CONVEX Architecture Handbook

Document No. 080-000120-000

Version 1.0

CONVEX Computer Corporation

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Preface
The CONVEX Family: Systems Methodology

The architecture of the CONVEX family of 64-bit supercomputers was designed and developed as the result of careful insight and in response to real customer demand.

The dominance of the 64-bit supercomputer as the mainstay of the computer market in the 80's is explained in part by the incessant demand for increased processing performance inherent in scientific applications.

However, for the high-end scientific market, general purpose computers do not supply the exceptional performance required. Coupling the host to an external array processor improves performance, but such systems are difficult to program and are expensive. Realizing this shortcoming and anticipating customer demand, in 1982 CONVEX Computer Corporation made the commitment to design and develop a family of sophisticated, high-performance computers for the 1980's and beyond.

At CONVEX Computer Corporation, the concern was to provide the scientific market with a computer family which would offer the same performance levels as the most sophisticated systems on the market while also providing superior software support and programmer productivity tools--all at a price favorable to our customers. Responding to customer demand and foreseeing the need for the total computer system, CONVEX architects and design engineers designed the CONVEX family of computers.

What resulted was the architecture for the whole CONVEX family of computers, and, notably, CONVEX-1, the world's first scientific 64-bit supercomputer with integrated vector processing capabilities. Offering the best performance levels in the minicomputer industry, our computer family combines proven high-performance architecture with advanced semiconductor technology, programmed using state-of-the-art software tools.

Central to the development of the CONVEX family of computers is the CONVEX architecture, where the incorporation of a vector processor as an integral part of the system resulted in new performance levels and user benefits. These customer benefits were achieved as a result of the dynamic marriage of VLSI, the CONVEX instruction set architecture, and interactive, user-friendly software development tools. For our customers, these benefits include the following:

* One integrated system means lower total system cost.

* Programmer productivity is optimized.

* Efficient implementation in VLSI and high throughput are achieved through the use of a RISC (Reduced Instruction Set Computer) architecture.

* A state-of-the-art globally optimizing and vectorizing FORTRAN compiler assures excellent programmer productivity and optimum speed for the scientific market.

* The CONVEX family is fully supported by a robust, flexible software system that combines efficient implementation with high performance in user-friendly distributed systems.

* UNIX**, the industry-standard operating system, supports all user application needs: real time for time-critical applications; highly-interactive multi-programming, and networking support for distributed environments.

Finally, the CONVEX architecture itself needs special consideration. Designed specifically for the scientific market from a solid base (the CONVEX-1), the architecture prescribes a systems methodology for future computers. Our pride in our achievement is derived from the architectural design itself:

* Central to the architectural design of the CONVEX family of supercomputers is the integrated vector processor for optimal performance.

* The family supports a full range of fixed and floating point data types for scientific applications.

* The system offers 4 Gigabytes of virtual memory with 2 Gigabytes available to each user for the support of massive user programs and data.

* The large, high speed register sets --address, scalar, and vector--guarantee high performance for address calculations in parallel with scalar and vector calculations.

* The CONVEX family is developed around a RISC (Reduced Instruction Set Computer) architecture for optimum implementation and throughput.

* A clean and powerful multi-level protection mechanism supports and separates users, enhancing total system reliability, and increasing the performance of operating system functions.

* Exceptional scalar performance for data management and system control functions is assured.

* Rapid subroutine entry and exit ensure the remarkable execution speeds specified in the architectural design.

This handbook has been prepared as an invitation to the user to share with us in the perspicuous design of the CONVEX family of supercomputers.

CHAPTER 1

1 Introduction

This document is an architectural specification for the CONVEX family of compatible 64-bit supercomputers. The term architecture has been crisply defined as "the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation" [Amdahl, 1]. Within this context, an architectural specification defines the instruction set, the structure of the logical address space "Logical Address Space" and protection mechanisms, the fault, trap, and interrupt facilities, and the I/O space as perceived by the programmer and properly embodied by the hardware designer. Specifically, in CONVEX machines, the central processing unit is the portion of the machine which is defined by this architecture. The document is not meant to convey a particular implementation. However, no architectural decisions and trade-offs are made without careful consideration of the potential effects on both hardware and software.

Throughout this document, we at CONVEX have presented rationales for many design decisions. This explication is included in order to forestall the obvious questions and to enable the user to understand the primary design motivations. These motivations are: the orthogonal projection of the instruction set; the reduction in hardware complexity (especially as it relates to performance and instruction decoding--hence the adoption of the RISC architecture), and the universal guidelines of "garbage in/garbage out." A frequently used term for this type of design is perspicuous: simple, elegant, and easy to understand.

In order to introduce the reader to the organization and content of this handbook, the following pages contain brief summaries of the book's various chapters. The chapters are arranged as follows:

- o Data Types
- o Register Set
- o Logical Address Space
- o Memory Management
- o Protection System
- o Exceptions
- o I/O and Interrupts
- o Instruction Set

1.1 Data Types

There are three generic scalar data types: numeric fixed point integer, numeric floating point, and unsigned numeric values. An array structure (ordered sequence) is provided for each of these data types. An array has four general characteristics:

- o data types
- o rank or dimension
- o length
- o stride

Refer to Figure 1-1 for a graphic representation of array terminology. In this example, A is a 3 by 4 array of words. An array is a data structure, composed of elements. In this case, the elements are words. Data types are manipulated by instructions defined in the latter sections of this handbook.

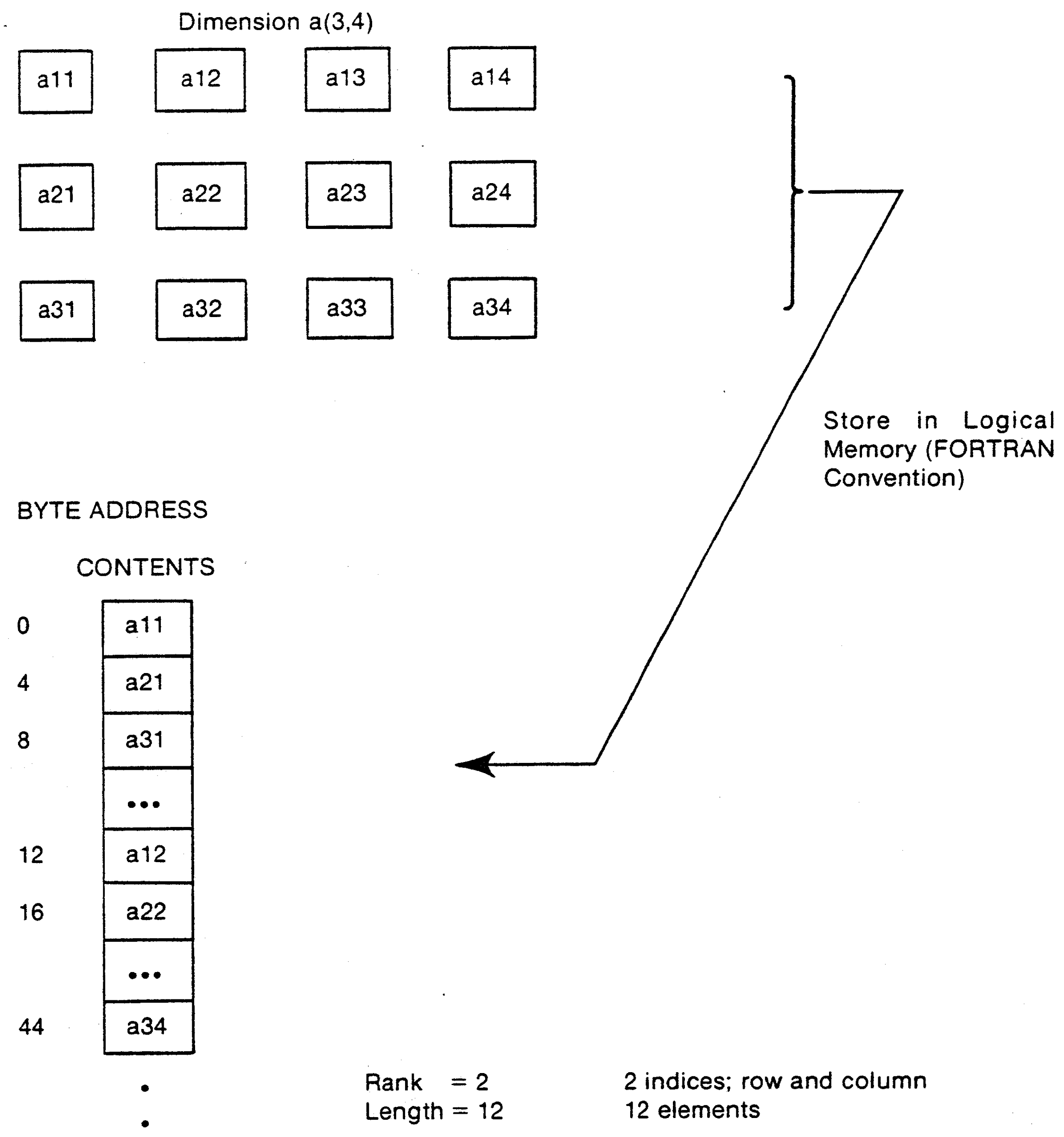
1.2 Register Sets

There are three general register sets and several status registers. The three general register sets are:

- o Address registers (8 x 32 bits)
- o Scalar registers (8 x 64 bits)
- o Vector Registers (8 vectors, each 128 elements x 64-bits)

The registers are partitioned according to the operand to be manipulated: addresses (and scalar indices), scalars, and vectors.

Figure 1-1: Vector Terminology



Rank	= 2	2 indices; row and column
Length	= 12	12 elements
Stride	= 4 or 12	Distance between elements in the same dimension: along columns, stride is 4 bytes; along rows, stride is 12 bytes.

1.2.1 Fixed Point Integer

The four fixed point integer representations--8, 16, 32, and 64 bits--(also referred to as byte, halfword, word, and longword, respectively) correspond to the following FORTRAN lengths: INTEGER*1, INTEGER*2, INTEGER*4, and INTEGER*8, respectively. Fixed point numbers use the 2's complement numbering system.

1.2.2 Floating Point

There are two floating point number representations: single precision word (32 bits) and double precision longword (64 bits). These formats are interpreted as binary normalized fractions with an implicit "1" bit in the most significant bit position of the fraction. The exponent is in biased form.

1.2.3 Unsigned Values/Addresses

An address or logical value is treated as unsigned. Addresses are 32 bits in length and reside in the address registers. For numeric purposes, an address is treated as an unsigned 32 bit integer.

1.2.4 Data Type Memory Alignment

The CONVEX logical address is byte granular; thus, all of the operands can begin on any byte boundary. However, performance may degrade if some data types are not aligned on an integral boundary (e.g., 64 bit integer operands are aligned if the least significant 3 bits of their byte address are 000).

1.3 Logical Address Space

The CONVEX architecture offers four Gigabytes (4.3 billion bytes) of virtual memory in its logical address space partitioned into 8 x 512 megabyte segments. Of these 8 segments, 4 are allocated to the user and 4 to the operating system: thus, the maximum user program (instructions and data) is 2 Gigabytes.

This allocation means that a user program (instructions and data) written in FORTRAN can occupy up to 2 Gigabytes of virtual storage. The operating system data structures and instructions necessary to manage the user program occupy the remaining 2 Gigabytes of virtual storage.

1.3.1 Memory Management

The memory management hardware permits the operating system to provide an extremely flexible and reliable virtual memory programming environment. As mentioned above, the logical address space of the CONVEX architecture is virtual; so, even though an address may be a valid logical address, the referenced data may or may not be in main or physical memory. Memory is managed on a fixed-size page basis. To manage the memory, the CONVEX architecture defines and supports several attributes:

- o Segment Descriptor Register--a 32-bit register that contains a pointer to the first level page table.
- o Page--a contiguous group of bytes, in particular, 4096. A page is both logically and physically contiguous.
- o Page Frame--a page that is stored in main memory.
- o Page Tables--a page that contains entries called Page Table Entries (PTE). A pagetable begins on an integral page boundary and is contained in one pageframe or less. First level page tables contain PTE's which have pointers to second level page tables; second level page tables contain pointers to physical page frames.
- o Page Table Entry (PTE)--a 4 byte entry (32 bits) that conveys information to determine, for instance, whether or not a page is resident in main memory.
- o Referenced and Modified bits--these determine whether a valid memory read or write has occurred.
- o Address Translation Unit--a programmer invisible address cache that maintains the most recently used logical to physical address translations.

1.3.2 Memory Protection System

The architecture of the CONVEX family of supercomputers provides a full 4 gigabyte virtual address space, supporting very large user programs. In addition, the program environment is enhanced by embedding the operating system in the user address space. From these vantage points, the user and the system benefit enormously. However, to realize these benefits fully, the operating system must be protected from the user. The protection system designed into the architecture protects the user, his programs, and other user's programs, while also supporting contemporary notions of sharing and operating system structures. This system is based on hierarchical structures called rings and has been designed to:

- o Support the embedding of the operating system in the user logical address space
- o Contain certain violations to the user's process
- o Permit the implementation of the reliable and robust UNIX operating system
- o Enhance the performance of operating system call processing by reducing the time for context switching

1.4 Exceptions, I/O, and Interrupts

Exceptions are invoked when problems occur in a currently executing program (arithmetic inconsistencies or Address Translation Faults, for example), or as a result of some asynchronous event (such as an interrupt). Control is then transferred to some predetermined location, the value of which is a function of the exception.

To the processor, all I/O data references are memory mapped, which means that there are no explicit I/O data reference instructions. I/O registers and status bits are referenced through the appropriate logical to physical address mapping. The I/O register space is 1 billion bytes: in essence, up to one billion I/O registers can be referenced. Generally, I/O operand references must be on an integral boundary, so that the least significant address bits equal to the precision of the referenced operand will be all 0.

Interrupts are a result of asynchronously occurring events and belong to the system and not to the executing process. When an interrupt occurs, the processor will vector to a particular handler as a function of the source of the interrupt.

1.5 Instruction Set

The CONVEX family of supercomputers offers an extremely powerful instruction set designed to provide maximum functionality per instruction consistent with ease of hardware decode and orthogonality of specification. The instruction set is used to generate logical addresses, load, store, and manage operands, and manipulate the virtual machine mechanisms.

A CONVEX instruction is one of three lengths: one, two, or three halfwords (equivalent to instructions of 16, 32, or 48 bits in length). Even though the fundamental unit of addressability is the byte, instructions are addressed on a halfword boundary. All instructions begin on even byte boundaries.

1.6 The Rest of the Document

The following sections of the document examine the architecture of the CONVEX computer family in detail.

Chapter 2 discusses the organization of data types and the addressing modes available.

Chapter 3 provides full coverage of the register set.

Chapter 4 describes in full the protection system.

Chapter 5 gives an overview of the logical address space and memory management structures.

Chapter 6 provides details on exceptions.

Chapter 7 discusses the I/O's and interrupts' capabilities.

Chapter 8 summarizes the instruction set format.

Chapter 9 offers a tutorial on the address register instructions and provides a detailed description of each instruction available. All entries are organized after the order in which they are discussed in the tutorial, and each includes a full description of the operation and effects of that instruction. Chapters 10, 11, 12, 13, 14, 15, and 16 are organized in the same way.

Chapter 10 covers the scalar register instruction set.

Chapter 11 describes the program control instruction set.

Chapter 12 details the privileged instruction set.

Chapter 13 defines the vector/scalar instruction set.

Comparisons/mask/merge/compress instructions are covered in Chapter 14.

Chapter 15 lists the vector reduction instructions.

Chapter 16 describes the VL, VS, and VM instruction set.

Appendix A lists notational conventions used in the handbook.

Appendix B sorts the operand codes by number.

Appendix C sorts the operand codes by name.

Appendix D contains the assembler notation.

Appendix E covers floating point computations.

Appendix F is a glossary of technical terms.

Further Reading

This handbook describing the architecture of the CONVEX computer family is complemented by The CONVEX Hardware Handbook. This text details the hardware characteristics of each CONVEX implementation and lists all of the particulars that would be of interest to a system or application programmer, as well as to a hardware designer.

Readers should note that a list of notational conventions, glossary of technical terms, and complete index are included in the latter sections of this text. The notational conventions have been established because a baseline methodology and set of consistent definitions are required for a proper understanding of this document. Finally, a feedback form is enclosed as the penultimate page in the handbook, and readers are invited to share with us their observations on the service and clarity of this text.

Notes

[Amdahl,1] G. M. Amdahl, G.A. Blaauw, and F. P. Brooks, Jr. "Architecture of the IBM System/360." IBM Journal of Research and Development 8, 2 (April 1964), 87-101.

CHAPTER 2

2 Data Types

There are 3 generic scalar data types: numeric fixed point integer, numeric floating point, and unsigned value. An address or logical value is treated as unsigned. An array structure is provided for each scalar data type, except unsigned. An array is an ordered sequence of any of these scalar data types. Arrays have four general characteristics: data type, rank or dimension, length, and stride. The rank or dimension is simply the number of indices necessary to reference a particular element. The length is the total number of elements in the entire array. For example, an array with 10 rows and 10 columns is a rank 2 array that has 100 elements. The length of the array is limited by the compiler and the logical address space. The stride is the distance in bytes between adjacent array elements along the same dimension. For example, for a one dimension word vector, the stride is 4 bytes. From the user's perspective, data types supported by the processor are specified below.

In the appropriate chapters, instructions are defined which manipulate the data types presented in this chapter. Unless otherwise specified mixed mode arithmetics on data types or manipulations on operands in registers must rigorously follow the conventions provided. Any attempt to circumvent these conventions through knowledge of an internal representation can be dangerous and is not recommended. If conventions are circumvented, it is not guaranteed that results can be reproduced from one implementation to another.

2.1 Fixed Point Integer

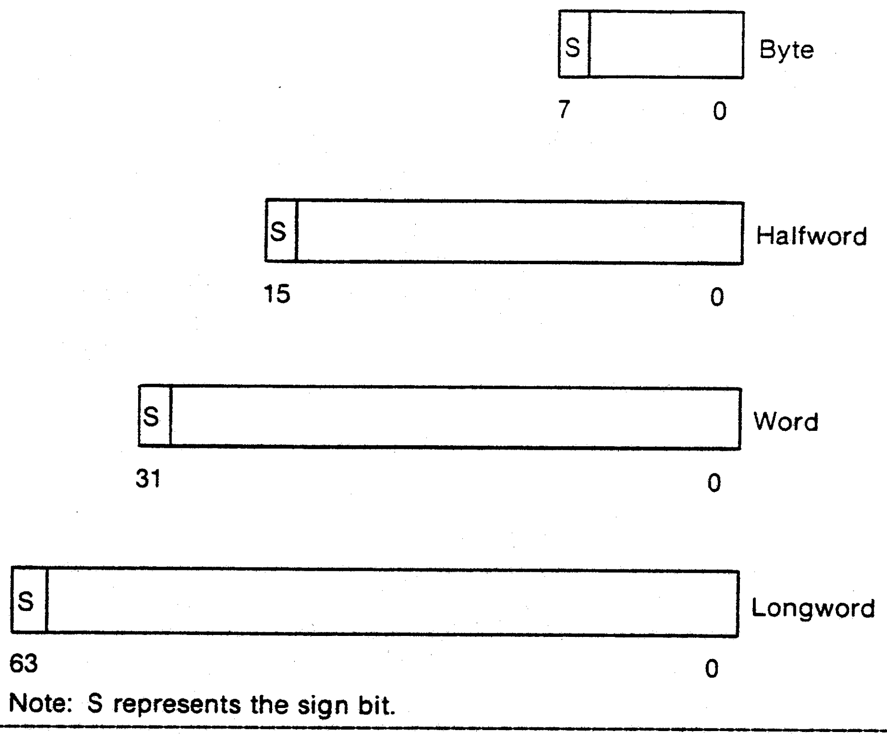
There are four fixed point integer representations: 8, 16, 32, and 64 bits. These numbers are referred to as; byte, halfword, word, and longword respectively. These numbers also correspond to the following Fortran lengths: INTEGER*1, INTEGER*2, INTEGER*4, INTEGER*8 respectively.

Fixed point numbers use the 2's complement numbering system, which means that the most significant bit, the sign bit, has a value equal to (where n is the bit position within the number num<n>):

(-1)*(num<n>)*(2**(n)).

All other bits have a value equal to (num<n>)*(2**(n)). The format of these four fixed point data types is shown in Figure 2-1.

Figure 2-1: Fixed Point Integer Representations



A scalar fixed point integer or unsigned value can be loaded into one of two types of registers (the register set is described in a later chapter): the address or scalar. Generally, operands that are to be used as an address or index value, or that are to be manipulated in parallel to a computation performed in the scalar or vector registers, are loaded into the address registers. The address registers are 32 bits in length. Thus, longword operands cannot be directly loaded into an address register. Operands that are to be used for numeric processing only are generally loaded into the scalar registers.

When operands with a precision less than the destination register are loaded, the remaining bits of the register are unchanged. For example, when a 16-bit integer is loaded into a 32-bit A register, the higher order 16 bits of the A register (A<32..16>) are unchanged.

Thus, a byte is loaded into bits<7..0> of a register; a halfword is loaded into bits<15..0> of a register; a word(integer or single precision) is loaded into bits<31..0> of a register, and longword is loaded into bits<63..0> of a register.

Because of this register and operand (address and scalar) partition, processor architectures can be constructed which allow for asynchronous and overlapped fetch and execute units. Consequently, all address calculations can be done in parallel to numeric calculations. This feature is highly desirable and provides for increased performance.

2.2 Floating Point

There are two floating point number representations: single precision word (32 bits) and double precision longword (64 bits). These formats and their interpretation follow the CONVEX F-format single and CONVEX G-format double precision architecture. These formats have normalized binary fractions and biased binary exponents. The fractions have an implicit "1" bit in the most significant bit position. The format of the single precision (32 bit) floating point number is:

```
 ----------------------------------
 |S| Exponent |  Fraction         |
 ----------------------------------
 31 30      23,22                 0
```

where:

S = the sign bit. A binary value of 0 denotes positive. A binary value of 1 denotes negative. Numbers in this form are termed sign and magnitude.

Exponent = A binary biased exponent. That is, the decimal value of the exponent is obtained by evaluating the unsigned binary value of bits<30..23>. Then 128 is subtracted from this value. This value is then used as a power of 2.

Fraction = A fractional value. An implicit 1 bit is to the left of bit 22. The decimal point is to the left of the implicit 1 bit.

The range of a single precision operand is approximately 10**-38 through 10**+38.

The format of the double precision (64 bit) floating point number is:

```
-----------------------------------------------------
|S| Exponent |      Fraction                        |
-----------------------------------------------------
63,62      52,51                                   0
```

where:

S = the sign bit. A binary value of 0 denotes positive. A binary value of 1 denotes negative. Numbers in this form are termed sign and magnitude.

Exponent = A binary biased exponent. That is the decimal value of the exponent is obtained by evaluating the unsigned binary value of bits <62..52>. Then 1024 is subtracted from this value. This value is then used as a power of 2

Fraction = A fractional value. An implicit 1 bit is to the left of bit 51. The decimal point is to the left of the implicit 1 bit.

The range of a double precision operand is approximately 10**-308 through 10**+308.

2.2.1 Floating Point Values

The value of a floating point operand is determined in the following manner. The sign of the operand is determined by the sign bit. Sign bit= 0 is positive; sign bit=1 is negative. The exponent is expressed as a biased exponent, which means that a positive number called the bias is added to the signed 2's complement of the true exponent. The following is an example of this bias. Assume that an exponent of 0 is desired. Then for single precision, the value 128 (the bias) is added to 0. Thus the binary exponent string of (1000 0000) represents an exponent of 0. For double precision, the bias is 1024.

The fraction has a binary point to the left of the most significant bit. Since the fraction is binary and is always normalized (the most significant fraction bit is a 1) for non-zero numbers, the fraction is expanded by one bit in the most significant bit within the processor during computation. The binary point is to the left of the implicit 1 bit. For computational purposes only, the fraction is interpreted as follows:

```
         -----------------------
        .|                     | Fraction in register
         -----------------------
         22                    0

     -------------------------
    .|1  bits <22..0>        | Numerical Processing
     -------------------------
    23                       0
```

Within these formats, there are certain reserved or special operands. In particular, these operands are used to represent zero or to initiate an exception.

2.2.2 Zero

A zero is a floating point number with a sign of 0 and an exponent of 0. The value of the fraction is unimportant. For example, if two floating point zeros with different fractions are compared for equality, the result is true.

For all computations that have a result of zero for the fraction, an all zero fraction is generated. A floating point zero with a fraction of all 0 is called a TRUE ZERO.

2.2.3 Reserved Operands

A floating point number (single or double) that has a sign bit of 1 and an exponent of all 0 is defined as a reserved operand. The value of the fraction bits is unimportant.

A reserved operand exception is detected if a reserved operand is encountered during a floating point numeric operation (e.g., ADD, SUBTRACT, COMPARE, MAX, and so on).

2.2.4 Rounding

All floating point algorithms use unbiased rounding, denoted as R*. For single precision, the processor determines the intermediate results of internal calculations by manipulating 26 bits. These bits include 23 fraction bits; the implicit 1 bit placed at "Unbiased Rounding, Fraction Bits" the left of the most significant fraction bit, and two guard bits placed at the right of the least significant fraction bit. In addition, a "sticky" bit is placed to the right of the guard bits. This bit is used in the intermediate calculations of floating point operands, and remembers whether or not any binary 1's were shifted out to the right during an alignment or partial product operation. For double precision, internal calculations use 55 bits plus the sticky bit.

2.3 Addresses

A logical address is 32 bits in length, resides in the address registers, and has the following format:

```
-------------------------------
| Logical Address             |
-------------------------------
31                            0
```

An address is treated, for numeric purposes, as an unsigned 32 bit integer; thus, an address is always positive. A logical address of all 0 is reserved. Note that for any particular implementation, the interpretation of an all 0 logical address may differ.

2.4 Unsupported Data Types

While unusual, it is important to know what data types are not supported by the instruction set. Byte and bit strings and commercial data types are not in the CONVEX architecture. CONVEX is a scientific machine, and, while Fortran '77 defines byte strings, it is up to the compiler to handle these data types through the appropriate basic instructions. Since the CONVEX logical address space is byte granular and all processors will be highly pipelined, an in-line string move, constructed of assembly language instructions, will be as efficient as a microcoded string move instruction. This is the essence of a RISC architecture.

2.5 Data Type Memory Alignment

The CONVEX logical address is byte granular. All the operands specified in this chapter can begin on any byte boundary, unless otherwise noted in an instruction definition.

However, there may be performance penalties for data types aligned on non-integral boundaries. In particular the application programmer and the compiler writer (when applicable), should follow the following alignment rules:

* Byte - No preference
* Halfword - least significant address bit = 0
* Word - least significant 2 address bits =0
* Longword - least significant 2 address bits = 0

CHAPTER 3

3 Register Set

There are three general register sets and additional status registers. The registers are partitioned according to the operand to be manipulated: addresses (and indices), scalars, and vectors. This partition permits the minimal machine state to be associated with the executing program--be it the operating system, compiler, scalar application program, or vector application program.

3.1 Address Registers/Program Counter/PSW

There are eight 32-bit address registers denoted as A0, A1,...,A7. A0, A5, A6, and A7 have predefined meanings (with the exception of A0, they can still be used as general purpose address registers). A5 is implicitly used by some complex instructions; A6 is the Argument Pointer; A7 is the Frame Pointer, and A0 is the Stack Pointer. In addition, A0 is interpreted in one of two ways. If A0 is specified as an addressing mode, then the value of 0 is used in place of the true value of A0. When A0 is used as a source or destination for an arithmetic operation, then the true value of A0 is used. All other address registers are available for general use. For complete information on stacks and the registers used to maintain them, see Chapter 5, "Logical Address Space and Memory Management".

Two additional 32 bit registers exist: the program counter (PC) and a processor status word (PSW). The PC is not part of the eight A register set. The separate definition and existence of a PC permits address generation not having to concern itself with the true state of the PC. Thus, in a highly pipelined processor with instruction overlap, no additional data paths or arithmetic logic units need exist to support PC relative addressing in a general manner.

The structure of the program counter is as follows:

```
---------------------------------
|SEG|  SEGMENT BYTE OFFSET    |R|
---------------------------------
3  2,2
1  9,8                       1 0
```

Program Counter

When the program counter increments to reference the next instruction, the bits incremented are a function of bit 31. If bit 31 is a 1, then bits<30..1> are incremented. If bit 31 is a 0, then bits <28..1> are incremented. Bit 0 of the program counter is not interpreted but is reserved for future use for hardware design.

3.1.1 Processor Status Word

The other 32 bit register is the user accessible processor status word. This register contains flags to indicate the results of numerical operations and flags to enable or disable exception processing. There are no privileged mode bits in the PSW. The structure of the PSW is as follows:

```
---------------------------------------------------------------------------
|C|AIV|ADZ|IVE|TR|FRL|SEQ|SC|SIV|SDZ|DZE|UN|OV|RO|FDZ|FE|FUE|    res    |
---------------------------------------------------------------------------
 3  3   2   2   2  2 2  2   2  2   2   2  1  1  1   1  1   1  1
 1  0   9   8   7  6 5  4   3  2   1   0  9  8  7   6  5   4  3        0
```

Processor Status Word

where the bits have the following meanings:

Bit 31 - C/Carry. The carry bit is set to the carry out for specified operations on the A (address) registers. For compare operations using the A registers, the sense of the compare is stored in the carry bit (i.e., was the compare true or false).

Bit 30 - AIV/Overflow. The address register integer overflow bit is set to indicate that a fixed point integer overflow occurred for specified operations on the A registers. An overflow trap occurs if bit 28 is a 1. If AIV is a 0, then no overflow has occurred since this bit was cleared. If AIV is a 1, then at least one overflow has occurred since this bit was last cleared.

Bit 29 - ADZ/Divide by Zero. A divide by zero was detected during an operation using the A registers. If ADZ is a 0, then no integer division with a zero divisor has occurred since this bit was cleared. If ADZ is a 1, then at least one integer division with a zero divisor has occurred since this bit was last cleared.

Bit 28 - IVE - Integer Overflow Enable. If bit 28 is a one, and either bit 22 or bit 30 is a one, an integer trap occurs. If bit 28 is a zero, no trap occurs.

Bit 27 - TR/Trace. A one causes an instruction trace trap before the execution of the instruction referenced by the program counter. A nonprivileged user can set or reset this bit. For trace mode to function properly, the user must set the SEQ bit (bit 24) to 1.

Bits<26..25> - FRL/Frame Length. Indicates whether the frame created by the last CALL instruction, TRAP, or fault was a short frame (FRL=11) or a long frame (FRL=10), an extended frame (FRL=01), or a context block (FRL=00). The FRL bits are used by the return (rtn) instruction to unwind the stack after a subroutine call or exception. When FRL=00, the current ring must be 0, and the rtnc instruction must be used.

Bit 24 - SEQ/Sequential. This bit determines the degree of pipelining within the processor. A 0 indicates that maximum pipelining and overlap is provided. A 1 indicates that all instructions are executed sequentially. That is, the execution of an instruction is initiated only after the previous instruction has completed its execution.

Bit 23 - SC/Scalar Carry. The carry out for operations involving the S (scalar) registers. This bit is also used to hold the result for scalar comparisons.

Bit 22 - SIV/Integer Overflow. If SIV is a 0, then no overflow has occurred since this bit was cleared. If SIV is a 1, then at least one overflow has occurred since this bit was last cleared.

Bit 21 -- SDZ/Divide by Zero. Indicates the occurrence of an integer divide by zero for a divide on a scalar or vector register. If SDZ is a 0, then no integer division with a zero divisor has occurred since this bit was cleared. If SDZ is a 1, then at least one integer division with a zero divisor has occurred since this bit was last cleared.

Bit 20 - DZE - Divide by Zero Enable. If bit 20 is a one, and either bit 21 or bit 29 is a one, a trap occurs. If bit 20 is a 0, no trap occurs.

Bit 19 - UN/Underflow. Indicates that a floating point underflow occurred during an operation on a scalar or vector register. If UN is a 0, then no underflow has occurred since this bit was last cleared. If UN is a 1, then at least one floating point underflow has occurred since UN was reset to 0.

Bit 18 - OV/Overflow. Indicates that a floating point overflow occurred during an operation on a scalar or vector register. If OV is a 0, then no operation on a scalar or vector register produced an overflow since this bit was last cleared. If OV is a 1, then at least one floating point overflow has occurred since OV was reset to 0.

Bit 17 - RO/Reserved Operand. Indicates that a floating point operation on a reserved operand during an operation on a scalar or vector register has been detected. A reserved operand is a floating point operand with a sign bit of 1 and an exponent of all 0's. A 1 indicates that a reserved operand has been detected. A 0 indicates that a reserved operand has not been detected. If RO is a 0, then no floating point operation on a scalar or vector register produced an overflow since this bit was last cleared to 0. If UN is a 1, then at least one floating point overflow has occurred since RO was reset to 0.

