE	F6,0,1D	1
MOV.AE2	AEGcnt,At	ASP	AE	F6,0,1D	2
MOV.AE3	AEGcnt,At	ASP	AE	F6,0,1D	3

Exceptions

Scalar Register Range (SRRE)

MOV – Move AEG Element to A Reg. with Immed. Index

MOV.AEx AEG,Immed,At ; where x=0,1,2,3

MOV AEG,Immed,At

31 30	29	28 24	23 18	17 12	11 6	5 0
ae	m	11101	opc	Immed12<11:6>	Immed12<5:0>	At

Description

The instruction moves an AEG register element to an address register. The AEG register element is specified by a 12-bit immediate value. The instruction's *m* bit is used to determine if the move instruction uses the AEC.RIM mask field to select which AEs participate (m=1), or the instruction directly specifies the single AE that participates in the operation (m=0).The result of each responding application engine is OR'ed together as the final result.

A scalar element range exception (SERE) is set if the immediate value is greater or equal to AEGcnt and the value zero is written to the destination A register.

Pseudo Code

```
atx = AregIdx(At);
if (m == 1) {
    A[atx] = 0;
    for (i = 0; i < 4; i += 1)
        if (AEC.RIM[i])                      // RIM mask specifies zero or more AEs
            A[atx] |= AE[i].AEG[ Immed12 ];
} else
    A[atx] = AE[ae].AEG[ Immed12  ];         // ae field specifies single AE
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*	*ae*
MOV.AE0	AEG,Immed,At	ASP	AE	F6,0,1C	0
MOV.AE1	AEG,Immed,At	ASP	AE	F6,0,1C	1
MOV.AE2	AEG,Immed,At	ASP	AE	F6,0,1C	2
MOV.AE3	AEG,Immed,At	ASP	AE	F6,0,1C	3
MOV	AEG,Immed,At	ASP	AE	F6,1,1C	0

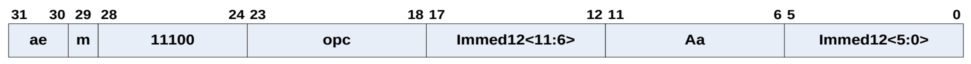
Exceptions

Scalar Register Range (SRRE)

Scalar Element Range (SERE)

MOV – Move A-Reg. to AEG Element with Immed. Index

```
MOV.AEx    Aa,Immed,AEG    ;  where x=0,1,2,3
MOV        Aa,Immed,AEG
```



Description

The instruction moves an address register value to an AEG register element on zero or more application engines. The instruction's *m* bit is used to determine if the move instruction uses the AEC.WIM mask field to select which AEs participate (m=1), or the instruction directly specifies the single AE that participates in the operation (m=0). The specific AEG register element is specified by a 12-bit immediate value.

A scalar element range exception (SERE) is set if the immediate value is greater or equal to AEGcnt and a write to an AEG register does not occur.

Pseudo Code

```
aax = AregIdx(Aa);
if (m == 1) {
    for (i=0; i<4; i+=1)
        if (AEC.WIM[i])                        // WIM mask specifies zero or more AEs
            AE[i].AEG[ Immed12 ] = A[aax];
} else
    AE[ae].AEG[ Immed12  ] = A[aax];          // ae field specifies single AE
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***ae***
MOV.AE0	Aa,Immed,AEG	ASP	AE	F5,0,18	0
MOV.AE1	Aa,Immed,AEG	ASP	AE	F5,0,18	1
MOV.AE2	Aa,Immed,AEG	ASP	AE	F5,0,18	2
MOV.AE3	Aa,Immed,AEG	ASP	AE	F5,0,18	3
MOV	Aa,Immed,AEG	ASP	AE	F5,1,18	0

Exceptions

Scalar Register Range (SRRE)   Scalar Element Range (SERE)

MOV – Move AEG Element to Scalar with Immed. Index

```
MOV.AEx    AEG,Immed,St    ; where x=0,1,2,3
MOV        AEG,Immed,St
```

31 30	29	28 25	24 18	17 12	11 6	5 0
ae	m	1101	opc	Immed12<11:6>	Immed12<5:0>	St

Description

The instruction moves an AEG element value to a scalar register. The AEG register element is specified by a 12-bit immediate value. The instruction's *m* bit is used to determine if the move instruction uses the AEC.RIM mask field to select which AEs participate (m=1), or the instruction directly specifies the single AE that participates in the operation (m=0). The results of all participating AEs are OR'ed together as the result written to the S register.

A scalar element range exception (SERE) is set if the immediate value is greater or equal to AEGcnt and the value zero is written to the destination S register.

Pseudo Code

```
stx = SregIdx(St);
if (m == 1) {
    S[stx] = 0;
    for (i = 0; i < 4; i += 1)
        if (AEC.RIM[i])                    // RIM mask specifies zero or more AEs
            S[stx] |= AE[i].AEG[ Immed12 ];
} else
    S[stx] = AE[ae].AEG[ Immed12  ];       // ae field specifies single AE
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*	*ae*
MOV.AE0	AEG,Immed,St	ASP	AE	F4,0,70	0
MOV.AE1	AEG,Immed,St	ASP	AE	F4,0,70	1
MOV.AE2	AEG,Immed,St	ASP	AE	F4,0,70	2
MOV.AE3	AEG,Immed,St	ASP	AE	F4,0,70	3
MOV	AEG,Immed,St	ASP	AE	F4,1,70	0

Exceptions

Scalar Register Range (SRRE)   Scalar Element Range (SERE)

MOV – Move Scalar to AEG Element with Immed. Index

```
MOV.AEx    Sa,Immed,AEG    ; where x=0,1,2,3
MOV        Sa,Immed,AEG
```

31 30	29	28 25	24 18	17 12	11 6	5 0
ae	m	1101	opc	Immed12<11:6>	Sa	Immed12<5:0>

Description

The instruction moves a scalar register value to an AEG register element on zero or more application engines. The instruction's *m* bit is used to determine if the move instruction uses the AEC.WIM mask field to select which AEs participate (m=1), or the instruction directly specifies the single AE that participates in the operation (m=0). The specific AEG register element is specified by a 12-bit immediate value.

A scalar element range exception (SERE) is set if the immediate value is greater or equal to AEGcnt and a write to an AEG register does not occur.

Pseudo Code

```
sax = SregIdx(Sa);
if (m == 1) {
    for (i=0; i<4; i+=1)
        if (AEC.WIM[i])                       // WIM mask specifies zero or more AEs
            AE[i].AEG[ Immed12 ] = S[sax];
} else
    AE[ae].AEG[ Immed12 ] = S[sax];           // ae field specifies single AE
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*	*ae*
MOV.AE0	Sa,Immed,AEG	ASP	AE	F5,0,20	0
MOV.AE1	Sa,Immed,AEG	ASP	AE	F5,0,20	1
MOV.AE2	Sa,Immed,AEG	ASP	AE	F5,0,20	2
MOV.AE3	Sa,Immed,AEG	ASP	AE	F5,0,20	3
MOV	Sa,Immed,AEG	ASP	AE	F5,1,20	0

Exceptions

Scalar Register Range (SRRE)   Scalar Element Range (SERE)

MOV – Move Scalar to AEG Register Element

MOV.AEx Sa,Ab,AEG ; where x=0,1,2,3

MOV Sa,Ab,AEG

31 30	29	28 25	24 18	17 12	11 6	5 0
ae	m	1101	opc	Ab	Sa	000000

Description

The instruction moves a scalar register value to an AEG register element on zero or more application engines. The instruction's *m* bit is used to determine if the move instruction uses the AEC.WIM mask field to select which AEs participate (m=1), or the instruction directly specifies the single AE that participates in the operation (m=0). The AEG element index is the least significant 18-bits of the A register value.

A scalar element range exception (SERE) is set if the A register value is greater or equal to AEGcnt and a write to an AEG register does not occur.

Pseudo Code

```
sax = SregIdx(Sa);
abx = AregIdx(Ab);
if (m == 1) {
    for (i=0; i<4; i+=1)
        if (AEC.WIM[i])                    // WIM mask specifies zero or more AEs
            AE[i].AEG[ abx ] = S[sax];
} else
    AE[ae].AEG[ abx ] = S[sax];            // ae field specifies single AE
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***ae***
MOV.AE0	Sa,Ab,AEG	ASP	AE	F4,0,40	0
MOV.AE1	Sa,Ab,AEG	ASP	AE	F4,0,40	1
MOV.AE2	Sa,Ab,AEG	ASP	AE	F4,0,40	2
MOV.AE3	Sa,Ab,AEG	ASP	AE	F4,0,40	3
MOV	Sa,Ab,AEG	ASP	AE	F4,1,40	0

Exceptions

Scalar Register Range (SRRE) Scalar Element Range (SERE)

MOV – Move AEG Register Element to Scalar

MOV.AEx	AEG,Ab,St	; where x=0,1,2,3
MOV	AEG,Ab,St	

31 30	29	28 25	24 18	17 12	11 6	5 0
ae	m	1101	opc	Ab	000000	St

Description

The instruction moves an AEG element value to a scalar register. The AEG register element is specified by the contents of an A register. The instruction's *m* bit is used to determine if the move instruction uses the AEC.RIM mask field to select which AEs participate (m=1), or the instruction directly specifies the single AE that participates in the operation (m=0). The results of all participating AEs are OR'ed together as the result written to the S register.

A scalar element range exception (SERE) is set if the A register value is greater or equal to AEGcnt and the value zero is written to the destination S register.

Pseudo Code

```
abx = AregIdx(Ab);
stx = SregIdx(St);
if (m == 1) {
    S[stx] = 0;
    for (i = 0; i < 4; i += 1)
        if (AEC.RIM[i])                         // RIM mask specifies zero or more AEs
            S[stx] |= AE[i].AEG[ A[abx] ];
} else
    S[stx] = AE[ae].AEG[ A[abx] ];              // ae field specifies single AE
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***ae***
MOV.AE0	AEG,Ab,St	ASP	AE	F4,0,68	0
MOV.AE1	AEG,Ab,St	ASP	AE	F4,0,68	1
MOV.AE2	AEG,Ab,St	ASP	AE	F4,0,68	2
MOV.AE3	AEG,Ab,St	ASP	AE	F4,0,68	3
MOV	AEG,Ab,St	ASP	AE	F4,1,68	0

Exceptions

Scalar Register Range (SRRE) Scalar Element Range (SERE)

MOV – Vector Register Move Double Word

MOV.DW	Va,Vt
MOV.DW.T	Va,Vt
MOV.DW.F	Va,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1100	opc	000000	Va	Vt

Description

The instruction copies the values of the 32-bit source register to the 32-bit destination register. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

Pseudo Code

```
vax = VregIdx(Va<4:0>);
vtx = VregIdx(Vt<4:0>);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1)
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j]. V32[Vt<5>][vtx][i] = VP[j].V32[Va<5>][vax][i];
        else
            VP[j]. V32[Vt<5>][vtx][i] = UndefinedValue();
    }
```

Instruction Encoding

Instruction	*Type*	*Encoding*	*VM*
MOV.DW	AE	F3,1,08	0
MOV.DW.T	AE	F3,1,08	2
MOV.DW.F	AE	F3,1,08	3

Exceptions

AE Register Range (AERRE)

MOV – Move Scalar to 32-Bit Vector Register Element

MOV.DW Sa,Ab,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
00	1	1101	opc	Ab	Sa	Vt

Description

The instruction moves a scalar register value to a 32-bit vector register element. The vector specific register element is specified by the contents of an A register.

The element index is checked to be within the valid range specified by the AEC field values VPM, VPL, and VL. An invalid element index sets the AES AEERE exception bit and no vector register element is modified.

Pseudo Code

```
sax = SregIdx(Sa);
abx = AregIdx(Ab);
vtx = VregIdx(Vt<4:0>);
VP[VPA].V32[Vt<5>][vtx][ A[abx] ] = S[sax]<31:0>;
```

Instruction Encoding

Instruction	***Type***	***Encoding***
MOV.DW	AE	F4,1,50

Exceptions

AE Register Range (AERRE)

Scalar Register Range (SRRE)

MOV – Move 32-bit Vector Register Element to Scalar

MOV.DW Va,Ab,St

31 30	29	28 25	24 18	17 12	11 6	5 0
00	1	1101	opc	Ab	Va	St

Description

The instruction moves a 32-bit vector register element to a scalar register. The vector element is specified by the contents of an A register and the partition is specified by the contents of the AEC register's VPA field.

The element index is checked to be within the valid range specified by the AEC field values VPM, VPL, and VL. An invalid element index sets the AES AEERE exception bit and stores the value zero to the destination S register.

Pseudo Code

```
vax = VregIdx(Va<4:0>);
abx = AregIdx(Ab);
stx = SregIdx(St);
S[stx] = VP[VPA].V32[Va<5>][vax][ A[abx]<11:0> ];
```

Instruction Encoding

Instruction	*Type*	*Encoding*
MOV.DW	AE	F4,1,60

Exceptions

AE Register Range (AERRE)	AE Element Range (AEERE)
Scalar Register Range (SRRE)	

MOVR – Move Reduction Result to Scalar

MOVR Va,Ab,St

31 30	29	28 25	24 18	17 12	11 6	5 0
00	1	1101	opc	Ab	Va	St

Description

The instruction moves the result(s) of a 64-bit vector reduction operation to a scalar register. Reduction operations may reduce a vector of elements to a single value per partition, or to multiple values per partition (implementation dependent). The Ab parameter indicates which partial reduction result is being requested. The VPA field of the AEC register specifies which partition is being accessed. The instruction ***MOV RED_CNT,At*** can be used to obtain the number of partial results a reduction operation produces. Each result can then be obtained using a loop or optimized straight line code.