Bit 16 - FDZ/Floating Divide by Zero. Indicates that a floating point divisor of 0 was used during a floating point operation on a scalar or vector register. A 0 indicates that no zero divisor was detected. A 1 indicates that a zero divisor was detected. If FDZ is a 0, then no floating point division with a zero divisor has occurred since this bit was cleared. If FDZ is a 1, then at least one floating point division with a zero divisor has occurred since this bit was last cleared.

Bit 15 - FE/Floating Point Trap Enable. If bit 15 is a 1, then if any of bits PSW<18..16> (Overflow, Reserved Operand, Divide By Zero) are a 1, a floating point trap occurs. If bit 15 is a 0, no floating point trap occurs.

Bit 14 - FUE/ Floating Point Underflow Enable. If bit 14 is a 0, a floating point underflow trap does not occur. If bit 14 is a 1, a floating point trap underflow does occur. In both cases, if a floating point underflow is detected, true zero is the result.

Bits <13..0> - RES/Reserved. These bits are reserved for future system use.

For bits 30, 29, 22, 21, 19, 18, 17 and 16 (AIV, ADZ, SIV, SDZ, OV, UN, RO, FDZ) a bit set to 1 stays a 1 unless otherwise cleared by an explicit store of the PSW. This permits the PSW to remember the occurrence of an exception which is masked out, but which is to be subsequently explicitly tested.

Thus, the logic equation for these exception bits is (using UN as an example):

UN = UN .OR. Indication of underflow.

NOTE:

For floating point exceptions, two trap enables are provided: one for underflow and one for every other floating point exception. The reasoning for this relates directly to the possibility of continuing computation after the trap. Underflow forces a true zero result, which for most circumstances, is sufficient. All other floating point exceptions force a reserved operand result. Reserved operands are generally markers for future trap handlers. Two trap enables permit the application programmer to choose the appropriate reaction to a floating point exception.

Please see the appropriate sections on arithmetic exceptions, subroutine calls, and system calls for a description of the modification of the PSW.

3.2 ION and VV Flags

There are two privileged binary flags, ION and VV, which control certain operations. The ION flag is used by the operating system to enable and disable external interrupts. There are instructions to test the state of the ION flag (bri.f, bri.t, jmp.f, and jmp.t). There are also privileged instructions to disable ION or set it to 0 (dsi), and to enable interrupts or set ION to 1 (eni).

The VV or vector valid flag is also used by the operating system to control the saving and restoring of the vector accumulators in a demand mode. Typically, when a program first uses vector instructions, a vector valid trap occurs, and the operating system will then allocate vector register (VM, VL, VS, and Vector Accumulators) to the user program. There are two instructions which operate on the VV flag. The privileged instruction mov Sk,VV loads the VV flag from Sk. The other instruction, tstvv, loads the value of VV into the SC bit in the PSW. Please see Section 6.3.3 for details on the vector valid trap.

3.3 Sequential Execution

When bit 24, SEQ, of the PSW is set, all pipelining is disabled. Owing to the pipelined characteristics of the processor, execution for debugging purposes (both for hardware and software) must sometimes be serialized. The numerical results produced are the same regardless of the setting of the SEQ bit. Only performance and the serial nature of the execution are affected. The user may freely set or reset this bit.

3.4 Data Accumulators

There are two general sets of data registers: scalar and vector. Within the vector register set, there are four types: vector accumulators(V), vector merge(VM), vector stride(VS), and vector length(VL).

Within the S and V registers, the architecturally supported data types occupy the following bit positions:

* BYTE - bits<7..0>
* HALFWORD - bits<15..0>
* WORD - bits<31..0>
* LOGICAL - bits<63..0>
* LONGWORD - bits<63..0>
* SINGLE PRECISION - bits<31..0>
* DOUBLE PRECISION - bits<63..0>

3.4.1 Scalar Registers

There are eight 64-bit scalar accumulators or S registers. The S registers contain either fixed point integer, logical, or floating point operands. When an operand less than 64 bits is loaded into a scalar register, the unused bits are left unchanged. The scalar registers are referenced as: S0, S1, ..., S7.

3.4.2 Vector Registers

3.4.2.1 Vector Accumulators

There are eight vector accumulators (V), each of which can contain up to 128 64-bit operands. These operands can be: integer, logical, or floating point. When an operand less than 64 bits is loaded into a vector accumulator, the unused bits are left unchanged.

Since a vector accumulator can contain up to 128 elements of the same data type and precision, a means is provided to specify the exact number of operands stored. The VL or Vector Length register provides this function.

The vector accumulators are referenced as: V0, V1, ..., V7. Individual elements within a vector accumulator are referenced by appending the element number to the vector accumulator designation. Thus, the 22nd element of V1 is referenced as V1(21), and the first element of V1 is V1(0) (origin 0 indexing).

3.4.2.2 Vector Merge

To support efficient element-by-element array comparisons and array manipulations such as compress, expand, and merge, a Vector Merge or VM register is provided. VM is 128 bits in length, with one bit position for each possible element in an array accumulator.

A binary 1 is used to contain the results of compare operations when those comparisons are true; likewise, a binary 0 is used to contain the results of compare operations when those comparisons are false.

Typical uses of VM (as supported by the instruction set) are:

1 Vector Clipping
2 Population Count (the number of successful compares)
3 The manipulation of sparse vectors.
4 Array compression, expansion, and merging.
5 The number and location of zero or threshold crossings.
6 Support arithmetic operations that are performed under mask.

3.4.2.3 Vector Stride

A 32-bit register, VS, is provided to specify the distance, in bytes, between adjacent array elements. If VS is positive, adjacent array elements are loaded and stored from memory by adding a multiple of VS to the initial address of the array base. If VS is negative, adjacent array elements are loaded and stored from memory by subtracting a multiple of VS from the initial address of the array base. In this latter case, logically adjacent elements are in decreasing locations in logical memory.

If VS is smaller than the precision of the operands fetched, undefined |

actions can occur. If VS is exactly 0, the referenced operand is extended to a vector whose length is equal to VL (scalar extension).

3.4.2.4 Vector Length

An 8-bit register, VL, is provided to specify the number of elements contained in a vector accumulator. Since VL is eight bits in length, up to 255 can be specified, but only 128 elements can reside in a vector accumulator. The VL register must have:

1 A specification of 128 elements or less.
2 An efficient means for compiler support of arrays greater than 128 where the length of the array is an integer multiple of 128 (often called strip mining or sectioning).
3 An efficient means for compiler support of arrays greater than 128 where the length of the array is not an integer multiple of 128.
4 The provision of one mechanism for handling loop control for the cases where the iteration count of a program loop is a variable.

Note: when VL is zero, no vector operation is performed. With these objectives in mind, the interpretation of VL is exactly equal to its decimal value.

Examples:

1 VL = 1000 0000. VL represents the value 128.
2 VL = 0000 0111. VL represents the value 7.
3 VL = 0000 0000. VL represents the value 0 (no-operation)

An attempt to load VL with a value greater than 128 will result in the decimal value of VL being exactly 128.

The following code sequences would suffice for handling an array of any length:

1 Load VS.
2 Load VL.
3 Execute the vector instruction sequence.
4 Subtract 128 from the value used to load VL in step 2.
5 If the resulting value is less than or equal to 0, then all of the array elements have been manipulated, and thus exit. Otherwise:
6 Adjust the address pointers to the array section (the next 128 elements) or unroll the DO loop in-line. GOTO step 2. (Note: VS need not be reloaded).

Please note that even though the VL register is 8 bits, a vector can be of any arbitrary length up to the user logical address space of 2 Gigabytes. As mentioned above, the compiler performs operations on vectors in groups of 128 elements, through a mechanism called sectioning or strip mining.

CHAPTER 4

4 Protection System

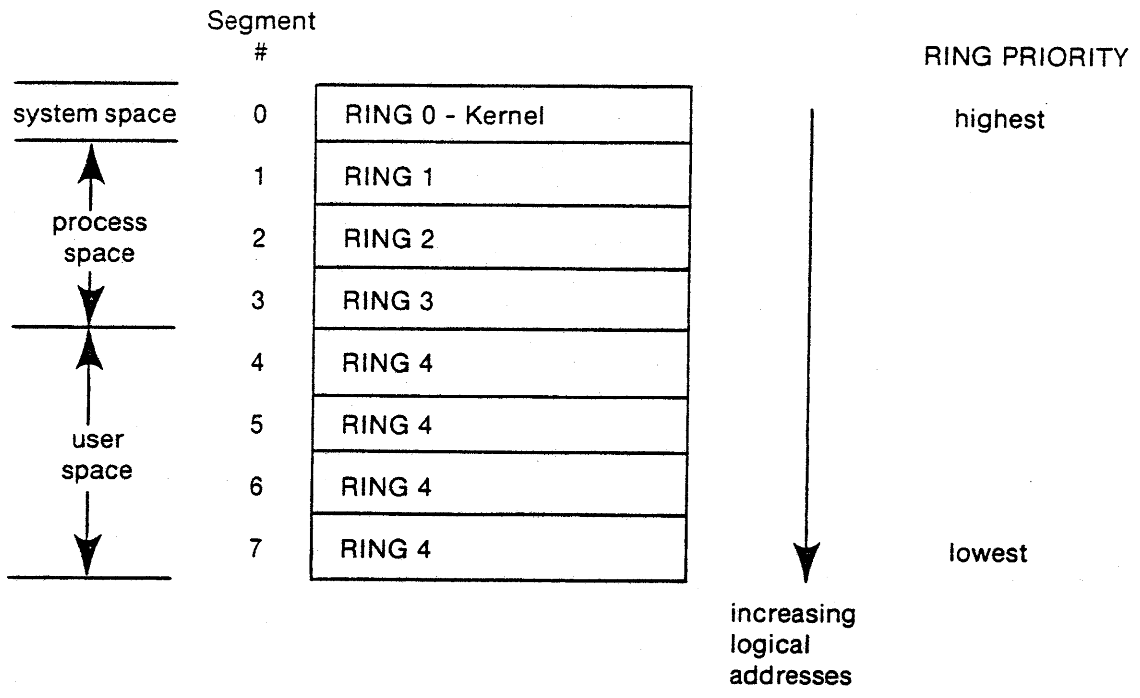
The protection system protects the user, his programs, and other user's programs, while also supporting contemporary notions of shared resources and operating system structures. These features do not get in the way of system performance or cause undue hardware complexity.

The protection system has been designed to:

1 Support the embedding of the operating system in the user logical address space (for reasons of performance and software reliability).
2 Contain certain violations to a user's process. A user can and may modify his/her own process, but not any others.
3 Detect runaway programs in a graceful manner.
4 Permit efficient implementations of virtual machine mechanisms.

In order to achieve these objectives, the protection system is structured in the following manner. The logical address space is partitioned into 5 hierarchical areas called rings. This partitioning is defined by the segment field (bits <31..29>) of the logical address. Segment 0 is always assigned to ring 0, which contains the operating system kernel. A set of instructions, referred to as privileged instructions, can only be executed in ring 0. Segment 1 is always assigned to ring 1. Segment 2 is always assigned to ring 2. Segment 3 is always assigned to ring 3. Segments 4,5,6, and 7 are always assigned to ring 4. The structure of the logical address space is graphically shown in Figure 4-1.

Figure 4-1: Logical Address Space Structure



This particular structure was chosen because the CONVEX family is a 64-bit supercomputer which supports a 32-bit address space. As a result, the computer can easily support large user programs in a virtual address space. Furthermore, the Operating System (OS) can be embedded in the user address space. From these vantage points, the user and the system benefit enormously. However, to realize these benefits fully, the OS must be protected from the user.

Another advantage to this structure is the segmentation support made available for users. Segmentation may be used to support the mechanisms of different address partitions for user code, static data, dynamic data (stack or unshared), and system library code. To these ends, four segments are explicitly provided for the user.

4.1 Logical Address Space Structure

Two other structures are needed to complete the protection system: the access field contained within a pagetable entry (PTE), and the access brackets pertaining to the enforcement of the ring structure.

The access bracket structure directly implements a mechanism called ring maximization. Ring maximization means that given a lower access priority, the references of the instruction are always at this lower priority. The reference starts at the priority implied by the ring field of the program counter (bits <31..29>). This initial ring is then compared to the ring of the referenced operand (if one exists). As a function of the numerical relation between these two ring numbers, a statement concerning the validity of the reference can be made. In this ring mechanism, higher ring numbers have lower priority than rings with lower numbers.

A memory reference that satisfies the ring maximization function is valid. All valid references must then satisfy the access requirements imposed by the access field of the PTE that references the target operand.

Table 4-1 defines the validity of logical address references. If an invalid reference is detected, a system exception occurs, and an error code is loaded into A5 while the fault is being serviced (see Chapter 6, "Exceptions"). The effective source is the ring of the program counter, and the effective target is the ring of the address of the referenced operand. If indirect addressing is specified, the effective source remains the ring of the program counter, and the new effective target is the ring number contained in the indirect pointer.

Table 4-1: Ring Maximization Source/Target

Program Counter Ring	Target				
	Most Privilege				Least Privilege
Ring 0	Valid-R0	Valid-R1	Valid-R2	Valid-R3	Valid-R4
Ring 1	Trap	Valid-R1	Valid-R2	Valid-R3	Valid-R4
Ring 2	Trap	Trap	Valid-R2	Valid-R3	Valid-R4
Ring 3	Trap	Trap	Trap	Valid-R3	Valid-R4
Ring 4	Trap	Trap	Trap	Trap	Valid-R4

Ring Maximization Source/Target

To determine whether or not a read, write, or execute access should be allowed for valid references, the system follows these procedures based on an access field within a PTE:

1 Read Access - The ring maximization function is used to determine whether or not the reference is valid. If it is, bit 3 of the valid PTE is examined. If bit 3 is a 1, the read is permitted. If a 0, the read is not permitted and a system exception occurs.

2 Write Access - The ring maximization function is used to determine whether or not the reference is valid. If it is, bit 2 of the valid PTE is examined. If bit 2 is a 1, the write access is permitted. If a 0, the write is not permitted, and a system exception occurs.

3 Execute Access - If transfer of control within the same ring is performed, bit 1 of the valid PTE is examined. If bit 1 is a 1, instructions can be fetched and executed from this page. If bit 1 is 0, instruction execution is not permitted, and a system exception occurs.

The protection mechanisms provided for inter-ring transfer of control (not within the same ring) are presented in a subsequent section of this chapter.

4.2 Protection Notes

The following notes will help users avoid some pitfalls when using this type of protection system:

1 PC relative addresses are granted no special privileges. The appropriate read, write, and execute privileges, as previously specified, apply.

2 Access checking is performed if and only if the pagetable entry associated with a valid address is valid. The state of the resident bit for checking access privileges is ignored. Thus an access violation can be detected for non-resident pages.

3 If an access privilege is changed for a process after that process has already established a context in the ATU, the ATU must be purged upon completion of the alteration. ATU entries are not altered automatically when a PTE is modified.

4 If an instruction specifies an immediate operand (e.g., Add immediate), the read access privilege of the page containing the immediate operand is not interpreted; it is treated as an execute access.

5 A ring check is not performed for instructions which produce effective addresses, but which do not immediately use them. For example, if a load effective address instruction executed in ring 3 develops a ring 1 address, no ring violation occurs. If

that ring 1 address is subsequently used by a ring 3 program to make an operand reference, a ring violation occurs.

6 The intermediate addresses of all instructions which can make multiple memory references (e.g., Vector Load) are always ring maximized with the current ring to determine the validity of the reference (i.e., the address of each array element).

7 When indirection is specified, the page containing the indirect pointer must permit read access. This read access is independent of the instruction type (i.e., load, store, jump)

8 I/O space operands must be addressed as single bytes. If a valid I/O reference is made using a non-byte operand, a protection violation occurs.

The protection structure uses a construction called effective source, which has these properties if indirection is specified:

1 The program counter is still the effective source.
2 The second target is the ring number contained within the indirect pointer. The first target is the ring number contained within the effective address that references this indirect pointer.

4.3 SDR Validity Bit Protection

If bit 31 of an SDR location is 0, a PTE violation occurs. The PTE violation is vectored through byte address 0C (hex) in page 0 of ring 0 (a system exception). To protect against an invalid SDR0 which would cause an endless PTE violation, the following special test is performed: if SDR0 is invalid, a machine exception occurs (see Chapter 6). The response to a machine exception is implementation-dependent.

4.4 Inter-ring Procedure Call/Return

Ring crossing can only occur as a result of an explicit attempt by a program control instruction to cross rings, or by a system exception. The conditions by which these explicit instructions can cross rings are as follows:

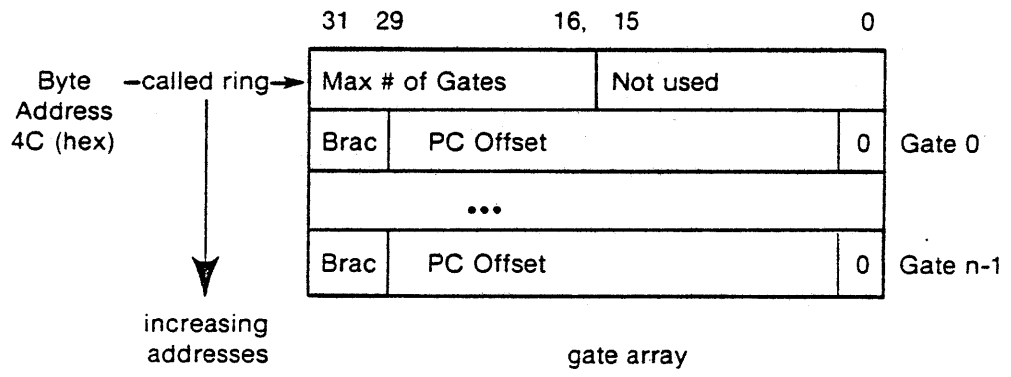
1 The explicit program control instruction is either a system call (sysc), return (rtn with FRL=01, extended return block), or the privileged instruction: return from context block (rtnc). All other program control instructions stay within the current ring of execution (i.e., the ring of the program counter). Thus the appropriate higher order bits of the target effective address are, in essence, ignored.

2 The direction of a subroutine system call is inward, toward ring 0. Outward calls are trapped as a ring violation. The direction of all subroutine system returns is outward, away from ring 0. Inward returns are trapped as a system exception.

3 The immediate field of the system call instruction is interpreted as an index into a table within the called ring. This index is referred to as a gate number. The table contained within the called ring is referred to as a gate array. The base of the gate array is pointed to by byte address 4C (hex) of page 0 of the called ring.

The structure of the gate array is illustrated in Figure 4-2.

Figure 4-2: Gate Array Structure



The format of the sysc instruction is:

```
-----------------    ------------------------------
|Opcode|L|000|Ak|    | #r  |      0      |    #g    |
-----------------    ------------------------------
15     7 6 5 4 3 0   31 29 28        16 15         0
```

Inward ring crossing functions in the following manner. The gate index field (g field) of the sysc instruction is used to indicate the desired entry point. This gate index field is compared with bits <31..16> of the first word of the gate array pointed to by byte address 4C (hex) of page 0 of the target ring. If the gate index field is greater than or equal to bits <31..16>, then a ring violation occurs, and the ring crossing does not occur (the gate is not defined). If the gate index field is less than bits <31..16>, the ring number of the segment containing the sysc instruction (current ring) is compared with bit <31..29> of the referenced gate index (r field). If the current ring is less than or equal to the bracket (Brac) field, then bits <28..1> of the gate are loaded into the program counter. Bits <31..29> are loaded with the target ring. If the current ring is greater than the bracket field, the PC is not loaded, the ring crossing does not take place, and a system exception occurs.

For example, assume that the operating system kernel has n gates. All the gates other than gate M are reserved for calls from rings 3,2,and 1. However, gate M (owing to the nature of this kernel call) can be directly called by ring 4. All gates in the kernel other than M have the value 3 in their gate bracket field. Gate M has the value 4 in its gate bracket field. If a ring 4 caller attempts to call a kernel gate other than M, the call fails (4 is greater than 3). If a ring 4 caller attempts to call kernel gate M, the call succeeds (4 is less than or equal to 4).

This mechanism permits individual segments to have entry points with unique gate brackets. Thus a particular operating system call can be restricted to a particular ring of origin.

All these actions are performed by the processor. There is no software overhead or involvement by the operating system kernel, unless an explicit kernel call is made.

4.4.1 <u>Stack Switching/Argument Reference</u>

There is one stack per ring, which means that the stack allocated to ring 4 is logically different from the stack for ring 3, for ring 2, and so on.

After a successful gate entry, as specified in the previous section, a new stack frame is created in the target ring, and an extended subroutine return block is pushed onto the target stack (the called routine's stack). The stack pointer of this stack is initially loaded from byte address 48 (hex) of page zero of the called ring. After the extended return block is pushed, the stack pointer (A0) is copied into the frame pointer (A7). The PC is loaded from the referenced gate.

The stack pointer value saved in the extended return block represents the value of the caller's stack pointer at the time of the call. It is necessary to save this value to permit a proper return from a multiplexed stack structure. Thus the link back to the outer ring's stack is contained within the extended return block pushed on the inner ring's stack. Additional details for an inward system call are covered in the description of the sysc instruction.

Arguments for the system call are maintained in a programmer-defined area. They may be in an argument packet or on the stack--whatever software-enforced conventions permit.

The converse of a system call is a system return, which is implemented with a return (rtn) instruction. Unlike a system call, no gate processing is necessary. An inner ring can unconditionally access an outer ring, so protection is unnecessary. A system return is like a normal return with the following four differences. First, the ring field of the Program Counter can change. Second, all returns must be the same ring or outward (away from ring 0). Third, the return block on the stack must be an extended or context type. Fourth, after the return block is popped from the stack, the updated stack pointer of the inner ring is restored to byte address 48 (hex) of page zero of the ring containing the rtn instruction. This guarantees that, with subsequent system calls to the same ring, the stack will be initialized to the proper values.

4.4.2 Trojan Horse Pointers

Trojan horse pointers can occur on system calls when a passed pointer references the operating system's data space. Usually, the software agent invoked as part of the inward ring call uses a passed pointer as part of system call processing. The agent expects these pointers to reference the logical address space of the caller (i.e., the ring of the user executing the sysc instruction). If a passed pointer references an agent's data space, unexpected (and generally undetected) disasters occur. To prevent such happenings, the following facilities are provided:

1 An instruction which checks to see that the ring maximization function is satisfied for passed pointers (compare immediate) is provided.
2 A load physical instruction is provided to obtain the access bits of appropriate pagetable entries.
3 Instructions which access data backwards (decreasing logical memory) always perform the ring maximization function to ensure that a dynamic Trojan horse pointer is not created.

All of these actions can occur outside the operating system kernel. One of the objectives of the protection and memory management structure is to reduce the size of the operating system's kernel. This permits a more reliable and secure kernel to be constructed. Additionally, virtual machine structures are easier to construct.

Generally, there is no algorithm which guarantees that Trojan Horse pointers will not occur. Experience has shown that, for a robust system call interface, arguments should be copied into the called ring's space, and then Trojan horse pointer checking initiated.

If the arguments are values, the level of required checking is somewhat mitigated. This provision prevents one argument pointer from modifying another argument after Trojan horse checking; this is essentially the self-modifying argument attack.

CHAPTER 5

5 Logical Address Space and Memory Management

The logical address space of CONVEX is virtual. This means that although an address may be a valid logical address, the referenced data may or may not be in main or physical memory. To manage this structure, various entities are defined and supported. Among these structures are:

1 Segment - A logically contiguous group of bytes--in particular, 512 MB (549,755,813,888 bytes).

2 Page--A contiguous group of bytes, in particular, 4096 bytes. A page is both logically and physically contiguous.

3 Segment Descriptor Register (SDR)--A 32-bit register that contains information necessary to translate a logical segment offset to a physical address in main memory.

4 Page Frame--A page that is stored in main memory.

5 Page Table--A Page that contains entries called Page Table Entries. A pagetable begins on an integral page boundary and is contained in one page frame or less.

6 Page Table Entry (PTE)--A 4 byte entry (32-bits) that conveys information necessary to determine if a page is resident in main memory or not. Other status bits within a PTE determine the validity of the memory reference from a protection viewpoint. A PTE is aligned on an integral word boundary.

7 Referenced Bit--A bit associated with a page frame. A referenced bit indicates that a valid read or write has occurred.

8 Modified Bit--A bit associated with a page frame. A modified bit indicates that a valid write has occurred.

9 Address Translation Unit (ATU)--A programmer invisible address cache that maintains the most recently used logical to physical address translations.

Sufficient referenced and modified bits exist for the total amount of physical memory for a particular implementation. These bits are stored in internal machine state and mapped into the I/O address space.

CONVEX's logical .address space is 4 Gigabytes (approximately 4.3 billion bytes). This space is partitioned into 8 x 512 Megabyte segments. The format of the logical address space is:

```
----------------------------------------
|SEG   | Segment Byte Offset           |
----------------------------------------
31   29,28                             0
```

There are no other hardware or architecturally supported pointer formats. Of these eight segments, four are allocated to the user (essentially hardware enforced), and four to the operating system. Thus the maximum user program (instructions and data) is 2 Gigabytes. This permits a user to share a common subroutine library in one segment, code in another segment, static (e.g., FORTRAN common) in the third segment, and dynamic (e.g., stack) in the fourth segment. This partitioning is a suggestion, not an enforced algorithm.

Logical addresses are generated through the use of the program counter, absolute references, or one of the eight address registers.

Data referenced by a byte logical address can begin on any arbitrary byte boundary. That is, a 64-bit operand can begin on any one of eight byte boundaries. The byte address generated by an instruction references the first byte (byte 0) of an operand. However, where storage allocation is controlled by the system, the preferred boundary is as follows:

- o byte - not applicable.
- o halfword - least significant address bit is a 0.
- o word - least significant 2 address bits are 00.
- o longword - least significant 2 address bits are 000.

The software linker must support, for performance reasons, alignment of various structures on programmer specified boundaries (e.g., the page boundary).

5.1 Indirection

All instructions which reference memory can optionally specify a mechanism called indirection. Indirection causes the logical address of the referenced operand to be contained in a memory location. The site of this memory location is obtained from the instruction specifying indirection.

Only one level of indirection can be explicitly specified by an instruction. The format of the indirect word is a byte pointer.

5.2 Stacks

A stack is an array of bytes organized in a "pushdown" manner. Stacks are used to contain operand temporaries and local variables, as well as to state information relevant to the environment of the currently executing program.

There are three architecturally-defined registers that are used to maintain a stack: the stack pointer (SP, A0), the argument pointer (AP, A6), and the frame pointer (FP, A7). As a function of the type of operation performed on the stack, one, two, or all three of these registers are affected. Generally, subroutine entry and exit use all three registers. The following subsections describe some of the types of operations performed on a stack. Please consult the appropriate instruction set chapter for specific details.

5.2.1 Push and Pop Operands

There are two primitive operations on a stack. One operation is a push and the other operation is a pop. A push stores an operand on the stack; a pop removes an operand from the stack. Address register A0 points to the top element of the stack (the last location used). A push decrements A0 by a multiple of 4 or 8; a pop increments A0 by a multiple of 4 or 8.

Pushing a word requires that A0 be decremented by 4; then the word is stored in the location referenced by the new value of A0. Popping a word from the stack requires that the top element is fetched from memory, and A0 is then incremented by 4.

The following example depicts these actions. Initially the top of stack is at byte 68 (decimal). Thus, a load word from the top of the stack fetches bytes 68, 69, 70, and 71. Pushing a word onto the stack requires that the stack pointer first be decremented by 4 (64 = 68-4); then the word to be pushed is stored into bytes 64, 65, 66, 67, or simply the word that begins at byte 64. This is illustrated in Figure 5-1.

Figure 5-1: Push and Pop Operands

```
                ------------------------
                |                      |
                |                      |
                |                      |
                |                      |
                ------------------------
A0 after push ->| 64  | 65  | 66  | 67  |<-New top of stack, A0 is 64
                ------------------------
A0 before push->| 68  | 69  | 70  | 71  |<-Prev. top of stack,A0 is 68
                ------------------------

                       Stack
```

The stack is managed as an aggregate of 32-bit words, which means that all instruction set primitives that manipulate the stack do so by incrementing or decrementing by stack and frame pointer by units of four. Even though the stack is referenced by a byte address, this convention is always obeyed. Overt modification of the stack pointer (by instructions which manipulate A0 and A7) by quantities other than multiples of four is not recommended. Even though the processor will continue to function, performance will be lost. The stack should be initialized to begin on an integral 4-byte address boundary.

There is no explicit stack overflow or stack underflow detection performed by the hardware. Stack overflow and underflow may be detected by surrounding the allocated stack by pages (described in a later chapter) of no access. Software reserved bits in the protection fields of the no access page table entries may be used to differentiate this type of access violation from other possible causes. (Page Table Entries are described in Chapter 4). Consequently, the protection trap handler can determine the reason for its invocation.

5.2.2 Creating and Deleting a Stack Frame

Stacks are generally used as dynamic storage, storage that is allocated and deallocated during the execution of a user program. To assist in the proper management of the stack, a frame pointer is defined. The frame pointer, A7, provides for dynamic linkage between frames contained on a stack. Typically, a frame consists of an area that contains saved copies of registers from the previous execution context, an area that contains storage for temporary variables local to this context, and values necessary to manage the present frame as well as a link back to the previous frame. See Figure 5-2 below for details on the stack structure.

5.2.3 Ring 0 Stack

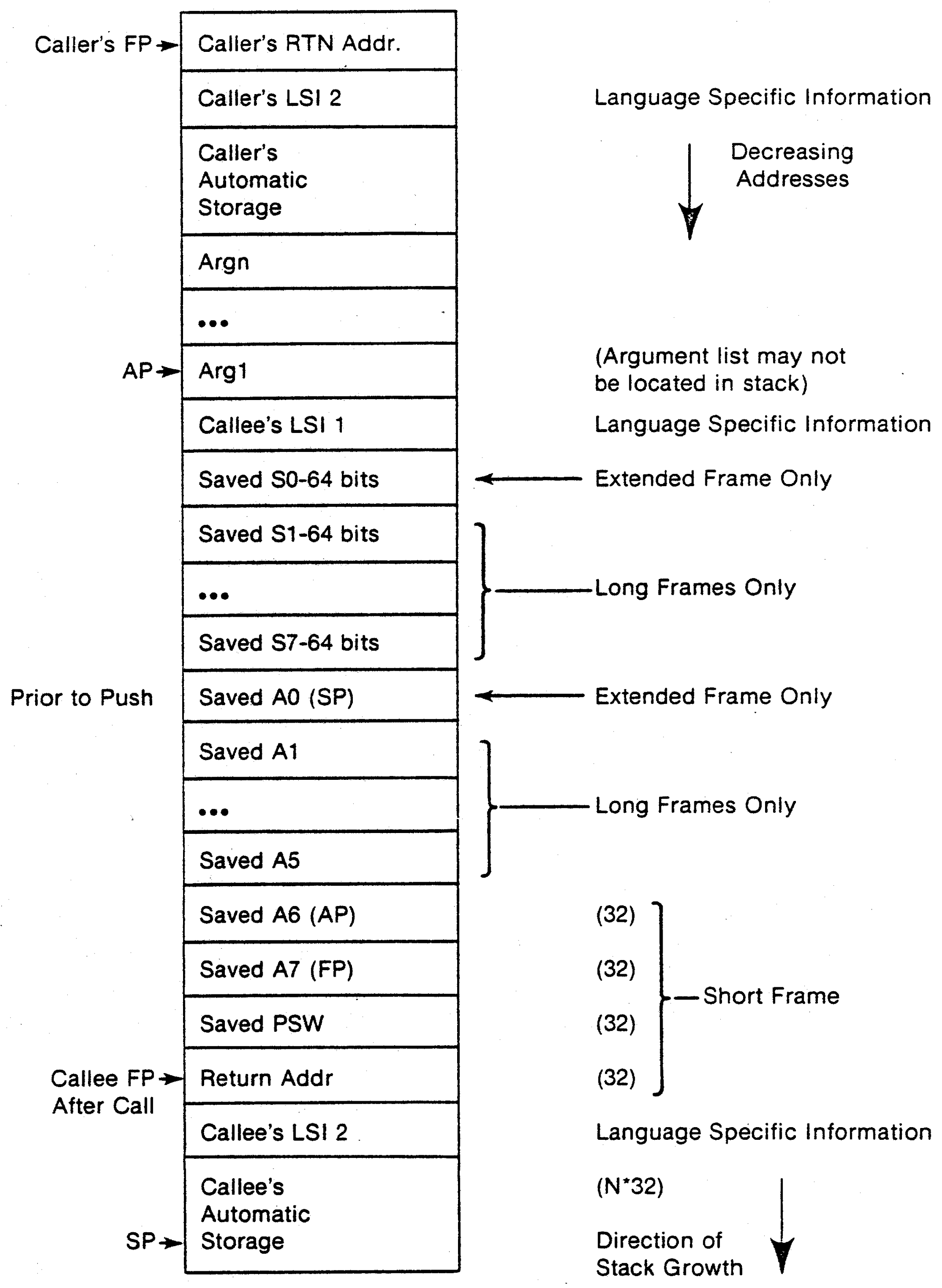
The ring 0 stack (the stack associated with the highest priority ring, ring 0) must always be aligned on a 32-bit word boundary. If it is not, a machine exception occurs (see Chapter 6).