Pseudo Code

```
vax = VregIdx(Va);
abx = AregIdx(Ab);
stx = SregIdx(St);
S[stx] = VP[VPA].V[vax][ ReductionMapFunction( A[abx] ) ];
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*
MOVR	Va,Ab,St	BVI	AE	F4,1,63

Exceptions

AE Register Range (AERRE)

AE Element Range (AEERE)

Scalar Register Range (SRRE)

MOVR – Move 32-Bit Reduction Result to Scalar

MOVR.DW Va,Ab,St

31 30	29	28 25	24 18	17 12	11 6	5 0
00	1	1101	opc	Ab	Va	St

Description

The instruction moves the result(s) of a double word vector reduction operation to a scalar register. Reduction operations may reduce a vector of elements to a single value per partition, or to multiple values per partition (implementation dependent). The Ab parameter indicates which partial reduction result is being requested. The VPA field of the AEC register specifies which partition is being accessed. The instruction ***MOV RED_CNT,At*** can be used to obtain the number of partial results a reduction operation produces. Each result can then be obtained using a loop or optimized straight line code.

Pseudo Code

```
vax = VregIdx(Va<4:0>);
abx = AregIdx(Ab);
stx = SregIdx(St);
S[stx]<63:32> = 0;
S[stx]<31:0> = VP[VPA].V32[Va<5>][vax][ ReductionMapFunction( A[abx] ) ];
```

Instruction Encoding

Instruction	*Type*	*Encoding*
MOVR.DW	AE	F4,1,62

Exceptions

AE Register Range (AERRE)

AE Element Range (AEERE)

Scalar Register Range (SRRE)

MUL – Address Multiply Integer with Immediate

MUL.UQ	Aa,Immed,At
MUL.SQ	Aa,Immed,At

31 29	28	27 22	21 12	11 6	5 0
it	**0**	**opc**	**immed10**	**Aa**	**At**

Description

The instruction performs multiplication between an address register and an immediate value using unsigned or signed integer arithmetic. The immediate value is either the zero or signed extended 10-bit Immed10 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code (example for MUL.SQ)

```
aax = AregIdx(Aa);
atx = AregIdx(At);
A[atx] = A[aax] * Simmed10();
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
MUL.UQ	Aa,Immed,At	Scalar	A	F1,34
MUL.SQ	Aa,Immed,At	Scalar	A	F1,35

Exceptions

Scalar Register Range (SRRE)

Scalar Integer Overflow (SIOE)

MUL – Scalar Multiply Integer with Immediate

MUL.UQ	Sa,Immed,St
MUL.SQ	Sa,Immed,St

31 30	29	28	27 22	21 12	11 6	5 0
it	1	0	opc	immed10	Sa	St

Description

The instruction performs multiplication between a scalar register and an immediate value using unsigned or signed integer arithmetic. The immediate value is either the zero or signed extended 10-bit Immed10 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code (example for MUL.SQ)

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx] = S[sax] * Simmed10();
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
MUL.UQ	Sa,Immed,St	Scalar	S	F1,1,34
MUL.SQ	Sa,Immed,St	Scalar	S	F1,1,35

Exceptions

Scalar Register Range (SRRE)

Scalar Integer Overflow (SIOE)

MUL – Scalar Multiply Float Double with Immediate

MUL.FD Sa,Immed,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	0	1100	opc	Immed6	Sa	St

Description

The instruction performs multiplication between a scalar register and an immediate value using float double arithmetic. The immediate value is either the zero extended 6-bit Immed6 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx] = S[sax] * Uimmed6();
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
MUL.FD	Sa,Immed,St	Scalar	S	F3,0,35

Exceptions

Scalar Register Range (SRRE)	Scalar Float Invalid Operand (SFIE)
Scalar Float Overflow (SFOE)	Scalar Float Underflow (SFUE)

MUL – Scalar Multiply Float Single with Immediate

MUL.FS Sa,Immed,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	0	1100	opc	Immed6	Sa	St

Description

The instruction performs multiplication between a scalar register and an immediate value using float single arithmetic. The immediate value is either the zero extended 6-bit Immed6 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx]<63:0> = 0;
S[stx]<31:0> = S[sax]<31:0> * Uimmed6()<31:0>;
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
MUL.FS	Sa,Immed,St	Scalar	S	F3,0,34

Exceptions

Scalar Register Range (SRRE)

Scalar Float Invalid Operand (SFIE)

Scalar Float Overflow (SFOE)

Scalar Float Underflow (SFUE)

MUL – Address Multiply Integer

MUL.UQ Aa,Ab,At
MUL.SQ Aa,Ab,At

31 29	28 25	24 18	17 12	11 6	5 0
it	1101	opc	Ab	Aa	At

Description

The instruction performs multiplication on two address register values using signed or unsigned integer arithmetic.

Pseudo Code

```
aax = AregIdx(Aa);
abx = AregIdx(Ab);
atx = AregIdx(At);
A[atx] = A[aax] * A[abx];
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
MUL.UQ	Aa,Ab,At	Scalar	A	F4,34
MUL.SQ	Aa,Ab,At	Scalar	A	F4,35

Exceptions

Scalar Register Range (SRRE)

MUL – Scalar Multiply Integer

MUL.UQ	Sa,Sb,St
MUL.SQ	Sa,Sb,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	1	1101	opc	Sb	Sa	St

Description

The instruction performs multiplication on two scalar register values using signed or unsigned integer arithmetic.

Pseudo Code

```
sax = SregIdx(Sa);
sbx = SregIdx(Sb);
stx = SregIdx(St);
S[stx] = S[sax] * S[sbx];
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
MUL.UQ	Sa,Sb,St	Scalar	S	F4,1,34
MUL.SQ	Sa,Sb,St	Scalar	S	F4,1,35

Exceptions

Scalar Register Range (SRRE)

MUL – Scalar Multiply Float Double

MUL.FD Sa,Sb,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	0	1101	opc	Sb	Sa	St

Description

The instruction performs multiplication on two scalar register values using float double arithmetic.

Pseudo Code

```
sax = SregIdx(Sa);
sbx = SregIdx(Sb);
stx = SregIdx(St);
S[stx] = S[sax] * S[sbx];
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
MUL.FD	Sa,Sb,St	Scalar	S	F4,0,35

Exceptions

Scalar Register Range (SRRE)	Scalar Float Invalid Operand (SFIE)
Scalar Float Overflow (SFOE)	Scalar Float Underflow (SFUE)

MUL – Scalar Multiply Float Single

MUL.FS Sa,Sb,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	0	1101	opc	Sb	Sa	St

Description

The instruction performs multiplication on two scalar register values using float single arithmetic.

Pseudo Code

```
sax = SregIdx(Sa);
sbx = SregIdx(Sb);
stx = SregIdx(St);
S[stx]<63:32> = 0;
S[stx]<31:0> = S[sax]<31:0> * S[sbx]<31:0>;
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
MUL.FS	Sa,Sb,St	Scalar	S	F4,0,34

Exceptions

Scalar Register Range (SRRE)

Scalar Float Invalid Operand (SFIE)

Scalar Float Overflow (SFOE)

Scalar Float Underflow (SFUE)

MUL – Vector Multiply 48x48 Signed Quad Word Scalar

MUL.SQ	Va,Sb,Vt
MUL.SQ.T	Va,Sb,Vt
MUL.SQ.F	Va,Sb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	if	1100	opc	Sb	Va	Vt

Description

The instruction multiplies a 64-bit vector register to a 64-bit scalar value using signed quad word integer data format. The multiply operation is performed using the least significant 48-bits of the input registers and produces a 48-bit result. An invalid input operand exception is issued if an input operand value exceeds 48 bits. An integer overflow exception is issued if the result exceeds 48 bits. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of the output vector register for masked elements is undefined and exceptions are not recorded for masked elements.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j]. V[vtx] [i] = VP[j]. V [vax][i] * S[sbx];
        else
            VP[j]. V[vtx] [i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
MUL.SQ	Va,Sb,Vt	BVI	AE	F3,1,35	0
MUL.SQ.T	Va,Sb,Vt	BVI	AE	F3,1,35	2
MUL.SQ.F	Va,Sb,Vt	BVI	AE	F3,1,35	3

Exceptions

AE Integer Overflow (AEIOE)	AE Register Range (AERRE)
Scalar Register Range (SRRE)	AE Invalid Input (AEIIE)

MUL – Vector Multiply 48x48 Signed Quad Word

MUL.SQ	Va,Vb,Vt
MUL.SQ.T	Va,Vb,Vt
MUL.SQ.F	Va,Vb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1101	opc	Vb	Va	Vt

Description

The instruction multiplies two 64-bit vector register using signed quad word integer data format. The multiply operation is performed using the least significant 48-bits of the input registers and produces a 48-bit result. An invalid input operand exception is issued if an input operand value exceeds 48 bits. An integer overflow exception is issued if the result exceeds 48 bits. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of the output vector register for masked elements is undefined and exceptions are not recorded for masked elements.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j].V[vtx][i] = VP[j]. V[vax][i] * VP[j].V[vbx][i];
        else
            VP[j].V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
MUL.SQ	Va,Vb,Vt	BVI	AE	F4,1,35	0
MUL.SQ.T	Va,Vb,Vt	BVI	AE	F4,1,35	2
MUL.SQ.F	Va,Vb,Vt	BVI	AE	F4,1,35	3

Exceptions

AE Integer Overflow (AEIOE)

AE Invalid Input (AEIIE)

AE Register Range (AERRE)

MUL – Vector Multiply 48x48 Unsigned Quad Word Scalar

MUL.UQ	Va,Sb,Vt
MUL.UQ.T	Va,Sb,Vt
MUL.UQ.F	Va,Sb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	if	1100	opc	Sb	Va	Vt

Description

The instruction multiplies a 64-bit vector register to a 64-bit scalar value using unsigned quad word integer data format. The multiply operation is performed using the least significant 48-bits of the input registers and produces a 48-bit result. An invalid input operand exception is issued if an input operand value exceeds 48 bits. An integer overflow exception is issued if the result exceeds 48 bits. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of the output vector register for masked elements is undefined and exceptions are not recorded for masked elements.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j]. V[vtx] [i] = VP[j]. V [vax][i] * S[sbx];
        else
            VP[j]. V[vtx] [i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		ISA	Type	Encoding	VM
MUL.UQ	Va,Sb,Vt	BVI	AE	F3,1,34	0
MUL.UQ.T	Va,Sb,Vt	BVI	AE	F3,1,34	2
MUL.UQ.F	Va,Sb,Vt	BVI	AE	F3,1,34	3

Exceptions

AE Integer Overflow (AEIOE)	AE Register Range (AERRE)
Scalar Register Range (SRRE)	AE Invalid Input (AEIIE)

MUL – Vector Multiply 48x48 Unsigned Quad Word

MUL.UQ	Va,Vb,Vt
MUL.UQ.T	Va,Vb,Vt
MUL.UQ.F	Va,Vb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1101	opc	Vb	Va	Vt

Description

The instruction multiplies two 64-bit vector register using unsigned quad word integer data format. The multiply operation is performed using the least significant 48-bits of the input registers and produces a 48-bit result. An invalid input operand exception is issued if an input operand value exceeds 48 bits. An integer overflow exception is issued if the result exceeds 48 bits. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of the output vector register for masked elements is undefined and exceptions are not recorded for masked elements.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j].V[vtx][i] = VP[j]. V[vax][i] * VP[j].V[vbx][i];
        else
            VP[j].V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*	*VM*
MUL.UQ	Va,Vb,Vt	BVI	AE	F4,1,34	0
MUL.UQ.T	Va,Vb,Vt	BVI	AE	F4,1,34	2
MUL.UQ.F	Va,Vb,Vt	BVI	AE	F4,1,34	3

Exceptions

AE Integer Overflow (AEIOE)

AE Register Range (AERRE)

AE Invalid Input (AEIIE)

NAND – Vector Logical Nand

NAND	Va,Vb,Vt
NAND.T	Va,Vb,Vt
NAND.F	Va,Vb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	if	1100	opc	Sb	Va	Vt

Description

The instruction performs a logical ‘nand’ operation between the two 64-bit source vector registers. The result of the operation is written to a 64-bit destination vector register. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of masked elements is undefined.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(VMA);
for (j = 0; j < VPL; j += 1) {
    for (i = 0; i < VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j].V[vtx][i] = ~(VP[j].V[vax][i]  & VP[j].V[vbx][i];
        else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
NAND	Va,Vb,Vt	BVI	AE	F4,1,22	0
NAND.T	Va,Vb,Vt	BVI	AE	F4,1,22	2
NAND.F	Va,Vb,Vt	BVI	AE	F4,1,22	3

Exceptions

AE Register Range (AERRE)

NAND – Vector Logical Nand with Scalar

NAND	Va,Sb,Vt
NAND.T	Va,Sb,Vt
NAND.F	Va,Sb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	if	1100	opc	Sb	Va	Vt

Description

The instruction performs a logical ‘nand’ operation between a 64-bit source vector register and an S register. The result of the operation is written to a 64-bit destination vector register. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of masked elements is undefined.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vtx = VregIdx(Vt);
vmax = VMregIdx(VMA);
for (j = 0; j < VPL; j += 1) {
    for (i = 0; i < VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j].V[vtx][i] = ~(VP[j].V[vax][i]  & S[sbx]);
        else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
NAND	Va,Sb,Vt	BVI	AE	F3,1,22	0
NAND.T	Va,Sb,Vt	BVI	AE	F3,1,22	2
NAND.F	Va,Sb,Vt	BVI	AE	F3,1,22	3

Exceptions

AE Register Range (AERRE)	Scalar Register Range (SRRE)

NEG – Scalar Negate Value Float Single

NEG.FS Sa,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	**0**	**1100**	**opc**	**000000**	**Sa**	**St**

Description

The instruction performs the negate value operation on the source scalar register using single precision floating point data format. The value of the operation is written to the destination scalar register.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx]<63:32> = 0;
S[stx]<31:0> = - S[sax]<31:0>;
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
NEG.FS	Sa,St	Scalar	S	F3,0,0C

Exceptions

Scalar Float Invalid Operand (SFIE)   Scalar Register Range (SRRE)

NEG – Scalar Negate Value Float Double

NEG.FD Sa,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	0	1100	opc	000000	Sa	St

Description

The instruction performs the negate value operation on the source scalar register using double precision floating point data format. The value of the operation is written to the destination scalar register.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx] = - S[sax];
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
NEG.FD	Sa,St	Scalar	S	F3,0,0D

Exceptions

Scalar Float Invalid Operand (SFIE) Scalar Register Range (SRRE)

NEG – Address Negate Value Signed Quad Word

NEG.SQ Aa,At

31 29	28 25	24 18	17 12	11 6	5 0
it	1100	opc	000000	Aa	At

Description

The instruction performs the negate value operation on the source address register using signed quad word integer data format. The value of the operation is written to the destination address register.

Pseudo Code

```
aax = AregIdx(Aa);
atx = AregIdx(At);
A[atx] = - A[aax];
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*
NEG.SQ	Aa,At	Scalar	A	F3,0D

Exceptions

Scalar Register Range (SRRE)

NEG – Scalar Negate Value Signed Quad Word

NEG.SQ Sa,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	1	1100	opc	000000	Sa	St

Description

The instruction performs the negate value operation on the source scalar register using signed quad word integer data format. The value of the operation is written to the destination scalar register.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx] = - S[sax];
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
NEG.SQ	Sa,St	Scalar	S	F3,1,0D

Exceptions

Scalar Register Range (SRRE)

NEG – Vector Negate Signed Quad Word

NEG.SQ	Va,Vt
NEG.SQ.T	Va,Vt
NEG.SQ.F	Va,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1100	opc	000000	Va	Vt

Description

The instruction performs the negate operation to the source vector register using signed quad word data format. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of the operation is written to a destination vector register. The masked elements results are undefined and exceptions are not signaled.

Pseudo Code

```
vax = VregIdx(Va);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j]. V[vtx][i] = - VP[j]. V[vax][i];
        else
            VP[j]. V[ vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
NEG.SQ	Va,Vt	BVI	AE	F3,1,0D	0
NEG.SQ.T	Va,Vt	BVI	AE	F3,1,0D	2
NEG.SQ.F	Va,Vt	BVI	AE	F3,1,0D	3

Exceptions

AE Integer Overflow (AEIOE)

AE Register Range (AERRE)

NOP – No Operation (A, S and AE)

NOP

31 30	29	28 25	24 18	17 12	11 6	5 0
it	1	1100	opc	000000	000000	000000

Description

The instruction does not modify any user state other than advancing the instruction pointer.

Pseudo Code

```
Nop();
```

Instruction Encoding

Instruction	***ISA***	***Type***	***Encoding***
NOP	Scalar	A	F3,00
NOP	Scalar	S	F3,1,00
NOP	BVI, ASP	AE	F3,1,00

Exceptions

None

NOR – Vector Logical Nor

NOR	Va,Vb,Vt
NOR.T	Va,Vb,Vt
NOR.F	Va,Vb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	if	1100	opc	Sb	Va	Vt

Description

The instruction performs a logical ‘nor’ operation between the two 64-bit source vector registers. The result of the operation is written to a 64-bit destination vector register. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of masked elements is undefined.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(VMA);
for (j = 0; j < VPL; j += 1) {
    for (i = 0; i < VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j].V[vtx][i] = ~(VP[j].V[vax][i]  | VP[j].V[vbx][i];
        else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
NOR	Va,Vb,Vt	BVI	AE	F4,1,23	0
NOR.T	Va,Vb,Vt	BVI	AE	F4,1,23	2
NOR.F	Va,Vb,Vt	BVI	AE	F4,1,23	3

Exceptions

AE Register Range (AERRE)

NOR – Vector Logical Nor with Scalar

NOR	Va,Sb,Vt
NOR.T	Va,Sb,Vt
NOR.F	Va,Sb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	if	1100	opc	Sb	Va	Vt

Description

The instruction performs a logical ‘nor’ operation between a 64-bit source vector register and an S register. The result of the operation is written to a 64-bit destination vector register. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of masked elements is undefined.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vtx = VregIdx(Vt);
vmax = VMregIdx(VMA);
for (j = 0; j < VPL; j += 1) {
    for (i = 0; i < VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j].V[vtx][i] = ~(VP[j].V[vax][i]  | S[sbx]);
        else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*	*VM*
NOR	Va,Sb,Vt	BVI	AE	F3,1,23	0
NOR.T	Va,Sb,Vt	BVI	AE	F3,1,23	2
NOR.F	Va,Sb,Vt	BVI	AE	F3,1,23	3

Exceptions

AE Register Range (AERRE)	Scalar Register Range (SRRE)

OR – Vector Logical Or

OR	Va,Vb,Vt
OR.T	Va,Vb,Vt
OR.F	Va,Vb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	if	1100	opc	Sb	Va	Vt

Description

The instruction performs a logical ‘or’ operation between the two 64-bit source vector registers. The result of the operation is written to a 64-bit destination vector register. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of masked elements is undefined.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(VMA);
for (j = 0; j < VPL; j += 1) {
    for (i = 0; i < VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j].V[vtx][i] = VP[j].V[vax][i]  | VP[j].V[vbx][i];
        else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
OR	Va,Vb,Vt	BVI	AE	F4,1,21	0
OR.T	Va,Vb,Vt	BVI	AE	F4,1,21	2
OR.F	Va,Vb,Vt	BVI	AE	F4,1,21	3

Exceptions

AE Register Range (AERRE)

OR – Vector Logical Or with Scalar

OR	Va,Sb,Vt
OR.T	Va,Sb,Vt
OR.F	Va,Sb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	if	1100	opc	Sb	Va	Vt

Description

The instruction performs a logical ‘or’ operation between a 64-bit source vector register and an S register. The result of the operation is written to a 64-bit destination vector register. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of masked elements is undefined.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vtx = VregIdx(Vt);
vmax = VMregIdx(VMA);
for (j = 0; j < VPL; j += 1) {
    for (i = 0; i < VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j].V[vtx][i] = VP[j].V[vax][i]  | S[sbx];
        else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
OR	Va,Sb,Vt	BVI	AE	F3,1,21	0
OR.T	Va,Sb,Vt	BVI	AE	F3,1,21	2
OR.F	Va,Sb,Vt	BVI	AE	F3,1,21	3

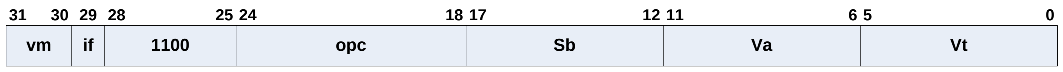
Exceptions

AE Register Range (AERRE)

Scalar Register Range (SRRE)

ORC – Vector Logical Or Compliment

ORC	Va,Vb,Vt
ORC.T	Va,Vb,Vt
ORC.F	Va,Vb,Vt



Description

The instruction performs a logical ‘or’ compliment operation between the two 64-bit source vector registers. The result of the operation is written to a 64-bit destination vector register. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of masked elements is undefined.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(VMA);
for (j = 0; j < VPL; j += 1) {
    for (i = 0; i < VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j].V[vtx][i] = ~VP[j].V[vax][i]  | VP[j].V[vbx][i];
        else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
ORC	Va,Vb,Vt	BVI	AE	F4,1,27	0
ORC.T	Va,Vb,Vt	BVI	AE	F4,1,27	2
ORC.F	Va,Vb,Vt	BVI	AE	F4,1,27	3

Exceptions

AE Register Range (AERRE)

ORC – Vector Logical Or Compliment with Scalar

ORC	Va,Sb,Vt
ORC.T	Va,Sb,Vt
ORC.F	Va,Sb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	if	1100	opc	Sb	Va	Vt

Description

The instruction performs a logical ‘or’ compliment operation between a 64-bit source vector register and an S register. The result of the operation is written to a 64-bit destination vector register. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of masked elements is undefined.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vtx = VregIdx(Vt);
vmax = VMregIdx(VMA);
for (j = 0; j < VPL; j += 1) {
    for (i = 0; i < VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j].V[vtx][i] = ~VP[j].V[vax][i]  | S[sbx];
        else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
ORC	Va,Sb,Vt	BVI	AE	F3,1,27	0
ORC.T	Va,Sb,Vt	BVI	AE	F3,1,27	2
ORC.F	Va,Sb,Vt	BVI	AE	F3,1,27	3

Exceptions

AE Register Range (AERRE)

Scalar Register Range (SRRE)

PLC – Population Count VM Register

PLC VMa,At

31 29	28 24	23 18	17 12	11 6	5 0
00	11101	opc	000000	VMa	At

Description

The instruction counts the number of one bits in the VMa register for the active partition. All bits of the specified VM register participate in the operation.

The AE Element Range Exception is set if VPA is invalid for the vector partition mode.

Pseudo Code

```
vmax = VMregIdx(VMa);
atx = AregIdx(At);
A[atx] = PopulationCount(VP[VPA].VM[vmax]);
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*	*VM*
PLC	VMa,At	BVI	AE	F6,1,0F	0

Exceptions

Scalar Register Range (SRRE)

AE Register Range (AERRE)

AE Element Range (AEERE)

RTN – Return from Subroutine

rtn		; return
rtn(.bl|.bu).(t|f)	CCt	; return conditional

31 29	28 27	26 22	21	20 6	5	4 0
it	10	opc	-	-	tf	cc

Description

If the call return stack is empty or a CRS overflow exception has occurred then the current coprocessor routine is complete, otherwise program flow is transferred to the return instruction pointer of the call return stack and the Window Base register is restored to the call return stack RWB value. Conditional returns can select from any of the condition code bits within the CC register. Conditional returns can use the value of the CC register or the complemented value. Additionally, conditional returns can be hinted as whether the return instruction is likely or unlikely to return.

Pseudo Code

```
if ( opc == 0x0C || CPS.CC<cc> == tf ) {
        If (CPS.CRT == 0 || CPS.SCOE) {
            UpdateUserModeStatus();
            StopExecution();
        } else {
             CPS.CRT –= 1;
            IP = CRS[CPS.CRT].RIP;
            WB = CRS[CPS.CRT].RWB;
        }
        CPS.WBV = 0;
} else
        IP = IP + InstLength();
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
RTN		Scalar	A	F2,0C
RTN	CCt	Scalar	A	F2,0D
RTN.BL	CCt	Scalar	A	F2,0E
RTN.BU	CCt	Scalar	A	F2,0F

Exceptions

Scalar Unaligned Reference (SURE)　　Scalar Register Range (SRRE)

SEL – Address Register Select

SEL	AC3.EQ,Aa,Ab,At
SEL	AC3.GT,Aa,Ab,At
SEL	AC3.LT,Aa,Ab,At
SEL	SC3.EQ,Aa,Ab,At
SEL	SC3.GT,Aa,Ab,At
SEL	SC3.LT,Aa,Ab,At

31 29	28 25	24 18	17 12	11 6	5 0
it	1101	opc	Ab	Aa	At

Description

The instruction selects between two address registers using an AC3 condition code bit.

Pseudo Code

```
aax = AregIdx(Aa);      abx = AregIdx(Ab);          atx = AregIdx(At);
switch (opc<1:0>) {
case 0x18: A[atx] = (AC3.EQ == 1) ? A[aax] : A[abx]; break;
case 0x19: A[atx] = (AC3.GT == 1) ? A[aax] : A[abx]; break;
case 0x1A: A[atx] = (AC3.LT == 1) ? A[aax] : A[abx]; break;
case 0x1C: A[atx] = (SC3.EQ == 1) ? A[aax] : A[abx]; break;
case 0x1D: A[atx] = (SC3.GT == 1) ? A[aax] : A[abx]; break;
case 0x1E: A[atx] = (SC3.LT == 1) ? A[aax] : A[abx]; break;
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
SEL	AC3.EQ,Aa,Ab,At	Scalar	A	F4,18
SEL	AC3.GT,Aa,Ab,At	Scalar	A	F4,19
SEL	AC3.LT,Aa,Ab,At	Scalar	A	F4,1A
SEL	SC3.EQ,Aa,Ab,At	Scalar	A	F4,1C
SEL	SC3.GT,Aa,Ab,At	Scalar	A	F4,1D
SEL	SC3.LT,Aa,Ab,At	Scalar	A	F4,1E

Exceptions

AE Register Range (AERRE)

SEL – Scalar Register Select

SEL	AC3.EQ,Sa,Sb,St
SEL	AC3.GT,Sa,Sb,St
SEL	AC3.LT,Sa,Sb,St
SEL	SC3.EQ,Sa,Sb,St
SEL	SC3.GT,Sa,Sb,St
SEL	SC3.LT,Sa,Sb,St

31 29	28 25	24 18	17 12	11 6	5 0
it	1101	opc	Ab	Aa	At

Description

The instruction selects between two address registers using an SC3 condition code bit.

Pseudo Code