5.3 Reserved Logical Memory

Reserved logical memory locations are used to obtain addresses or status when exceptions occur. Generally when one of these conditions occurs, an implicit subroutine call occurs. The processor provides the subroutine call opcode, and the reserved area in memory provides the address. Because a stack has already been defined, arguments may be passed and a handler routine executed.

The reserved area in logical memory is the first page in the segment referenced by the ring field of the program counter. This page is referred to as Page 0. Since there are five rings, there are five page 0's. For ring 4, page 0 is always in segment 4. The only page 0 that must be memory resident

Figure 5-2: Stack Structure



is page 0 of ring 0.

Page 0 is used in one of two ways, depending on the classification of exception (trap or fault) which has occurred. (Exceptions are detailed in Chapter 6.) The two types of exceptions which access Page 0 are process exceptions and system exceptions. In addition, interrupts also access Page 0.

Process exceptions are characterized by the fact that they are handled by the user program in the current ring of execution. Examples include arithmetic traps and instruction trace. This approach supports the PL/1 "ON" condition and anticipated revisions to FORTRAN. It also permits each user to have his or her own debugger. The operating system is not involved in handling process exceptions.

A system exception involves the operating system. Examples include address translation faults and ring-crossing traps. When such an exception arises, a ring crossing to ring 0 occurs, and the ring 0 process stack is used. Rings are discussed in more detail in Chapter 4, "Protection System."

Interrupts also cause a ring crossing to ring 0. An interrupt is an asynchronous event which the operating system must handle. A unique stack in ring 0 is established for interrupts; this stack is different from the ring 0 process stack.

5.3.1 System Page 0

Table 5-1 shows the logical memory organization of Page 0 of Ring 0. See Chapter 6, "Exceptions," for definitions of the terms "exception," "trap," and "fault."

Table 5-1: Page 0 Logical Memory Organization

BYTE ADDRESS (hex)	31	16,15 0	
0	Reserved		System
4	Inter. Level	Reserved	
8	I/O Interrupt		
C	System Exception Handler		
10	Interval Timer Interrupt		
14	Reserved		
18	Reserved		
1C	Vector Valid Trap		
20	Interrupt Stack Pointer		
24	Context Stack Pointer		
28	Reserved		
2C	Previous Stack Pointer		
30-3C	Reserved		
40	Instruction Trace Trap		Process
44	Arithmetic Exception Trap		
48	Stack Pointer		
4C	Segment Entry Point		
50	Breakpoint Trap		

Each of the above entries has the following meanings.

0 Reserved. Should not be used by software. May be used in the future.

1 Interrupt Level. A 16-bit memory-based counter that indicates the number of nested interrupts currently being processed. If Interrupt Level is 0, then no interrupts are being processed. If Interrupt Level is not 0, then interrupts are being processed, and the ring 0 stack is the interrupt stack.

2 I/O Interrupt. A byte pointer to the handler for I/O interrupts.

3 System Exception Handler. A byte pointer to a system exception handler. The exceptions that transfer control to this system exception handler are: error exit trap; undefined opcode trap; ring violation; PTE violation, and non-resident page.

4 Interval Timer. A byte pointer to the interrupt handler that responds to an interval timer interrupt.

5 Reserved.

6 Reserved.

7 Vector Valid Trap. A byte pointer to a trap handler that responds to the vector valid trap. A vector valid trap occurs if an attempt to execute a vector instruction occurs and the vector valid bit is 0. A vector instruction is an instruction which manipulates the V, VL,VS, or VM registers.

8 Interrupt Stack Pointer. A byte pointer that specifies the stack to be used when an interrupt occurs.

9 Context Stack Pointer. A byte pointer that specifies the stack to be used when a system exception occurs.

10 Reserved.

11 Previous Stack Pointer. A save area used for interrupt processing. When an interrupt first occurs and the ring 0 stack is initialized to the value of the interrupt stack pointer, the process stack pointer is saved in byte address 2C (hex). This ensures that there is a proper linkage for stack switching in ring 0 for interrupt processing.

12 Reserved.

13 Reserved.

14 Reserved.

15 Reserved.

16 Instruction Trace. A byte pointer to the handlér that responds to an instruction trace trap.

17 Arithmetic Exception. A byte pointer to the handler that responds to an arithmetic exception. The PSW contains bits which indicate the type of arithmetic exception(s) that occurred.

18 Stack Pointer. A save area that maintains the stack pointer for cross ring call processing.

19 Segment Entry Point. A byte pointer to the base of the gate array defined in the called ring. Each ring has a unique entry point and associated gate array.

20 Breakpoint Trap. A byte pointer to the handler that is executed when the bkpt instruction is executed.

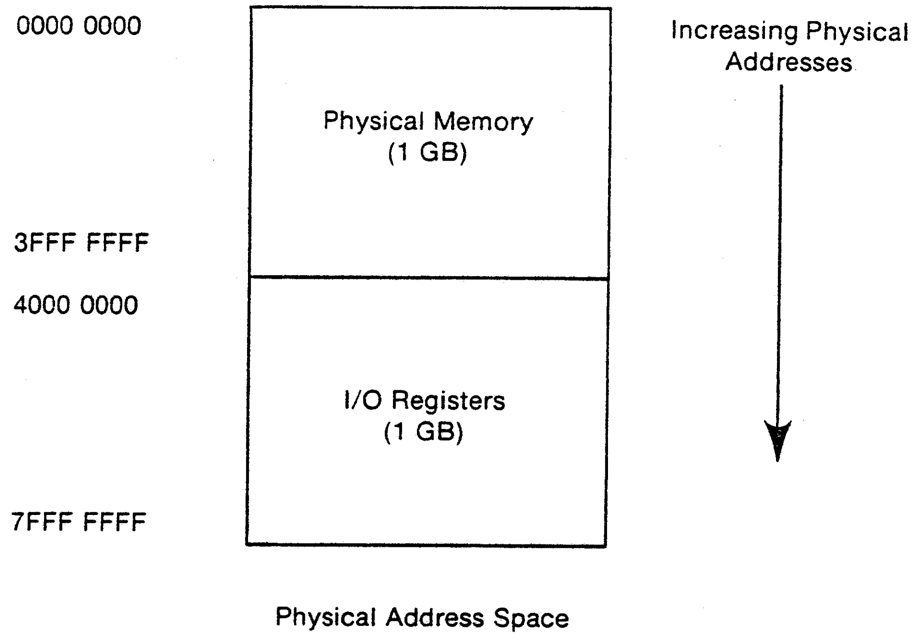
5.3.2 Process Page 0

If a trap is classified as belonging to a user process, page 0 of the current ring has the same format as specified above with one exception: the first 16 words are reserved.

5.4 Physical Address Space

The physical address space is 2 Gigabytes. One Gigabyte is allocated to main memory (i.e., up to 1 billion bytes of physical memory can be configured), and 1 Gigabyte is allocated to I/O registers. Physical addresses 0 through 3FFF FFFF (hex) reference main memory. Physical addresses 4000 0000 (hex) through 7FFF FFFF (hex) reference I/O registers. Figure 5-3 depicts this partition.

Figure 5-3: Physical Address Space



Within the I/O space, one set of registers is presently defined. These registers contain the referenced and modified bits maintained for each 4096 byte page of physical memory (a page frame). The architectural limit of one Gigabyte of physical memory means that 2**15 bits must be maintained. The following allocation has been made:

1 Addresses 4000 0000 to 4000 7FFF (hex) allocated to Referenced Bits.

2 Addresses 4000 8000 to 4000 FFFF (hex) allocated to Modified Bits.

All other I/O registers are reserved for future system use. It should be noted that I/O registers are not encached in a processor cache. This permits the I/O registers to change asynchronously without the processor having to concern itself with the presence of a cache.

All operands within the I/O register space must be one byte. If a valid reference is made to an operand other than a byte, a process exception (class C (hex), qualifier 7 Invalid I/O access) occurs. See Table 6-1. The referenced and modified bits are accessed as byte operands. Thus, the load and store byte instructions should be used. These instructions load and store eight referenced or modified bits at a time. The use of any other type of instructions will produce undefined actions.

5.5 Translating Logical To Physical Addresses

Presently, logical memory is substantially larger than physical memory. Consequently, a means must be provided to determine if there exists a physical page or page frame for a valid logical address. This determination is accomplished by a two level pagetable mechanism. A two level pagetable is used for accessing data in memory in the same way that a two level file index might be used for accessing data on disk.

5.5.1 Segment Descriptor Register

A segment descriptor register is a 32 bit word aligned to word boundary, which controls the validity of a segment (the basic partition of the logical memory space) and provides information relating to address translation. There are 8 SDR's, one for each segment. Each SDR is 32 bits long. An SDR has the following format:

```
--------------------------------------
|V|0|Page frame Base     | hw | sw   |
--------------------------------------
 3 3 2
 1 0 9                  9 8  7 6    0
```

Segment Descriptor Register

The meaning of each of these bits is as follows:

Bit<31> - Valid - If 0, this segment is not valid. A system exception is signaled and an error code is loaded into A5 after a context block is saved. (See Chapter 6).

Bit<30> - Hardware reserved. Must be zero.

Bits<29..9> - Page frame base. The page frame base is the higher order 21 bits of a 30-bit physical address. Bits <8..2> of this physical address come from bits <28..22> of the logical address to be translated. Bits <1..0> of the physical address are 0. This physical address references a page table entry in main memory. The page frame base is modulo 512 bytes. See note below.

Bits<8..7> - Hardware Reserved. These bits presently have no meaning. System software must not use these bits.

Bits<6..0> - Software Reserved.

The page frame base in the SDR permits, if desired by the operating system, for one page (4 KB) to be used to contain the first level page table for multiple contiguous segments. The first level page table can be structured so that it is contained in a 512 byte page rather than in a 4096 byte page. This 512 byte page can be used in one of two ways. It can be used to conserve physical memory by only allocating 512 bytes rather than 4096 to the first level lookup. Or, the 512 byte page can be configured to be one of the eight possible 512 byte partitions in a 4096 byte page. This last feature permits multiple first level lookups to be physically contained in one page frame.

5.6 Page Table Entry (PTE)

A pagetable entry (PTE) is a 32-bit word aligned on an integral 32-bit boundary (the least significant two bits of the byte address are 00). A PTE is one of 128 entries for the first index level or one of 1024 entries for the second index level. A PTE is used to determine the validity of a reference and the physical memory location of a valid reference. A valid reference meets two requirements: first, the PTE must be valid (bit 31=1), and second, the type of access being made (Read, Write, or Execute) must be allowed by the appropriate protection bit (bits <3..1> of the PTE). The format of a valid, resident PTE is shown for first level and second level page table entries, below:

```
-----------------------------------------------
|V|O| Page frame Address   |hw | sw |rd|wr|ex|nr|
-----------------------------------------------
 3 3 2                     1 1
 1 0 9                     2 1 87  4 3  2   1  0
```

First Level Page Table Entry

```
---------------------------------------------
|V| Page frame Address   |hw | sw |rd|wr|ex|nr|
---------------------------------------------
 3 3                     1 1
 1 0                     2 1 87  4 3  2   1  0
```

Second Level Page Table Entry

The meaning of each of these bits is as follows:

Bit<31> - Valid. Indicates the validity of the PTE. A 0 indicates an invalid reference; a 1 indicates a valid reference. A segment out-of-bounds error is detected when an invalid PTE is accessed. A reference to an invalid PTE results in a system exception. See Table 6-1.

When bit <31> and bit <30> are both 1, bit<0> is ignored. A valid PTE that references I/O space is always assumed to be resident.

Bits<30..12> - Page Frame Address. If a valid reference to a resident page occurs, then bits <30..12> become the most significant 19 bits of a 31-bit physical byte address. The page frame base is modulo 4096 bytes. For a first level page table entry, bit 30 is always 0, while for a second level page table entry, bit 30 may be either 0 or 1.

Bits<11..9> - Hardware Reserved. These bits are reserved for potential use by hardware. Presently, there is no interpretation of these bits. It is not recommended that these bits be used for software.

Bit<8> - Encache. When bit 8 is zero, the data associated with the reference are encached. When bit 8 is 1, the referenced data are NOT encached.

Bits<7..4> - Software Reserved. These bits are reserved for potential use for software.

Bit<3> - Read Access. Indicates the validity of a read access to the referenced page. A 0 indicates that no read access is permitted. A 1 indicates that a read access is permitted to the referenced page. If a read access is attempted, and bit 3=0, a system exception is signaled, and an error code is loaded into A5.

Bit<2> - Write Access. Indicates the validity of a write access to the referenced page. A 0 indicates that no write access is permitted. A 1 indicates that a write access is permitted to the referenced page. If a write access is attempted, and bit 2=0, a system exception is signaled, and an error code is loaded into A5.

Bit<1> - Execute Access. Indicates the validity of an execute access (branch or jump to instruction) to the referenced page. A 0 indicates that no execute access is permitted. A 1 indicates that an execute access is permitted to the referenced page. If an execute access is attempted, and bit 1=0, a system exception is signaled, and an error code is loaded into A5.

Bit<0> - Non-Resident. Indicates the presence or absence of the referenced page frame in the physical address space of the process. A 0 indicates the absence of the referenced page in physical memory. In this case, a page fault occurs and causes a system exception. A 1 indicates the presence of the referenced page. In this latter case, bits<30..12> are used as the physical page frame address of the referenced page. Bit 0 is interpreted for valid references only.

NOTE: Segment out-of-bounds errors may be detected by resetting all of the unused PTE's valid bits to zero. Thus, during logical to physical address translation for invalid pages, an out-of-bounds reference causes a system exception.

The physical addresses of all pagetables must reside in physical memory (physical addresses 0 through 3FFF FFFF (hex)). Thus, all physical address of pagetable entries are 30 bits in length.

The format of a valid non-resident PTE is:

```
-----------------------------------------------------
| V | 0 |   Software Reserved    | rd | wr | ex | nr |
-----------------------------------------------------
  31 30                         4   3    2    1    0
```

Note that bit <30> must be zero. If bit <30> is a 1, an I/O reference can occur. The read, write, and execute bits are then interpreted to determine if the reference is valid.

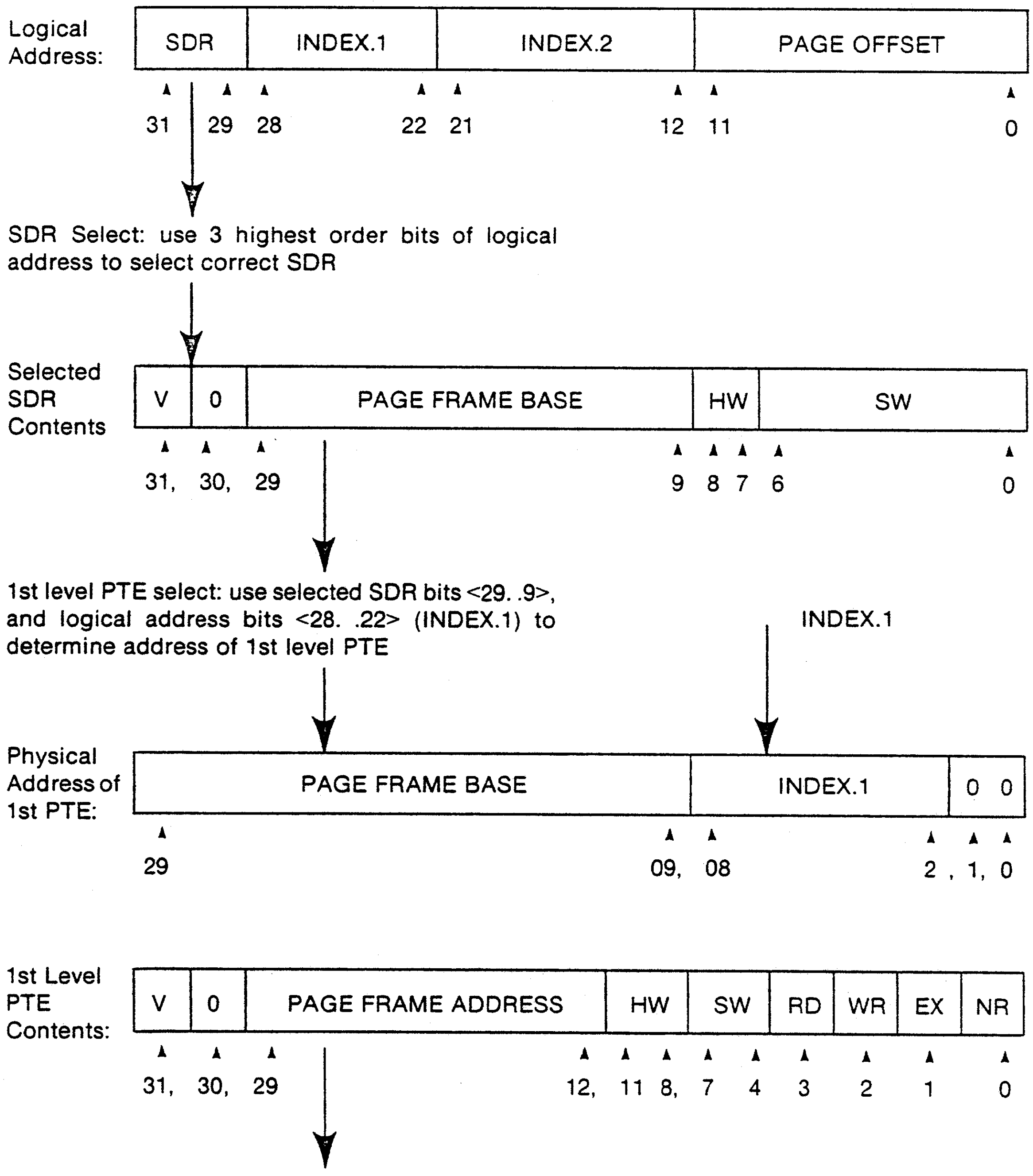
5.6.1 Final Translation

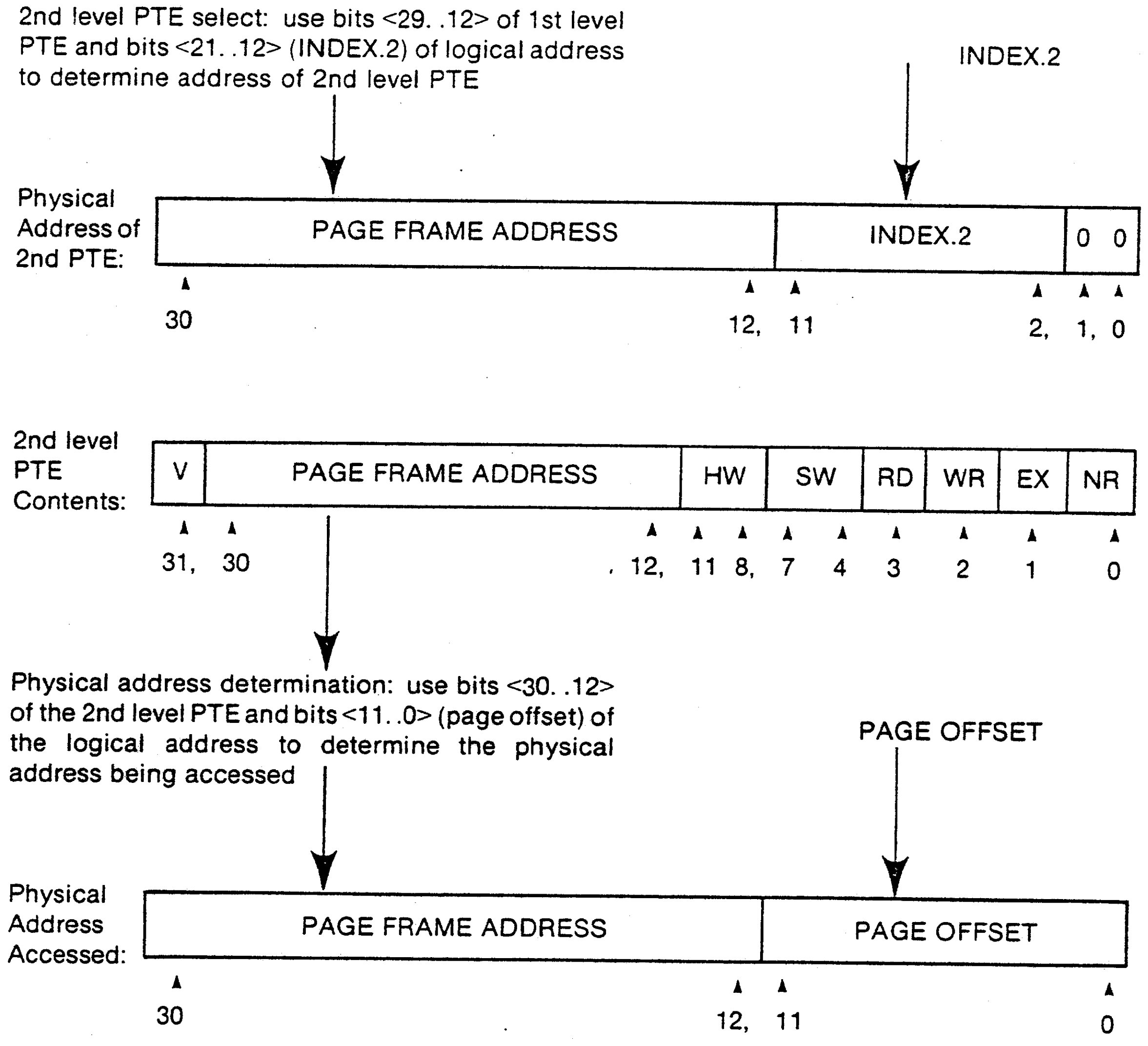
This section presents a pictorial representation of the logical to physical translation. The following attributes of this translation are worth noting:

1 The pagetable referenced by the first level index is always resident in physical memory.
2 The pagetable referenced by the second level index may not be resident in physical memory. A page fault can occur when referencing a second level pagetable page.
3 The access bits in the first level pagetable entry are never interpreted. That is, no protection access checks are performed when a first level pagetable entry is used to reference a second level pagetable entry.
4 If a pagetable entry is invalid, no further translation occurs.
5 A page fault occurs only for valid references.

For logical to physical address translation purposes, a 32-bit byte logical address has the structure shown in Figure 5-2:

Figure 5-4: 32-bit Byte Address: Logical to Physical Translation




2nd level PTE select: use bits <29..12> of 1st level PTE and bits <21..12> (INDEX.2) of logical address to determine address of 2nd level PTE
INDEX.2
Physical Address of 2nd PTE:
PAGE FRAME ADDRESS
INDEX.2
0 0
30
12, 11
2, 1, 0
2nd level PTE Contents:
V
PAGE FRAME ADDRESS
HW
SW
RD
WR
EX
NR
31, 30
12, 11 8, 7 4 3 2 1 0
Physical address determination: use bits <30..12> of the 2nd level PTE and bits <11..0> (page offset) of the logical address to determine the physical address being accessed
PAGE OFFSET
Physical Address Accessed:
PAGE FRAME ADDRESS
PAGE OFFSET
30
12, 11
0

5.7 Referenced/Modified Bits

Associated with each page frame are two flags: "referenced" and "modified". The referenced bit is used to indicate that a successful reference (read, write, or execute access) has occurred (bit is set to 1 in the page corresponding to that page frame). The modified bit is used to indicate that a successful write has occurred (bit is set to 1). A successful write also sets the referenced bit to 1.

For the purposes of memory management, a successful reference is defined as a memory reference which does not result in a PTE violation on a resident page.

I/O memory references do not affect the state of the referenced and modified bits. When power is first applied, the state of the referenced and modified bits is indeterminate.

The referenced and modified bits are mapped into the I/O space. Thus the operating system accesses the referenced and modified bits by mapping the appropriate I/O space into the physical address space of a process.

In particular, the referenced and modified bits are grouped into bytes. Consequently, referenced and modified bits are addressed through the use of Load and Store byte instructions. References using any other instructions produce undefined results.

5.8 Address Translation Unit

The Address Translation Unit is used to accelerate the translation of logical to physical addresses. This unit contains a cache of recently translated logical addresses.

The steps necessary to translate a logical to physical address have already been described. Once a translation occurs, the association between the logical to physical addresses should be placed in memory for the following reasons:

1 The steps necessary to translate the logical addresses require machine cycles that would otherwise be used to execute instructions.
2 Programs exhibit temporal and spatial locality of reference. Thus it is probable that once a logical to physical translation is accomplished and encached (remembered), subsequent logical addresses will reference the same page associated with the initial translation.

Because previous translations can be placed in memory, the number of processor cycles allocated to logical address translations is significantly reduced, a feature which greatly enhances program performance, and makes the ATU an address cache.

The ATU accelerates address translations by associating a logical address with an ATU entry. This ATU entry contains three types of information. One type is a page frame physical address. This address holds the contents of the PTE that referenced the addressed operand. A second type is the higher order bits of the logical address encached. Since the page-offset field of the logical address is not translated, this entry contains, at most, the most significant 20 bits of the translated logical address. The third type of information is concerned with the access privileges associated with the addressed page. The ATU entry provides a convenient place to store these privileges. Some characteristics of the ATU are relevant to the system programmer because:

1 The size and structure of the ATU are implementation dependent.
2 Individual entries within the ATU are not explicitly addressable.
3 Modification of a PTE in memory does not necessarily have an immediate effect, if any, on ATU entries.
4 Several privileged mode instructions exist to permit a level of control over ATU address translation in a manner that is ATU implementation independent. These instructions purge the entire ATU or selective entries. Purging the entire ATU is necessary for process multiplexing. Purging selective ATU entries is used when selective PTE modifications occur (e.g., when an address translation fault finds the physical page in main memory but not in the physical space of the process).

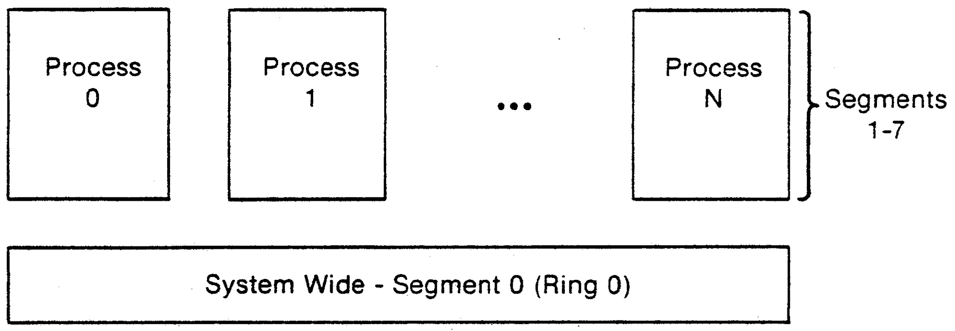
5.9 Process Multiplexing

A process may be defined as an abstraction of the locus of control that passes through an executing program. CONVEX processes are unique in construction and are composed of two general partitions: one partition is the user program; the other is that part of the operating system that is shared by all user processes. This user part of the operating system usually includes such features as pagetables used for translating logical to physical addresses, buffers for disc or terminal records, and various control blocks that are created by the operating system on behalf of the user.

CONVEX processes are unusual in that each has its own private logical address space. Thus logical addresses, though identical in two or more processes, need not translate to the same physical address. As a result of this logical address space structure, the ATU must be purged when a new user process is dispatched. Purging an ATU simply involves marking all entries as invalid, so that no encached translations exist.

Owing to the characteristics of ring 0 of the protection system, encached entries for ring 0 translations need not be purged. This freedom is possible because ring 0 is system-wide (not process-wide), which means that every process shares the same ring 0. Interrupt processing is an example of a system-wide service that is performed in ring 0. This partition is depicted in Figure 5-4.

Figure 5-5: Process/System/Segment Partition



Process/System/Segment Partition

5.10 Power Up/Bootstrap/Physical Addressing

When power is first applied and the system is bootstrapped, logical addresses equal physical addresses. There is no memory mapping or page faulting, and the least significant bits of the logical address space representing the physical address space are passed directly to the main memory system.

An instruction exists to initiate virtual address mapping and the control of the Address Translation Unit. Additional instructions and/or facilities exist to turn off address translation. Generally the existence of the ATU is known only to the operating system kernel, and its internal structure is implementation dependent. All instructions which manipulate or control the ATU are privileged--they can only be executed in ring 0.

When logical addressing is disabled, the processor executes as if it were always in privileged mode (ring 0).

CHAPTER 6

6 Exceptions

6.1 Overview

An exception is an event which disrupts the running of a program, process, or system. Exceptions occur because of problems in the currently executing program (for example, arithmetic inconsistencies or address translation faults), or as a result of some asynchronous event (such as an interrupt or hardware failure). Exceptions result in the transfer of control to a predetermined address known as an exception handler. The starting addresses of the exception handlers are located in tables in memory referred to as Page 0. The definition of these tables is provided in Chapter 5. State information is saved on the appropriate stack.

The primary goals for exception processing are:

1 Wherever possible, the operating system kernel will not be involved.

2 The hardware will structure exceptions as asynchronous kernel calls. This permits the OS kernel to use a single procedure for call processing.

3 The hardware will provide a reasonable and open-ended means to indicate the cause of the exception.

4 The hardware will provide a reasonable means to mask out those exceptions which are under user control.

To achieve these goals, exceptions are grouped into three different classes: process, system, and machine.

1 Process exceptions belong to the currently running process, and may be handled with an exception handler in that process. The exception handler is in the current ring of execution.

2 System exceptions cannot be handled by the current process and require intervention by the kernel executing in ring 0.

3 Machine exceptions include fatal errors in the system which cannot be handled by the operating system.

If exceptions of different classes are pending simultaneously, machine exceptions have the highest priority, followed by system exceptions.

Exceptions may also be subdivided by the way in which they are normally treated by the exception handler. The exception handler will handle exceptions in one of two ways depending on the nature of the exception. In many cases, the exception handler can correct the underlying cause of an exception and permit the original program to resume; on the other hand, the exception handler may choose not to correct the cause of the exception and not to return control to the original program. Two terms signify the way in which the system treats each exception type:

1 A fault is an exception which the handler can normally correct. The exception handler returns control to the program at the place which it was interrupted.

2 A trap is an exception which the handler cannot normally correct. Often, the handler terminates the program or process and supplies error information to the user.

6.2 Process Exceptions

Process exceptions occur at the process level and the user can handle them without system intervention. The exception handler which is called resides in the current ring of execution. The process exceptions are arithmetic trap and instruction trace. Instruction trace, together with sequential execution and the breakpoint instruction (bkpt), provides support for program debugging. In addition, the user can disable or mask out many of the process exceptions.

6.2.1 Arithmetic Trap

An arithmetic trap occurs when an operation encounters or produces an illegal value, one which is not within the representable range of numbers for the machine. However, the user can mask out these exceptions using the appropriate enable bits provided in the PSW. Arithmetic traps are processed thus:

1 The processor sets the appropriate bits within the PSW to 1 to indicate the cause of the trap. Since a CONVEX processor has multiple arithmetic units, it can set more than one bit in the saved PSW. Sufficient bits exist to identify multiple trap types simultaneously.

2 The processor pushes an extended return block onto the current

stack (no ring crossing occurs).

3 The processor clears PSW bits (C, SC, AIV, ADZ, VN, OV, FDZ, RO, SIV, SDZ, FRL) of the newly-generated PSW to 0.

4 Instruction execution for the trap handler begins at the address contained at byte address 44 (hex) of page 0 of the current ring.

5 The machine initiates the trap as soon as steps 1 through 4 have occurred, unless an exception of higher priority is also pending.

The following PSW bits report the occurrence of exceptions AIV, ADZ, SIV, SDZ, UN, OV, RO, and FDZ. The PSW bits IVE, DZE, FE, and FUE selectively enable groups of arithmetic exceptions. The user may choose to ignore certain exceptions by clearing the appropriate enable bit to 0. The IVE/Integer Overflow Trap Enable bit corresponds to the SIV and AIV bits; the DZE/Divide by Zero Enable bit corresponds to the ADZ and SDZ bits; the FE/Floating Point Trap Enable bit corresponds to the OV, RO, and FDZ bits, and finally, the FUE/Floating Point Underflow Enable bit corresponds to the UN bit.