```
sax = SregIdx(Sa);      sbx = SregIdx(Sb);          stx = SregIdx(St);
switch (opc) {
case 0x18: S[stx] = (AC3.EQ == 1) ? S[sax] : S[sbx]; break;
case 0x19: S[stx] = (AC3.GT == 1) ? S[sax] : S[sbx]; break;
case 0x1A: S[stx] = (AC3.LT == 1) ? S[sax] : S[sbx]; break;
case 0x1C: S[stx] = (SC3.EQ == 1) ? S[sax] : S[sbx]; break;
case 0x1D: S[stx] = (SC3.GT == 1) ? S[sax] : S[sbx]; break;
case 0x1E: S[stx] = (SC3.LT == 1) ? S[sax] : S[sbx]; break;
}
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*
SEL	AC3.EQ,Sa,Sb,St	Scalar	S	F4,1,18
SEL	AC3.GT,Sa,Sb,St	Scalar	S	F4,1,19
SEL	AC3.LT,Sa,Sb,St	Scalar	S	F4,1,1A
SEL	SC3.EQ,Sa,Sb,St	Scalar	S	F4,1,1C
SEL	SC3.GT,Sa,Sb,St	Scalar	S	F4,1,1D
SEL	SC3.LT,Sa,Sb,St	Scalar	S	F4,1,1E

Exceptions

AE Register Range (AERRE)

SEL – Vector Element Select

SEL.T	Va,Vb,Vt
SEL.F	Va,Vb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	0	1101	opc	Vb	Va	Vt

Description

The instruction selects element by element between two vector registers into a third vector register. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1)
        if (vm==2 && VP[j].VM[vmax]<i>==1 || vm==3 && VP[j].VM[vmax]<i>==0)
            VP[j].V[vtx][i] = VP[j].V[vax][i];
        else
            VP[j].V[vtx][i] =VP[j]. V[vbx][i];
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***vm***
SEL.T	Va,Vb,Vt	BVI	AE	F4,0,2F	2
SEL.F	Va,Vb,Vt	BVI	AE	F4,0,2F	3

Exceptions

AE Register Range (AERRE)

SEL – Vector / Scalar Element Select

SEL.T Va,Sb,Vt

SEL.F Va,Sb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1100	opc	Sb	Va	Vt

Description

The instruction selects element by element between a vector register and a scalar register into a destination vector register. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1)
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i>==1
                  || vm==3 && VP[j].VM[vmax]<i>==0)
            VP[j].V[vtx][i] = VP[j]. V[vax][i];
        else
            VP[j].V[vtx][i] = S[sbx];
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***vm***
SEL.T	Va,Sb,Vt	BVI	AE	F3,0,2F	2
SEL.F	Va,Sb,Vt	BVI	AE	F3,0,2F	3

Exceptions

AE Register Range (AERRE)

Scalar Register Range (SRRE)

SHFL – Address Shift Left Unsigned with Immediate

SHFL.UQ Aa,Immed,At

31 29	28 25	24 18	17 12	11 6	5 0
it	1100	opc	Immed6	Aa	At

Description

The instruction shifts the source register left by the amount specified by the immediate field. Zero bits are shifted in on the right. The immediate value is either the zero extended 6-bit Immed6 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
aax = AregIdx(Aa);
atx = AregIdx(At);
A[atx] = A[aax] << Uimmed6()<5:0>;
```

Instruction Encoding

Instruction	*ISA*	*Type*	*Encoding*
SHFL.UQ Aa,Immed,At	Scalar	A	F3,2C

Exceptions

Scalar Register Range (SRRE)

SHFL – Address Shift Left Unsigned

SHFL.UQ Aa,Ab,At

31 29	28 25	24 18	17 12	11 6	5 0
it	**1101**	**opc**	**Ab**	**Aa**	**At**

Description

The instruction shifts the source register left by the amount specified by the least significant six bits of the Ab register. Zero bits are shifted in on the right.

Pseudo Code

```
aax = AregIdx(Aa);
abx = AregIdx(Ab);
atx = AregIdx(At);
A[atx] = A[aax] << A[abx]<5:0>;
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
SHFL.UQ	Aa,Ab,At	Scalar	A	F4,2C

Exceptions

Scalar Register Range (SRRE)

SHFL – Address Shift Left Signed with Immediate

SHFL.SQ Aa,Immed,At

31 29	28 25	24 18	17 12	11 6	5 0
it	1100	opc	Immed6	Aa	At

Description

The instruction shifts the source register left by the amount specified by the immediate field. Zero bits are shifted in on the right. The immediate value is either the zero extended 6-bit Immed6 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
aax = AregIdx(Aa);
atx = AregIdx(At);
A[atx] = A[aax] << Uimmed6()<5:0>;
```

Instruction Encoding

Instruction	***ISA***	***Type***	***Encoding***
SHFL.SQ Aa,Immed,At	Scalar	A	F3,2D

Exceptions

Scalar Register Range (SRRE)

Scalar Integer Overflow (SIOE)

SHFL – Address Shift Left Signed

SHFL.SQ Aa,Ab,At

31 29	28 25	24 18	17 12	11 6	5 0
it	1101	opc	Ab	Aa	At

Description

The instruction shifts the source register left by the amount specified by the least significant six bits of the Ab register. Zero bits are shifted in on the right.

Pseudo Code

```
aax = AregIdx(Aa);
abx = AregIdx(Ab);
atx = AregIdx(At);
A[atx] = A[aax] << A[abx]<5:0>;
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
SHFL.SQ	Aa,Ab,At	Scalar	A	F4,2D

Exceptions

Scalar Register Range (SRRE)

Scalar Integer Overflow (SIOE)

SHFL – Scalar Shift Left Unsigned with Immediate

SHFL.UQ Sa,Immed,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	**if**	**1100**	**opc**	**Immed6**	**Sa**	**St**

Description

The instruction shifts the source register left by the amount specified by the immediate field. Zero bits are shifted in on the right. The immediate value is either the zero extended 6-bit Immed6 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx] = S[sax] << Uimmed6()<5:0>;
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*
SHFL.UQ	Sa,Immed,St	Scalar	S	F3,1,2C

Exceptions

Scalar Register Range (SRRE)

SHFL – Scalar Shift Left Unsigned

SHFL.UQ Sa,Sb,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	if	1101	opc	Sb	Sa	St

Description

The instruction shifts the source register left by the amount specified by the least significant six bits of the Sb register. Zero bits are shifted in on the right.

Pseudo Code

```
sax = SregIdx(Sa);
sbx = SregIdx(Sb);
stx = SregIdx(St);
S[stx] = S[sax] << S[sbx]<5:0>;
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
SHFL.UQ	Sa,Sb,St	Scalar	S	F4,1,2C

Exceptions

Scalar Register Range (SRRE)

SHFL – Scalar Shift Left Signed with Immediate

SHFL.SQ Sa,Immed,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	if	1100	opc	Immed6	Sa	St

Description

The instruction shifts the source register left by the amount specified by the immediate field. Zero bits are shifted in on the right. The immediate value is either the zero extended 6-bit Immed6 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
sax = SregIdx(Sa);
atx = SregIdx(St);
S[stx] = S[aax] << Uimmed6()<5:0>;
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*
SHFL.SQ	Sa,Immed,St	Scalar	S	F3,1,2D

Exceptions

Scalar Register Range (SRRE)

Scalar Integer Overflow (SIOE)

SHFL – Scalar Shift Left Signed

SHFL.SQ   Sa,Sb,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	**if**	**1101**	**opc**	**Sb**	**Sa**	**St**

Description

The instruction shifts the source register left by the amount specified by the least significant six bits of the Sb register. Zero bits are shifted in on the right.

Pseudo Code

```
sax = SregIdx(Sa);
sbx = SregIdx(Sb);
stx = SregIdx(St);
S[stx] = S[sax] << S[sbx]<5:0>;
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
SHFL.SQ	Sa,Sb,St	Scalar	S	F4,1,2D

Exceptions

Scalar Register Range (SRRE)

Scalar Integer Overflow (SIOE)

SHFL – Vector Shift Left Unsigned Quad Word with Scalar

SHFL.UQ Va,Sb,Vt

SHFL.UQ.T Va,Sb,Vt

SHFL.UQ.F Va,Sb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1100	opc	Sb	Va	Vt

Description

The instruction shifts each register element left by the amount specified in the least significant bits of the S register value. Zero bits are shifted in on the right. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1)
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            VP[j].V[vtx][i] = VP[j].V[vax][i] << S[sbx]<5:0>;
        } else
            VP[j]. V[vtx][i] = UndefinedValue();
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
SHFL.UQ	Va,Sb,Vt	BVI	AE	F3,1,2C	0
SHFL.UQ.T	Va,Sb,Vt	BVI	AE	F3,1,2C	2
SHFL.UQ.F	Va,Sb,Vt	BVI	AE	F3,1,2C	3

Exceptions

AE Register Range (AERRE)

Scalar Register Range (SRRE)

SHFL – Vector Shift Left Unsigned Quad Word

SHFL.UQ Va,Vb,Vt

SHFL.UQ.T Va,Vb,Vt

SHFL.UQ.F Va,Vb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1101	opc	Vb	Va	Vt

Description

The instruction shifts each register element left by the amount specified in the least significant bits of a vector register element values. Zero bits are shifted in on the right. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1)
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            VP[j].V[vtx][i] = VP[j].V[vax][i] << VP[j].V[vbx][i]<5:0>;
        } else
            VP[j]. V[vtx][i] = UndefinedValue();
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
SHFL.UQ	Va,Vb,Vt	BVI	AE	F4,1,2C	0
SHFL.UQ.T	Va,Vb,Vt	BVI	AE	F4,1,2C	2
SHFL.UQ.F	Va,Vb,Vt	BVI	AE	F4,1,2C	3

Exceptions

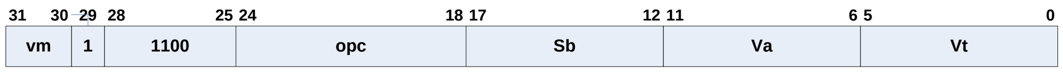
AE Register Range (AERRE)

SHFL – Vector Shift Left Signed Quad Word with Scalar

SHFL.SQ Va,Sb,Vt

SHFL.SQ.T Va,Sb,Vt

SHFL.SQ.F Va,Sb,Vt



Description

The instruction shifts each register element left by the amount specified in the least significant 6-bits of the S register value. Zero bits are shifted in on the right.

An overflow exception is produced if the bits shifted out are not all ones or all zeros, or if the value changes sign by the shift operation. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of the output vector register for masked elements is undefined and exceptions are not recorded for masked elements.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1)
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            VP[j].V[vtx][i] = VP[j].V[vax][i] << S[sbx]<5:0>;
        } else
            VP[j]. V[vtx][i] = UndefinedValue();
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
SHFL.SQ	Va,Sb,Vt	BVI	AE	F3,1,2D	0
SHFL.SQ.T	Va,Sb,Vt	BVI	AE	F3,1,2D	2
SHFL.SQ.F	Va,Sb,Vt	BVI	AE	F3,1,2D	3

Exceptions

AE Integer Overflow (AEIOE)

AE Register Range (AERRE)

Scalar Register Range (SRRE)

SHFL – Vector Shift Left Signed Quad Word

SHFL.SQ Va,Vb,Vt

SHFL.SQ.T Va,Vb,Vt

SHFL.SQ.T Va,Vb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1101	opc	Vb	Va	Vt

Description

The instruction shifts each register element left by the amount specified in the least significant bits of a vector register element values. Zero bits are shifted in on the right. An overflow exception is produced if the bits shifted out are not all ones or all zeros, or if the value changes sign by the shift operation. The 'vm' field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1)
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            VP[j].V[vtx][i] = VP[j].V[vax][i] << VP[j].V[vbx][i]<5:0>;
        } else
            VP[j]. V[vtx][i] = UndefinedValue();
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
SHFL.SQ	Va,Vb,Vt	BVI	AE	F4,1,2D	0
SHFL.SQ.T	Va,Vb,Vt	BVI	AE	F4,1,2D	2
SHFL.SQ.F	Va,Vb,Vt	BVI	AE	F4,1,2D	3

Exceptions

AE Integer Overflow (AEIOE)

AE Register Range (AERRE)

SHFR – Address Shift Right Unsigned with Immediate

SHFR.UQ Aa,Immed,At

31 29	28 25	24 18	17 12	11 6	5 0
it	1100	opc	Immed6	Aa	At

Description

The instruction shifts the source register right by the amount specified by the immediate field. Zero bits are shifted in on the left. The immediate value is either the zero extended 6-bit Immed6 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
aax = AregIdx(Aa);
atx = AregIdx(At);
A[atx] = A[aax] >> Uimmed6()<5:0>;
```

Instruction Encoding

Instruction	***ISA***	***Type***	***Encoding***
SHFR.UQ Aa,Immed,At	Scalar	A	F3,2E

Exceptions

Scalar Register Range (SRRE)

SHFR – Address Shift Right Unsigned

SHFR.UQ Aa,Ab,At

31 29	28 25	24 18	17 12	11 6	5 0
it	1101	opc	Ab	Aa	At

Description

The instruction shifts the source register right by the amount specified by the least significant six bits of the Ab register. Zero bits are shifted in on the left.

Pseudo Code

```
aax = AregIdx(Aa);
abx = AregIdx(Ab);
atx = AregIdx(At);
A[atx] = A[aax] >> A[abx]<5:0>;
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
SHFR.UQ	Aa,Ab,At	Scalar	A	F4,2E

Exceptions

Scalar Register Range (SRRE)

SHFR – Address Shift Right Signed with Immediate

SHFR.SQ Aa,Immed,At

31 29	28 25	24 18	17 12	11 6	5 0
it	1100	opc	Immed6	Aa	At

Description

The instruction shifts the source register right by the amount specified by the immediate field. The most significant bit is replicated on the left. The immediate value is either the zero extended 6-bit Immed6 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
aax = AregIdx(Aa);
atx = AregIdx(At);
A[atx] = A[aax] >> Uimmed6()<5:0>;
```

Instruction Encoding

Instruction	*ISA*	*Type*	*Encoding*
SHFR.SQ Aa,Immed,At	Scalar	A	F3,2F

Exceptions

Scalar Register Range (SRRE)

Scalar Integer Overflow (SIOE)

SHFR – Address Shift Right Signed

SHFR.SQ Aa,Ab,At

31 29	28 25	24 18	17 12	11 6	5 0
it	1101	opc	Ab	Aa	At

Description

The instruction shifts the source register right by the amount specified by the least significant six bits of the Ab register. The most significant bit is replicated on the left.

Pseudo Code

```
aax = AregIdx(Aa);
abx = AregIdx(Ab);
atx = AregIdx(At);
A[atx] = A[aax] >> A[abx]<5:0>;
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
SHFR.SQ	Aa,Ab,At	Scalar	A	F4,2F

Exceptions

Scalar Register Range (SRRE)

Scalar Integer Overflow (SIOE)

SHFR – Scalar Shift Right Unsigned with Immediate

SHFR.UQ Sa,Immed,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	if	1100	opc	Immed6	Sa	St

Description

The instruction shifts the source register right by the amount specified by the immediate field. Zero bits are shifted in on the left. The immediate value is either the zero extended 6-bit Immed6 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx] = S[sax] >> Uimmed6()<5:0>;
```

Instruction Encoding

Instruction	*ISA*	*Type*	*Encoding*
SHFR.UQ Sa,Immed,St	Scalar	S	F3,1,2E

Exceptions

Scalar Register Range (SRRE)

SHFR – Scalar Shift Right Unsigned

SHFR.UQ Sa,Sb,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	if	1101	opc	Sb	Sa	St

Description

The instruction shifts the source register right by the amount specified by the least significant six bits of the Sb register. Zero bits are shifted in on the left.

Pseudo Code

```
sax = SregIdx(Sa);
sbx = SregIdx(Sb);
stx = SregIdx(St);
S[stx] = S[sax] >> S[sbx]<5:0>;
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*
SHFR.UQ	Sa,Sb,St	Scalar	S	F4,1,2E

Exceptions

Scalar Register Range (SRRE)

SHFR – Scalar Shift Right Signed with Immediate

SHFR.SQ   Sa,Immed,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	**if**	**1100**	**opc**	**Immed6**	**Sa**	**St**

Description

The instruction shifts the source register right by the amount specified by the immediate field. The most significant bit is replicated on the left. The immediate value is either the zero extended 6-bit Immed6 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
sax = SregIdx(Sa);
atx = SregIdx(St);
S[stx] = S[aax] >> Uimmed6()<5:0>;
```

Instruction Encoding

Instruction	***ISA***	***Type***	***Encoding***
SHFR.SQ Sa,Immed,St	Scalar	S	F3,1,2F

Exceptions

Scalar Register Range (SRRE)   Scalar Integer Overflow (SIOE)

SHFR – Scalar Shift Right Signed

SHFR.SQ Sa,Sb,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	if	1101	opc	Sb	Sa	St

Description

The instruction shifts the source register right by the amount specified by the least significant six bits of the Sb register. The most significant bit is replicated on the left.

Pseudo Code

```
sax = SregIdx(Sa);
sbx = SregIdx(Sb);
stx = SregIdx(St);
S[stx] = S[sax] >> S[sbx]<5:0>;
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
SHFR.SQ	Sa,Sb,St	Scalar	S	F4,1,2F

Exceptions

Scalar Register Range (SRRE) Scalar Integer Overflow (SIOE)

SHFR – Vector Shift Right Unsigned Quad Word with Scalar

SHFR.UQ Va,Sb,Vt

SHFR.UQ.T Va,Sb,Vt

SHFR.UQ.F Va,Sb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1100	opc	Sb	Va	Vt

Description

The instruction shifts each register element right by the amount specified in the least significant bits of the S register value. Zero bits are shifted in on the left. The 'vm' field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1)
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            VP[j].V[vtx][i] = VP[j].V[vax][i] >> S[sbx]<5:0>;
        } else
            VP[j]. V[vtx][i] = UndefinedValue();
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
SHFR.UQ	Va,Sb,Vt	BVI	AE	F3,1,2E	0
SHFR.UQ.T	Va,Sb,Vt	BVI	AE	F3,1,2E	2
SHFR.UQ.F	Va,Sb,Vt	BVI	AE	F3,1,2E	3

Exceptions

AE Register Range (AERRE) Scalar Register Range (SRRE)

SHFR – Vector Shift Right Unsigned Quad Word

SHFR.UQ Va,Vb,Vt

SHFR.UQ.T Va,Vb,Vt

SHFR.UQ.F Va,Vb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1101	opc	Vb	Va	Vt

Description

The instruction shifts each register element right by the amount specified in the least significant bits of a vector register element values. Zero bits are shifted in on the left. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1)
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            VP[j].V[vtx][i] = VP[j].V[vax][i] >> VP[j].V[vbx][i]<5:0>;
        } else
            VP[j]. V[vtx][i] = UndefinedValue();
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
SHFR.UQ	Va,Vb,Vt	BVI	AE	F4,1,2E	0
SHFR.UQ.T	Va,Vb,Vt	BVI	AE	F4,1,2E	2
SHFR.UQ.F	Va,Vb,Vt	BVI	AE	F4,1,2E	3

Exceptions

Scalar Register Range (SRRE) AE Register Range (AERRE)

SHFR – Vector Shift Right Signed Quad Word with Scalar

SHFR.SQ Va,Sb,Vt

SHFR.SQ.T Va,Sb,Vt

SHFR.SQ.F Va,Sb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1100	opc	Sb	Va	Vt

Description

The instruction shifts each register element right by the amount specified in the least significant bits of the S register value. The value of bit 63 (the sign bit) is replicated on the left. The 'vm' field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1)
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            VP[j].V[vtx][i] = VP[j].V[vax][i] >> S[sbx]<5:0>;
        } else
            VP[j]. V[vtx][i] = UndefinedValue();
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
SHFR.SQ	Va,Sb,Vt	BVI	AE	F3,1,2F	0
SHFR.SQ.T	Va,Sb,Vt	BVI	AE	F3,1,2F	2
SHFR.SQ.F	Va,Sb,Vt	BVI	AE	F3,1,2F	3

Exceptions

Scalar Register Range (SRRE)

AE Register Range (AERRE)

SHFR – Vector Shift Right Signed Quad Word

SHFR.SQ Va,Vb,Vt

SHFR.SQ.T Va,Vb,Vt

SHFR.SQ.F Va,Vb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1101	opc	Vb	Va	Vt

Description

The instruction shifts each register element right by the amount specified in the least significant bits of a vector register element values. The value of bit 63 (the sign bit) is replicated on the left. The 'vm' field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1)
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            VP[j].V[vtx][i] = VP[j].V[vax][i] >> VP[j].V[vbx][i]<5:0>;
        } else
            VP[j]. V[vtx][i] = UndefinedValue();
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
SHFR.SQ	Va,Vb,Vt	BVI	AE	F4,1,2F	0
SHFR.SQ.T	Va,Vb,Vt	BVI	AE	F4,1,2F	2
SHFR.SQ.F	Va,Vb,Vt	BVI	AE	F4,1,2F	3

Exceptions

AE Register Range (AERRE)

SQRT – Scalar Square Root Value Float Single

SQRT.FS Sa,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	0	1100	opc	000000	Sa	St

Description

The instruction performs the square root operation on the source scalar register using single precision floating point data format. The value of the operation is written to the destination scalar register.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx]<63:32> = 0;
S[stx]<31:0> = Sqrt( S[sax]<31:0> );
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
SQRT.FS	Sa,St	Scalar	S	F3,0,0E

Exceptions

Scalar Float Invalid Operand (SFIE)   Scalar Register Range (SRRE)

SQRT – Scalar Square Root Value Float Double

SQRT.FD Sa,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	0	1100	opc	000000	Sa	St

Description

The instruction performs the square root operation on the source scalar register using double precision floating point data format. The value of the operation is written to the destination scalar register.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx] = Sqrt( S[sax] );
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
SQRT.FD	Sa,St	Scalar	S	F3,0,0F

Exceptions

Scalar Float Invalid Operand (SFIE) Scalar Register Range (SRRE)

ST – Store with offset (A Register)

ST.(UB|UW|UD|UQ|SB|SW|SD) At,offset(Aa)

31 29	28	27 22	21 12	11 6	5 0
it	0	opc	immed10	Aa	At

Description

Store an A register to memory. The store can be Byte, Word, Double Word or Quad Word in size. Stores are checked for overflow when the memory size is less than a quad word. The effective address is formed by adding an immediate value to an A register value.

Pseudo Code

```
aax = AregIdx(Aa);
atx = AregIdx(At);
effa = A[aax] + Offset(opc<1:0>);
switch ( opc<2:0> ) {
case 0: ZovflChk(A[atx], 8); MemStore(effa, 8, A[atx]); break;      // unsigned byte
case 1: ZovflChk(A[atx], 16); MemStore(effa, 16, A[atx]); break;    // unsigned word
case 2: ZovflChk(A[atx], 32); MemStore(effa, 32, A[atx]); break;    // unsigned double word
case 3: MemStore(effa, 64, A[atx]); break;                          // quad word
case 4: SovflChk(A[atx], 8); MemStore(effa, 8, A[atx]); break;      // signed byte
case 5: SovflChk(A[atx], 16); MemStore(effa, 16, A[atx]); break;    // signed word
case 6: SovflChk(A[atx], 32); MemStore(effa, 32, A[atx]); break;    // signed double word
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
ST.UB	At,offset(Aa)	Scalar	A	F1,08
ST.UW	At,offset(Aa)	Scalar	A	F1,09
ST.UD	At,offset(Aa)	Scalar	A	F1,0A
ST.UQ	At,offset(Aa)	Scalar	A	F1,0B
ST.SB	At,offset(Aa)	Scalar	A	F1,0C
ST.SW	At,offset(Aa)	Scalar	A	F1,0D
ST.SD	At,offset(Aa)	Scalar	A	F1,0E

Exceptions

Scalar Unaligned Reference (SURE) Scalar Register Range (SRRE)

Scalar Store Overflow (SSOE)

ST – Store with offset (S Register)

ST.(UB|UW|UD|UQ|SB|SW|SD) St,offset(Aa)

31 30	29	28	27 22	21 12	11 6	5 0
it	1	0	opc	immed10	Aa	St

Description

Store an S register to memory. The store can be Byte, Word, Double Word or Quad Word in size. Stores are checked for overflow when the memory size is less than a quad word. The effective address is formed by adding an immediate value to an S register value.

Pseudo Code

```
aax = AregIdx(Aa);
stx = SregIdx(St);
effa = A[aax] + Offset(opc<1:0>);
switch ( opc<2:0> ) {
case 0: ZovflChk(S[stx], 8); MemStore(effa, 8, S[stx]); break;     // unsigned byte
case 1: ZovflChk(S[stx], 16); MemStore(effa, 16, S[stx]); break;   // unsigned word
case 2: ZovflChk(S[stx], 32); MemStore(effa, 32, S[stx]); break;   // unsigned double word
case 3: MemStore(effa, 64, S[stx]); break;                          // quad word
case 4: SovflChk(S[stx], 8); MemStore(effa, 8, S[stx]); break;     // signed byte
case 5: SovflChk(S[stx], 16); MemStore(effa, 16, S[stx]); break;   // signed word
case 6: SovflChk(S[stx], 32); MemStore(effa, 32, S[stx]); break;   // signed double word
}
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*
ST.UB	St,offset(Aa)	Scalar	S	F1,1,08
ST.UW	St,offset(Aa)	Scalar	S	F1,1,09
ST.UD	St,offset(Aa)	Scalar	S	F1,1,0A
ST.UQ	St,offset(Aa)	Scalar	S	F1,1,0B
ST.SB	St,offset(Aa)	Scalar	S	F1,1,0C
ST.SW	St,offset(Aa)	Scalar	S	F1,1,0D
ST.SD	St,offset(Aa)	Scalar	S	F1,1,0E

Exceptions

Scalar Unaligned Reference (SURE)

Scalar Register Range (SRRE)

Scalar Store Overflow (SSOE)

ST – Store with offset Float Single (S Register)

ST.FS St,offset(Aa)

31 30	29	28	27 22	21 12	11 6	5 0
it	0	0	opc	immed10	Aa	St

Description

Store the least significant 32-bits of an S register to memory. The effective address is formed by adding an immediate value to an S register value.

Pseudo Code

```
aax = AregIdx(Aa);
stx = SregIdx(St);
effa = A[aax] + Offset(opc<1:0>);
MemStore(effa, 32, S[stx]);
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
ST.FS	St,offset(Aa)	Scalar	S	F1,0,0A

Exceptions

Scalar Unaligned Reference (SURE) Scalar Register Range (SRRE)

ST – Store indexed (A Register)

ST.(UB|UW|UD|UQ|SB|SW|SD) At,Ab(Aa)

31 29	28 25	24 18	17 12	11 6	5 0
it	1101	opc	Ab	Aa	At

Description

Store an A register to memory. The store can be Byte, Word, Double Word or Quad Word in size. Stores are checked for overflow when the memory size is less than a quad word. The effective address is formed by adding two A register values.

Pseudo Code

```
aax = AregIdx(Aa);                    abx = AregIdx(Ab);
atx = AregIdx(At);
effa = A[aax] + A[abx];
switch ( opc<2:0> ) {
case 0: ZovflChk(A[atx], 8); MemStore(effa, 8, A[atx]); break;      // unsigned byte
case 1: ZovflChk(A[atx], 16); MemStore(effa, 16, A[atx]); break;    // unsigned word
case 2: ZovflChk(A[atx], 32); MemStore(effa, 32, A[atx]); break;    // unsigned double word
case 3: MemStore(effa, 64, A[atx]); break;                          // quad word
case 4: SovflChk(A[atx], 8); MemStore(effa, 8, A[atx]); break;      // signed byte
case 5: SovflChk(A[atx], 16); MemStore(effa, 16, A[atx]); break;    // signed word
case 6: SovflChk(A[atx], 32); MemStore(effa, 32, A[atx]); break;    // signed double word
}
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*
ST.UB	At,Ab(Aa)	Scalar	A	F4,08
ST.UW	At,Ab(Aa)	Scalar	A	F4,09
ST.UD	At,Ab(Aa)	Scalar	A	F4,0A
ST.UQ	At,Ab(Aa)	Scalar	A	F4,0B
ST.SB	At,Ab(Aa)	Scalar	A	F4,0C
ST.SW	At,Ab(Aa)	Scalar	A	F4,0D
ST.SD	At,Ab(Aa)	Scalar	A	F4,0E

Exceptions

Scalar Unaligned Reference (SURE)

Scalar Register Range (SRRE)

Scalar Store Overflow (SSOE)

ST – Store indexed (S Register)

ST.(UB|UW|UD|UQ|SB|SW|SD) St,Ab(Aa)

31 30	29	28 25	24 18	17 12	11 6	5 0
it	1	1101	opc	Ab	Aa	St

Description

Store an S register to memory. The store can be Byte, Word, Double Word or Quad Word in size. Stores are checked for overflow when the memory size is less than a quad word. The effective address is formed by adding two A register values.