Because of the pipelined nature of the machine, more than one instruction may be executing when a trap occurs. This sequence is:

1 When the machine detects an arithmetic exception that requires a trap, it places all pending instructions on hold.

2 The system allows all current instructions to complete their execution.

3 The system honors the exception only after completing steps 1 and 2, and only if there are no events pending with a higher priority (such as interrupts).

6.2.1.1 Details Of Arithmetic Traps

This section details the characteristics of each type of arithmetic exception.

Integer Overflow. Integer overflow occurs when a result is too large to occupy the specified destination. When an integer overflow occurs, the AIV or SIV bit in the PSW is set to 1. The result loaded into the destination is correct in the least significant bits. When an integer overflow exception occurs for integer longword multiplication (64 bits), the result is correct in the least significant 53 bits. Bits<63..53> are undefined.

Integer Divide By Zero. When the divisor is zero, the processor sets the appropriate divide by zero bit in the PSW (ADZ or SDZ) to 1. The output of the divide is the dividend.

Floating Divide By Zero. When the divisor is zero, the processor sets the FDZ bit to 1. The output of the divide is a reserved operand.

Floating Point Overflow. When the resulting exponent requires more precision than is allowed (greater than 127 (unbiased) for single and greater than 1023 for double), a floating point overflow occurs. The result operand is forced to a reserved operand (sign=1, exponent and fraction all 0's). The OV bit in the PSW is set.

Floating Point Underflow. When the resulting exponent requires an exponent less than -127 (unbiased) for single precision and less than -1023 for double precision, a floating point underflow occurs. The resulting operand is forced to true zero (sign =0, exponent=0, fraction=0). True zero is forced regardless of the value of the underflow trap enable bit. The machine sets the UN bit in the PSW.

Reserved Operand. When an input to a floating point arithmetic operation has a sign=1 and an exponent of all 0, a reserved operand exception is detected. The fraction value is a "don't care." The output of an arithmetic operation with a reserved operand input is a reserved operand output. A reserved operand output has an all 0's fraction. The RO bit in the PSW is set.

6.2.2 Debugging Support

The remainder of the process exceptions are useful debugging tools. Instruction trace allows a single instruction to execute between each exception, the sequential bit (SEQ) in the PSW forces instructions to execute one at a time without overlap, and a breakpoint instruction (bkpt) causes transfer of control when it is executed.

6.2.2.1 Instruction Trace

Instruction trace is a useful debugging tool, one which the user can directly control. Setting bit 27 of the PSW (TR) enables instruction trace and causes a trace trap to occur. After the execution of each instruction, the processor pushes an extended return block onto the stack. The Program

Counter pushed references the next instruction to be executed in the program. The exception handler is located at the address contained at address 40 (hex) of the current ring. Since no ring crossing occurs, the processing of this trap does not involve the operating system. For instruction trace to function properly, bit 24 of the PSW, SEQ, must also be set to 1.

6.2.3 Sequential Execution

Although sequential execution is not an exception, its value affects the operation of the machine and perhaps the specific conditions which exist when an exception occurs. As noted in Chapter 3, when a program sets bit 24, SEQ, of the PSW, all overlapped execution is disabled. Owing to the pipelined characteristics of the machine, multiple instructions are often executing simultaneously. The SEQ bit forces execution in a serial manner for debugging purposes (both for hardware and software). The numerical results produced are the same regardless of the setting of the SEQ bit. The procedure only affects performance and the serial nature of the execution. The user may freely set or reset this bit.

6.2.3.1 Breakpoint

Although the breakpoint instruction (bkpt) is also not a true exception, it is included here since it qualifies as a debugging tool. The user may insert bkpt instructions anywhere within a program, and execution of the bkpt instruction causes a call to the routine addressed by byte 50 (hex) of the current ring, and pushes an extended return block on the stack.

6.3 System Exceptions

All system exceptions result in a ring crossing to Ring 0, the operating system kernel. The return block saved in each case is either extended (FRL=01) or context (FRL=00). Many of the system exceptions are related to virtual memory address translation, as described in Chapter 5. Chapter 6 describes the processing of system exceptions.

All system exceptions have the following characteristics:

1 They are not maskable.

2 They always result in a cross ring call to ring 0.

3 There are residency and alignment requirements for Page 0. The ring 0 stack must always be aligned on a 32-bit (word) boundary, and ring 0 page 0 must be resident. If not, a machine exception results.

6.3.1 Error Exit Trap

An error exit trap occurs if the processor encounters an all-zero opcode. If the processor attempts to execute code from memory which resides beyond the boundaries of a program, this trap will occur, assuming that the memory in question has previously been cleared to zero.

6.3.2 Undefined Opcode Trap

An undefined opcode trap is executed whenever the processor attempts an illegal instruction.

6.3.3 Vector Valid Fault

The vector valid fault, coupled with the Vector Valid bit (VV), permits the operating system to save and restore the vector accumulators on demand. Additionally, it permits the OS to detect unsuspected use of vector instructions. The example below details the use of this fault.

Assume that 10 programs are running but that only 2 use the vector accumulators. Upon interrupt processing, during one of these 2 programs, the system need not save the vector accumulators since the interrupt service routine does not use them. If subsequent programs do not use the vector accumulators (either statically because there is no need, or dynamically because the particular code segment is not vector in nature), no time is wasted in saving vector machine state. This enhances the real-time characteristics of the system. However, if one of these subsequent programs correctly uses the vector machine state, a recoverable fault occurs. This fault indicates to the operating system that it is time to save a previous process's vector machine state. Once this state is saved, the affected program can resume.

The vector valid fault functions in this sequence:

1 The Vector Valid (VV) bit must be a 0.

2 The processor attempts execution of a vector instruction.

3 The machine now performs a ring crossing to ring 0, and pushes an extended return block on the ring 0 stack.

4 Finally, the machine jumps to the instruction pointed to by address 1C (hex) of page 0 of ring 0.

6.3.4 Ring Violation Traps

Ring violation traps are a group of system exceptions concerning invalid access to rings. The operation of the ring structure is defined in Chapter 4. The following ring violations are defined:

1 Privileged Instruction. This trap occurs when the system attempts a privileged instruction outside of ring 0.

2 Inward Address. A reference to an address which is in an inner ring causes this trap.

3 Outward Call. A call which crosses rings must proceed to an inner ring; otherwise this trap will occur.

4 Inward Return. This trap occurs when a return instruction attempts to move to an inward ring; all returns must be to the same or an outward ring.

5 Invalid Gate. If the gate number specified in a call which crosses rings is incorrect, this trap will occur.

6.3.5 PTE Violation Traps

PTE violation traps encompass a group of illegal Page Table Entry accesses. Pagetables and their usage are defined in Chapter 5. PTE violations include:

1 Read Protect. This trap is invoked when the processor attempts a read access to a page whose valid Page Table Entry does not allow reads.

2 Write Protect. A write protect trap occurs during an attempted write to a page whose valid PTE does not have write enabled.

3 Execute Protect. This trap occurs when the processor attempts an instruction fetch on a page without execute enabled in its valid PTE.

4 Invalid SDR. A memory access to a segment whose SDR's valid bit is not set causes this trap to occur.

5 Invalid Level 1 PTE. A memory reference to an address whose first level PTE's valid bit is not set results in this trap.

6 Invalid Level 2 PTE. If an address's corresponding Level 2 PTE

is not valid, an invalid level 2 PTE trap occurs.

7 Invalid I/O Access.

6.3.6 Non-resident Page Faults

A non-resident page fault occurs when the processor attempts to reference a memory location which is part of the logical address space but is not part of the physical address space. The system initiates a page fault only after it has interpreted the validity and appropriate access bits in a Page Table Entry. The two forms of this fault are:

1 Non-resident data page. If the actual data page corresponding to the logical address is not in physical memory, this fault occurs.

2 Non-resident Level 2 pagetable. A page fault can also occur as a result of a logical reference which, as part of its logical address translation, accesses a non-resident pagetable.

Note: if the system detects another page fault while the processor is responding to a page fault as described in the above sequence, a machine exception occurs. This check prevents the generation of an infinite number of page faults.

6.3.7 Processing of System Exceptions

When the machine detects a system exception, it accesses the exception handler by one of two methods. First, address 1C (hex) of page 0 of ring 0 points to the vector valid fault's exception handler. Second, a single exception handler serves all other system exceptions; address C (hex) of page 0 of ring 0 contains its address. Information passed in registers A3, A4, and A5 describes these exceptions. As soon as the system pushes a return block on the current ring 0 stack, it loads an exception code into A5. Byte 0 and 1 of this code are always 0. Byte 2 specifies the class of the exception, and Byte 3 is an optional qualifier for that particular class.

In addition to the codes loaded into A5, the processor loads the logical address of the failure into A4, and loads the number of bytes stored in the return block into A3 if a context block is saved (FRL=00). In fact, the FRL field of the PSW pushed specifies the type of return block: short, long, extended, or context. Whereas the number of bytes for short, long, and extended are invariant for the life of the architecture, the context block is implementation dependent.

Table 6-1 lists the class codes and qualifiers placed in A5 for each exception.

Table 6-1: System Exceptions: Class Codes and Qualifiers

	BYTE 2	BYTE 3
<u>EXCEPTION</u>	<u>CLASS</u> (<u>Hex</u>)	<u>QUALIFIERS</u>
Error Exit	0 Highest Priority	None
Undefined Opcode	4	None
Ring Violation	8	0 Privileged Instruction 1 Inward Address 2 Outward Call 3 Inward Return 4 Invalid Gate 5 Invalid Frame Length on Return Instruction
PTE Violation	C	0 Read Protect 1 Write Protect 2 Execute Protect 4 Invalid SDR 5 Invalid Level 1 PTE 6 Invalid Level 2 PTE 7 Invalid I/O Access
Non-Resident Page	10 Lowest Priority	0 Level 1 PTE2 Page 1 Level 2 DATA Page

6.4 <u>Machine Exceptions</u>

Machine exceptions comprise the most drastic of exceptions. Hardware failures, such as memory errors, which cannot be corrected, and parity errors cause machine exceptions. In addition, the following programmer visible conditions result in machine exceptions:

Exceptions

1 Unaligned Ring 0 stack. The stacks in ring 0 must always be aligned on a 32-bit boundary. If not, a machine exception occurs.

2 Page fault during a page fault. If either a PTE violation trap or a non-resident page fault occurs while the machine is changing context to service one of these two exceptions, a machine exception occurs. If this condition were not detected, an infinite number of page faults would occur.

3 Non-resident data for segment descriptor registers. If a non-resident page fault occurs for the data read by either the load kernel segment descriptor registers (ldkdr) or load process segment descriptor registers (ldsdr), a machine exception results.

4 Unaligned data for segment descriptor registers. If the data to be loaded by either an ldkdr or an ldsdr are not word aligned on a 32-bit boundary, a machine exception also occurs.

5 Execution of ldkdr after virtual memory is enabled. The execution of an ldkdr (load kernel segment descriptor registers) instruction after virtual memory has been enabled causes a machine exception.

6 Invalid SDR0 after virtual memory is enabled. If SDR0 is accessed while its valid bit is cleared, the resulting exception cannot be processed and a machine exception results.

The processing of machine exceptions is implementation dependent. In general, however, further processing is not possible. The machine might post a message to the console or to an error logging device, might alert a diagnostic processor (if one exists), or might simply halt the machine. The details of machine exception processing are presented in The Hardware Handbook.

CHAPTER 7

7 I/O and Interrupts

All I/O is memory mapped, which means that there are no explicit central processing unit (CPU) instructions that reference I/O control or data registers. I/O registers and status bits are referenced through an appropriate logical to physical address mapping. The I/O register space is 1 billion bytes. In essence, up to 1 billion I/O registers can be referenced. As a function of the implementation, however, certain types of operand references may cause undesirable side effects. Generally, I/O operand references should be on an integral byte boundary, such that the least significant address bits equal to the precision of the referenced operand will be all 0's.

Interrupts are a result of asynchronously occurring events and belong to the system and not to the executing process. They are processed on an interrupt stack in ring 0. Because interrupts can occur during interrupt processing, a means to nest them must be provided. When an interrupt occurs, the processor will vector to a particular handler as a function of the source of the interrupt.

7.1 JP and I/O Interrupt Channels

There are two types of channels: central processing unit virtual channels and I/O channels. There are 256 interrupt channels in a CONVEX system. Within these 256 channels, 8 channels are specifically allocated to the central processing unit (CPU). These 8 channels are referred to as CPU virtual channels 0-7; they are also addressed as channels 0-7 of the 256 possible system-wide channels. The remaining 248 channels are allocated to I/O processors. The number of I/O processors and the number of channels allocated to I/O processors are a function of a particular implementation. Instructions exist for any one channel to interrupt another channel.

All external devices and controllers, regardless of their local intelligence, interrupt the CPU on one of eight channel ports. These eight ports bear no relationship to the number of actual I/O channels. In fact, one I/O channel may initiate interrupts using more than one CPU virtual channel. Conversely, there may be up to 248 I/O channels competing for eight CPU virtual channels. The "mski" instruction is used by the CPU to mask out interrupts selectively from a particular CPU virtual channel.

There are up to 248 I/O virtual channels. The CPU can interrupt, individually, any of the I/O channels through the use of the "xmti" instruction. In some cases, one physical I/O controller may be viewed as multiple I/O channels. The CPU can interrupt itself by addressing channels 0 through 7.

7.2 Interrupt Mechanism

When an interrupt (e.g. I/O, or Internal Timer) occurs, the following actions take place. The 16 bit halfword located at bytes 4 and 5 of page 0 of ring 0 is fetched. If this halfword is 0, then the interrupt is the first interrupt processed. This condition is referred to as base level interrupt processing. If this halfword is not the first interrupt, then the interrupt is not the first interrupt, and the processor is already at interrupt level. (Thus the ring 0 stack used is the interrupt stack. In effect, the ring 0 process stack pointer has temporarily become the interrupt stack pointer).

The fundamental difference between the two classifications is the existence of an interrupt stack. When the interrupt level is 0, an interrupt stack must be established in ring 0. Once the determination is made, the interrupted level's halfword is incremented by 1 and stored back into bytes 4 and 5 (the increment by 1 cannot be interrupted).

In the following discussion, the program counter which is pushed onto the stack references the instruction that would have been executed if the interrupt had not occurred. Also, during the sequences described, all further interrupts are kept pending (ION is reset to 0).

If the interrupt is initiated by an I/O device interrupting a CPU virtual channel, then the CPU virtual channel interrupt is reset after the processor responds to the interrupt.

7.2.1 Ring 0 Stack Alignment

The ring 0 stack must always be aligned on a 32-bit (word) boundary. If it is not, a machine exception occurs.

7.2.2 Base-Level Processing

Base-level processing occurs when the current interrupt level is 0. The actions that subsequently occur are determined by the current ring of execution, be it ring 0 or any other ring.

7.2.3 Base-Level Ring 0

It is assumed that the stack pointer is already initialized to the ring 0 address space. Since the interrupt is at base-level, stack multiplexing to the interrupt stack must occur, and the following sequences are initiated:

1 FRL is set to 01. (Extended Return Block).

2 The extended return block is saved on the current ring 0 stack.

3 The PSW is cleared.

4 The updated stack pointer is saved in byte address 2C (hex) of page 0, of ring 0. This value is the process stack pointer at the top of the ring 0 process stack. This procedure is preparatory to stack multiplexing to the interrupt stack.

5 The stack pointer (A0) and frame pointer (A7) are loaded from byte address 20 (hex) of page 0, of ring 0. This is the interrupt stack pointer.

6 A common hardware interrupt sequence is then executed.

When a return to base-level is performed, the interrupt dismissal routine moves the previous process SP (in byte address 2C (hex) of ring 0) to A0 prior to executing the rtn instruction.

7.2.3.1 Base-Level Processing--Non Ring 0

In this case, the hardware performs a crossing to ring 0 and establishes an interrupt stack.

1 A ring crossing to ring 0 is executed (as if the sysc instruction were executed).
2 The steps described in the above section, for Base Level ring 0, are now executed.

7.2.4 Interrupt

At interrupt level, the ring 0 stack has already been initialized to the interrupt stack.

7.2.4.1 Ring 0--Interrupt Level

An extended return block is pushed onto the current stack and the common interrupt sequence is then entered.

7.2.4.2 Non-Ring 0--Interrupt Level

In this case, the following actions are taken:

1 A ring crossing to ring 0 is executed.
2 Since the ring 0 stack has already been initialized to the interrupt stack, an extended return block is pushed on the ring 0 stack, and the common interrupt sequence entered.

7.3 Common Interrupt Sequence

The following section describes the actions undertaken after a crossing to ring 0 and the establishment of an interrupt stack have occurred. There are two causes of an interrupt: an I/O device via a virtual channel (byte address 08 (hex)), and an Interval Timer (byte address 14 (hex)).

1 Byte addresses 08, 10, and 14 (all hex) of ring 0 contain the address of the appropriate interrupt handler. This address, selected by hardware, is loaded into the program counter. If the interrupt is caused by an I/O device, the identification of the interrupting device is loaded into A5 after the return block is pushed. This identification takes the form of 29 0 bits followed by a 3-bit encoding. The 3-bit encoding identifies which CPU virtual channel initiated the interrupt.

2 The first instruction of the interrupt handler is now executed. Interrupts are not enabled. The interrupt handler must explicitly reenable interrupts.

7.4 General Notes

1 The interrupt return sequence determines whether or not the return is to base-level or interrupt-level as a function of the interrupt halfword in ring 0, page 0.

2 The return to base-level is achieved by executing a rtn instruction. The return to interrupt level is also achieved by executing an rtn instruction.

3 In order to return from an interrupt, the following steps must be taken by the software:

a. First, the interrupt level is decremented by one.

b. If the level is now zero, A7 (FP) is loaded from byte address 2C (hex) of page 0, and the rtn instruction is executed.

c. If the level is not zero, the rtn instruction is executed. A7 need not be restored, since the ring 0 stack must still be the interrupt stack.

4 The process stack pointer in page zero, of ring 0 bytes <72..75> is not modified during the hardware initiated interrupt processing.

CHAPTER 8

8 Instruction Set Overview

This chapter and subsequent chapters describe the instruction set. The instruction set is used to generate logical addresses, load, store, and manipulate operands, and manipulate the virtual machine mechanisms.

8.1 Overview

A CONVEX instruction is one of three lengths: one, two, or three halfwords. This is equivalent to instructions that are 16, 32, or 48 bits in length. Even though the fundamental unit of addressability is the byte, instructions are addressed on a halfword boundary. All instructions begin on even byte boundaries. Thus bit 0 of the Program Counter (PC) is never interpreted.

The instruction set was designed to meet high standards and can be characterized as follows:

1 The instruction set is simple and easy to understand and decode--hence the adoption of a RISC (Reduced Instruction Set Architecture).

2 All manipulations are generally register to register. Loads and stores are needed to transfer information to and from the register set.

3 Orthogonality of instructions is assured; for every data type manipulation, there exists the same operation. (In some cases, this makes no sense, and thus there are exceptions.)

8.2 Instruction Formats

There are 8 instruction formats. The first 8 bits of the instruction (<15..8>) are encoded using a modified leading 1's (HUFFMAN) encoding method to indicate the instruction format. The 8 instruction formats are:

Instruction Set Overview

FORMAT	
0	1, [op=6], [Ri], [Rj], [Rk]
1	00, [op=6], [@,L], [Ak], [Rj] [disp=16/32]
2	010, [op=7], [Rj], [Rk]
3	0110, [op=6], [Rj], [Rk]
4	0111 0, [op=3], [disp=8]
5	0111 10, [op=4], [Rj], [Rk]
6	0111 110, [op=6], [Rk]
7	0111 1110, [op=5], [Rk]

Note that R is either A, S, or V as a function of the particular opcode.

8.3 Addressing Modes

There are two generic types of addressing modes: register to register and register to/from memory.

In the register to register mode, it is assumed that all source operands to be manipulated have been pre-loaded into one of the various machine registers. The destination of the result is also a register. Register to register instructions specify none, one, two, or three unique registers. Thus some instructions can have up to 3 3-bit register designation fields. The opcode specifies the number of register fields in the remaining bits of the instruction. All register to register instructions are 16-bits in length.

8.3.1 Referencing Memory

This section describes how to build Effective Addresses. Generally, instructions which reference memory to load and store operands are either 32-bits or 48-bits in length. The difference is the length of the displacement field. The structure of these memory reference instructions is:

```
            ------------------------  ------------------
 L=0        | Opcode |@|L| Aj   | Rk|  |  Displacement  |
            ------------------------  ------------------
            15      8, 7,6,5  3,2  0  15             0


           ------------------------   ----------------------------
 L=1       | Opcode |@|L| Aj   | Rk|  |  Displacement             |
           ------------------------   ----------------------------
           15      8,7,6,5  3,2  0    31                        0
```

MEMORY LOAD AND STORE FORMATS

The meaning of these fields is as follows:

OPCODE = bits<15..8>. Specifies the operation to be performed on the referenced data.

@ = bit<7> - Indirection. Specifies the existence or absence of indirection. If @=0, no indirection is specified. If @=1, indirection is specified.

L = bit<6> - Length. Specifies the length of the displacement field. If L=0, the displacement field is a 2's complement 16-bit integer. This field is sign extended to 32 bits prior to being added to the contents of the address register specified in bits <5..3> of the opcode halfword of the instruction.

Rk = bits<2..0>. Specifies the source or destination of the referenced operand. The opcode specifies the precision, datatype, structure, and direction of the move. Rk can either be a scalar register (Sk), an address register (Ak), or a vector register (Vk).

Aj = bits<5..3>. Specifies the A (address) register to be used to generate the logical address. If A0 is specified, the value of 0 is used as the contents of A0 for absolute addressing. The true contents of A0 are unused. If A1-7 is specified, then the contents (all 32 bits) of the specified address register are added to the displacement field to generate the final effective address of the target address. The first byte of the target operand is referenced. If indirection is specified, the effective target address references byte 0 of a 32-bit indirect word.

Displacement. A 16 bit or 32 bit value that is algebraically added to the entire contents of an A register (A1-A7) or used directly as a byte address. A 16 bit displacement is sign extended to 32 bits before it is used.

8.3.2 Indirection

Indirection is the means by which an address in logical memory can be used to reference the ultimate target operand. Indirection occurs after indexing (adding of the address register). The format of an indirect word is:

```
------------------------------------
|       Byte Pointer               |
------------------------------------
31                                0
```

Only one level of indirection can be specified.

8.3.3 Branches

Some forms of transfers of control are Program Counter relative. In these cases, no indirection or indexing is specified. There is a special form of transfer which permits an 8-bit signed displacement to be specified within a 16-bit (halfword) instruction. This is covered in more detail in Chapter 11, "Program Control Instruction Set."

8.4 Undefined Opcodes

When an attempt is made to execute an undefined opcode, a system exception occurs. An undefined opcode is a syntactically correct instruction whose opcode field has no associated definition.

An undefined opcode results in a system call through byte address 0C (hex) of page 0 of ring 0 (the system exception handler). The class code loaded into byte 2 of A5 after a push of the return block is 1. No qualifier code is loaded into byte 3 of A5.

8.5 Form Of Presentation

The following chapters contain the definition of the CONVEX instruction set.

Some of the conventions used throughout the instruction set definition are:

Data types are specified with the following suffixes appended to the opcode:

b - Byte/8 bits integer
h - Halfword/16 bits integer
w - Word/32 bits integer
l - Long/64 bits integer
s - Single Precision Floating Point/32 bits
d - Double Precision Floating Point/64 bits

t - True
f - False

Other notation used includes:

VS - Vector Stride
VL - Vector Length
VM - Vector Merge
#n - a 3 bit immediate with value 0,1,..,7
#N - a 32 bit 2's complement signed immediate

In the instruction set definition, two attributes are the alteration of the PSW and the detection of exceptions. If these fields in the instruction set definition are blank, then there is no change to the PSW and no exceptions are detected.

8.5.1 Meta-notation for Instruction Syntax

The following meta-notation is used to describe the contents of memory in the assembly language syntax:

1 Effective Address means the effective memory address of an operand.

2 c(effective address) = Sk means that the contents of the Sk register are stored into the memory location specified by the effective address.

3 Sk = c(effective address) means that the contents of the memory location specified by the effective address are loaded into Sk.

4 | = means alternation. Thus (a|b) means a or b.

5 ! = is used as a comment delimiter. All text to the right of the ; is an English language comment relative to the metalanguage on that line.

8.5.2 Instruction Page Layout

The generic format and layout used in this handbook for an instruction are shown in Figure 8-1.

Figure 8-1: Instruction Page Layout

The Name of the instruction(s) The assembler syntax

Purpose:

"The purpose or intent of the instruction"

Format:

The physical format of the instruction, including field locations and use.

Operation:

The FORTRAN metalanguage description of the instruction.

PSW:

The listing of alterations to the PSW, if any. If there is none, this space is left blank.

Exceptions:

The listing of exceptions that are detected. If a trap is enabled, a trap occurs. Please note that for all instructions which reference memory, exceptions related to address translation (such as page faults or protection violations) can occur.

Opcode:

A listing of the assembly language syntax, binary opcode, and opcode name. One page may contain the definition of many orthogonal instructions. The binary shown for each opcode is to be placed in the opcode field of the instruction format. Additional fixed subfields within the instruction are shown in the above format section. The opcode is presented in three columns. The first column is the opcode name; the second is the binary encoding of the opcode, and the third is the opcode function. For example, for the following instruction,

add.w Aj,Ak	0101100001	Add addr. reg. word

add.w Aj,Ak is the opcode name and the assembly language syntax, and 0101100001 is the binary encoding of the opcode. Add address registers is a brief description of the opcode function.

Description:

An English description of the functions performed by the instruction.

Notes:

A listing of notes that may be of interest to the understanding and use of this or other appropriate instructions.

CHAPTER 9

9 Address Register Instruction Set

This chapter describes the instructions which manipulate the Address registers:

1 Loads and Stores
2 Arithmetics
3 Logical Operations
4 Shift/Push/Pop/Move/PSW/Effective Address
5 Compares
6 A Register Conversions

9.1 Overview

The instructions defined in this chapter manipulate operands in the A registers. When these operands are less than 32 bits, only the specified bits of the A registers are modified in a known way. All other bits are left unchanged.

Instructions are provided which load and store bytes, halfwords, and words. Arithmetic instructions are provided to: add, subtract, multiply, and divide halfwords and words. All arithmetics are performed using 2's complement manipulations. Overflow is generated and stored in the PSW. Integer traps can be disabled. Logical operations operate on words only. A full set of signed and unsigned compares is provided. These compares set or reset the carry flag.

9.2 Loads and Stores

These instructions describe the means by which operands are loaded and stored to/from address registers. These instructions include Load Address Register, Store Address Register, and Load Address/ Immediate. Load and Store work with byte, halfword, and word operands, and Load Immediate with halfword and word. These instructions do not affect the flags in the Program Status Word.

9.3 Arithmetics

These instructions perform 2's complement arithmetics on the contents of the specified address registers. Included are instructions for Add, Subtract, Multiply, Divide, and Negate Address/Address, and the following Immediates: Add Address/Immediate, Subtract Address/Immediate, Multiply Address/Immediate, and Divide Address/Immediate. Add Scalar Address is also included here. All Arithmetic instructions are either halfword or word and affect the flags in the Program Status Word as follows:

Arithmetic Overflow is signified by the AIV bit in the PSW, the C bit reflects a carry or borrow out of the most significant bit of the operation, and the ADZ bit is set on the occurrence of a divide by zero.

9.4 Logical Operations

Logical operations perform a bitwise operation between two address registers. Four sets of logicals are provided (AND, OR, XOR, Complement). All 32 bits of Ak and Aj participate in the logical operation. The instructions are AND Address/Address, OR Address/Address, Exclusive OR Address/Address, and Complement Address/Address, and the following Logical Immediates: AND Address/Immediate, OR Address/Immediate, and Exclusive OR Address/Immediate.

9.5 Shift/Push/Pop/Move/PSW/Effective Address

These instructions include Logical Shift Address/Address, Logical Shift Address/Immediate, Load Effective Address, Push Effective Address, Move PC/Address, Move Address/Address, Push Address Register, Pop Address Register, Test and Set, Load Physical, Move PSW/Address, and Move Address/PSW. Move PC/Address, Push, and Pop Address Registers, Move PSW/Address and Move Address/PSW all use word operands; Test and Set operates on a byte operand.

9.6 Compares

Address register comparisons result in the setting or clearing of the carry bit (bit 31 of the PSW). If the specified comparison is true, the carry bit is set to 1. If the comparison is false, the carry bit is cleared to 0. There are 2 sets of comparisons: signed and unsigned. Unsigned comparisons treat both operands as positive values. For each set of comparisons, an instruction group is provided which permits the specification of an immediate. One form of the immediates permits a register within a 16 bit instruction to compare with the values: 0,1,...,7. This is particularly useful in comparing with 0.

Typically, after the execution of one of these compare instructions, a branch on carry instruction is executed. The strategy adopted for compares and branches is as follows: provide a compare register-to-register instruction for one of three of the possible six compare relations. If the required relation is not one of the three provided, then the complement of the relation is used along with the complement of the branch condition.

The following are the complementary relations used for compares:

```
.LE. <-> .GT.
.LT. <-> .GE.
.NE. <-> .EQ.
```

For example A .LE. B , BRANCH TRUE is equivalent to A .GT. B, BRANCH FALSE. Coincidentally, for many operators A .REL. B is identical to B .not. .REL. A. Thus A .LE. B, BRANCH TRUE is equivalent to B .GE. A, BRANCH TRUE.

The Compare instructions include Compare Address/Address, Compare Address/Address Unsigned, Compare Address/Immediate, and Compare Address/Immediate Unsigned. These operands are all halfword and word instructions, and affect the C flag as the result of a compare.

9.7 A Register Conversions

This instruction--Convert Integer Address/Address--converts the various A register integer operands from one precision to another and affects the AIV flag in integer overflow.

LOAD ADDRESS REGISTER

ld.(b|h|w) <effa>,Ak

Purpose:

To load memory operands into the address registers.

Format:

```
----------------------------    ------------------
|  Opcode   |@|L| Aj   | Ak  |    | Displacement   |
----------------------------    ------------------
15         8,7,6,5   3,2   0      (31,15)          0
```

Operation:

Ak = c(Effective Address)

PSW:

Exceptions:

Opcode:

ld.b <effa>,Ak	001010000	Load addr. reg. byte
ld.h <effa>,Ak	001010010	Load addr. reg. halfword
ld.w <effa>,Ak	001010100	Load addr. reg. word

Description:

The operand referenced by the effective address is loaded into the address register Ak.

Notes:

Byte operands are loaded into bits<7..0> of the specified A register. Halfword operands are loaded into bits<15..0> of the specified A register. All other bits are left unchanged.

STORE ADDRESS REGISTER st.(b|h|w) Ak,<effa>

Purpose:

To store the contents of an address into memory.

Format:

```
----------------------------     ------------------
|  Opcode  |@|L| Aj  | Ak  |     |  Displacement  |
----------------------------     ------------------
15         8,7,6,5   3,2   0     (31,15)          0
```

Operation:

c(Effective Address) = Ak

PSW:

Exceptions:

Opcode:

st.b Ak,<effa>	001011000	Store addr. reg. byte
st.h Ak,<effa>	001011010	Store addr. reg. halfword
st.w Ak,<effa>	001011100	Store addr. reg. word

Description:

The contents of the source address register Ak are stored into the memory location referenced by the effective address.

Notes:

Higher order bits of the Ak register used as the source are ignored for byte and halfword stores.

## LOAD ADDRESS/IMMEDIATE 			ld.(h|w) #(n|N),Ak

Purpose:
To load an address register with an immediate operand.

Format:
Short Immediate
```
---------------------------
| Opcode  |  n    |  Ak   |
---------------------------
15        6,5    3,2     0
```

Long Immediate
```
----------------------------  --------------------
|  Opcode |L| 000  |  Ak   |  |        N         |
----------------------------  --------------------
15,       7,6 5    3,2     0  (31|15)           0
```

Operation:
```
Short: Ak = #n          !#n is 0,1,...,7.
Long:  Ak = #N          !#N is an Immediate.
```

PSW:

Exceptions:

Opcode:

ld.h #N,Ak	000100010	Load halfword imm. into Ak
ld.w #N,Ak	000100011	Load imm. into Ak
ld.h #n,Ak	0100010010	Load short imm. into Ak
ld.w #n,Ak	0100010011	Load short imm. into Ak

Description:
The specified immediate field is loaded into the specified address register.

ADD ADDRESS/ADDRESS add.(h|w) Aj,Ak

Purpose:
To add the contents of one address register to another.

Format:
```
----------------------------
| Opcode  |  Aj   |  Ak   |
----------------------------
15        6,5     3,2     0
```

Operation:
Ak = Ak + Aj

PSW:
C = Carry out of the most significant bit
AIV = Integer Overflow

Exceptions:
Integer Overflow

Opcode:

add.h Aj,Ak	0101100000	Add addr. reg. halfword
add.w Aj,Ak	0101100001	Add addr. reg. word

Description:
The contents of the address register Ak are added to the address register Aj and the result loaded into the Ak.