Pseudo Code

```
aax = AregIdx(Aa);                          abx = AregIdx(Ab);
stx = SregIdx(St);
effa = A[aax] + A[abx];
switch ( opc<2:0> ) {
case 0: ZovflChk(S[stx], 8); MemStore(effa, 8, S[stx]); break;      // unsigned byte
case 1: ZovflChk(S[stx], 16); MemStore(effa, 16, S[stx]); break;    // unsigned word
case 2: ZovflChk(S[stx], 32); MemStore(effa, 32, S[stx]); break;    // unsigned double word
case 3: MemStore(effa, 64, S[stx]); break;                          // quad word
case 4: SovflChk(S[stx], 8); MemStore(effa, 8, S[stx]); break;      // signed byte
case 5: SovflChk(S[stx], 16); MemStore(effa, 16, S[stx]); break;    // signed word
case 6: SovflChk(S[stx], 32); MemStore(effa, 32, S[stx]); break;    // signed double word
}
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*
ST.UB	St,Ab(Aa)	Scalar	S	F4,1,08
ST.UW	St,Ab(Aa)	Scalar	S	F4,1,09
ST.UD	St,Ab(Aa)	Scalar	S	F4,1,0A
ST.UQ	St,Ab(Aa)	Scalar	S	F4,1,0B
ST.SB	St,Ab(Aa)	Scalar	S	F4,1,0C
ST.SW	St,Ab(Aa)	Scalar	S	F4,1,0D
ST.SD	St,Ab(Aa)	Scalar	S	F4,1,0E

Exceptions

Scalar Unaligned Reference (SURE)

Scalar Register Range (SRRE)

Scalar Store Overflow (SSOE)

ST – Store indexed Float Single (S Register)

ST.FS Ab(Aa),St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	0	1101	opc	Ab	Aa	St

Description

Store the least significant 32-bits of an S register to memory. The effective address is formed by adding two A register values.

Pseudo Code

```
aax = AregIdx(Aa);
abx = AregIdx(Ab);
stx = SregIdx(St);
effa = A[aax] + A[abx];
MemStore(effa, 32, S[stx]);
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
ST.FS	St,Ab(Aa)	Scalar	S	F4,0,0A

Exceptions

Scalar Unaligned Reference (SURE)
Scalar Register Range (SRRE)

ST – Store Vector Dual Double Word

ST.DDW	Vt,offset(Aa)
ST.DDW.T	Vt,offset(Aa)
ST.DDW.F	Vt,offset(Aa)

31 30	29	28	27 22	21 12	11 6	5 0
vm	**0**	**0**	**opc**	**immed10**	**Aa**	**Vt**

Description

The instruction stores a 64-bit vector register to memory. The 'vm' field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. Note that dual double word type can be aligned on four byte memory boundaries.

Pseudo Code

```
aax = AregIdx(Aa);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
effaBase = A[aax] + Offset(2);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            effa = effaBase + VPS * j + VS * i;
            MemStore( effa<47:0>, 64, VP[j].[vtx]<i> );
        }
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
ST.DDW	Vt,offset(Aa)	BVI	AE	F1,0,0E	0
ST.DDW.T	Vt,offset(Aa)	BVI	AE	F1,0,0E	2
ST.DDW.F	Vt,offset(Aa)	BVI	AE	F1,0,0E	3

Exceptions

AE Register Range (AERRE)

Scalar Register Range (SRRE)

AE Unaligned Reference (AEURE)

ST – Store Vector Double Word

ST.DW	Vt,offset(Aa)
ST.DW.T	Vt,offset(Aa)
ST.DW.F	Vt,offset(Aa)

31 30	29	28	27 22	21 12	11 6	5 0
vm	0	0	opc	immed10	Aa	Vt

Description

The instruction stores a 32-bit vector register to memory. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

Pseudo Code

```
aax = AregIdx(Aa);
vtx = VregIdx(Vt<4:0>);
vmax = VMregIdx(WB.VMA);
effaBase = A[aax] + Offset(opc<1:0>);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            effa = effaBase + VPS * j + VS * i;
            MemStore( effa<47:0>, 32, VP[j]. V32[Vt<5>][vtx]<i> );
        }
    }
}
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*	*VM*
ST.DW	Vt,offset(Aa)	BVI	AE	F1,0,0A	0
ST.DW.T	Vt,offset(Aa)	BVI	AE	F1,0,0A	2
ST.DW.F	Vt,offset(Aa)	BVI	AE	F1,0,0A	3

Exceptions

AE Register Range (AERRE)

AE Unaligned Reference (AEURE)

Scalar Register Range (SRRE)

ST – Store Vector Indexed Double Word

ST.DW	Vt,Vb(Aa)
ST.DW.T	Vt,Vb(Aa)
ST.DW.F	Vt,Vb(Aa)

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	0	1101	opc	Vb	Aa	Vt

Description

The instruction stores a 32-bit vector register to memory using a vector of indexes. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

Pseudo Code

```
aax = AregIdx(Aa);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt<4:0>);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            effa = A[aax] + VP[j].V[vbx][i];
            MemStore( effa<47:0>, 32, VP[j]. V32[Vt<5>] [vtx]<i> );
        }
    }
}
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*	*VM*
ST.DW	Vt,Vb(Aa)	BVI	AE	F4,0,0A	0
ST.DW.T	Vt,Vb(Aa)	BVI	AE	F4,0,0A	2
ST.DW.F	Vt,Vb(Aa)	BVI	AE	F4,0,0A	3

Exceptions

AE Register Range (AERRE)

Scalar Register Range (SRRE)

AE Unaligned Reference (AEURE)

ST – Store Vector Unsigned Double Word

ST.UD	Vt,offset(Aa)
ST.UD.T	Vt,offset(Aa)
ST.UD.F	Vt,offset(Aa)

31 30	29	28	27 22	21 12	11 6	5 0
vm	1	0	opc	immed10	Aa	Vt

Description

The instruction stores the least significant four bytes of a 64-bit vector register to memory. If the upper 32-bits of the source vector register are non-zero then an AE Store Overflow Exception is signaled. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

Pseudo Code

```
aax = AregIdx(Aa);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
effaBase = A[aax] + Offset(opc<1:0>);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            effa = effaBase + VPS * j + VS * i;
            MemStore( effa<47:0>, 32, VP[j]. V[vtx][i] );
        }
    }
}
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*	*VM*
ST.UD	Vt,offset(Aa)	BVI	AE	F1,1,0A	0
ST.UD.T	Vt,offset(Aa)	BVI	AE	F1,1,0A	2
ST.UD.F	Vt,offset(Aa)	BVI	AE	F1,1,0A	3

Exceptions

AE Register Range (AERRE)	Scalar Register Range (SRRE)
AE Unaligned Reference (AEURE)	AE Store Overflow (AESOE)

ST – Store Vector Unsigned Quad Word

ST.UQ	Vt,offset(Aa)
ST.UQ.T	Vt,offset(Aa)
ST.UQ.F	Vt,offset(Aa)

31 30	29	28	27 22	21 12	11 6	5 0
vm	1	0	opc	immed10	Aa	Vt

Description

The instruction stores eight bytes of a vector register to memory. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

Pseudo Code

```
aax = AregIdx(Aa);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
effaBase = A[aax] + Offset(opc<1:0>);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            effa = effaBase + VPS * j + VS * i;
            MemStore( effa<47:0>, 64, VP[j]. V[vtx][i] );
        }
    }
}
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*	*VM*
ST.UQ	Vt,offset(Aa)	BVI	AE	F1,1,0B	0
ST.UQ.T	Vt,offset(Aa)	BVI	AE	F1,1,0B	2
ST.UQ.F	Vt,offset(Aa)	BVI	AE	F1,1,0B	3

Exceptions

AE Register Range (AERRE)

Scalar Register Range (SRRE)

AE Unaligned Reference (AEURE)

ST – Store Vector Signed Double Word

ST.SD	Vt,offset(Aa)
ST.SD.T	Vt,offset(Aa)
ST.SD.F	Vt,offset(Aa)

31 30	29	28	27 22	21 12	11 6	5 0
vm	1	0	opc	immed10	Aa	Vt

Description

The instruction stores the least significant four bytes of a 64-bit vector register to memory. If the upper 32-bits of the source vector register are not a sign extension of the lower 32-bits then an AE Integer Store Exception is signaled. The 'vm' field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

Pseudo Code

```
aax = AregIdx(Aa);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
effaBase = A[aax] + Offset(opc<1:0>);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            effa = effaBase + VPS * j + VS * i;
            MemStore( effa<47:0>, 32, VP[j]. V[vtx][i] );
        }
    }
}
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*	*VM*
ST.SD	Vt,offset(Aa)	BVI	AE	F1,1,0E	0
ST.SD.T	Vt,offset(Aa)	BVI	AE	F1,1,0E	2
ST.SD.F	Vt,offset(Aa)	BVI	AE	F1,1,0E	3

Exceptions

AE Register Range (AERRE)	Scalar Register Range (SRRE)
AE Unaligned Reference (AEURE)	AE Store Overflow (AESOE)

ST – Store Vector Indexed Unsigned Double Word

ST.UD	Vt,Vb(Aa)
ST.UD.T	Vt,Vb(Aa)
ST.UD.F	Vt,Vb(Aa)

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1101	opc	Vb	Aa	Vt

Description

The instruction stores the least significant four bytes of a 64-bit vector register to memory using a vector of indexes. If the upper 32-bits of the source vector register are non-zero then an AE Integer Store Exception is signaled. The 'vm' field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

Pseudo Code

```
aax = AregIdx(Aa);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            effa = A[aax] + VP[j].V[vbx][i];
            MemStore( effa<47:0>, 32, VP[j]. V[vtx][i] );
        }
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
ST.UD	Vt,Vb(Aa)	BVI	AE	F4,1,0A	0
ST.UD.T	Vt,Vb(Aa)	BVI	AE	F4,1,0A	2
ST.UD.F	Vt,Vb(Aa)	BVI	AE	F4,1,0A	3

Exceptions

AE Register Range (AERRE)	Scalar Register Range (SRRE)
AE Unaligned Reference (AEURE)	AE Store Overflow (AESOE)

ST – Store Vector Indexed Unsigned Quad Word

ST.UQ	Vt,Vb(Aa)
ST.UQ.T	Vt,Vb(Aa)
ST.UQ.F	Vt,Vb(Aa)

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1101	opc	Vb	Aa	Vt

Description

The instruction stores a 64-bit vector register to memory using a vector of indexes. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

Pseudo Code

```
aax = AregIdx(Aa);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            effa = A[aax] + VP[j].V[vbx][i];
            MemStore( effa<47:0>, 64, VP[j]. V[vtx][i] );
        }
    }
}
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*	*VM*
ST.UQ	Vt,Vb(Aa)	BVI	AE	F4,1,0B	0
ST.UQ.T	Vt,Vb(Aa)	BVI	AE	F4,1,0B	2
ST.UQ.F	Vt,Vb(Aa)	BVI	AE	F4,1,0B	3

Exceptions

AE Register Range (AERRE)

Scalar Register Range (SRRE)

AE Unaligned Reference (AEURE)

ST – Store Vector Indexed Signed Double Word

ST.SD	Vt,Vb(Aa)
ST.SD.T	Vt,Vb(Aa)
ST.SD.F	Vt,Vb(Aa)

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1101	opc	Vb	Aa	Vt

Description

The instruction stores the least significant four bytes of a 64-bit vector register to memory using a vector of indexes. If the upper 32-bits of the source vector register are not the sign extension of the least significant 32-bits then an AE Integer Store Exception is signaled. The 'vm' field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

Pseudo Code

```
aax = AregIdx(Aa);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            effa = A[aax] + VP[j].V[vbx][i];
            MemStore( effa<47:0>, 32, VP[j]. V[vtx][i] );
        }
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
ST.SD	Vt,Vb(Aa)	BVI	AE	F4,1,0E	0
ST.SD.T	Vt,Vb(Aa)	BVI	AE	F4,1,0E	2
ST.SD.F	Vt,Vb(Aa)	BVI	AE	F4,1,0E	3

Exceptions

AE Register Range (AERRE)	Scalar Register Range (SRRE)
AE Unaligned Reference (AEURE)	AE Store Overflow (AESOE)

ST – Store Vector Mask Register

ST VMt, offset(Aa)

31 30	29	28	27 22	21 12	11 6	5 0
00	0	0	opc	immed10	Aa	Vt

Description

The instruction stores the specified VM register to memory. The effective address must be aligned on an 8-byte boundary. The format of the VM bits within memory is implementation specific.

HC-1 Implementation Details

The instruction writes four bytes per function pipe to memory (a total of 128 bytes are written). The VPM, VL and VPL must be set such that 32 4-byte writes occur. The following table specifies the legal values of VL and VPL for the defined values of VPM. An AE Element Range Exception (AEERE) is signaled if VPM, VL and VPL have values other than those in the table. The table also shows values of VS and VPS that will result in the VM register being written to contiguous locations in memory (no exception is signaled for values of VS and VPS).

VPM	*VL*	*VPL*	*VS*	*VPS*
0 (Classical)	32	ignored	4	ignored
1 (Physical)	8	4	4	32
2 (Short Vector)	1	32	Ignored	4

Future product generations may change the number of function pipes, and or elements per function pipe. These changes would result in a different number of bytes being written to memory.

Pseudo Code

```
aax = AregIdx(Aa);
vmtx = VMregIdx(VMt<5:0>);
effaBase = A[aax] + Offset(opc<1:0>);
MemStore( effa<47:0>, 1024, VM[vmtx] );
```

Instruction Encoding

Instruction	***ISA***	***Type***	***Encoding***	***VM***
ST VMt, offset(Aa)	BVI	AE	F1,1,1B	0

Exceptions

AE Register Range (AERRE)

Scalar Register Range (SRRE)

AE Unaligned Reference (AEURE)

AE Element Range (AEERE)

SUB – Address Subtraction Integer with Immediate

SUB.UQ	Aa,Immed,At
SUB.SQ	Aa,Immed,At

31 29	28	27 22	21 12	11 6	5 0
it	0	opc	immed10	Aa	At

Description

The instruction performs subtraction between an address register and an immediate value using unsigned or signed integer arithmetic. The immediate value is either the zero or sign extended 10-bit Immed10 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code (example for SUB.SQ)

```
aax = AregIdx(Aa);
atx = AregIdx(At);
A[atx] = A[aax] – Simmed10();
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
SUB.UQ	Aa,Immed,At	Scalar	A	F1,32
SUB.SQ	Aa,Immed,At	Scalar	A	F1,33

Exceptions

Scalar Register Range (SRRE)

Scalar Integer Overflow (SIOE)

SUB – Scalar Subtraction Integer with Immediate

SUB.UQ	Sa,Immed,St
SUB.SQ	Sa,Immed,St

31 30	29	28	27 22	21 12	11 6	5 0
it	1	0	opc	immed10	Sa	St

Description

The instruction performs subtraction between a scalar register and an immediate value using unsigned or signed integer arithmetic. The immediate value is either the zero or sign extended 10-bit Immed10 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx] = S[sax] – Simmed10();
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
SUB.UQ	Sa,Immed,St	Scalar	S	F1,1,32
SUB.SQ	Sa,Immed,St	Scalar	S	F1,1,33

Exceptions

Scalar Register Range (SRRE)

Scalar Integer Overflow (SIOE)

SUB – Scalar Subtraction Float Double with Immediate

SUB.FD	Sa,Immed,St
SUB.FD	Immed,Sa,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	0	1100	opc	Immed6	Sa	St

Description

The instruction performs subtraction between a scalar register and an immediate value using float double arithmetic. The immediate value is either the zero or sign extended 6-bit Immed6 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
switch (opc) {
case 0x33: S[stx] = S[sax] – Uimmed6(); break;
case 0x13: S[stx] = Uimmed6() – S[sax]; break;
}
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*
SUB.FD	Sa,Immed,St	Scalar	S	F3,0,33
SUB.FD	Immed,Sa,St	Scalar	S	F3,0,13

Exceptions

Scalar Register Range (SRRE)	Scalar Float Invalid Operand (SFIE)
Scalar Float Overflow (SFOE)	Scalar Float Underflow (SFUE)

SUB – Scalar Subtraction Float Single with Immediate

SUB.FS	Sa,Immed,St
SUB.FS	Immed, Sa,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	0	1100	opc	Immed6	Sa	St

Description

The instruction performs subtraction between a scalar register and an immediate value using float single arithmetic. The immediate value is either the zero or sign extended 6-bit Immed6 field of the instruction, or a 64-bit extended immediate value at IP+8.

Pseudo Code

```
sax = SregIdx(Sa);
stx = SregIdx(St);
S[stx]<63:0> = 0;
switch (opc) {
case 0x33: S[stx]<31:0> = S[sax]<31:0> – Uimmed6()<31:0>; break;
case 0x13: S[stx]<31:0> = Uimmed6()<31:0> – S[sax]<31:0>; break;
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
SUB.FS	Sa,Immed,St	Scalar	S	F3,0,32
SUB.FS	Immed,Sa,St	Scalar	S	F3,0,12

Exceptions

Scalar Register Range (SRRE)	Scalar Float Invalid Operand (SFIE)
Scalar Float Overflow (SFOE)	Scalar Float Underflow (SFUE)

SUB – Address Subtraction Integer

SUB.UQ	Aa,Ab,At
SUB.SQ	Aa,Ab,At

31 29	28 25	24 18	17 12	11 6	5 0
it	1101	opc	Ab	Aa	At

Description

The instruction performs subtraction on two address register values using signed or unsigned integer arithmetic.

Pseudo Code

```
aax = AregIdx(Aa);
abx = AregIdx(Ab);
atx = AregIdx(At);
A[atx] = A[aax] – A[abx];
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
SUB.UQ	Aa,Ab,At	Scalar	A	F4,32
SUB.SQ	Aa,Ab,At	Scalar	A	F4,33

Exceptions

Scalar Register Range (SRRE)

SUB – Scalar Subtraction Integer

SUB.UQ	Sa,Sb,St
SUB.SQ	Sa,Sb,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	1	1101	opc	Sb	Sa	St

Description

The instruction performs subtraction on two scalar register values using signed or unsigned integer arithmetic.

Pseudo Code

```
sax = SregIdx(Sa);
sbx = SregIdx(Sb);
stx = SregIdx(St);
S[stx] = S[sax] – S[sbx];
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
SUB.UQ	Sa,Sb,St	Scalar	S	F4,1,32
SUB.SQ	Sa,Sb,St	Scalar	S	F4,1,33

Exceptions

Scalar Register Range (SRRE)

SUB – Scalar Subtraction Float Double

SUB.FD Sa,Sb,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	0	1101	opc	Sb	Sa	St

Description

The instruction performs subtraction on two scalar register values using float double arithmetic.

Pseudo Code

```
sax = SregIdx(Sa);
sbx = SregIdx(Sb);
stx = SregIdx(St);
S[stx] = S[sax] – S[sbx];
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
SUB.FD	Sa,Sb,St	Scalar	S	F4,0,33

Exceptions

Scalar Register Range (SRRE)	Scalar Float Invalid Operand (SFIE)
Scalar Float Overflow (SFOE)	Scalar Float Underflow (SFUE)

SUB – Scalar Subtraction Float Single

SUB.FS Sa,Sb,St

31 30	29	28 25	24 18	17 12	11 6	5 0
it	0	1101	opc	Sb	Sa	St

Description

The instruction performs subtraction on two scalar register values using float single arithmetic.

Pseudo Code

```
sax = SregIdx(Sa);
sbx = SregIdx(Sb);
stx = SregIdx(St);
S[stx]<63:32> = 0;
S[stx]<31:0> = S[sax]<31:0> – S[sbx]<31:0>;
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***
SUB.FS	Sa,Sb,St	Scalar	S	F4,0,32

Exceptions

Scalar Register Range (SRRE)	Scalar Float Invalid Operand (SFIE)
Scalar Float Overflow (SFOE)	Scalar Float Underflow (SFUE)

SUB – Vector Subtract Signed Quad Word Scalar

SUB.SQ	Va,Sb,Vt
SUB.SQ.T	Va,Sb,Vt
SUB.SQ.F	Va,Sb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	if	1100	opc	Sb	Va	Vt

Description

The instruction subtracts a 64-bit scalar register from each element of a 64-bit vector register using signed quad word integer data format. T The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of the output vector register for masked elements is undefined and exceptions are not recorded for masked elements.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j]. V[vtx] [i] = VP[j]. V [vax][i] - S[sbx];
        else
            VP[j]. V[vtx] [i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
SUB.SQ	Va,Sb,Vt	BVI	AE	F3,1,33	0
SUB.SQ.T	Va,Sb,Vt	BVI	AE	F3,1,33	2
SUB.SQ.F	Va,Sb,Vt	BVI	AE	F3,1,33	3

Exceptions

AE Integer Overflow (AEIOE)

AE Register Range (AERRE)

Scalar Register Range (SRRE)

SUB – Vector Subtract Signed Quad Word Scalar

SUB.SQ	Sb,Va,Vt
SUB.SQ.T	Sb,Va,Vt
SUB.SQ.F	Sb,Va,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	**if**	**1100**	**opc**	**Sb**	**Va**	**Vt**

Description

The instruction subtracts each element of a 64-bit vector register from 64-bit scalar register using signed quad word integer data format. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of the output vector register for masked elements is undefined and exceptions are not recorded for masked elements.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j]. V[vtx] [i] = S[sbx] - VP[j]. V [vax][i];
        else
            VP[j]. V[vtx] [i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
SUB.SQ	Sb,Va,Vt	BVI	AE	F3,1,13	0
SUB.SQ.T	Sb,Va,Vt	BVI	AE	F3,1,13	2
SUB.SQ.F	Sb,Va,Vt	BVI	AE	F3,1,13	3

Exceptions

AE Integer Overflow (AEIOE)	AE Register Range (AERRE)
Scalar Register Range (SRRE)	

SUB – Vector Subtract Signed Quad Word

SUB.SQ	Va,Vb,Vt
SUB.SQ.T	Va,Vb,Vt
SUB.SQ.F	Va,Vb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1101	opc	Vb	Va	Vt

Description

The instruction subtracts two 64-bit vector registers using signed quad word integer data format. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of the output vector register for masked elements is undefined and exceptions are not recorded for masked elements.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j].V[vtx][i] = VP[j]. V[vax][i] - VP[j].V[vbx][i];
        else
            VP[j].V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
SUB.SQ	Va,Vb,Vt	BVI	AE	F4,1,33	0
SUB.SQ.T	Va,Vb,Vt	BVI	AE	F4,1,33	2
SUB.SQ.F	Va,Vb,Vt	BVI	AE	F4,1,33	3

Exceptions

AE Integer Overflow (AEIOE)

AE Register Range (AERRE)

SUB – Vector Subtract Unsigned Quad Word Scalar

SUB.UQ	Va,Sb,Vt
SUB.UQ.T	Va,Sb,Vt
SUB.UQ.F	Va,Sb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	**if**	**1100**	**opc**	**Sb**	**Va**	**Vt**

Description

The instruction subtracts a 64-bit scalar register from each element of a 64-bit vector register using unsigned quad word integer data format. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of the output vector register for masked elements is undefined and exceptions are not recorded for masked elements.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j]. V[vtx] [i] = VP[j]. V [vax][i] - S[sbx];
        else
            VP[j]. V[vtx] [i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
SUB.UQ	Va,Sb,Vt	BVI	AE	F3,1,32	0
SUB.UQ.T	Va,Sb,Vt	BVI	AE	F3,1,32	2
SUB.UQ.F	Va,Sb,Vt	BVI	AE	F3,1,32	3

Exceptions

AE Integer Overflow (AEIOE)	AE Register Range (AERRE)
Scalar Register Range (SRRE)	

SUB – Vector Subtract Unsigned Quad Word Scalar

SUB.UQ	Sb,Va,Vt
SUB.UQ.T	Sb,Va,Vt
SUB.UQ.F	Sb,Va,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	if	1100	opc	Sb	Va	Vt

Description

The instruction subtracts each element of a 64-bit vector register from 64-bit scalar register using unsigned quad word integer data format. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of the output vector register for masked elements is undefined and exceptions are not recorded for masked elements.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j]. V[vtx] [i] = S[sbx] - VP[j]. V [vax][i];
        else
            VP[j]. V[vtx] [i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
SUB.UQ	Sb,Va,Vt	BVI	AE	F3,1,12	0
SUB.UQ.T	Sb,Va,Vt	BVI	AE	F3,1,12	2
SUB.UQ.F	Sb,Va,Vt	BVI	AE	F3,1,12	3

Exceptions

AE Integer Overflow (AEIOE)

AE Register Range (AERRE)

Scalar Register Range (SRRE)

SUB – Vector Subtract Unsigned Quad Word

SUB.UQ	Va,Vb,Vt
SUB.UQ.T	Va,Vb,Vt
SUB.UQ.F	Va,Vb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1101	opc	Vb	Va	Vt

Description

The instruction subtracts two 64-bit vector registers using unsigned quad word integer data format. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of the output vector register for masked elements is undefined and exceptions are not recorded for masked elements.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j].V[vtx][i] = VP[j]. V[vax][i] - VP[j].V[vbx][i];
        else
            VP[j].V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
SUB.UQ	Va,Vb,Vt	BVI	AE	F4,1,32	0
SUB.UQ.T	Va,Vb,Vt	BVI	AE	F4,1,32	2
SUB.UQ.F	Va,Vb,Vt	BVI	AE	F4,1,32	3

Exceptions

AE Integer Overflow (AEIOE) AE Register Range (AERRE)

SUMR – Vector Summation Reduction Integer

SUMR.UQ	Va,Vt	SUMR.SQ	Va,Vt
SUMR.UQ.T	Va,Vt	SUMR.SQ.T	Va,Vt
SUMR.UQ.F	Va,Vt	SUMR.SQ.F	Va,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	1	1100	opc	000000	Va	Vt

Description

The instruction performs a summation reduction on a vector register. The result is one or more partial results per partition (implementation dependent). If more than one result is produced per partition then scalar instructions must be used to complete the reduction operation to a single value. The ***MOV REDcnt,At*** instruction can be used to obtain the number of partial reduction results produced per partition. The ***MOVR Va,Ab,St*** instruction must be used to obtain the reduction results with the A register value indicating the index of the partial result being accessed. The order that the vector elements are combined is implementation dependent. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

If a load instruction with VS=0 was previously used to load the MAXR source vector register (Va), then the result of the reduction is undefined and an AEERE exception is issued. Note that the exception is issued independent of the number of instructions that separate the load with VS=0 and the reduction instruction.

Pseudo Code (example code produces one result per partition)

```
vmax = VMregIdx(WB.VMA);
vax = VregIdx(Va);
vtx = VregIdx(Vt);
for (j = 0; j < AEC.VPL; j += 1) {
    tmp = 0;
    for (i = 0; i < AEC.VL; i += 1)
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            tmp += VP[j].V[vax][i];
    VP[j]. V[vtx][0] = tmp;
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
SUMR.UQ	Va,Vt	BVI	AE	F3,1,48	0
SUMR.UQ.T	Va,Vt	BVI	AE	F3,1,48	2
SUMR.UQ.F	Va,Vt	BVI	AE	F3,1,48	3
SUMR.SQ	Va,Vt	BVI	AE	F3,1,49	0

SUMR.SQ.T	Va,Vt	BVI	AE	F3,1,49	2
SUMR.SQ.F	Va,Vt	BVI	AE	F3,1,49	3

Exceptions

AE Register Range (AERRE)	AE Element Range (AEERE)

TZC – Trailing Zero Count VM Register

TZC VMa,At

31 29	28 24	23 18	17 12	11 6	5 0
00	11101	opc	000000	VMa	At

Description

The instruction counts the number of trailing zero bits (starting with the least significant bit, bit zero) in the VMa register for the active partition.