Notes:

MULTIPLY ADDRESS/ADDRESS mul.(h|w) Aj,Ak

Purpose:

To multiply the contents of one address register with another.

Format:

```
----------------------------
| Opcode  |  Aj   |  Ak    |
----------------------------
15        6,5     3,2     0
```

Operation:

Ak = Ak * Aj

PSW:

C = Unchanged
AIV = Integer overflow

Exceptions:

Integer Overflow

Opcode:

mul.h Aj,Ak	0101110000	Multiply addr. reg. halfword
mul.w Aj,AK	0101110001	Multiply addr. reg. word

Description:

The contents of the address register Aj are multiplied by the address register Ak and the results loaded into Ak.

Notes:

The precision of the result is equal to the precision of the Ak.

DIVIDE ADDRESS/ADDRESS div.(h|w) Aj,Ak

Purpose:

To divide the contents of one address register by another.

Format:

```
----------------------------
| Opcode  |  Aj    |  Ak   |
----------------------------
15        6,5     3,2     0
```

Operation:

Ak = Ak / Aj

PSW:

AIV = Integer overflow
ADZ = Address Register divide by zero

Exceptions:

Integer Overflow
Divide by Zero

Opcode:

div.h Aj,Ak	0101111000	Divide addr. reg. halfword
div.w Aj,Ak	0101111001	Divide addr. reg. word

Description:

The contents of the address register Ak are divided by the address register Aj and the result loaded into the address register Ak.

Notes:

1 Integer overflow occurs if the largest negative number is divided by -1.
2 If a divide by 0 is detected, the result is the original dividend.

NEGATE ADDRESS/ADDRESS neg. (h|w) Aj,Ak

Purpose:

To negate the contents of an address register.

Format:

```
----------------------------
|  Opcode  |  Aj   |  Ak   |
----------------------------
15         6,5     3,2     0
```

Operation:

Ak = 0 - Aj

PSW:

AIV = Integer overflow
C = Carry Out

Exceptions:

Integer Overflow

Opcode:

neg.h Aj,Ak	0101011010	Negate addr. reg. halfword
neg.w Aj,Ak	0101011011	Negate addr. reg. word

Description:

The two's complement of Aj are loaded into Ak.

Notes:

1 The negate operation is identical to subtracting the contents of a register from 0.

2 Overflow can occur only for the negation of the most negative number.

ADD ADDRESS/IMMEDIATE　add.(h|w) #(n|N),Ak

Purpose:

To Add an immediate field to an address register.

Format:

Short Immediate

```
---------------------------
| Opcode  |  n     |  Ak   |
---------------------------
15        6,5     3,2     0
```

Long Immediate

```
----------------------------   --------------------
|  Opcode |L| 000  |  Ak   |   |       N          |
----------------------------   --------------------
15,       7,6 5    3,2     0   (31|15)            0
```

Operation:

```
Short: Ak = Ak + #n      !#n is 0,1,...,7.
Long:  Ak = Ak + #N      !#N is an Immediate.
```

PSW:

C = Carry Out
AIV = Integer Overflow

Exceptions:

Integer Overflow

Opcode:

add.h #n,Ak	0101100010	Add short imm. address halfword
add.w #n,Ak	0101100011	Add short imm. address word
add.h #N,Ak	000101000	Add imm. address halfword
add.w #N,Ak	000101001	Add imm. address word

Description:

The immediate field is added to the destination field with the result loaded into the destination field.

SUBTRACT ADDRESS/IMMEDIATE sub.(h|w) #(n|N),Ak

Purpose:

To Subtract an immediate field from an address register.

Format:

Short Immediate

```
---------------------------
| Opcode  |  n     |  Ak   |
---------------------------
15        6,5     3,2     0
```

Long Immediate

```
----------------------------   --------------------
|  Opcode |L| 000  |  Ak   |   |        N         |
----------------------------   --------------------
15,       7,6 5    3,2     0   (31|15)            0
```

Operation:

```
Short: Ak = Ak - #n       !#n is 0,1,...,7.
Long : Ak = Ak - #N       !#N is an Immediate.
```

PSW:

C = Carry Out
AIV = Integer Overflow

Exceptions:

Integer Overflow

Opcode:

sub.h #N,Ak	000101010	Subtract imm. address halfword
sub.w #N,Ak	000101011	Subtract imm. address word
sub.h #n,Ak	0101101010	Subtract short imm. address halfword
sub.w #n,AK	0101101011	Subtract short imm. address word

Description:

The immediate field is subtracted from the address register Ak and the result loaded into Ak.

Notes:

MULTIPLY ADDRESS/IMMEDIATE mul.(h|w) #(n|N),Ak

Purpose:
    To multiply an immediate field with an address register.

Format:

Short Immediate
```
---------------------------
| Opcode  |  n    |  Ak   |
---------------------------
15        6,5     3,2     0
```

Long Immediate
```
----------------------------   --------------------
|  Opcode |L| 000  |  Ak   |   |       N          |
----------------------------   --------------------
15,       7,6 5    3,2     0   (31|15)            0
```

Operation:
```
Short: Ak = Ak * #n     !#n is 0,1,...,7.
Long : Ak = Ak * #N     !#N is an Immediate.
```

PSW:
    AIV = Integer Overflow

Exceptions:
    Integer Overflow

Opcode:

mul.h #n,Ak	0101110010	Multiply short imm. address halfword
mul.w #n,Ak	0101110011	Multiply short imm. address word
mul.h #N,Ak	000101100	Multiply imm. address halfword
mul.w #N,Ak	000101101	Multiply imm. address word

Description:
    The address register Ak is multiplied by the immediate and the result loaded into Ak.

Notes:

DIVIDE ADDRESS/IMMEDIATE div.(h|w) #(n|N),Ak

Purpose:
To divide an address register by an immediate.

Format:
Short Immediate

```
---------------------------
| Opcode  |  n     |  Ak   |
---------------------------
15        6,5     3,2     0
```

Long Immediate

```
----------------------------   --------------------
|  Opcode |L| 000  |  Ak    |   |        N         |
----------------------------   --------------------
15,       7,6 5    3,2      0  (31|15)            0
```

Operation:

```
Short: Ak = Ak / #n      !#n is 0,1,...,7.
Long : Ak = Ak / #N      !#N is an Immediate.
```

PSW:
AIV = integer overflow
ADZ = integer divide by 0

Exceptions:
Integer Overflow
Divide by Zero

Opcode:

div.h #n,Ak	0101111010	Divide short imm. address halfword
div.w #n,Ak	0101111011	Divide short imm. address word
div.h #N,Ak	000101110	Divide imm. address halfword
div.w #N,Ak	000101111	Divide imm. address word

Description:
The address register Ak is divided by the immediate and the result loaded into the address register Ak.

Notes:
1 Integer overflow occurs if the largest negative number is divided by -1.
2 For divide by 0, the result is the original dividend.

ADD SCALAR/ADDRESS

add.w Sj,Ak

Purpose:

To add a scalar register to an address register.

Format:

```
--------------------------
|  Opcode       | Sj | Ak |
--------------------------
15             6,5  3,2  0
```

Operation:

Ak = Ak + Sj<31..0>

PSW:

C = Carry out of the most significant bit
AIV = Integer Overflow

Exceptions:

Integer Overflow

Opcode:

add.w Sj,Ak	0101000000	Add scalar to addr word

Description:

The contents of the Sj register are added to the contents of the Ak register.

Notes:

Purpose:
To AND the contents of two address registers.

Format:

```
----------------------------
| Opcode  |  Aj   |  Ak   |
----------------------------
15        6,5    3,2     0
```

Operation:
Ak = Ak .AND. Aj

PSW:

Exceptions:

Opcode:

and Aj,Ak	0101001000	AND addr. reg.

Description:
The contents of the address register Ak are ANDed with the address register Aj and the result loaded into the address register Ak. All 32 bits of the A registers participate in the operation.

Notes:

OR ADDRESS/ADDRESS

or Aj,Ak

Purpose:

To or the contents of two address registers.

Format:

```
----------------------------
| Opcode  |  Aj    |  Ak   |
----------------------------
15        6,5      3,2    0
```

Operation:

Ak = Ak .OR. Aj

PSW:

Exceptions:

Opcode:

or Aj,Ak	0101001001	OR addr. reg.

Description:

The contents of the address register Ak are ored with the address register Aj and the result loaded into the address register Ak. All 32 bits of the A registers participate in the operation.

Notes:

EXCLUSIVE OR ADDRESS/ADDRESS xor Aj,Ak

Purpose:
To exclusive OR the contents of two address registers.

Format:

```
----------------------------
| Opcode  |  Aj   |  Ak   |
----------------------------
15        6,5     3,2     0
```

Operation:
Ak = Ak .XOR. Aj

PSW:

Exceptions:

Opcode:

xor Aj,Ak	0101001010	Exclusive OR addr. reg.

Description:
The contents of the address register Ak are exclusive ORed with the address register Aj and the result loaded into the address register Ak. All 32 bits of the A registers participate in the operation.

COMPLEMENT ADDRESS/ADDRESS — not Aj,Ak

Purpose:

To COMPLEMENT an address register.

Format:

```
----------------------------
| Opcode  |  Aj   |  Ak   |
----------------------------
15        6,5    3,2      0
```

Operation:

Ak = .NOT. Aj

PSW:

Exceptions:

Opcode:

not Aj,Ak	0101001011	Complement addr. reg.

Description:

The contents of the Aj address register are complemented; this value is then loaded into the Ak address register.

Notes:

AND ADDRESS/IMMEDIATE and #N,Ak

Purpose:

To AND an immediate field to an address register.

Format:

```
----------------------------  --------------------
|  Opcode  |L| 000 |  Ak   |  |        N         |
----------------------------  --------------------
15,         6,5   3,2      0  31|16              0
```

Operation:

Ak = Ak .AND. Immediate

PSW:

Exceptions:

Opcode:

and #N,Ak	000100100	AND imm. to addr. reg.

Description:

The address register Ak is logically ANDed with the immediate operand, and the result is loaded into Ak.

Notes:

1 16-bit immediates are sign-extended to 32 bits.

OR ADDRESS/IMMEDIATE

Purpose:
To or an immediate field with an address register.

Format:

```
----------------------------  --------------------
| Opcode  |L| 000 |  Ak   |  |        N          |
----------------------------  --------------------
15,        6,5   3,2      0  31|16               0
```

Operation:
Ak = Ak .OR. immediate

PSW:

Exceptions:

Opcode:

or #N,Ak	000100101	OR imm. to addr. reg.

Description:
The address register Ak is logically ORed with the immediate operand and the result loaded into Ak.

Notes:
1 16-bit immediates are sign-extended to 32 bits.

EXCLUSIVE OR ADDRESS/IMMEDIATE xor #N,Ak

Purpose:
To exclusive OR an immediate field with an address register.

Format:

```
---------------------------   --------------------
| Opcode   |L| 000 |  Ak   |   |         N        |
---------------------------   --------------------
15,         6,5   3,2     0   31|16              0
```

Operation:
Ak = Ak .XOR. Immediate

PSW:

Exceptions:

Opcode:

xor #N,AK	000100110	Exclusive OR imm. to addr. reg.

Description:
The address register Ak is logically exclusive ORed with the immediate operand, and the result is loaded into Ak.

Notes:
1 16-bit immediates are sign extended to 32 bits.

LOGICAL SHIFT ADDRESS/ADDRESS shf Aj,Ak

Purpose:

To logically shift the contents of an address register.

Format:

```
----------------------------
| Opcode  |  Aj    |  Ak    |
----------------------------
15         6,5     3,2      0
```

Operation:

Ak = shift Ak by Aj<7..0>

PSW:

Exceptions:

Opcode:

shf Aj,Ak	0101000001	Shift an address

Description:

The contents of the address register Ak are logically shifted according to a count contained within Aj<7..0>. If the count is positive, then Ak is shifted left. If the count is negative, then Ak is shifted right. Zeros fill vacated positions. All 32 bits of Ak participate in the shift.

Bits<7..0> of Aj are examined to determine the shift count.

Notes:

A compare immediate instruction should be used to determine if the shift count in Aj is greater than 127 or less than -128.

LOGICAL SHIFT ADDRESS/IMMEDIATE shf #(n|N),Ak

Purpose:

To logically shift with an immediate field to an address register.

Format:

Short Immediate

```
---------------------------
| Opcode  |  n     |  Ak   |
---------------------------
15       6,5      3,2     0
```

Immediate

```
----------------------------   --------------------
|  Opcode |L| 000  |  Ak   |   |        N         |
----------------------------   --------------------
15,      8,7 6    3,2      0   32|16              0
```

Operation:

```
Short: Ak = Shift Ak by #n   ! #n is 0,1,...,7.
Long : Ak = Shift Ak by #N   ! #N is an Immediate.
```

PSW:

Exceptions:

Opcode:

shf #n,Ak	0100010001	Logical shift left short imm.
shf #N,AK	000100111	Logical shift imm. to addr. reg.

Description:

The address register Ak is logically shifted. The immediate field determines the direction and number of bits shifted. A positive immediate indicates a shift left. A negative immediate indicates a shift right. For shf by #n only a left shift is supported.

Notes:

shf by #n provides a convenient way to adjust an ordinal index to a byte offset. An example of this is A(I), where A is an array containing 64-bit integers, and I is contained in an address register. I must be shifted left by 3 (multiply by 8) to convert I into into a byte offset relative to the start of A.

LOAD EFFECTIVE ADDRESS

ldea <effa>,Ak

Purpose:

To load an address register with a byte pointer.

Format:

```
----------------------------    ------------------
| Opcode  |@|L| Aj   | Ak  |    | Displacement  |
----------------------------    ------------------
15         8,7,6,5   3,2   0    (31,15)           0
```

Operation:

Ak = Effective Address

PSW:

Exceptions:

Opcode:

ldea <effa>,Ak	000010010	Load effective address

Description:

The effective address, determined by evaluating the L,@,Aj fields, is loaded into Ak.

Notes:

No ring violation occurs if the developed effective address references an inner ring.

PUSH EFFECTIVE ADDRESS pshea <effa>

Purpose:
To push a byte pointer onto the stack.

Format:
```
----------------------     -------------------
|Opcode |@|L| Aj|  k |     |  Displacement   |
----------------------     -------------------
15      8,7 6,5,3,2  0    (31|15)            0
```

Operation:
```
temp  = Effective Address
A0 = A0 -4
c(A0) = temp
```

PSW:

Exceptions:

Opcode:

pshea <effa>	000011010	Push effective address

Description:
The effective address determined by evaluating the L,@,Aj fields is pushed onto the stack.

Notes:

1 No ring violation occurs if the developed effective address references an inner ring.

2 The K field of the instruction is not used.

MOVE PC/ADDRESS

Purpose:

To move the address of the next instruction into an address register.

Format:

```
-----------------------------
| Opcode              | Ak   |
-----------------------------
15                   3,2     0
```

Operation:

Ak = Current-Address + 2

PSW:

Exceptions:

Opcode:

mov PC,Ak	0111110001010	Load next PC address

Description:

The address of the instruction following this instruction is loaded into Ak.

Notes:

MOVE ADDRESS/ADDRESS mov Aj,Ak

Purpose:

To move the contents of one address register to another.

Format:

```
----------------------------
| Opcode  |  Aj   |  Ak    |
----------------------------
15        6,5     3,2      0
```

Operation:

Ak = Aj

PSW:

Exceptions:

Opcode:

mov Aj,Ak	0101000010	Move addr. reg.

Description:

The contents of Aj are moved to Ak. Aj remains unchanged.

Notes:

PUSH ADDRESS REGISTER psh.w Ak

Purpose:

To push an address register onto the stack.

Format:

```
----------------------------
| Opcode            | Ak   |
----------------------------
15                  3,2    0
```

Operation:

```
AO = AO - 4
c(AO) = Ak
```

PSW:

Exceptions:

Opcode:

```
psh.w Ak        0111110100000   Push an addr. reg.
```

Description: .

The contents of the Ak address register are pushed onto the stack

Notes:

1 When the register to be pushed is the stack pointer itself, AO, the value pushed is the value after AO is decremented by 4; in other words, the value after the instruction is executed. |

POP ADDRESS REGISTER

Purpose:

To pop a word from the stack into an address register.

Format:

```
-----------------------------
| Opcode             | Ak   |
-----------------------------
15                  3,2     0
```

Operation:

```
Ak = c(A0)
A0 = A0 + 4
```

Exceptions:

Opcode:

pop.w Ak	0111110100010	Pop word into addr. reg.

Description:

The contents of Ak are loaded from a word at the top of the stack. The stack pointer is then incremented by 4 to reference the new top of stack.

Notes:

TEST AND SET   tas <effa>

Purpose:

To indivisibly set a byte in memory.

Format:

```
----------------------     -------------------
|Opcode |@|L| Aj|  k |     |  Displacement   |
----------------------     -------------------
15      8,7 6,5,3,2  0    (31|15)           0
```

Operation:

```
IF (c(Effective Address) .EQ. 0000 0000) THEN
        C = 1
ELSE
        C = 0
c(effa) = 1111 1111
The Read and Write of memory is non-divisible.
```

Exceptions:

Opcode:

tas <effa>	000011000	Test and Set a memory byte

Description:

The referenced byte is tested for all 0. If the byte is all 0, then carry is set to 1. Otherwise carry is set to 0.

Notes:

1. The test and set byte is used to test a byte in memory indivisibly. No I/O operation is permitted between the read and write of the referenced byte.
2. The K field is unused.

LOAD PHYSICAL ldpa Aj,Ak

Purpose:

To convert a logical address to a physical address and load it into an address register.

Format:

```
----------------------------
| Opcode  |  Aj   |  Ak    |
----------------------------
15        6,5     3,2      0
```

Operation:

The logical address in Aj is converted into a physical address. The address is checked for validity in the following sequence:

ATU enabled;
Ring maximization;
Valid SDR;
Valid Level 1 PTE;
Valid Level 2 PTE;
Resident Level 2 Page Table;
Resident Data Page.

If any of the above checks fail, the carry (C) is set to 1, and A5 receives an error code. If the translation succeeds, the carry (C) is set to 0, and A5 receives the physical address. In most cases, Aj receives the address of the last Page Table Entry (PTE), and Ak receives the PTE itself. However, Aj and Ak will be set to 0 if:

Logical equals physical,
Ring maximization fails, or
The SDR is invalid.

PSW:

C = 1 if invalid reference.
C = 0 if valid reference.

Exceptions:

Opcode:

ldpa Aj,Ak	0100010000	Load a physical byte address into Ak

Description:

The contents of Aj are assumed to be a logical address, and this address is translated to its equivalent physical address. If the logical address is valid, the physical address is placed in A5 and the carry bit (C) in the PSW is set to 0. If the logical address in invalid, error information is returned in A5, and C is set to 1. In either case, the physical address of the last Page Table Entry (PTE) is placed in Aj, and the PTE itself is loaded in Ak.

The error information placed in A5 is consistent with System Exception Conditions, as described in Chapter 6. The following table shows the potential errors which can occur with their corresponding codes:

Error	Class	Qualifier	
ATU not enabled	0	none	
Ring Violation	8	1	Inward Address Reference
PTE Violation	c	4	Invalid SDR
		5	Invalid Level 1 PTE
		6	Invalid Level 2 PTE
Non-resident Page	10	0	Non-resident Level 2 Page Tabl
		1	Non-resident Data Page

Notes:

1 After the instruction has completed, the carry, C, signifies the validity of the translation. If C=1, the address was invalid, and A5 contains an error code as a result. If C=0, the address was valid and A5 contains the physical address as a result.

2 For Logical=Physical, Inward Address Reference, and Invalid SDR errors, the contents of Aj and Ak are undefined.

3 The PTE returned in Ak is useful for checking for Trojan Horse Pointers (addresses provided by the user to the system for system call processing -- see Chapter 4 of the handbook for details). Also contained in the PTE are the access privileges of the referenced address.

4 This instruction can also be used to determine whether or not a page is resident in main memory.

5 No access checks are performed (i.e. Read, Write, or Execute).

Purpose:
To move the PSW into an address register.

Format:

```
---------------------------
| Opcode             | Ak  |
---------------------------
15                  3,2    0
```

Operation:
Ak = PSW

PSW:

Exceptions:

Opcode:

mov PSW,Ak	0111110001000	Store the PSW into an addr. reg.

Description:
The PSW is loaded into the specified address register Ak. The PSW is unchanged after the loading of Ak.

Notes:
Before the PSW is moved to Ak, all existing concurrent processing is completed. This ensures that all exception condition flags accurately reflect the state of the processor.

MOVE ADDRESS/PSW

Purpose:

To move an address register into the PSW

Format:

```
----------------------------
|  Opcode           |  Ak   |
----------------------------
15                 3,2     0
```

Operation:

PSW = Ak

PSW:

Exceptions:

Opcode:

```
mov Ak,psw      0111110001001   Load an addr. reg. into the PSW
```

Description:

The contents of the specified Ak are loaded into the PSW.

Notes:

Before Ak is moved to the PSW, all existing concurrent processing is completed. This ensures that all exception flags and trap enables accurately reflect the sequential state of the processor.

COMPARE ADDRESS/ADDRESS (le|lt|eq).(h|w) Aj,Ak

Purpose:

To compare the contents of 2 address registers.

Format:

```
----------------------------
|  Opcode  |  Aj   |  Ak   |
----------------------------
15        6,5     3,2     0
```

Operation:

```
IF (Aj .OPCODE-TEST. Ak)  THEN
        C=1
ELSE
        C=0
ENDIF
```

PSW:

C is affected (see above).

Exceptions:

None

Opcode:

le.h Aj,Ak	0100110000	Compare less than or equal signed halfword
le.w Aj,AK	0100110001	Compare less than or equal signed word
lt.h Aj,Ak	0100111000	Compare less than signed halfword
lt.w Aj,Ak	0100111001	Compare less than signed word
eq.h Aj,Ak	0100011000	Compare equal halfword
eq.w Aj,Ak	0100011001	Compare equal word

Description:

The contents of Ak are compared with Aj. The result of the comparison is used to modify the C bit of the PSW. If the comparison is true, the C bit is set to 1. If the comparison is false, the C bit is reset to 0.

Notes:

1 The contents of the Ak and Aj registers are unmodified.
2 The comparison A .NE. B, A .GT. B and A .GE. B is implemented by comparing A .EQ. B, A .LT. B and A .LE. B respectively, and testing for complemented C.
3 Unsigned equal is equivalent to signed equal.

COMPARE ADDRESS/ADDRESS UNSIGNED (le|lt)u.(h|w) Aj,Ak

Purpose:

To perform unsigned comparisons between the contents of two address registers.

Format:

```
----------------------------
| Opcode  |  Aj   |  Ak   |
----------------------------
15         6,5    3,2     0
```

Operation:

```
IF (Aj .OPCODE-TEST. Ak)  THEN
        C = 1
ELSE
        C = 0
ENDIF
```

PSW:

C is affected (see above).

Exceptions:

None

Description:

The contents of Ak are compared with Aj. The result of the unsigned comparison is used to modify the C bit of the PSW. If the comparison is true, the C bit is set to 1. If the comparison is false, the C bit is reset to 0.

Opcode:

leu.h Aj,Ak	0100100000	Compare unsigned less than or equal halfwor
ltu.h gt,Aj,Ak	0100101000	Compare unsigned less than halfword
leu.w Aj,Ak	0100100001	Compare unsigned less or equal than word
ltu.w gt,Aj,Ak	0100101001	Compare unsigned less than word

Notes:

1 Ak and Aj are not modified.
2 Unsigned equality and inequality are performed by the signed comparisons for equality and inequality.

COMPARE ADDRESS/IMMEDIATE (le|lt|eq).(h|w) #(n|N),Ak

Purpose:

To compare the contents of an address register with an immediate.

Format:

Short Immediate

```
---------------------------
| Opcode  |  n     |  Ak   |
---------------------------
15        6,5     3,2     0
```

Long Immediate

```
----------------------------   --------------------
|  Opcode |L| 000  |  Ak   |   |        N         |
----------------------------   --------------------
15,       7,6 5    3,2     0   (31|15)            0
```

Operation:

```
Short:
IF ( Ak .OPCODE-TEST. #n) THEN  !#n is (0,1,...,7).
        C = 1
ELSE
        C = 0
ENDIF

Long:
IF ( Ak .OPCODE-TEST. #n) THEN  !#N is an Immediate.
        C = 1
ELSE
        C = 0
ENDIF
```

PSW:

C is affected (see above).

Exceptions:

None

Opcode:

le.h #n,Ak	0100110010	Compare less than or equal halfword
le.w #n,Ak	0100110011	Compare less than or equal word
lt.h #n,Ak	0100111010	Compare less than halfword
lt.w #n,Ak	0100111011	Compare less than word
eq.h #n,Ak	0100011010	Compare equal halfword
eq.w #n,Ak	0100011011	Compare equal word
le.h #N,Ak	000111100	Compare less than or equal halfword

le.w #N,Ak	000111101	Compare less than or equal word
lt.h #N,Ak	000111110	Compare less than halfword
lt.w #N,Ak	000111111	Compare less than word
eq.h #N,Ak	000110110	Compare equal halfword
eq.w #N,Ak	000110111	Compare equal word

Description:

The contents of the Ak register are compared to the specified immediate. If the comparison is true, the C bit of the PSW is set to 1. If the comparison is false, the C bit of the PSW is reset to 0.

Notes:

1 The contents of the Ak register are not modified.

COMPARE ADDRESS/IMMEDIATE UNSIGNED (le|lt)u.(h|w) #(n|N),Ak

Purpose:

To compare the unsigned contents of an address register with an immediate.

Format:

Short Immediate

```
---------------------------
| Opcode  |  n    |  Ak   |
---------------------------
15       6,5     3,2     0
```

Long Immediate

```
----------------------------  --------------------
|  Opcode |L| 000  |  Ak   |  |       N          |
----------------------------  --------------------
15,      7,6 5    3,2      0  (31|15)            0
```

Operation:

```
        IF ( Ak .OPCODE-TEST. #n)  THEN !#n is (0,1,...,7).
                C = 1
        ELSE
                C = 0
        ENDIF

        IF ( Ak .OPCODE-TEST. #n)  THEN !#N is an Immediate.
                C = 1
        ELSE
                C = 0
        ENDIF
```

PSW:

C is affected (see above).

Exceptions:

None

Opcode:

leu.h #n,Ak	0100100010	Compare unsigned less than or equal ha
ltu.h #n,Ak	0100101010	Compare unsigned less than halfword
leu.h #N,Ak	000111000	Compare unsigned less than halfword
ltu.h #N,Ak	000111010	Compare unsigned less than halfword
leu.w #n,Ak	0100100011	Compare unsigned less than or equal wo
ltu.w #n,Ak	0100101011	Compare unsigned less than word
leu.w #N,Ak	000111001	Compare unsigned less than word
ltu.w #N,Ak	000111011	Compare unsigned less than word

Description:

The contents of the Ak register are compared to the specified immediate. If the comparison is true, the C bit of the PSW is set to 1. If the comparison is false, the C bit of the PSW is reset to 0.

Notes:

1 The contents of the Ak register are not modified.
2 Compare equal immediate N is performed using the eq.x #N,Ak instructions.

CONVERT INTEGER ADDRESS/ADDRESS cvt Aj,Ak

Purpose:
To convert the integer contents of one A register to an integer of different precision.

Format:

```
----------------------------
| Opcode  |  Aj   |  Ak   |
----------------------------
15        6,5    3,2     0
```

Operation:
Ak = Convert (Aj)   ! according to the opcode

PSW:
AIV = Integer Overflow

Exceptions:
Integer Overflow

Opcode:

cvtw.b Aj,Ak	0100000000	Convert word to byte
cvtw.h Aj,Ak	0100000001	Convert word to halfword
cvtb.w Aj,Ak	0100000010	Convert byte to word
cvth.w Aj,AK	0100000011	Convert half to word

Description:
The specified integer is converted to specified destination integer and the result is loaded into Ak. An overflow exception can occur for the word to byte and word to halfword instructions. Bytes and halfwords are sign extended to words.

Notes:

1 The other possible signed conversions are implemented by executing two of the provided conversions. Halfword to byte is performed by cvth.w and cvtw.b. Byte to half-word is performed by cvtb.w and cvtw.h.

2 Unsigned conversions may be implemented using logical AND instructions.

CHAPTER 10

10 Scalar Register Instruction Set

This chapter describes the instructions which primarily manipulate the scalar register set. The instruction categories include Scalar Loads and Stores, Scalar Load Immediates, Scalar/Scalar Arithmetics, Scalar/Scalar Logical Operations, Arithmetic Immediates, Logical Immediates, Push/Pop Scalar Registers, Scalar/Scalar Compares Signed/Unsigned, S Register Compare Immediates, S Register Conversions, and Shifts/Moves/Counts. There are 8 x 64 bit scalar registers. The scalar registers are used to perform computation on fixed and floating point operands. Instructions which affect both scalar and vector registers are described in the chapter on the Vector/ Scalar Instruction Set. Details are also provided on the scheduling of these instructions.

10.1 Scalar Loads and Stores

These instructions include Load Scalar Register and Store Scalar Register; both instruction types work with byte, halfword, word and longword operands and do not affect flags in the Processor Status Word. The subgroup, Scalar Load Immediates, includes Load Scalar/Immediate as the only instruction, and its data types include word and longword. Load Scalar/Immediate does not affect the flags in the Processor Status.

10.2 Scalar/Scalar Arithmetic

Instructions included in this group include Add Scalar/Scalar, Subtract Scalar/Scalar, Multiply Scalar/Scalar, Divide Scalar/Scalar, and Negate Scalar/Scalar. All Scalar/Scalar Arithmetics are either byte, halfword, word, or longword integers, or single or double precision floating point. Integer Overflow is signified by the SIV bit in the PSW, Exponent Overflow by the OV bit, Exponent Underflow by the UN bit, and Carry output by the SC bit. RO is used to signify a reserved operand. The SDZ bit shows a Divide by Zero. An integer FDZ indicates a floating point divide by zero.

10.3 Scalar/Scalar Logical Operations

These instructions include AND Scalar/Scalar, OR Scalar/Scalar, Exclusive OR Scalar/Scalar, and Complement Scalar/Scalar. No flags are affected in the PSW.

10.4 S Register Immediates

These instructions provide a means to perform an operation between an S register and an immediate. The section has two sub groups: Arithmetic Immediates and Logical Immediates. Arithmetic Immediates include Add, Subtract, Multiply, and Divide Scalar/Immediate instructions which work with halfword, word, or Single Precision Floating Point operands and affect the following flags in the PSW:

SIV	=	Integer Overflow; Integer only
OV	=	Exponent Overflow; Floating Point only
UN	=	Exponent Underflow; Floating Point only
SC	=	Carry Output; Integer only
RO	=	Reserved Operand; Floating Point only
SDZ	=	Integer Divide by Zero
FDZ	=	Floating Point Divide by Zero

Logical Immediates include AND, OR, Exclusive OR, and Logical Shift Scalar/Immediate instructions, which affect no flags in the PSW.

10.5 Push/Pop Scalar Registers

These instructions--Push Scalar Register and Pop Scalar Register--operate on word and longword and do not affect the flags in the PSW.

10.6 Scalar/Scalar Compares Signed/Unsigned

This section includes Compare Scalar/Scalar and Compare Scalar/Scalar Unsigned, and the following S Register Compare Immediates: Compare Scalar/Immediate and Compare Scalar/Immediate Unsigned. Compare Scalar/Scalar works with byte, halfword, word, longword, single and double precision floating point and affects only the SC bit in the PSW. The Compare Scalar/Scalar Unsigned instruction can be byte, word, halfword, or longword and likewise affects only the SC bit in the PSW. S Register Compare Immediate instructions provide immediate to S register operations. The immediates provided are either 16 or 32 bit and affects only the SC bit.

10.7 S Register Conversions

This instruction provides conversions from one S register to another. Integer Overflow is signified by the SIV bit, and Floating Point Overflow by the OV bit in the PSW.

10.8 Shifts/Moves/Counts

These instructions include Logical Shift Scalar/Scalar, Trailing Zero Count, Population Count Scalar, Move Scalar to Address, Move Address/Scalar, and Move Scalar/Scalar. None of these instructions affects the flags in the PSW.

LOAD SCALAR REGISTER ld.(b|h|w|l|s|d) <effa>,Sk

Purpose:

To load an operand into a scalar accumulator.