The AE Element Range Exception is set if VPA is invalid for the vector partition mode.

Pseudo Code

```
vmax = VMregIdx(VMa);
atx = AregIdx(At);
A[atx] = TrailingZeroCount(VP[VPA].VM[vmax]);
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
TZC	VMa,At	BVI	AE	F6,1,0E	0

Exceptions

Scalar Register Range (SRRE)

AE Register Range (AERRE)

AE Element Range (AEERE)

VIDX – Vector Index

VIDX	Aa,SH,Vt
VIDX.T	Aa,SH,Vt
VIDX.F	Aa,SH,Vt

31 30	29	28 24	23 18	17 12	11 6	5 0
vm	1	11101	opc	SH	Aa	Vt

Description

Generate vector element indices, shift the index by the constant SH value, and add the value of an A register. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements.

Pseudo Code

```
shftAmt = SH & 0x3;    // valid shift amount is 0-3
vtx = VregIdx(Vt);
aax = AregIdx(At);
vmax = VMregIdx(WB.VMA);
for (j = 0; j < AEC.VPL; j += 1) {
    idx = 0;
    for (i = 0; i < AEC.VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>) {
            VP[j].V[vtx][i] = (idx << shftAmt) + A[aax];
            idx += 1;
        } else
            VP[j].V[vtx][i] = UndefinedValue();
    }
}
```

Opc Field Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
VIDX	Aa,SH,Vt	BVI	AE	F5,1,00	0
VIDX.T	Aa,SH,Vt	BVI	AE	F5,1,00	2
VIDX.F	Aa,SH,Vt	BVI	AE	F5,1,00	3

Exceptions

AE Register Range (AERRE)   Scalar Register Range (SRRE)

VRRINC – Increment the VRRO field of the WB Register

VRRINC

31 29	28 24	23 18	17 12	11 6	5 0
it	**11110**	**opc**	**000000**	**000000**	**000001**

Description

The instruction increments the VRRO field of the WB register. If the VRRO field is greater than or equal to the VRRS field after the increment, then VRRO field is set to zero.

Pseudo Code

```
WB.VRRO = (WB.VRRO + 1) >= WB.VRRS ? 0 : (WB.VRRO + 1);
```

Instruction Encoding

Instruction	*ISA*	*Type*	*Encoding*
VRRINC	Scalar	A	F7,08

Exceptions

None

VRRSET – Set VRR fields of WB Register

VRRSET VRRB,VRRS,VRRO

31 29	28 24	23 18	17 12	11 6	5 0
it	**11110**	**opc**	**VWRB**	**VWRS**	**VWRO**

Description

The instruction moves three 6-bit immediate values to the Vector Register Rotate fields of the WB register.

Pseudo Code

```
WB.VRRB = VRRB;                          // Vector Register Rotate Base field
WB.VRRS = VRRS;                          // Vector Register Rotate Size field
WB.VRRO = VRRO < VRRS ? VRRO : 0;        // Vector Register Rotate Offset field
```

Instruction Encoding

Instruction	***ISA***	***Type***	***Encoding***
VRRSET VRRB,VRRS,VRRO	Scalar	A	F7,09

Exceptions

None

VSHF – Shift Vector Elements

VSHF Va,Sb,Vt

31 30	29	28 25	24 18	17 12	11 6	5
00	if	1100	opc	Sb	Va	Vt

Description

The instruction shifts elements down to lower indices from a source vector register into a destination vector register. The most significant vector register element of the destination register is loaded with the scalar register value.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vtx = VregIdx(Vt);
for (j = 0; j < AEC.VPL; j += 1) {
    for (i = 0; i < VL - 1; i += 1)
        VP[j].V[vtx][i] = VP[j].V[vax][i+1];    // shift vector element down by one element
    VP[j].V[vtx][i] = S[sbx];                   // most significant vector element gets scalar
}
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*
VSHF	Va,Sb,Vt	BVI	AE	F3,1,57

Exceptions

AE Register Range (AERRE)
Scalar Register Range (SRRE)

WBDEC – Decrement the WB fields of WB Register

WBDEC AWB,SWB,VWB,VMWB

31 29	28 24	23 18	17 13	12 8	7 3	2 0
it	11110	opc	AWB	SWB	VWB	VMWB

Extended Immediate Format:

63	51	39	29	23	17	11	5 0
AWB	SWB	VWB	000000	000000	000000	VMWB	000000

Description

The instruction decrements the four Window Base fields of the WB register using the four immediate values. A Scalar Register Range Exception is signaled decrementing the window base fields would result in a negative value.

Pseudo Code

```
WB.AWB -= AWB;        // Address Register Window Base
WB.SWB -= SWB;        // Scalar Register Window Base
WB.VWB -= VWB;        // Vector Register Window Base
WB.VMWB -= VMWB; // Vector Mask Register Window Base
```

Instruction Encoding

Instruction	***ISA***	***Type***	***Encoding***
WBDEC AWB,SWB,VWB,VMWB	Scalar	A	F7,02

Exceptions

Scalar Register Range (SRRE)

WBINC – Increment the WB fields of WB Register

WBINC	AWB,SWB,VWB,VMWB
WBINC.P	AWB,SWB,VWB,VMWB

31 29	28 24	23 18	17 13	12 8	7 3	2 0
it	11110	opc	AWB	SWB	VWB	VMWB

Extended Immediate Format:

63	51	39	29	23	17	11	5	0
AWB	SWB	VWB	000000	000000	000000	VMWB	000000	

Description

The instruction increments the four Window Base fields of the WB register using the four immediate values. A Scalar Register Range Exception is signaled when incrementing the window base fields results in a field overflow.

The WBINC.P instruction additionally writes the WB register (prior to modification) to the top entry of the call / return stack (CRS) and sets the WBV bit of the CRS register.

Pseudo Code

```
If (WBINC.P) {
    CRS[CPS.CRT  8].RWB = WB;
    CPS.WBV = 1;
    If (CPS.CRT >= 8)
        CPS.SCOE = 1;
}
WB.AWB += AWB;       // Address Register Window Base
WB.SWB += SWB;       // Scalar Register Window Base
WB.VWB += VWB;       // Vector Register Window Base
WB.VMWB += VMWB;  // Vector Mask Register Window Base
WB.VMA = 0;
WB.VRRS = 0;
```

Instruction Encoding

Instruction		*ISA*	*Type*	*Encoding*
WBINC	AWB,SWB,VWB,VMWB	Scalar	A	F7,00
WBINC.P	AWB,SWB,VWB,VMWB	Scalar	A	F7,01

Exceptions

Scalar Register Range (SRRE)

WBSET – Set WB fields of WB Register

WBSET   AWB,SWB,VWB,VMWB

31 29	28 24	23 18	17 13	12 8	7 3	2 0
it	11110	opc	AWB	SWB	VWB	VMWB

Extended Immediate Format:

63	51	39	29	23	17	11	5 0
AWB	SWB	VWB	000000	000000	000000	VMWB	000000

Description

The instruction moves four immediate values to the Window Base fields of the WB register.

Pseudo Code

```
WB.AWB = AWB;       // Address Register Window Base
WB.SWB = SWB;       // Scalar Register Window Base
WB.VWB = VWB;       // Vector Register Window Base
WB.VMWB = VMWB;  // Vector Mask Register Window Base
```

Instruction Encoding

Instruction	***ISA***	***Type***	***Encoding***
WBSET AWB,SWB,VWB,VMWB	Scalar	A	F7,03

Exceptions

None

XNOR – Vector Logical Xnor

XNOR	Va,Vb,Vt
XNOR.T	Va,Vb,Vt
XNOR.F	Va,Vb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	**if**	**1100**	**opc**	**Sb**	**Va**	**Vt**

Description

The instruction performs a logical ‘xnor’ operation between the two 64-bit source vector registers. The result of the operation is written to a 64-bit destination vector register. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of masked elements is undefined.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(VMA);
for (j = 0; j < VPL; j += 1) {
    for (i = 0; i < VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j].V[vtx][i] = ~(VP[j].V[vax][i]  ^ VP[j].V[vbx][i];
        else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
XNOR	Va,Vb,Vt	BVI	AE	F4,1,25	0
XNOR.T	Va,Vb,Vt	BVI	AE	F4,1,25	2
XNOR.F	Va,Vb,Vt	BVI	AE	F4,1,25	3

Exceptions

AE Register Range (AERRE)

XNOR – Vector Logical Xnor with Scalar

XNOR	Va,Sb,Vt
XNOR.T	Va,Sb,Vt
XNOR.F	Va,Sb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	if	1100	opc	Sb	Va	Vt

Description

The instruction performs a logical ‘xnor’ operation between a 64-bit source vector register and an S register. The result of the operation is written to a 64-bit destination vector register. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of masked elements is undefined.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vtx = VregIdx(Vt);
vmax = VMregIdx(VMA);
for (j = 0; j < VPL; j += 1) {
    for (i = 0; i < VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j].V[vtx][i] = ~(VP[j].V[vax][i]  ^ S[sbx]);
        else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
XNOR	Va,Sb,Vt	BVI	AE	F3,1,25	0
XNOR.T	Va,Sb,Vt	BVI	AE	F3,1,25	2
XNOR.F	Va,Sb,Vt	BVI	AE	F3,1,25	3

Exceptions

AE Register Range (AERRE)

Scalar Register Range (SRRE)

XOR – Vector Logical Xor

XOR	Va,Vb,Vt
XOR.T	Va,Vb,Vt
XOR.F	Va,Vb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	if	1100	opc	Sb	Va	Vt

Description

The instruction performs a logical ‘xor’ operation between the two 64-bit source vector registers. The result of the operation is written to a 64-bit destination vector register. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of masked elements is undefined.

Pseudo Code

```
vax = VregIdx(Va);
vbx = VregIdx(Vb);
vtx = VregIdx(Vt);
vmax = VMregIdx(VMA);
for (j = 0; j < VPL; j += 1) {
    for (i = 0; i < VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j].V[vtx][i] = VP[j].V[vax][i]  ^ VP[j].V[vbx][i];
        else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
XOR	Va,Vb,Vt	BVI	AE	F4,1,24	0
XOR.T	Va,Vb,Vt	BVI	AE	F4,1,24	2
XOR.F	Va,Vb,Vt	BVI	AE	F4,1,24	3

Exceptions

AE Register Range (AERRE)

XOR – Vector Logical Xor with Scalar

XOR	Va,Sb,Vt
XOR.T	Va,Sb,Vt
XOR.F	Va,Sb,Vt

31 30	29	28 25	24 18	17 12	11 6	5 0
vm	if	1100	opc	Sb	Va	Vt

Description

The instruction performs a logical ‘xor’ operation between a 64-bit source vector register and an S register. The result of the operation is written to a 64-bit destination vector register. The ‘vm’ field can be used to specify masked operations. The VMA register specifies the VM register used to mask vector elements. The value of masked elements is undefined.

Pseudo Code

```
vax = VregIdx(Va);
sbx = SregIdx(Sb);
vtx = VregIdx(Vt);
vmax = VMregIdx(VMA);
for (j = 0; j < VPL; j += 1) {
    for (i = 0; i < VL; i += 1) {
        if (vm==0 || vm==2 && VP[j].VM[vmax]<i> || vm==3 && !VP[j].VM[vmax]<i>)
            VP[j].V[vtx][i] = VP[j].V[vax][i]  ^ S[sbx];
        else
            VP[j]. V[vtx][i] = UndefinedValue();
    }
}
```

Instruction Encoding

Instruction		***ISA***	***Type***	***Encoding***	***VM***
XOR	Va,Sb,Vt	BVI	AE	F3,1,24	0
XOR.T	Va,Sb,Vt	BVI	AE	F3,1,24	2
XOR.F	Va,Sb,Vt	BVI	AE	F3,1,24	3

Exceptions

AE Register Range (AERRE) Scalar Register Range (SRRE)