Format:

```
----------------------     -------------------
| Opcode |@|L|Aj |Sk |     |  Displacement   |
----------------------     -------------------
15      8,7 6,5,3,2 0      (31|15)           0
```

Operation:

Sk = c(Effective Address)

PSW:

Exceptions:

Opcode:

ld.b <effa>,Sk	001100000	Load scalar byte
ld.h <effa>,Sk	001100010	Load scalar halfword
ld.w <effa>,Sk	001100100	Load scalar word
ld.l <effa>,Sk	001100110	Load scalar longword
ld.s <effa>,Sk	001100100	Load scalar single float
ld.d <effa>,Sk	001100110	Load scalar double float

Description:

The referenced data is loaded into the specified scalar register.

Notes:

1 Single precision floating point and 32 bit integer occupy the same bit positions (<31..0>) within a scalar register.

2 Byte data occupies bit positions <7..0> within a scalar register.

3 Halfword data occupies bit positions <15..0> within a scalar register.

4 The .s and .w forms of this instruction are equivalent, as are the .d and .l forms. The .s and .d forms are added for convenience.

STORE SCALAR REGISTER st.(b|h|w|l|s|d) Sk,<effa>

Purpose:

To store an operand from a scalar register.

Format:

```
----------------------     -------------------
| Opcode |@|L|Aj |Sk |     |  Displacement   |
----------------------     -------------------
15      8,7 6,5,3,2 0      (31|15)           0
```

Operation:

c(Effective Address) = Sk

PSW:

Exceptions:

Opcode:

st.b Sk,<effa>	001101000	Store scalar byte
st.h Sk,<effa>	001101010	Store scalar halfword
st.w Sk,<effa>	001101100	Store scalar word
st.l Sk,<effa>	001101110	Store scalar longword
st.s Sk,<effa>	001101100	Store scalar single float
st.d Sk,<effa>	001101110	Store scalar double float

Description:

The referenced data is stored from the specified scalar register Sk into the referenced memory location.

Notes:

1 Single precision floating pointer and 32-bit integer occupy the same bit positions within a scalar register.

2 The .s and .w forms of this instruction are equivalent, as are the .d and .l forms. The .s and .d forms are added for convenience.

LOAD SCALAR/IMMEDIATE ld.(w|l|d|u|du|dl|lu|ll) #N,Sk

Purpose:

To load a word immediate into a scalar register.

Format:

```
------------------------     ---------------
| Opcode |L| 001 | Sk |      |     N        |
------------------------     ---------------
15       7 6 5   3,2  0     31|16          0
```

Operation:

```
Sk<63..32> = Immediate           ! ld.d,#N,Sk
Sk<31..0> = 0

Sk<63..32> = Immediate<31>       ! ld.l,#N,Sk
Sk<31..0> = Immediate
                                 ! ld.(w|dl|ll),#N,Sk
Sk<63..32> = Sk<63..32> !(unchanged)
Sk<31..0> = Immediate
                                 ! ld.(u|du|lu) #N,Sk

Sk<63..32> = Immediate
Sk<31..0> = Sk<31..0>   !(unchanged)
```

PSW:

Exceptions:

Opcode:

ld.d #N,Sk	000100000	Load immediate, most significant bits
ld.l #N,Sk	000100010	Load 64 bit sign extended immediate
ld.w #N,Sk	000100011	Load a 32 bit immediate
ld.u #N,Sk	000100001	Load immediate, upper half
ld.du #N,Sk	000100001	Load 64 bit floating immed., upper hal
ld.dl #N,Sk	000100011	Load 64 bit floating immed., lower hal
ld.lu #N,Sk	000100001	Load 64 bit integer immed., upper half
ld.ll #N,Sk	000100011	Load 64 bit integer immed., lower half

Description:

The immediate is loaded into Sk.

Notes:

1. A full 64-bit immediate can be developed by first performing a ld.d,#N,Sk and then by executing a ld.w,#N,Sk.

1. The forms using .du, .lu, .dl, and .ll allow convenient use of 64 bit immediate operators, and load either the upper half (.du and .lu) or the lower half (.dl and .ll), respectively. For example, "ld.du #3.14159,s0" causes the upper half of the 64 bit floating point constant PI to be loaded into the upper half of register s0. These four forms are necessary since immediate operands are only 32 bits in length.

ADD SCALAR/SCALAR add.(b|h|w|l|s|d) Sj,Sk

Purpose:

To add a scalar to a scalar.

Format:

```
--------------------------
| Opcode         | Sj | Sk |
--------------------------
15              6,5  3,2  0
```

Operation:

Sk = Sk + Sj

PSW:

```
SIV = Integer Overflow! Integer Only
OV = Exponent Overflow! Floating Point only
UN = Exponent Underflow! Floating point only
SC = Carry Output! Integer Only
RO = Reserved Operand! Floating point only
```

Exceptions:

Integer Overflow
Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

add.b Sj,Sk	0101100100	Add scalar/scalar integer byte
add.h Sj,Sk	0101100101	Add scalar/scalar integer halfword
add.w Sj,Sk	0101100110	Add scalar/scalar integer word
add.l Sj,Sk	0101100111	Add scalar/scalar integer longword
add.s Sj,Sk	0101010100	Add scalar/scalar single float
add.d Sj,Sk	0101010101	Add scalar/scalar double float

Description:

The contents of the scalar register Sk are added to the contents of the scalar register Sj, and the scalar result is loaded into Sk.

Notes:

SUBTRACT SCALAR/SCALAR   sub.(b|h|w|l|s|d) Sj,Sk

Purpose:

To subtract a scalar from a scalar.

Format:

```
---------------------------
| Opcode        | Sj | Sk |
---------------------------
15             6,5  3,2  0
```

Operation:

Sk = Sk - Sj

PSW:

SIV = Integer Overflow; Integer Only
OV = Exponent Overflow; Floating Point Only
UN = Exponent Underflow; Floating Point Only
SC = Carry output
RO = Reserved Operand Only; Floating Point Only

Exceptions:

Integer Overflow
Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

sub.b Sj,Sk	0101101100	Subtract scalar/scalar integer byte
sub.h Sj,Sk	0101101101	Subtract scalar/scalar integer halfword
sub.w Sj,Sk	0101101110	Subtract scalar/scalar integer word
sub.l Sj,Sk	0101101111	Subtract scalar/scalar integer longword
sub.s Sj,Sk	0101010110	Subtract scalar/scalar single float
sub.d Sj,Sk	0101010111	Subtract scalar/scalar double float

Description:

The contents of the scalar register Sj are subtracted from the contents of the scalar register Sk, and the scalar result is loaded into Sk.

Notes:

MULTIPLY SCALAR/SCALAR mul.(b|h|w|l|s|d) Sj,Sk

Purpose:

To multiply a scalar by a scalar.

Format:

```
---------------------------
|  Opcode         | Sj  | Sk |
---------------------------
15              6,5  3,2  0
```

Operation:

Sk = Sk * Sj

PSW:

SIV = Integer Overflow; Integer Only
OV = Exponent Overflow; Floating Point Only
UN = Exponent Underflow; Floating Point Only
RO = Reserved Operand; Floating Point Only

Exceptions:

Integer Overflow
Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

mul.b Sj,Sk	0101110100	Multiply scalar/scalar integer byte
mul.h Sj,Sk	0101110101	Multiply scalar/scalar integer halfwor
mul.w Sj,Sk	0101110110	Multiply scalar/scalar integer word
mul.l Sj,Sk	0101110111	Multiply scalar/scalar integer longwor
mul.s Sj,Sk	0101011100	Multiply scalar/scalar single float
mul.d Sj,Sk	0101011101	Multiply scalar/scalar double float

Description:

The contents of the scalar register Sk are multiplied with the contents of the scalar register Sj, and the scalar result is loaded into Sk.

Notes:

DIVIDE SCALAR/SCALAR div.(b|h|w|l|s|d) Sj,Sk

Purpose:

To divide a scalar by a scalar.

Format:

```
---------------------------
| Opcode        | Sj | Sk |
---------------------------
15             6,5  3,2  0
```

Operation:

Sk = Sk / Sj

PSW:

SIV = Integer Overflow! Integer only
OV = Exponent Overflow! Floating Point only
UN = Exponent Underflow! Floating Point only
SDZ = Divide by zero! Integer only
RO = Reserved Operand! Floating Point only
FDZ= Divide by zero! Floating Point only

Exceptions:

Integer Overflow
Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

div.b Sj,Sk	0101111100	Divide scalar/scalar integer byte
div.h Sj,Sk	0101111101	Divide scalar/scalar integer halfword
div.w Sj,Sk	0101111110	Divide scalar/scalar integer word
div.l Sj,Sk	0101111111	Divide scalar/scalar integer longword
div.s Sj,Sk	0101011110	Divide scalar/scalar single float
div.d Sj,Sk	0101011111	Divide scalar/scalar double float

Description:

The contents of a scalar Sk are divided by a scalar Sj, and the result is loaded into Sk.

Notes:

NEGATE SCALAR/SCALAR — neg.(b|h|w|l|s|d) Sj,Sk

Purpose:

To negate a scalar.

Format:

```
---------------------------
|  Opcode        | Sj | Sk |
---------------------------
15              6,5  3,2  0
```

Operation:

Sk = 0 - Sj

PSW:

SIV = Integer Overflow; Integer only
SC = Integer Carry Out.
OV = Exponent Overflow; Floating Point only
UN = Exponent Underflow; Floating point only
RO = Reserved Operand; Floating Point Only

Exceptions:

Integer Overflow
Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

neg.b Sj,Sk	0110111100	Negate scalar/scalar integer byte
neg.h Sj,Sk	0110111101	Negate scalar/scalar integer halfword
neg.w Sj,Sk	0110111110	Negate scalar/scalar integer word
neg.l Sj,Sk	0110111111	Negate scalar/scalar integer longword
neg.s Sj,Sk	0110010110	Negate scalar/scalar single float
neg.d Sj,Sk	0110010111	Negate scalar/scalar double float

Description:

The algebraic negation of Sj is loaded into Sk. The result is identical to subtracting Sj from 0.

Notes:

AND SCALAR/SCALAR

Purpose:
To AND the contents of two scalars.

Format:

```
---------------------------
| Opcode        | Sj | Sk |
---------------------------
15             6,5  3,2  0
```

Operation:
Sk = Sk .AND. Sj

PSW:

Exceptions:

Opcode:

and Sj,Sk	0101001100	AND scalar/scalar

Description:
The contents of the Sk register are ANDed with the contents of the Sj register. The results of the AND are loaded into Sk. All 64-bits of each scalar register participate in the operation.

Notes:

OR SCALAR/SCALAR or Sj,Sk

Purpose:
To OR the contents of two scalars.

Format:

```
---------------------------
|  Opcode        | Sj | Sk |
---------------------------
15              6,5  3,2  0
```

Operation:
Sk = Sk .OR. Sj

PSW:

Exceptions:

Opcode:

or Sj,Sk	0101001101	OR scalar/scalar

Description:
The contents of the Sj scalar register are ORed with the contents of the Sk register. The results of the OR are loaded into Sk. All 64-bits of the scalar registers participate in the operation.

Notes:

EXCLUSIVE OR SCALAR/SCALAR xor Sj,Sk

Purpose:
To Exclusive OR the contents of two scalar registers.

Format:

```
--------------------------
| Opcode        | Sj | Sk |
--------------------------
15             6,5  3,2  0
```

Operation:
Sk = Sk .XOR. Sj

PSW:

Exceptions:

Opcode:

xor Sj,Sk	0101001110	Exclusive OR scalar/scalar

Description:
The contents of the Sj vector register are exclusive ORed with the contents of the Sk register. The results of the exclusive or are loaded into Sk. All 64-bits of the scalar registers participate in the operation.

Notes:

COMPLEMENT SCALAR/SCALAR

Purpose:
To COMPLEMENT the contents of a scalar.

Format:

```
---------------------------
| Opcode        | Sj | Sk |
---------------------------
15            6,5  3,2  0
```

Operation:
Sk = .NOT. Sj

PSW:

Exceptions:

Opcode:

not Sj,Sk	0101001111	Complement scalar/scalar

Description:
The contents of the Sj scalar register are complemented and the results loaded into the Sk scalar register. All 64-bits of the scalar registers participate in the operation.

Notes:

ADD SCALAR/IMMEDIATE add.(h|w|s) #N,Sk

Purpose:

To add an immediate to a scalar.

Format:

```
-----------------------     ---------------
| Opcode |L| 001 | Sk |     |     N       |
-----------------------     ---------------
15       7 6 5   3,2  0    31|16          0
```

Operation:

Sk = Sk + Immediate

PSW:

```
SIV = Integer Overflow   ! Integer Only
OV = Exponent Overflow   ! Floating Point only
UN = Exponent Underflow  ! Floating point only
SC = Carry Output        ! Integer Only
RO = Reserved Operand    ! Floating point only
```

Exceptions:

Integer Overflow
Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

add.h #N,Sk	000101000	Add scalar/immed. integer halfword
add.w #N,Sk	000101001	Add scalar/immed. integer word
add.s #N,Sk	000110000	Add scalar/immed. single float

Description:

The contents of the scalar register Sk are added to the contents of immediate, and the scalar result is loaded into Sk.

SUBTRACT SCALAR/IMMEDIATE sub.(h|w|s) #N,Sk

Purpose:

To subtract an immediate from a scalar.

Format:

```
------------------------     ---------------
|  Opcode |L| 001 | Sk |     |      N      |
------------------------     ---------------
15        7 6 5   3,2  0     31|16         0
```

Operation:

Sk = Sk - Immediate

PSW:

SIV = Integer Overflow; Integer Only
OV = Exponent Overflow; Floating Point Only
UN = Exponent Underflow; Floating Point Only
SC = Carry output
RO = Reserved Operand Only; Floating Point Only

Exceptions:

Integer Overflow
Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

sub.h #N,Sk	000101010	Subtract scalar/immed. integer halfwor
sub.w #N,Sk	000101011	Subtract scalar/immed. integer word
sub.s #N,Sk	000110001	Subtract scalar/immed. single float

Description:

The contents of the immediate are subtracted from the contents of the scalar register Sk, and the scalar result is loaded into Sk.

Notes:

MULTIPLY SCALAR/IMMEDIATE mul.(h|w|s) #N,Sk

Purpose:
To multiply a scalar by a scalar.

Format:

```
------------------------     ---------------
| Opcode |L| 001 | Sk |      |     N       |
------------------------     ---------------
15       7 6 5   3,2  0     31|16          0
```

Operation:
Sk = Sk * Sj

PSW:
SIV = Integer Overflow; Integer Only
OV = Exponent Overflow; Floating Point Only
UN = Exponent Underflow; Floating Point Only
RO = Reserved Operand; Floating Point Only

Exceptions:
Integer Overflow
Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

mul.h #N,Sk	000101100	Multiply scalar/immed. integer halfword
mul.w #N,Sk	000101101	Multiply scalar/immed. integer word
mul.s #N,Sk	000110010	Multiply scalar/immed. single float

Description:
The contents of the scalar register Sk are multiplied with the contents of the scalar register Sj, and the scalar result is loaded into Sk.

Notes:

DIVIDE SCALAR/IMMEDIATE div.(h|w|s) #N,Sk

Purpose:

To divide a scalar by a scalar.

Format:

```
 ------------------------      ---------------
 |  Opcode |L| 001 | Sk |      |      N      |
 ------------------------      ---------------
 15       7 6 5   3,2  0     31|16          0
```

Operation:

Sk = Sk / Immediate

PSW:

SIV	= Integer Overflow; Integer only
OV	= Exponent Overflow; Floating Point only
UN	= Exponent Underflow; Floating Point only
SDZ	= Divide by zero; Integer only
RO	= Reserved Operand; Floating Point only
FDZ	= Divide by zero; Floating Point only

Exceptions:

Integer Overflow
Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

div.h #N,Sk	000101110	Divide scalar/scalar integer halfword
div.w #N,Sk	000101111	Divide scalar/scalar integer word
div.s #N,Sk	000110011	Divide scalar/scalar single float

Description:

The contents of a scalar Sk are divided by an immediate, and the result is loaded into Sk.

Notes:

AND SCALAR/IMMEDIATE and #N,Sk

Purpose:
    To AND the contents of a scalar and an immediate

Format:

```
-------------------------     ---------------
|  Opcode  |L| 001  | Sk |     |      N      |
-------------------------     ---------------
15        7 6 5   3,2  0     31|16          0
```

Operation:
    Sk<31..0> = Sk<31..0> .AND. Immediate

PSW:

Exceptions:

Opcode:

and #N,Sk	000100100	AND scalar/immediate

Description:
    The contents of the Sk vector register are ANDed with the immediate. The results of the AND are loaded into Sk. The least significant 32 bits of Sk participate in the operation.

Notes:
    1 16-bit immediates are sign-extended to 32 bits.

OR SCALAR/IMMEDIATE

Purpose:

To OR the contents of a scalar and an immediate

Format:

```
------------------------     ---------------
| Opcode |L| 001 | Sk |      |     N       |
------------------------     ---------------
15       7 6 5  3,2  0      31|16          0
```

Operation:

Sk<31..0> = Sk<31..0> .OR. Immediate

PSW:

Exceptions:

Opcode:

or #N,Sk	000100101	OR scalar/immediate

Description:

The immediate is ORed with contents of the Sk register. The results of the OR are loaded into Sk. The least significant 32 bits of Sk participate in the operation.

Notes:

1 16-bit immediates are sign-extended to 32 bits.

EXCLUSIVE OR SCALAR/IMMEDIATE xor #N,Sk

Purpose:
To Exclusive OR the contents of an immediate with a scalar

Format:

```
------------------------     ---------------
| Opcode |L| 001 | Sk |     |      N       |
------------------------     ---------------
15       7 6 5   3,2  0    31|16          0
```

Operation:
Sk<31..O> = Sk<31..O> .XOR. Immediate

PSW:

Exceptions:

Opcode:

xor #N,Sk	000100110	Exclusive OR scalar/immediate

Description:
The contents of an immediate are exclusive ORed with the contents of the Sk register. The results of the exclusive or are loaded into Sk. The least significant 32 bits of Sk participate in the operation.

Notes:
1 16-bit immediates are sign-extended to 32 bits.

LOGICAL SHIFT SCALAR/IMMEDIATE shf #N,Sk

Purpose:

To logically shift the contents of a scalar as controlled by an immediate

Format:

```
------------------------     ---------------
| Opcode |L| 001 | Sk |      |      N      |
------------------------     ---------------
15      7 6 5  3,2  0      31|16          0
```

Operation:

Sk<63..0> = Shift Sk by immediate<7..0>

PSW:

Exceptions:

Opcode:

shf #N,Sk	000100111	Shift Scalar/immediate

Description:

The contents of the Sk scalar register are shifted according to the value specified in the immediate. Only bits<7..0> are used to control the shift. The shift count is interpreted as an 8-bit two's complement number. Thus, if bit<7> is a 0, the logical shift is left. If bit<7> is a 1, the logical shift is to the right. Vacated positions are zero filled.

PUSH SCALAR REGISTER psh.(w|l) Sk

Purpose:

To push an Sk register onto the stack.

Format:

```
--------------------------
|  Opcode           | Sk |
--------------------------
15                 3,2  0
```

Operation:

```
IF (PSH.W-OPCODE)  THEN
        AO = AO - 4
ELSE
        AO = AO - 8     ! psh.l
ENDIF
c(AO) = Sk<63..O>       ! if long
c(AO) = Sk<31..O>       ! if word
```

PSW:

Exceptions:

Opcode:

psh.w Sk	0111110100100	Push Sk<31..0> onto the stack
psh.l Sk	0111110100101	Push Sk<63..0> onto the stack.

Description:

The scalar register, Sk, is pushed onto the stack. Either all 64 bits of Sk or the least significant 32 bits of Sk are pushed.

Notes:

POP SCALAR REGISTER

pop.(w|l) Sk

Purpose:

To pop an Sk register from the stack.

Format:

```
--------------------------
|  Opcode            | Sk |
--------------------------
15                  3,2  0
```

Operation:

```
IF (POP.W-OPCODE) THEN
        Sk<31..0> = c(A0)
        A0 = A0 - 4
ELSE
        Sk = c(A0)
        A0 = A0 - 8 ! pop.l
ENDIF
```

PSW:

Exceptions:

Opcode:

pop.w Sk	0111110100110	Pop Sk<31..0> from the stack
pop.l Sk	0111110100111	Pop Sk<63..0> from the stack.

Description:

The scalar register, Sk, is popped from the stack. Either all 64 bits or the least significant 32 bits of Sk are loaded from the stack.

Notes:

COMPARE SCALAR/SCALAR (le|lt|eq).(b|h|w|l|s|d) Sj,Sk

Purpose:
To sign compare a scalar and scalar and load SC

Format:

```
---------------------------
|  Opcode        | Sj  | Sk |
---------------------------
15              6,5  3,2  0
```

Operation:

```
IF (Sj .OPCODE-TEST. Sk) THEN
        SC = 1
ELSE
        SC = 0
ENDIF
```

PSW:
SC is affected (see above).

Exceptions:
Reserved Operand (floating point only).

Opcode:

le.b Sj,Sk	0100110100	Compare less than or equal byte
lt.b Sj,Sk	0100111100	Compare less than byte
eq.b Sj,Sk	0100011100	Compare equal byte
le.h Sj,Sk	0100110101	Compare less than or equal halfword
lt.h Sj,Sk	0100111101	Compare less than halfword
eq.h Sj,Sk	0100011101	Compare equal halfword
le.w Sj,Sk	0100110110	Compare less than or equal word
lt.w Sj,Sk	0100111110	Compare less than word
eq.w Sj,Sk	0100011110	Compare equal word
le.l Sj,Sk	0100110111	Compare less than or equal longword
lt.l Sj,Sk	0100111111	Compare less than longword
eq.l Sj,Sk	0100011111	Compare equal longword
le.s Sj,Sk	0101010000	Compare less than or equal single float
lt.s Sj,Sk	0101010010	Compare less than single float
eq.s Sj,Sk	0101011000	Compare equal single float
le.d Sj,Sk	0101010001	Compare less than or equal double float
lt.d Sj,Sk	0101010011	Compare less than double float
eq.d Sj,Sk	0101011001	Compare equal double float

Description:

The scalar registers Sk and Sj are signed compared. The result is loaded into SC. If the comparison is .TRUE., then SC is set to 1. If the comparison is .FALSE., then SC is reset to 0.

Notes:

COMPARE SCALAR/SCALAR UNSIGNED (le|lt)u.(b|h|w|l) Sj,Sk

Purpose:

To unsigned compare a scalar and scalar and load SC

Format:

```
---------------------------
|  Opcode        | Sj | Sk |
---------------------------
15             6,5  3,2  0
```

Operation:

```
IF (Sj .OPCODE-TEST. Sk)  THEN
        C = 1
ELSE
        C = 0
ENDIF
```

PSW:

SC is affected (see above).

Exceptions:

Opcode:

leu.b Sj,Sk	0100100100	Compare less than or equal byte
ltu.b Sj,Sk	0100101100	Compare less than byte
leu.h Sj,Sk	0100100101	Compare less than or equal halfword
ltu.h Sj,Sk	0100101101	Compare less than halfword
leu.w Sj,Sk	0100100110	Compare less than or equal word
ltu.w Sj,Sk	0100101110	Compare less than word
leu.l Sj,Sk	0100100111	Compare less than or equal longword
ltu.l Sj,Sk	0100101111	Compare less than longword

Description:

The scalar registers Sk and Sj are unsigned compared. The result is loaded into SC. If the comparison is .TRUE., then SC is set to 1. If the comparison is .FALSE., then SC is reset to 0.

Notes:

1 Unsigned compares for equality are performed using the signed compares.

COMPARE SCALAR/IMMEDIATE (le|lt|eq).(h|w|s) #N,Sk

Purpose:

To signed compare a scalar and an immediate and load SC

Format:

```
-------------------------     ---------------
| Opcode |L| 001 | Sk |       |      N       |
-------------------------     ---------------
15       7 6 5  3,2  0      31|16           0
```

Operation:

```
IF (Ak .OPCODE-TEST. Immediate) THEN
        SC=1
ELSE
        SC=0
ENDIF
```

PSW:

SC is affected (see above).

Exceptions:

Reserved Operand (floating point only).

Opcode:

le.h #N,Sk	000111100	Compare less than or equal halfword
lt.h #N,Sk	000111110	Compare less than halfword
eq.h #N,Sk	000110110	Compare equal halfword
le.w #N,Sk	000111101	Compare less than or equal word
lt.w #N,Sk	000111111	Compare less than word
eq.w #N,Sk	000110111	Compare equal word
le.s #N,Sk	000110100	Compare less than or equal single
lt.s #N,Sk	000110101	Compare less than single

Description:

The scalar register Sk and the immediate are signed compared. The result is loaded into SC. If the comparison is .TRUE., then SC is set to 1. If the comparison is .FALSE., then SC is reset to 0.

Notes:

The signed compare for equality for single precision float is NOT provided. Reserved operand detection for equality can be achieved by performing a compare word immediate with the immediate being a reserved operand. Then a compare word instruction can be used.

COMPARE SCALAR/IMMEDIATE UNSIGNED (le|lt)u.(h|w) #N,Sk

Purpose:

To unsigned compare a scalar and an immediate.

Format:

```
------------------------     ---------------
| Opcode |L| 001 | Sk |      |      N      |
------------------------     ---------------
15       7 6 5  3,2  0     31|16          0
```

Operation:

```
IF ( Immediate .OPCODE-TEST. Sk) THEN
        C = 1
ELSE
        C = 0
ENDIF
```

PSW:

SC is affected (see above).

Exceptions:

Opcode:

leu.h #N,Sk	000111000	Compare unsigned less than or equal halfword
ltu.h #N,Sk	000111010	Compare unsigned less than halfword
leu.w #N,Sk	000111001	Compare unsigned less than or equal word
ltu.w #N,Sk	000111011	Compare unsigned less than word

Description:

The scalar register Sk and an immediate are unsigned compared. The result is loaded into SC. If the comparison is .TRUE., then SC is set to 1. If the comparison is .FALSE., then SC is reset to 0.

Notes:

1 Compare equal unsigned is performed using the compare signed instruction.

CONVERT SCALAR/SCALAR — cvt Sj,Sk

Purpose:

To convert the integer or floating point contents of one S register to an integer or floating point value of different precision.

Format:

```
---------------------------
|  Opcode        | Sj | Sk |
---------------------------
15              6,5  3,2  0
```

Operation:

```
Sk = Convert (Sj)        ! according to the opcode
```

PSW:

SIV = Integer Overflow
OV = Exponent Overflow
RO = Reserved Operand

Exceptions:

Integer Overflow
Exponent Overflow

Opcode:

cvtw.s Sj,Sk	0100001000	Convert word to single float
cvts.w Sj,Sk	0100001001	Convert single float to word
cvtd.s Sj,Sk	0100001010	Convert double float to single float
cvts.d Sj,Sk	0100001011	Convert single float to double float
cvtw.b Sj,Sk	0100000100	Convert word to byte
cvtw.h Sj,Sk	0100000101	Convert word to halfword
cvtb.w Sj,Sk	0100000110	Convert byte to word
cvth.w Sj,Sk	0100000111	Convert halfword to word
cvts.l Sj,Sk	0100001100	Convert single float to longword
cvtd.l Sj,Sk	0100001101	Convert double float to longword
cvtl.s Sj,Sk	0100001110	Convert longword to single float
cvtl.d Sj,Sk	0100001111	Convert longword to double float
cvtl.w Sj,Sk	0100010100	Convert longword to word
cvtw.l Sj,Sk	0100010101	Convert word to longword

Description:

The specified integer or floating point operand is converted to specified destination data type and the result is loaded into Sk. An overflow exception can occur for the word to byte and word to halfword instructions. Bytes and halfwords are sign extended to words. Conversions from float to fix use truncation (rounding towards 0) as the rounding algorithm.

Notes:

1 The other possible signed conversions are implemented by executing two of the provided conversions. Halfword to byte is performed by cv.hw and cv.wb. Byte to halfword is performed by cv.bw and cv.wh.

2 Unsigned conversions may be implemented using logical AND instructions.

3 Only the specified precision of the destination operand is modified. All other bits are unchanged.

4 If an input operand is a floating point reserved operand, then the destination is unchanged, and the RO and SIV flags are both set to 1.

5 Truncation from float to fix follows the FORTRAN standard. Thus, -5.9 is truncated to -5, and 5.9 is truncated to 5.

6 Conversion from integer to floating point types is performed thus:

a. The fixed point number is normalized.

b. The most significant 24 bits (for single) or the most significant 53 bits (for double) of the normalized fixed point number become the fraction of the result. The lower order bits of the normalized fixed point number are truncated.

LOGICAL SHIFT SCALAR/SCALAR — shf Sj,Sk

Purpose:

To logically shift a scalar.

Format:

```
--------------------------
| Opcode       | Sj | Sk |
--------------------------
15             6,5  3,2  0
```

Operation:

Sk = Shift Sk by Sj<7..0>

PSW:

Exceptions:

Opcode:

shf Sj,Sk	0101000101	Shift a scalar

Description:

The contents of Sk are shifted according to the contents of Sj. When Sj is positive, Sk is shifted left. When Sj is negative, Sk is shifted right. All 64 bits of Sk participate in the shift. Vacated positions are zero filled. Only Sj<7..0> are used to control the shift. Sj<63..8> are ignored.

Notes:

Arithmetic shifts are implemented using multiplies and divides.

TRAILING ZERO COUNT tzc Sj,Sk

Purpose:
To determine the number of trailing zeros in a scalar register.

Format:

```
---------------------------
|  Opcode       | Sj | Sk |
---------------------------
15              6,5  3,2  0
```

Operation:
Sk = Trailing-zero-count(Sj)

Exceptions:

Opcode:

tzc Sj,Sk	0100010111	Count of trailing zeroes in Sj

Description:
The number of trailing zeros in Sj counting from bit 0 through bit 63 are loaded into bits<6..0> of Sk. The tzc instruction searches for the first binary 1 from right to left, the same way bits are numbered. If Sj is all 0, the number 64 is loaded into Sk<63..0>. Otherwise, if bit n is set, then the binary value of n is loaded into Sk<63..0>.

POPULATION COUNT SCALAR

plc.t Sj,Sk

Purpose:

To determine the number of 1's in a scalar register.

Format:

```
---------------------------
|  Opcode        | Sj  | Sk |
---------------------------
15              6,5  3,2  0
```

Operation:

```
Sk = 0

DO 10 a = 0, 63
        IF (Sj<a> .EQ.1)  THEN
                Sk =Sk + 1
        ENDIF
10 CONTINUE
```

PSW:

Exceptions:

Opcode:

plc.t Sj,Sk	0100010110	Count the number of 1's in Sj

Description:

The number of 1's in Sj are loaded into bits<6..0> in Sk. All other bits of Sk are reset to 0.

Notes:

MOVE SCALAR/ADDRESS   mov Sj,Ak

Purpose:

To move a scalar register into an address register.

Format:

```
--------------------------
|  Opcode      | Sj | Ak |
--------------------------
15            6,5  3,2  0
```

Operation:

Ak = Sj<31..0>

PSW:

Exceptions:

Opcode:

mov Sj,Ak	0101000011	Move 32 bits of Sj into Ak.

Description:

Move the least significant 32 bits of Sj into Ak. The most significant 32 bits of Sj are ignored.

Notes:

MOVE ADDRESS/SCALAR   mov Aj,Sk

Purpose:

To move an address register into a scalar register.

Format:

```
---------------------
| Opcode  | Aj | Sk |
---------------------
15       6,5  3,2  0
```

Operation:

```
Sk<31..0> = Aj
Sk<63..32> = Sk<63...62> ! unchanged
```

PSW:

Exceptions:

Opcode:

mov Aj,Sk	0101000111	Move an address to a scalar

Description:

The Aj address register is moved into the least significant 32 bits of Sk. The most significant 32 bits of Sk are unchanged.

Notes:

MOVE SCALAR/SCALAR mov.(w|l|s|d) Sj,Sk

Purpose:
To move one scalar register to another

Format:

```
--------------------------
|  Opcode        | Sj | Sk |
--------------------------
15              6,5  3,2  0
```

Operation:

```
Sk = Sj       ! longword
Sk<31..0> = Sj<31..0>    ! word
```

PSW:

Exceptions:

Opcode:

mov.l Sj,Sk	0101000110	Move scalar register longword
mov.w Sj,Sk	0101000100	Move scalar register word
mov.d Sj,Sk	0101000110	Move scalar register single float
mov.s Sj,Sk	0101000100	Move scalar register double float

Description:
The contents of the specified portion of scalar register Sj are moved to Sk. Sj and any unreferenced portion of Sk remain unchanged.

Notes:

1 The .s and .w forms of this instruction are equivalent, as are the .d and .l forms. The .s and .d forms are added for convenience.

CHAPTER 11

11 Program Control Instruction Set

The instructions defined in this chapter alter the program counter (PC). Alterations of the program counter may occur as a result of a branch performed after a comparison, an unconditional jump, a subroutine call or return, or an operating system call or return. A return from interrupt processing is also defined.

11.1 Branches/Jumps

These instructions include Branch on PSW Bit, Jump, and Breakpoint. The branch instructions provide a convenient way to branch relative to the present value of the PC. All branch instructions are 16-bits in length. Jump instructions load the program counter with an effective address. The effective address is developed as an operand address. Only Breakpoint affects the flags in the PSW, as follows:

FRL, C, AIV, ADZ, SC, SIV, SDZ, OV, UN, FDZ, RO = 0

When the PC increments to reference the next sequential instruction or a new value is loaded into the PC, the PC is altered in the following way for branches and jumps. If the current ring is 4, bits<30..1> are loaded. If the current ring is not ring 4, bits<28..1> are loaded. In all cases, the branch or jump is constrained to be within the current ring.

11.2 Subroutine Call/Save/Return

This section describes the facilities provided for subroutine call/save/return. The general strategy is to provide the following:

1 An efficient mechanism in both time and space.
2 Flexibility in supporting precompiled argument packets or arguments pushed onto a stack.
3 Passing arguments by reference (a byte pointer) or by value.
4 Support of the notion of a common run-time environment and linking subroutines written in one or more languages.

With these objectives in mind, the following facilities and/or methodologies are supplied.

1 An architecturally defined stack pointer, argument pointer, and frame pointer within the address register space.
2 Three call instructions: call, calls, and callq. The calls instruction pushes minimal machine state. The call instruction pushes all the scalar machine state. FRL in the saved (pushed) PSW denotes the amount of machine state pushed. A third, callq,

is used for local subroutines that do not require the definition of a new stack frame.

3 A set of instructions which push and pop registers from the stack. Also an instruction which pushes effective addresses onto the stack.

The following instructions could be generated for the two most likely subroutine invocation sequences: precompiled argument packets, and arguments pushed on the stack.

Precompiled Arguments

1 Ldea (the effective address) into AP the address of the packet.
2 Call the subroutine.
3 Rtn instruction executed in called subroutine's space.
4 Load the former AP from the stack (the effective address).

Arguments Pushed onto the Stack

1 Push arguments onto the stack.
2 Move SP to AP (address of the argument packet in the stack).
3 Call the subroutine.
4 Rtn instruction executed in the callee's space.
5 Clear off the pushed arguments by an add.w,#(n|N),Ak instruction.
6 Load the former AP from the stack.

11.3 <u>Stack Structure/Return Blocks</u>

The structure of the stack for subroutine entry and exits is described below (where LSI means Language Specific Information):

There are 4 types of return blocks:

* Short
* Long
* Extended
* Context

A short return block is formed as a result of executing a calls instruction. The return address, psw, A7, and A6 are saved. (FRL<1...0>=11).

A long return block is formed as a result of executing a call instruction. The return address, psw, A7, A6, A5 through A1, and S7 through S1 are saved. A0 and S0 are not saved. (FRL<1...0>=10).

An extended return block is formed as a result of: system call (sysc), trap, or a breakpoint. The return block contains the return address, psw, all the A registers, and all the S registers. The stack pointer saved (A0) references the value of A0 PRIOR to the saving of the extended return block. (FRL<1...0>=01).

A context block may be formed as a result of a system exception. The context block contains an extended return block plus internal machine state. This internal state will change for each implementation. A context block is pushed on the ring 0 process stack. (FRL<1...0>=00).

Returning from each of these return blocks is implemented as follows: the rtn instruction is used for the short, long and extended return blocks. The rtnc instruction is used to return from system exceptions that save a context block. The structure of the stack is shown in Figure 11-1.

11.4 Quick Calls and Return

Instructions are provided for subroutine entry that only save and restore the program counter. The PSW and frame pointer are unaltered. The callq instruction pushes the address of the next instruction and branches to the subroutine. The rtnq pops the program counter value on top of the stack.

11.5 System Call and Return

These instructions are used to perform system calls and system returns, and perform ring crossings in a protected manner. The System Call instruction is used to perform the call, and the flags in the PSW are cleared to all zero. To perform a system return, the standard return instruction (rtn) is used, and the flags are loaded from the return block.

Figure 11-1: Stack Structure for Subroutine Entry

```
                |--------------------|
 Caller's FP->| Caller's RTN Addr.|
                |--------------------|
                |  Caller's LSI 2    |      Language Specific Information
                |--------------------|
                |                    |
                |      Caller's      |             ||   Decreasing
                |      Automatic     |             ||   Addresses
                |      Storage       |            \  /
                |--------------------|             \/
                |      ARGn          |
                |--------------------|
                |       ...          |
                |--------------------|      (Arg. list may not
         AP ->  |      Arg1          |       be located in stack)
                |--------------------|
                |   Callee's LSI 1   |      Language Specific Information
                ----------------------
                |   Saved S0         |<-- Extended Frame Only
                ----------------------
                |   Saved S1         |<--|
                |--------------------|   |
                |      ...           |   |
                |--------------------|   |---Long Frames Only
                |   Saved S7         |   |
                |--------------------|<--|
 Prior to Push|   Saved A0  (SP)   |<-- Extended Frame Only
                |--------------------|<--|
                |   Saved A1         |   |
                |--------------------|   |
                |      ...           |   |---Long Frames Only
                |--------------------|   |
                |   Saved A5         |<--|
                ---------------------------------
                |   Saved A6  (AP)   |     (32)    |
                |--------------------|             |
                |   Saved A7  (FP)   |     (32)    |
                |--------------------|             |-Short Frame
                |   Saved PSW        |     (32)    |
                |--------------------|             |
 Callee   FP->  |   Return Addr      |     (32)    |
 After Call     |--------------------|--------------
                |  Callee's LSI 2    |     Language Specific Information
                |--------------------|
                |   Callee's         |     (N*32)       ||
                |   Automatic        |                  ||
          SP->  |   Storage          |   Direction of \  /
                |--------------------|   Stack Growth  \/
```

Purpose:

To conditionally or unconditionally perform a short PC relative branch.

Format:

```
------------------------------
| Opcode     |  Displacement  |
------------------------------
15          8,7              0
```

Operation:

```
IF (test .EQ. .TRUE.) THEN
        PC = PC + sign-extended-displacement
ELSE
        PC = PC+2
END IF
```

PSW:

unchanged

Exceptions:

Opcode:

nop	01110000	No Operation
br	01110001	Branch Always
bri.f	01110010	Branch on ION false
bri.t	01110011	Branch on ION true
bra.f	01110100	Branch on address carry false
bra.t	01110101	Branch on address carry true
brs.f	01110110	Branch on scalar carry false
brs.t	01110111	Branch on scalar carry true

Description:

The specified condition is tested. If the tested condition is asserted, the 8-bit displacement is sign extended to 31 bits. These 31 bits are then added to bits<31..1> of the program counter.

The program counter value added to contains the address of the branch instruction. If the tested condition is not asserted, then the next sequential instruction is executed.

Notes:

1 These instructions are used for short PC relative branches. All branches are constrained to stay within the current ring (indicated by bits<31..29> of the PC).
2 Additional instructions exist to jump to any arbitrary instruction. See the instruction.
3 The range of the branch instruction is +127 to -128 halfwords, or +254 to -256 bytes.
4 The displacement is unused for the instruction.

Purpose:

To conditionally/unconditionally jump to an address.

Format:

```
---------------------------    ------------------
| Opcode   |@|L| Aj  | Ak  |    | Displacement   |
---------------------------    ------------------
15         8,7,6,5   3,2   0    (31,15)           0
```

Operation:

```
IF (OPCODE-COND .EQ. .TRUE.) THEN
        PC = Effective Address
ELSE
        PC = PC + Instruction Length
END IF
```

PSW:

Unchanged

Exceptions:

Opcode:

exit	000000000	Error Exit Instruction
jmp <effa>	000000010	Jump Always
jmpi.f <effa>	000000100	Jump on ION false
jmpi.t <effa>	000000110	Jump on ION true
jmpa.f <effa>	000001000	Jump on address carry false
jmpa.t <effa>	000001010	Jump on address carry true
jmps.f <effa>	000001100	Jump on scalar carry false
jmps.t <effa>	000001110	Jump on scalar carry true

Description:

The specified condition is tested. If the condition is asserted, the Program Counter is loaded with the effective address. If the condition is not asserted, the Program Counter is updated to reference the next sequential instruction.

For the error exit instruction, a system call to ring 0, byte address C (hex) is performed. A class code of 0 is loaded into byte 2 of A5. Zero is loaded into byte 3 of A5.

Notes:

1 All jumps (except exit) are constrained to be within the current ring. That is, if the current ring is 4, the most significant bit of the effective address is ignored. If the current ring is 3,2,1, or 0 the most significant 3 bits of the effective address are ignored.

2 The error exit instruction is an instruction whose opcode is all 0. It detects a common programming error (transfer to a page that was initialized to all 0 by the OS).

3 For the error exit instruction, the L (length) bit is interpreted. This means that the error exit can have a 16- or 32-bit displacement. The @ bit (bit 7) is not interpreted; thus, indirection can never occur for the error exit instruction.

BREAKPOINT bkpt

Purpose:
To jump to a debugger via a breakpoint call.

Format:

```
----------------------------
| Opcode           | Ak   |
----------------------------
15               3,2     0
```

Operation:
An extended call is performed to the address contained in byte address 50 (hex) of the current ring.

PSW:
FRL= 00
C,AIV,ADZ,SC,SIV,SDZ,OV,UN,FDZ,RO = 0

Exceptions:

Opcode:

bkpt	0111110101010	Breakpoint

Description:
A subroutine call is executed. The callee's address is the current ring, byte address 50 (hex). An extended return block is pushed on the user stack. Thus all the A and S registers are pushed onto the current stack. The PC saved in the return block references the instruction immediately following the bkpt instruction.

Notes:

1 The length of the bkpt instruction is 1 halfword. Thus a bkpt instruction can be substituted for any instruction in the CONVEX architecture.
2 The Ak field is not used.

CALL A SUBROUTINE (call|calls) <effa>

Purpose:

To call a subroutine.

Format:

```
 ---------------------------    ------------------
 | Opcode  |@|L| Aj  | Ak  |    | Displacement   |
 ---------------------------    ------------------
 15         8,7,6,5  3,2  0     (31,15)          0
```

Operation:

```
IF (CALLS-OPCODE) THEN
        PSW-FRL-BITS = 11  ! short frame
ELSE
        PSW-FRL-BITS = 10  ! long frame
END IF

IF (CALL-OPCODE) THEN  ! long frame
        PUSH S1..S7
        PUSH A1..A5     !SP is not updated
ENDIF

PUSH A6         ! short and long frame
PUSH A7
PUSH PSW
PUSH next instruction address

PSW(FRL,C,SC,AIV,ADZ,UN,OV,FDZ,RO,SIV,SDZ)=0

IF (CALLS-OPCODE) THEN
        A0 = A0 -16 ! short frame
ELSE
        A0 = A0 -92 ! long frame
END IF

A7 = A0
PC = Effective Address
```

PSW:

```
PSW<FRL>=00
PSW<C,SC,AIV,ADZ,UN,OV,FDZ,RO,SIV,SDZ>=0
```

Exceptions:

Opcode:

call <effa>	001000000	Call a subroutine, make a long frame
calls <effa>	001000010	Call a subroutine, make a short frame

Description:

A new stack frame is created. A short frame is created for the calls instruction. A long frame is created for the call instruction. S0 is never saved. The FRL bits in the PSW saved indicate the frame size created. A0 and A7 are updated to reference the new top of stack. The effective address of the call instruction then becomes the new value of the program counter. The trap enable bits of the PSW are propagated from the caller to the callee (i.e., left unchanged). Thus, if the caller had floating point overflow traps enabled, the callee also has floating point trap enabled. The status bits of the callee's PSW are reset to 0.

Notes:

1 A local area for variables is created by executing a sub.w #N,SP on entry to the called routine.
2 The previous values of the frame and argument pointers are saved on the stack.
3 Arguments are referenced with positive displacement from an argument pointer if defined by software convention.
4 S0 is propagated but not saved. Typically, S0 is used to hold return values of functions (by software convention).
5 A0, the stack pointer, is not saved. A0, however, is updated to reference the new top of stack.
6 All A and S registers (except A0 and A7 -- these reference the new top of stack) are propagated through the call.
7 The Ak field is unused.
8 The FRL bits in the PSW register are always reset to 0. To determine the type of frame created, the FRL bits of the PSW in the saved frame must be examined.

Purpose:

To return from a subroutine.

Format:

```
----------------------------
| Opcode           | Ak   |
----------------------------
15                3,2     0
```

Operation:

SP = FP ! remove local save area
Unwind stack to previous frame (restore stack built by call, calls, sysc, or exception condition handler). If this is an outward return, then the SP after the pop is stored into bytes <72..75> of page 0 of the ring containing the rtn instruction.

PSW:

Load from current stack frame

Exceptions:

Ring Violation; Inward Return
Ring Violation; Invalid Frame Length

Opcode:

rtn	0111110010010	Return from subroutine call

Description:

An exit from a subroutine is performed. FP is moved to SP to reestablish the top of the stack. It is assumed that the subroutine was entered using a call, calls instruction, sysc (system call), or an exception condition that pushed an extended return block.

There are 4 types of return blocks: short, long, extended, and context. Short and long, are created as a result of a subroutine call; extended and context are created as a result of a system call, fault, or trap. When FRL=00 (indicating a context block), an rtnc (return from context) must be executed. When a ring crossing is encountered (as indicated by an extended return block and a saved PC whose ring field is not the current ring), then the following occurs. First, a check for outward ring crossing is made. If the ring crossing is inward, a system exception condition occurs. If outward, the extended return block is popped. The stack pointer after the pop is stored in bytes <72..75> of page 0 of the ring containing the rtn instruction.

Notes:

1 The FRL field from the PSW in the stack indicates the type of return block saved: 11=short, 10=long, 01=extended, and 00=context.
2 Ak is not used in this instruction.
3 The rtnc instruction must be used to return when FRL=00, indicating a context block.

PUSH PROGRAM COUNTER

callq <effa>

Purpose:

To push the PC onto the stack and branch.

Format:

```
----------------------------     ------------------
| Opcode  |@|L| Aj  | Ak  |     | Displacement  |
----------------------------     ------------------
15         8,7,6,5   3,2   0     (31,15)          0
```

Operation:

```
TEMP = PC-NEXT-INSTRUCTION
PC = Effective Address
PUSH TEMP
```

PSW:

Exceptions:

Opcode:

callq <effa>	001000100	Push the program counter and jump

Description:

The address of the instruction immediately following the callq instruction is pushed onto the stack. The PC is then loaded with the calculated effective address.

Notes:

This instruction is used as a fast subroutine call when the current address context need not be altered. The rtnq instruction is used to return from the callq instruction.

Purpose:

To pop the top element of the stack and load the PC.

Format:

```
----------------------------
| Opcode            | Ak   |
----------------------------
15                3,2     0
```

Operation:

```
Temp = c(A0)
A0 = A0 + 4
IF (PC<31> = 1) THEN
   PC<30..0> = TEMP <30..0>
ELSE
   PC<28..0> = TEMP <28..0>
ENDIF
```

PSW:

Exceptions:

Opcode:

rtnq	0111110010000	Pop the program counter and jump

Description:

The top of the stack contains a previously pushed PC value. The PC is popped into the present PC, the stack is adjusted, and execution continues at the instruction referenced by the popped PC.

Notes:

The Ak field is not used.
The current ring of execution does not change.

SYSTEM CALL sysc #r,#g

Purpose:

To perform a system call to ring #r.

Format:

```
-----------------   ------------------------------
|Opcode|L|000|Ak|   | #r |      0      |    #g    |
-----------------   ------------------------------
15     7 6 5 4 3 0  31 29 28         16 15         0
```

Operation:

An inward or current ring crossing with protection checks is performed. Temp = PC+instruction length, where PC is the address of the sysc instruction.

If the call is to the current ring, then the current SP is used. If the call is to an inner ring (#r), then the stack for ring #r is used. The SP contained in page 0 of the inner ring is used to define the base of the inner ring stack.

```
FRL<1..0> = 01 ! indicates an extended frame

Execute a call to ring #r, GATE_ARRAY(#g).
Get new stack pointer (as described above).
SP = SP - 104   ! Extended return block
FP = SP
PC<31..29> = #r
PC<28..1> = GATE_ARRAY(#g)
```

PSW:

Cleared to all zero

Exceptions:

Ring Violation; Outward call
Ring Violation; Invalid Gate

Opcode:

sysc #r,#g	000100001	Perform a system call

Description:

The #g field is used to reference an entry in a gate array. The #r field is the number of the called ring. The base of the gate array is referenced by bytes <76..79> of page zero of the ring named by #r.

After the new ring stack is established, an extended return block is pushed. All of the A and S (including S0 and A0) registers are saved, as well as the PSW and the address of the instruction following this instruction (which is the system call instruction).

The #g field is used to obtain the address of the called instruction for both intra (same ring) or inward ring crossing.

Notes:

1 The Ak field is not used.
2 See the protection chapter for a more detailed explanation of stack switching and the structure of the gate array.
3 The stack pointer saved in the extended return block references the top of stack of the caller's ring.
4 If the ring to be called is 0, and the gate entry is less than 32,768, then the immediate length can be specified as 16 bits.
5 If L in the instruction is 0, the 16-bits immediately following the opcode are sign extended to 32 bits. These 32 bits are interpreted as the #r, #g fields.

CHAPTER 12

12 Privileged Control/Status Instruction Set

This chapter contains the definition of the privileged instruction set. A privileged instruction is like any other CONVEX instruction with one difference: the current ring of execution must be 0. This means that bits<31..29> of the program counter must be all 0. Privileged instructions are used by the operating system kernel to control process multiplexing, virtual address space management, and system clock stores.

Other instructions are also included in this chapter. These instructions generally are used by parts of the extended operating system contained in a ring other than 0.

An attempt to execute a privileged instruction in a ring other than 0 results in a system exception and the generation of a system call through byte address 0C (hex) of page 0 of ring 0.

12.1 Clocks

A series of clocks are available to the operating system and user. These clocks are used for:

1 Scheduling
2 Maintaining a time chronometer
3 Maintaining an alarm
4 Measuring the time it takes for a program to execute.

The clocks associated with items 2 and 3 are maintained external to the job processor. By addressing main memory locations appropriately, the time of day can be determined.

Supporting items 1 and 4 are a 32-bit interval timer register and 2 associated support registers. Manipulation of all three of these registers is privileged.

The 3 32-bit registers are the:

* Next Internal Timer Counter (NITC)
* Interval Timer Counter (ITC)
* Interval Timer Status Register (ITSR)

The format of the NITC (Next Interval Timer Count) is:

```
-----------------------------------------
|      |Next Interval Timer Count |     |
-----------------------------------------
31   28,27                       8,7    0
```

When ITC overflows, NITC is loaded into ITC to resume counting.

The format of the ITC (Interval Timer Counter) is:

```
-----------------------------------------
|     | Interval Timer Counter    |     |
-----------------------------------------
31   28,27                       8,7    0
```

The ITC is a 32-bit register that is incremented every microsecond in bit 8. It is loaded from NITC when ITC becomes all 0. At that time an interrupt may also be generated (as controlled by ITSR).

The format of the ITSR in read mode is:

```
---------------------------------------
|ON|INE|FULL|OVF|   Reserved          |
---------------------------------------
31  30   29  28  27                   0
```

where the ITSR bits have the following meanings:

* Bit 28 - OVF. When ITC overflows and if FULL is already set, then OVF is set to one. This condition occurs when 2 or more interval timer overflows occur without an interrupt being processed. OVF is cleared to 0 when a value is loaded into ITSR.

* Bit 29 - FULL. When ITC overflows, FULL is set to 1. If INE is also set to 1, then an interval timer interrupt occurs (vectors through byte address 10 (hex) of ring 0). FULL is reset to 0 when a value is loaded into ITSR.

* Bit 30 - INE. When INE is set to 1 and bit 29 (Full) is set to 1, an interval timer interrupt (vectors through byte address 10 (hex) of ring 0) occurs. When INE is 0, no interrupt occurs.

* Bit 31 - ON. When ON is reset to 0, ITC does not increment. When ON is set to 1, ITC increments every microsecond.

For write mode, the ITSR will operate as follows:

```
   31      30       29       28
------------------------------
| ON    |  INE   |  DEC   | --- |
------------------------------
```

The FULL and OVFL bits act as a pseudo two-bit counter, incremented by an overflow of the timer counter, and decremented by a write to the ITSR with DEC (bit 61) set to a one. OVF and FULL count as follows:

Before FULL	WRITE OVF	After FULL	WRITE OVF	(DEC =1)
0	0	0	0	
1	0	0	0	
1	1	1	0	

Note that two writes with DEC=1 will clear both FULL and OVF. A write to ITSR with DEC=0 will not affect the FULL and OVF bits.

The interval timer interrupt handler should execute a store into the ITSR to clear FULL. An interval timer interrupt is masked out by either INE being reset to 0 in the ITSR or by the ION flag being set to 1. When all three interval timer registers are concurrently moved to/from a scalar register, the format of Sk is:

```
-----------------------------------------------------
| ITSR |    NITC     |      RES      |    ITC    | RES |
-----------------------------------------------------
63    60 59        40 39           28 27        8 7    0
```

12.2 Instruction Set

The instructions defined in this chapter are:

dsi
eni
ldsdr
ldkdr
patu
pate
rtnc
pich
plch
Load ITR
Store ITR
Load ITSR
Load CPUID
xmti
mski

Privileged Control/Status Instruction Set

```
halt
mov Sk,VV
tstvv
```

Purpose:

To load values into the process SDR's.

Format:

```
-----------------------------
| Opcode            | Ak    |
-----------------------------
15                  3,2     0
```

Operation:

SDR(1,2,...,7) = Effective Address(Ak) ! memory data must be resident and word aligned.

PSW:

Exceptions:

Ring Violation; Privileged Instruction

Opcode:

ldsdr Ak	0111110000000	Load process SDR's

Description:

Ak contains the effective address of a 7 entry table. Each entry is a word. Entry 1 contains a value to be stored into SDR1. Entry 2, at (Ak)+4, contains a value to be stored into SDR2, and so on. Upon completing the store of the 7 table entries into SDR(1:7), the ATU, logical cache, and instruction cache are purged of any entries associated with segments 1-7.

Notes:

1 The data located in main memory to be loaded into the SDR's must be resident to ensure that an address translation fault does not occur and must be aligned on 32-bit words. If the data are not resident or aligned, a machine exception occurs.

2 The data located in main memory to be loaded into the SDR's must be word aligned on a 32-bit boundary. If they are not, a machine exception occurs.

LOAD KERNEL SDRs

ldkdr Ak

Purpose:

To load values in all 8 SDR's.

Format:

```
----------------------------
| Opcode            | Ak   |
----------------------------
15                 3,2    0
```

Operation:

SDR(0,1,...,7) = Effective Address(Ak)

PSW:

Exceptions:

Ring Violation; Privileged Instruction

Opcode:

ldkdr Ak	0111110000001	Load all 8 SDR's

Description:

Ak contains the effective address of an 8 entry table. Each entry is a word. Entry 0 is stored in SDR0. Entry 1 is stored into SDR1 and so on. (Ak) references the table entry 0. (Ak)+4 references table entry 1. Upon completing the store of all 8 entries into SDR(0,1,...,7), the ATU, logical, and instruction caches are purged. If, prior to ATU purging, logical addressing equaled physical addressing, then the ATU is enabled. Thus, all subsequent addresses are virtual.

Notes:

1 The data located in main memory which are to be loaded into the SDR's must be resident to ensure that no address translation fault occurs. If the data are not resident, a machine exception occurs.

2 The data located in main memory which are to be loaded into the SDR's must be word aligned on a 32-bit boundary, or a machine exception results.

3 The addressing environment must be physical (the ATU must be turned off). If the ATU is turned on, a machine exception occurs.

4 Memory data must be resident and word aligned.

PURGE ATU

Purpose:

Purge the entire ATU.

Format:

```
----------------------------
| Opcode            | Ak   |
----------------------------
15                 3,2    0
```

Operation:

```
ATU_valid_bits = 0
Purge L_cache
Purge I_cache
```

PSW:

Exceptions:

Ring Violation; Privileged Instruction

Opcode:

patu	0111110000100	Purge the entire ATU

Description:

All the entries in the ATU are purged, which means that any virtual addresses that were encached are invalidated. The logical and instruction caches are also purged.

Notes:

1 The Ak field of the instruction is unused.

Purpose:

To purge an ATU entry.

Format:

```
----------------------------
|  Opcode            |  Ak  |
----------------------------
15                  3,2    0
```

Operation:

```
ATU_valid_bit(Ak) = 0
Purge L_cache
Purge I_cache
```

PSW:

Exceptions:

Ring Violation; Privileged Instruction; class=8, qualifier=0

Opcode:

```
pate Ak        0111110000101   Purge ATU entry
```

Description:

Ak contains a virtual address. If there is an ATU entry associated with the address in Ak, that ATU entry is purged (i.e., marked invalid). All other ATU entires are left unchanged.

Notes:

1 This instruction is typically used when a pageframe is added to the working set of a process after an address translation fault. The page associated with the fault is found in physical memory but is not part of the process's working set. Thus no I/O request is necessary to resolve the fault.

RETURN FROM CONTEXT BLOCK rtnc

Purpose:
To return from a context block.

Format:

```
---------------------------
| Opcode            | Ak  |
---------------------------
15                 3,2    0
```

Operation:

```
Pop context block using A0.
Processor State = Context Block (FRL=00).
A0 = A0 + context block size
Ring 0, bytes <36..39> = A0
A0 = stack pointer from context block.
```

PSW:

Exceptions:
Ring Violation; Privileged Instruction

Opcode:

rtnc	0111110010101	Return from a context block

Description:
The context block on the ring 0 stack is popped; then the new value of the stack pointer after the pop, A0, is stored into the context stack pointer in bytes <36..39> of ring 0. Finally, the stack pointer value contained within the context block just popped is loaded into A0.

Notes:

1 The entire processor state was stored in the context block at the time that the condition that initiated the exception occurred. The cause of the exception was loaded into A5 after the context block was stored. This permits the operating system to recover from the exception condition, if possible.

2 The Ak field of the instruction is unused.

3 The processor context block is the entire hardware machine state at the time of the system exception condition. After the OS resolves the exception (if it can), the machine state (the context block) is restored. Then the faulted instruction can resume execution.

4 The paradigm for system exception conditions is identical to kernel calls, and differs only in the size of the saved return block, and the stack used.

5 The processor does not check the FRL bits in the PSW. They are assumed to be 00.

ENABLE/DISABLE INTERRUPTS (eni|dsi)

Purpose:
To enable or disable interrupts.

Format:

```
-------------------------
| Opcode          | k |
-------------------------
15               3,2  0
```

Operation:
ion = 1 ! eni ion = 0 ! dsi

PSW:

Exceptions:
Ring Violation; Privileged Instruction

Opcode:

eni	0111110101000	Enable interrupts, set ion to 1
dsi	0111110101001	Disable interrupts,reset ion to 0

Description:
The ion flag is used to enable or disable interrupts. When ion is a 1, interrupts are enabled. When ion is a 0, interrupts are disabled.

The next sequential instruction is always executed even though interrupts may be pending when ion is set to 1.

The ion flag enables or disables the interval timer, power fail, and the CPU virtual channel interrupts.

Notes:
1 Instructions exist to test the value of the ion.
2 One additional instruction is always executed before any interrupts are taken.

Purpose:
Purge the entire ICACHE.

Format:

```
---------------------------
| Opcode          | Ak   |
---------------------------
15                3,2    0
```

Operation:
ICACHE_valid_bits = 0

PSW:

Exceptions:

Opcode:

pich	0111110000110	Purge the instruction cache

Description:
All the entries in the instruction cache are purged. After the execution of the pich instruction, the instruction cache is reloaded from main memory.

Notes:

1 The Ak field of the instruction is unused. The pich instruction does not affect the results produced by the currently executing program, only its performance. The pich instruction is typically used by a language debugger to PURGE the Instruction Cache after a modification of instruction space is performed.

2 This is NOT a privileged instruction.

PURGE LOGICAL CACHE

Purpose:
To purge the logical cache.

Format:

```
---------------------------
| Opcode            | Ak  |
---------------------------
15                 3,2    0
```

Operation:
LCACHE_valid_bits = 0

PSW:

Exceptions:

Opcode:

plch	0111110000111	Purge the logical cache

Description:
All the entries in the logical cache are purged. After the execution of the plch instruction, the logical cache must be reloaded from main memory.

Notes:

1 This plch instruction is used by a process that is concurrently performing I/O and executing another task (multi-tasking within a process). The typical use of plch occurs when the pended task blocked or on I/O is completed and is subsequently dispatched and becomes running. As part of the dispatching mechanism, the logical cache must be purged. When process multiplexing occurs (by reloading the process SDR's), the logical cache is automatically purge.
2 The plch instruction prevents previous data (stale data) from the pended task from being re-referenced when the pended task makes the transition to running. This instruction is only necessary for multiple tasks within the same process.
3 This instruction is NOT privileged, and Ak is a don't care.

MOVE SCALAR/ITR mov Sk,ITR

Purpose:
To move Sk to the interval timer registers (nitc,itsr,itc)

Format:

```
--------------------------
|  Opcode           | Sk |
--------------------------
15                3,2   0
```

Operation:

```
nitc = Sk<59..40>
itsr = Sk<63..60>
itc = Sk<27..8>
```

PSW:

Exceptions:
Ring Violation; Privileged Instruction

Opcode:

mov Sk,ITR	0111110001101	Load NITC, ITC, ITSR from Sk

Description:
The contents of Sk are used to load the next iteration count register, the iteration counter, and the iteration status register of Sk.

Notes:

1 For a description of the interval timer and its associated registers, please see the introduction to Chapter 12 in the CONVEX Architecture Handbook.

MOVE ITR/SCALAR mov ITR,Sk

Purpose:
To move the interval timer registers (itc,nitc,itsr) to Sk

Format:

```
--------------------------
| Opcode             | Sk |
--------------------------
15                  3,2  0
```

Operation:

```
Sk<59..40> = nitc
Sk<63..60> = itsr
Sk<27..8> = itc
```

PSW:

Exceptions:
Ring Violation; Privileged Instruction; class=8, qualifier=0

Opcode:

```
mov ITR,Sk      0111110001100   Move the itc,itsr,nitc into Sk
```

Description:
The current value of the iteration counter, next iteration counter, and the interval timer status register are loaded into Sk.

Notes:
See the description of the interval timer and its associated registers at the beginning of this chapter.

Purpose:

To Sk<63..0> to the ITSR register

Format:

```
--------------------------
|  Opcode            | Sk |
--------------------------
15                  3,2  0
```

Operation:

ITSR = Sk<63..60>

PSW:

Exceptions:

Ring Violation; Privileged Instruction; class=8, qualifier=0

Opcode:

mov Sk,itsr	0111110001111	Load ITSR with a scalar

Description:

The current value of itsr is loaded from the specified Sk.

Notes:

See the description of the interval timer and its associated registers at the beginning of this chapter.

TRANSMIT INTERRUPT xmti Sk

Purpose:
To interrupt a channel

Format:

```
--------------------------
|  Opcode            | Sk |
--------------------------
15                  3,2  0
```

Operation:
Assert the channel interrupt line of virtual channel c(Sk<7..0>)

PSW:

Exceptions:
Ring Violation; Privileged Instruction

Opcode:

xmti Sk	0111110101101	Transmit Interrupt

Description:
An interrupt to the specified virtual channel is asserted. The least significant 8 bits of Sk (Sk<7..0>) indicate which of the 256 possible virtual channels is interrupted.

Notes:

1 Channels 0, 1,...,7 are associated with the CPU. These CPU channels are referred to as CPU virtual channels. Interrupts to these channels are maskable; thus, a CPU can interrupt itself by referencing virtual channels (0,1,...,7). Please see the mski instruction.

2 Sk bits <63..8> are not used.

MASK INTERRUPT   mski Sk

Purpose:
Mask the virtual channels

Format:

```
--------------------------
|  Opcode            | Sk |
--------------------------
15                  3,2  0
```

Operation:
Mask out, individually, the virtual channels using Sk<7..0>

PSW:

Exceptions:
Ring Violation; Privileged Instruction

Opcode:

```
mski Sk         0111110101100   Mask Out Interrupt
```

Description:
The least significant 8 bits of Sk are used as a mask. Bit 0 masks out virtual channel 0, bit 1 masks out virtual channel 1, and so on. A 0 inhibits the interrupt from a channel. A 1 enables the interrupt from the channel. Each channel can be individually masked out independently from the others.

If concurrent interrupts are pending on multiple enabled virtual channels, then the interrupts are responded to in the following order:

```
0 -- highest priority
7 -- lowest  priority
```

Notes:

1 The operating system must explicitly perform mask outs upon interrupt service. This may require the saving of any previous mask values.

HALT halt #N,Ak

Purpose:
To halt the central processing unit.

Format:

```
----------------------------  --------------------
| Opcode  |L| 000 |  Ak    |  |        N          |
----------------------------  --------------------
15,         6,5  3,2      0  31|16                0
```

Operation:

```
assert an I/O interrupt request

Ak = immediate  ! Ak is loaded to indicate the halt reason
```

PSW:

Exceptions:
Ring Violation; Privileged Instruction

Opcode:

halt #N,Ak	000100000	Halt the central processing unit

Description:
The immediate field is loaded into Ak, and the central processing unit is halted. Further action is machine implementation dependent.

Notes:
This instruction is typically used for diagnostic and debugging purposes.

EXECUTE DIAGNOSTIC MICROCODE diag Ak

Purpose:
To execute a desired sequence of non-standard microcode

Format:

```
----------------------------
|  Opcode            |  Ak  |
----------------------------
15                  3,2     0
```

Operation:
Execute microcode sequence pointed to by the contents of Ak

PSW:

Exceptions:

```
Undefined  Opcode        ! Ak does not contain valid operation code
Ring Violation           ! Privileged Instruction; class=8, qualifier=0
```

Opcode:

```
diag Ak         0111110111000   Execute non-standard microcode sequence
```

Description:
This instruction invokes one of a set of privileged instructions. Ak contains an opcode used by the microcode to jump to a desired sequence of microcode. Thus, while only one instruction opcode is used, multiple non-standard operations are accessible using this instruction. For a list of these operations, see below. An illegal opcode trap occurs if the contents of Ak are not supported by this instruction.

Ak contents	Operation
1	enable lcache
2	disable lcache
3	store P_cache at (a5)
4	load P_cache from (a5)
5	store Addr_Trans_caches at (a5)
6	load Addr_Trans_caches from (a5)
11	store sdrs 0-7 at (a5)
12	enable forced faults
13	disable forced faults
14	flush physical cache
15	store hardware physical address contents
16	load hardware physical address contents
17	enable halt in rings 1-4
18	disable halt in rings 1-4

A5 is used as a logical address which references an area in memory. All loads and stores use A5 as a base address, incrementing A5 as successive addresses are accessed.

Notes:

1 This instruction is used by diagnostics to reference internal processor registers not accessible to the user program and is specific to the C-1 implementation.

Purpose:

To move the least significant bit of Sk to the vector valid flag.

Format:

```
--------------------------
|  Opcode            | Sk |
--------------------------
15                  3,2  0
```

Operation:

```
IF   (Sk<0> = 1) THEN
        VV = 1
ELSE
        VV = 0
```

PSW:

Exceptions:

Ring Violation; Privileged Instruction

Opcode:

mov Sk,VV	0111110101110	Move scalar to vector valid flag

Description:

The current ring of execution must be 0; otherwise, a privileged instruction exception occurs. If the current ring is 0, the least significant bit of Sk (bit<0>) is moved to the vector valid flag.

TEST VECTOR VALID tstvv

Purpose:

To test the value of the vector valid flag.

Format:

```
--------------------------
| Opcode           | Sk |
--------------------------
15                 3,2  0
```

Operation:

```
IF  (VV = 1) THEN
        SC = 1
ELSE
        SC = 0
```

PSW:

SC is affected (see above).

Exceptions:

Opcode:

tstvv	0111110101111	Test value of vector valid flag

Description:

The SC bit is loaded with the value of the vector valid flag.

Notes:

1. Sk is unused.
1. This instruction is not privileged.
1. A brs instruction is typically used to determine the value of the SC bit after the execution of the tstvv instruction.

CHAPTER 13

13 Vector/Scalar Instruction Set

13.1 Overview

This chapter defines the instructions which manipulate the vector accumulators (V), scalar accumulators (S), and the vector merge (VM) register. The instructions which manipulate these registers are separate and distinct from the instructions which deal with the address registers. This distinction permits overlapped execution of instructions, which then perform operations on these registers (V, S, and VM), for increased performance. The vector/scalar instructions include Loads and Stores, Vector/Vector Arithmetics, Vector/Scalar Arithmetics, Vector/Vector Logical Operations, and Vector/Scalar Logical Operations.

As with the A registers, the basic instruction set philosophy is for all memory operands to be loaded first into a scalar or vector accumulator; then a register to register instruction performs the specified operation.

There are some additional features applicable to the S and V registers that are not applicable to the A registers. These features are: data types manipulated, chaining, functional unit reservation, register unit reservation, and register topology.

13.1.1 Data Types

There are 6 different data types; integer 8, 16, 32, and 64, referenced as byte, halfword, word, and longword, respectively; and single and double precision floating point (32 and 64 bit). Logical data types are a special case of integer 64 bit.

13.1.2 Vector Register Specification

As previously discussed in Chapter 3, "Register Set," a vector accumulator can hold up to 128 elements, where each element can be up to 64 bits of precision. The VL register specifies the exact number of elements contained in the vector accumulator. VL applies to all vector accumulators.

Note: when VL is zero, no vector operation is performed.

Care should be taken when loading the VL register. When an operation is initiated, VL is copied into the internal machine state. Length control is generated from this internal VL value. However, when chaining is initiated, VL should not be changed until after the last chained instruction begins executing.

The first element in a vector accumulator ,i, is Vi(0). If VL is less than 128, then the Vi(0) through Vi(VL-1) elements are manipulated. All other elements are unimportant and are left unchanged.

13.1.3 <u>Chaining</u>

Chaining is a vector mechanism that permits the output of one vector instruction to be immediately used as the input to another vector instruction. For example, the DOT product operation requires a sum (sigma) of a series of products. Chaining permits the sum to be initiated while the products are being produced. This form of concurrency results in significantly higher performance.

To facilitate chaining, vector register operations can specify up to three operands; two sources and one destination. This feature permits an output register to be different from either of the two possible input registers (3 operand addressing). The output register can then be used as an input for the second (chained) operation.

The following is an example of chaining:

```
        V3 = V2 + V1
        V5 = V4 * V3
```

In the above example, the output of the "+" that is stored into V3 can be immediately used as an input to the "*" operation, which is essentially equivalent to executing the single statement:

```
        V5 = V4 * V3 = V2 + V1
```

This single statement equivalent results in execution rates twice that achieved through sequential execution.

Another example is:

```
        V1 = Merge (VM,V0;S0)
        V3 = V2+V1
        V5 = V4*V3
```

This executes as the single expression:

```
        V5 = V4*V3 = V2 + V1 = Merge(VM,V0;S0)
```

which is essentially adding V2 to the merged elements of V0, and multiplying the results by V4. The output of one functional unit may be chained into the input of a different functional unit. The actual time of chaining can occur anytime the output is available. The output can be chained into stores, masks, reduction operations, etc. Unless otherwise specified, there are no exceptions to this chaining criterion. In this last example, chaining exists across three functional units.

13.1.4 Functional Unit Reservation

As explained above, several instructions can be executed at the same time because more than one arithmetic or functional unit is provided. Multiple functional units (e.g., ADD, MULTIPLY, and DIVIDE)--rather than multiple units of the same type (i.e., two or more adders)--are provided to ensure this higher performance. However, when a vector instruction is decoded which requires a functional unit currently being used, a functional unit reservation occurs. This implementation means that the second instruction CANNOT execute simultaneously with the first.

The types of independent functional units that are present are implementation dependent. Generally, the possible arithmetic units are:

1 ADD/SUB
2 MULTIPLY/DIVIDE
3 LOGICAL
4 MASK/MERGE/COMPRESS
5 LOAD FROM MEMORY
6 STORE TO MEMORY

These arithmetic units are structured into three separate and distinct functional units, grouped as follows:

1 ADD/SUB, LOGICAL, COMPARE, POPULATION COUNT, SHIFTING
2 MULTIPLY/DIVIDE
3 LOAD/STORE FROM MEMORY, MASK/MERGE/COMPRESS

The following is an example of functional unit reservation:

```
V2 = V1 + V0
V5 = V4 + V3
```

Both of these operations on the indicated vector registers require the ADD functional unit, and thus cannot function simultaneously.

13.1.5 Register Unit Reservation

To permit simultaneous instruction execution with multiple functional units, multiple registers must be provided, which means that access must be provided for all data to be manipulated. If this access is NOT provided, register reservation occurs, and instruction execution is sequential. In the following example,

```
V1 = V1 + V0
V4 = V3 * V1
```

the two instructions are executed sequentially since there are three references to V1 across the two instructions. This sequential execution occurs even though different functional units are specified. If the two instructions were,

```
V2 = V1 + V0
V5 = V4 * V3
```

then both instructions would execute simultaneously.

13.1.6 Accumulator Topology

The CONVEX architecture has 8 vector accumulators. Within the processor of CONVEX-1, these registers are structured in the following manner across 4 memory elements.

```
----------   -----------   -----------    ------------
|        |   |         |   |         |    |          |
|  V4    |   |   V5    |   |   V6    |    |   V7     |
----------   -----------   -----------    ------------
|  V0    |   |   V1    |   |   V2    |    |   V3     |
|        |   |         |   |         |    |          |
----------   -----------   -----------    ------------
```

Each of the above rectangles represents a high speed memory array contained within the processor. There are 4 such arrays; each has 2 vector accumulators with 64 bits in each element. The following are the access rules governing manipulations of elements in the array.

NOTE: The noted vector accumulator topology is implementation specific to CONVEX-1 and is NOT part of the architecture.

1) Two independent accesses can occur during each cycle for each array.

2) These accesses can be any combination of read and write. Thus (read,read), (read,write), (write,read), and (write,write) are permitted.

3) If more than two accesses are specified to the same array, the operation will still continue to function, but at reduced performance.

4) Read and writes to the same array during the same operation are permitted. There are no unusual side effects; the array functions exactly as one would expect. For example, the operation V0 = V0 + V1 adds vector accumulator 0 to vector accumulator 1 and stores the result in vector accumulator 0.

The following are some examples of full and partial speed operations as they relate to this array accessing.

The operation V2 = V1 + V0 proceeds at full speed. The operation V2 = V2 + V0 proceeds at full speed. The operation V2 = V2 + V6 proceeds at half speed, since both V2 and V6 are in the same array.

13.1.7 Recursion/Reduction

Reduction operators are explicitly provided and are not just a byproduct of vector accumulator specification. Explicit operators are provided to perform SUM, PROD, MAX, MIN, and various logical reductions (ANY, ALL, PARITY). The specification of a vector accumulator as both a source and destination causes the accumulator to function in the expected way, and there are no unusual side effects. The output of an operation can be chained into a reduction operation. For example, a dot product is a multiply chained into a sum operation.

13.1.8 Scalar Functional Units

From the compiler's viewpoint, there is an infinite number of scalar functional units. Thus, for the arithmetic expression,

```
        Z = A + B + C + D + E + F
```

the preferred way of evaluating the expression is:

```
Temp1 = A+B
Temp2 = C+D
Temp3 = E+F

Temp1 = Temp1 + Temp2
Temp1 = Temp1 + Temp3
```

This expression is in preference to:

```
Temp = A +B
Temp = Temp + C
Temp = Temp + D
Temp = Temp + E
Temp = Temp + F
```

This technique is referred to as "tree height reduction."

13.2 Loads and Stores (Gather and Scatter)

These instructions include Load Vector Register, Store Vector Register, Load Vector Register/Vector Index, Store Vector Register/Vector Index, Store Scalar Extended/Vector Index, and Store Scalar Extended. The Load and Store with index instructions allow the user to use a vector register to specify those indices of another vector which will be affected. These operations are commonly referred to as gather and scatter. None of the flags in the PSW are affected. This permits scatter stores and gather loads to be implemented.

VL elements of a vector are loaded into the specified vector accumulator Vk. The first element is referenced by the effective address produced by evaluating the L,@,A fields. The address of the next element accessed is obtained by adding the signed value in VS. This signed value is the distance in BYTEs. The address of every successive element is obtained by adding VS to the address of the previous element.

13.3 Vector/Vector Arithmetics

Included in this section are the following instructions: Add, Subtract, Multiply, Divide, and Negate Vector/Vector. The flags in the PSW are affected, as follows:

SIV	= Integer Overflow;	Integer Only
OV	= Exponent Overflow;	Floating Point Only
UN	= Exponent Underflow;	Floating Point Only
SDZ	= Divide by Zero;	Integer Only
RO	= Reserved Operand;	Floating Point Only
FDZ	= Divide by Zero;	Floating Point Only

13.4 Vector/Scalar Arithmetics

The instructions in this section include Add, Subtract, Multiply, and Divide Vector/Scalar. They affect the following flags in the PSW:

SIV	= Integer Overflow;	Integer Only
OV	= Exponent Overflow;	Floating Point Only
UN	= Exponent Underflow;	Floating Point Only
SDZ	= Divide by Zero;	Integer Only
RO	= Reserved Operand;	Floating Point Only
FDZ	= Divide by Zero;	Floating Point Only

13.5 Vector/Vector Logical Operations

These instructions include AND, OR, Exclusive OR, and Complement Vector/Vector; none of the flags in the PSW is affected by their operation.

13.6 Vector/Scalar Logical Operations

Instructions included in this group are AND, OR, and Exclusive OR Vector/ Scalar. No flags in the PSW are affected by their operation.

13.7 Shifts and Moves

Logical Shift Vector/Scalar, Move Scalar/Vector, and Move Vector Element/ Scalar are the instructions in this section, and none affects the flags in the PSW.

LOAD VECTOR REGISTER ld.(b|h|w|l|s|d) <effa>,Vk

Purpose:

To load a vector into a vector accumulator.

Format:

```
----------------------   --------------------
| Opcode |@|L| Aj| Vk|   | Displacement     |
----------------------   --------------------
15       8,7,6,5  3,2 0  (31|15)            0
```

Operation:

```
temp = Effective Address

DO 10  a = 0,(VL-1)
       Vk(a)  = c(temp)
       temp = temp +  VS
10 CONTINUE
```

PSW:

Exceptions:

Opcode:

ld.b <effa>,Vk	001110000	Load vector byte
ld.h <effa>,Vk	001110010	Load vector halfword
ld.w <effa>,Vk	001110100	Load vector word
ld.l <effa>,Vk	001110110	Load vector longword
ld.s <effa>,Vk	001110100	Load vector single float
ld.d <effa>,Vk	001110110	Load vector double float

Description:

VL elements of a vector are loaded into the specified vector accumulator Vk. The first element is referenced by the effective address produced by evaluating the L,@,A fields. The address of the next element accessed is obtained by adding the signed value in VS. This signed value is the distance in BYTEs.

The address of every successive element is obtained by adding VS to the address of the previous element.

Notes:

1 The value contained in VS can either be positive or negative.
2 64 bit integers or 64 bit floating point operands are loaded using the ld.l instruction.

3 32 bit integer or 32 bit singal precision floating point operands are loaded using the ld.w instruction.
4 If the distance between successive elements is less than the precision of an element, unpredictable actions occur.
5 VS is not changed during the execution of this instruction.
6 The .s and .w forms of this instruction are equivalent, as are the .d and .l forms. The .s and .d forms are added for convenience.

STORE VECTOR REGISTER st.(b|h|w|l|s|d) Vk,<effa>

Purpose:

To store a vector from a vector accumulator.

Format:

```
----------------------   --------------------
| Opcode |@|L| Aj| Vk|   | Displacement     |
----------------------   --------------------
15       8,7,6,5  3,2 0   (31|15)            0
```

Operation:

```
temp = Effective Address
DO 10 a = 0,(VL-1)
        c(temp) = Vk(a)
        temp = temp + VS
10 CONTINUE
```

PSW:

Exceptions:

Opcode:

st.b Vk,<effa>	001111000	Store vector byte
st.h Vk,<effa>	001111010	Store vector halfword
st.w Vk,<effa>	001111100	Store vector word
st.l Vk,<effa>	001111110	Store vector longword
st.s Vk,<effa>	001111100	Store vector single float
st.d Vk,<effa>	001111110	Store vector double float

Description:

VL elements of a vector are stored from the specified vector accumulator Vk. The first element is referenced by the effective address produced by evaluating the L,@,A fields. The address of the next element accessed is obtained by adding the signed value in VS. This signed value is the distance in BYTEs.

The address of every successive element is obtained by adding VS to the address of the previous element.

Notes:

1 The value contained in VS can either be positive or negative.

2 64-bit integers or 64-bit floating point operands are stored using the st.l instruction.

3 32-bit integers or 32-bit single precision floating point operands are stored using the st.w instruction.

4 If the distance between successive elements is less than the precision of an element, unpredictable actions occur.

5 VS is not changed during the execution of this instruction.

6 The .s and .w forms of this instruction are equivalent, as are the .d and .l forms. The .s and .d forms are added for convenience.

LOAD VECTOR REGISTER/VECTOR INDEX ldvi.(b|h|w|l|s|d) Vj,Vk

Purpose:

To load a vector into a vector accumulator using a vector of indices (commonly referred to as "vector gather").

Format:

```
----------------------------
| Opcode      | Vj    | Vk  |
----------------------------
15           6,5     3,2    0
```

Operation:

```
DO 10 a =0,(VL-1)
        temp = A5 + Vj(a)<31..0>
        Vk(a) = c<temp>
10 CONTINUE
```

PSW:

Exceptions:

Opcode:

ldvi.b Vj,Vk	0111100000	Index Load vector byte
ldvi.h Vj,Vk	0111100001	Index Load vector halfword.
ldvi.w Vj,Vk	0111100010	Index Load vector word
ldvi.l Vj,Vk	0111100011	Index Load vector longword
ldvi.s Vj,Vk	0111100010	Index Load vector single float
ldvi.d Vj,Vk	0111100011	Index Load vector double float

Description:

VL elements of a vector are loaded into the specified vector accumulator, Vk. The first element is referenced by the effective address produced by adding the contents of A5 and Vj(0). The second element is referenced by the effective address produced by adding the contents of A5 and the least significant 32 bits of Vj(1), and so on.

Notes:

1 A5 is typically loaded with an ldea (load effective address instruction.)

2 64 bit integers or 64 bit floating point operands are loaded using the ldvi.l instruction.

3 If the distance between successive elements is less than the precision of an element, unpredictable actions occur.

4 A5 is not changed during the execution of this instruction.

5 The contents of Vj(i) are treated as a byte offset. The appropriate shift of Vj may have to be performed to account for the precision of the operand to be loaded.

6 The .s and .w forms of this instruction are equivalent, as are the .d and .l forms. The .s and .d forms are added for convenience.

STORE VECTOR REGISTER/VECTOR INDEX stvi.(b|h|w|l|s|d) Vk,Vj

Purpose:

To store a vector from a vector accumulator using a vector of indices (commonly referred to as "vector scatter").

Format:

```
-----------------------------
| Opcode     |  Vj   |  Vk  |
-----------------------------
15          6,5     3,2     0
```

Operation:

```
DO 10 a = 0, (VL-1)
        temp = Vj(a<31..0>) + A5
        c(temp) = Vk(a)
10 CONTINUE
```

PSW:

Exceptions:

Opcode:

stvi.b Vk,Vj	0111101000	Index Store vector byte
stvi.h Vk,Vj	0111101001	Index Store vector halfword
stvi.w Vk,Vj	0111101010	Index Store vector word
stvi.l Vk,Vj	0111101011	Index Store vector longword
stvi.s Vk,Vj	0111101010	Index Store vector single float
stvi.d Vk,Vj	0111101011	Index Store vector double float

Description:

VL elements of a vector are stored from the specified vector accumulator, Vk. The first element is referenced by the effective address produced by adding the contents of A5 and the least significant 32 bits Vj(0). The second element is referenced by the effective address produced by adding the contents of A5 and Vj(1), and so on.

Notes:

1 A5 is typically loaded with an ldea (load effective address instruction.)

2 64-bit integers or 64-bit floating point operands are stored using the stvi.l instruction.

3 32-bit integers or 32-bit single precision floating point operands are stored using the stvi.w instruction.

4 If the distance between successive elements is less than

the precision of an element, unpredictable actions occur.

5 A5 is not changed during the execution of this instruction.

6 The contents of Vj(i) are treated as a byte offset. The appropriate shift of Vj may have to be performed to account for the precision of the operand to be stored.

7 The .s and .w forms of this instruction are equivalent, as are the .d and .l forms. The .s and .d forms are added for convenience.

Purpose:

To store a vector from a scalar accumulator using a vector of indices

Format:

```
-----------------------------
| Opcode      | Vj    | Sk  |
-----------------------------
15           6,5     3,2     0
```

Operation:

```
DO 10 a = 0,(VL-1)
        temp = Vj(a<31..0>) + A5
        c(temp) = Sk
10 CONTINUE
```

PSW:

Exceptions:

Opcode:

stvi.b Sk,Vj	0111101100	Scalar Index Store vector byte
stvi.h Sk,Vj	0111101101	Scalar Index Store vector halfword
stvi.w Sk,Vj	0111101110	Scalar Index Store vector word
stvi.l Sk,Vj	0111101111	Scalar Index Store vector longword
stvi.s Sk,Vj	0111101110	Scalar Index Store vector single float
stvi.d Sk,Vj	0111101111	Scalar Index Store vector double float

Description:

VL elements of a vector are stored from the specified scalar accumulator, Sk. The first vector element to be stored into is referenced by the effective address produced by adding the contents of A5 and the least significant 32 bits Vj(0). The second element is referenced by the effective address produced by adding the contents of A5 and Vj(1), and so on.

Notes:

1 A5 is typically loaded with an ldea (load effective address instruction.)

2 64-bit integers or 64-bit floating point operands are stored using the stvi.l instruction.

3 32-bit integers or 32-bit floating point operands are stored using the stvi.w instruction.

4 If the distance between successive elements is less than

the precision of an element, unpredictable actions occur.

5 A5 is not changed during the execution of this instruction.

6 The contents of Vj(i) are treated as a byte offset. The appropriate shift of Vj may have to be performed to account for the precision of the operand to be stored.

7 The .s and .w forms of this instruction are equivalent, as are the .d and .l forms. The .s and .d forms are added for convenience.

STORE SCALAR EXTENDED ste.(b|h|w|l|s|d) Sk,<effa>

Purpose:

To store a scalar register repetitively into memory.

Format:

```
----------------------     -------------------
| Opcode |@|L|Aj |Sk |     |   Displacement  |
----------------------     -------------------
15       8,7 6,5,3,2 0     (31|15)          0
```

Operation:

```
temp = Effective Address
DO 10 a = 0,(VL-1)
        c<temp> = Sk
        temp = temp + VS
10 CONTINUE
```

Exceptions:

Opcode:

ste.b Sk,<effa>	001001000	Store an extended scalar byte
ste.h Sk,<effa>	001001010	Store an extended scalar halfword
ste.w Sk,<effa>	001001100	Store an extended scalar word
ste.l Sk,<effa>	001001110	Store an extended scalar longword
ste.s Sk,<effa>	001001100	Store an extended scalar single float
ste.d Sk,<effa>	001001110	Store an extended scalar double float

Description:

The contents of the scalar register Sk are repetitively stored into memory until VL.

Notes:

1 This instruction is used to perform the loop construct A(I)=K.

2 Sk is unchanged at the completion of this instruction.

3 This instruction can be used to clear memory by initializing Sk with 0.

4 The .s and .w forms of this instruction are equivalent as are the .d and .l forms. The .s and .d forms are added for convenience.

ADD VECTOR/VECTOR

add.(b|h|w|l|s|d) Vi,Vj,Vk

Purpose:

To add two vectors.

Format:

```
---------------------------
| Opcode    | Vi | Vj | Vk |
---------------------------
15          9,8  6,5  3,2  0
```

Operation:

```
   DO 10 a = 0, (VL-1)
      Vk(a) = Vi(a) + Vj(a)
10 CONTINUE
```

PSW:

SIV = Integer Overflow! Integer Only
OV = Exponent Overflow! Floating Point Only
UN = Exponent Underflowz! Floating Point Only
RO = Reserved Operand! Floating Point Only

Exceptions:

Integer Overflow
Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

add.b Vi,Vj,Vk	1100000	Add vector/vector integer byte
add.h Vi,Vj,Vk	1100001	Add vector/vector integer halfword
add.w Vi,Vj,Vk	1100010	Add vector/vector integer word
add.l Vi,Vj,Vk	1100011	Add vector/vector integer longword
add.s Vi,Vj,Vk	1011000	Add vector/vector single float
add.d Vi,Vj,Vk	1011001	Add vector/vector double float

Description:

The contents of the vector register Vi are added to the contents of the vector register Vj, and the vector result is loaded into the vector register Vk. The number of elements added is determined by the value of VL at the time execution begins.

Notes:

Hold issues: functional unit and register reservation.

Purpose:

To subtract two vectors.

Format:

```
----------------------------
|  Opcode    | Vi  | Vj  | Vk |
----------------------------
15          9,8  6,5  3,2   0
```

Operation:

```
        DO 10 a = 0, (VL-1)
             Vk(a) = Vi(a) - Vj(a)
    10 CONTINUE
```

PSW:

SIV = Integer Overflow; Integer Only
OV = Exponent Overflow; Floating Point Only
UN = Exponent Underflow; Floating Point Only
RO = Reserved Operand; Floating Point Only

Exceptions:

Integer Overflow
Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

sub.b Vi,Vj,Vk	1101000	Subtract vector/vector integer byte
sub.h Vi,Vj,Vk	1101001	Subtract vector/vector integer halfword
sub.w Vi,Vj,Vk	1101010	Subtract vector/vector integer word
sub.l Vi,Vj,Vk	1101011	Subtract vector/vector integer longword
sub.s Vi,Vj,Vk	1011010	Subtract vector/vector single float
sub.d Vi,Vj,Vk	1011011	Subtract vector/vector double float

Description:

The contents of the vector register Vj are subtracted from the contents of the vector register Vi, and the vector result is loaded into the vector register Vk. The number of elements subtracted is determined by the value of VL at the time execution begins.

Notes:

MULTIPLY VECTOR/VECTOR mul.(b|h|w|l|s|d) Vi,Vj,Vk

Purpose:

To multiply two vectors.

Format:

```
---------------------------
| Opcode    | Vi | Vj | Vk |
---------------------------
15        9,8  6,5  3,2   0
```

Operation:

```
    DO 10 a = 0, (VL-1)
         Vk(a) = Vi(a) * Vj(a)
10 CONTINUE
```

PSW:

SIV = Integer Overflow; Integer only
OV = Exponent Overflow; Floating Point Only
UN = Exponent Underflow; Floating Point Only
RO = Reserved Operand; Floating Point Only

Exceptions:

Integer Overflow
Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

mul.b Vi,Vj,Vk	1110000	Multiply vector/vector integer byte
mul.h Vi,Vj,Vk	1110001	Multiply vector/vector integer halfword
mul.w Vi,Vj,Vk	1110010	Multiply vector/vector integer word
mul.l Vi,Vj,Vk	1110011	Multiply vector/vector integer longword
mul.s Vi,Vj,Vk	1001000	Multiply vector/vector single float
mul.d Vi,Vj,Vk	1001001	Multiply vector/vector double float

Description:

The contents of the vector register Vi are multiplied by the contents of the vector register Vj, and the vector result is loaded into the vector register Vk. The number of elements multiplied is determined by the value of VL at the time execution begins.

Notes:

DIVIDE VECTOR/VECTOR div.(b|h|w|l|s|d) Vi,Vj,Vk

Purpose:

To divide two vectors.

Format:

```
----------------------------
|  Opcode     | Vi | Vj | Vk |
----------------------------
15         9,8  6,5  3,2   0
```

Operation:

```
   DO 10 a = 0, (VL-1)
        Vk(a) = Vi(a) / Vj(a)
10 CONTINUE
```

PSW:

```
SIV = Integer Overflow         ! Integer Only
OV  = Exponent Overflow        ! Floating Point Only
UN  = Exponent Underflow       ! Floating Point Only
SDZ = Divide by Zero           ! Integer Only
RO  = Reserved Operand         ! Floating Point Only
FDZ = Divide by Zero           ! Floating Point Only
```

Exceptions:

Integer Overflow
Exponent Overflow
Exponent Underflow
Divide by Zero
Reserved Operand

Opcode:

div.b Vi,Vj,Vk	1111000	Divide vector/vector integer byte
div.h Vi,Vj,Vk	1111001	Divide vector/vector integer halfword
div.w Vi,Vj,Vk	1111010	Divide vector/vector integer word
div.l Vi,Vj,Vk	1111011	Divide vector/vector integer longword
div.s Vi,Vj,Vk	1001010	Divide vector/vector single float
div.d Vi,Vj,Vk	1001011	Divide vector/vector double float

Description:

The contents of the vector register Vi are divided by the contents of the vector register Vj, and the vector result is loaded into the vector register Vk. The number of elements divided is determined by the value of VL at the time execution begins.

Notes:

Hold issues: functional unit and register reservation.

NEGATE VECTOR/VECTOR neg.(b|h|w|l|s|d) Vj,Vk

Purpose:
To negate a vector.

Format:

```
-----------------------------
| Opcode      |  Vj   |  Vk  |
-----------------------------
15           6,5     3,2     0
```

Operation:

```
    DO 10 a = 0, (VL-1)
        Vk(a) = 0 - Vj(a)
10 CONTINUE
```

PSW:
SIV = Integer Overflow; Integer Only
OV = Exponent Overflow; Floating Point Only
UN = Exponent Underflow; Floating Point Only
RO = Reserved Operand; Floating Point Only

Exceptions:
Integer Overflow
Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

neg.b Vj,Vk	0110111000	Negate vector/vector integer byte
neg.h Vj,Vk	0110111001	Negate vector/vector integer halfword
neg.w Vj,Vk	0110111010	Negate vector/vector integer word
neg.l Vj,Vk	0110111011	Negate vector/vector integer longword
neg.s Vj,Vk	0110010010	Negate vector/vector single float
neg.d Vj,Vk	0110010011	Negate vector/vector double float

Description:
The algebraic negation of vector register Vj is loaded into Vk. The number of elements negated is determined by the value of VL at the time execution begins.

Notes:

ADD VECTOR/SCALAR add.(b|h|w|l|s|d) Vi,Sj,Vk

Purpose:
To add a scalar to a vector.

Format:

```
----------------------------
|  Opcode    | Vi | Sj | Vk |
----------------------------
15          9,8  6,5  3,2  0
```

Operation:

```
   DO 10 a = 0, (VL-1)
      Vk(a) = Vi(a)+Sj
10 CONTINUE
```

PSW:

SIV = Integer Overflow; Integer Only
OV = Exponent Overflow; Floating Point Only
UN = Exponent Underflow; Floating Point Only
RO = Reserved Operand; floating point only

Exceptions:

Integer Overflow
Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

add.b Vi,Sj,Vk	1100100	Add vector/scalar integer byte
add.h Vi Sj,Vk	1100101	Add vector/scalar integer halfword
add.w Vi,Sj,Vk	1100110	Add vector/scalar integer word
add.l Vi,Sj,Vk	1100111	Add vector/scalar integer longword
add.s Vi,Sj,Vk	1011100	Add vector/scalar single float
add.d Vi,Sj,Vk	1011101	Add vector/scalar double float

Description:
The contents of the scalar register Sj are added to the contents of the vector register Vi, and the vector result is loaded into the vector register Vk. The number of elements added is determined by the value of VL at the time execution begins.

Notes:

SUBTRACT VECTOR/SCALAR sub.(b|h|w|l|s|d) Vi,Sj,Vk

Purpose:

To subtract a scalar from a vector.

Format:

```
----------------------------
|  Opcode    | Vi  | Sj  | Vk |
----------------------------
15          9,8  6,5  3,2  0
```

Operation:

```
   DO 10 a = 0, (VL-1)
        Vk(a) = Vi(a) - Sj
10 CONTINUE
```

PSW:

SIV = Integer Overflow; Integer Only
OV = Exponent Overflow; Floating Point Only
UN = Exponent Underflow; Floating Point Only
RO = Reserved Operand; Floating Point Only

Exceptions:

Integer Overflow
Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

sub.b Vi,Sj,Vk	1101100	Subtract vector/scalar integer byte
sub.h Vi,Sj,Vk	1101101	Subtract vector/scalar integer halfword
sub.w Vi,Sj,Vk	1101110	Subtract vector/scalar integer word
sub.l Vi,Sj,Vk	1101111	Subtract vector/scalar integer longword
sub.s Vi,Sj,Vk	1011110	Subtract vector/scalar single float
sub.d Vi,Sj,Vk	1011111	Subtract vector/scalar double float

Description:

The contents of the scalar register Sj are subtracted from the contents of the vector register Vi, and the vector result is loaded into the vector register Vk. The number of elements subtracted is determined by the value of VL at the time execution begins.

Notes:

MULTIPLY VECTOR/SCALAR

mul.(b|h|w|l|s|d) Vi,Sj,Vk

Purpose:

To multiply a scalar with a vector.

Format:

```
----------------------------
|  Opcode    | Vi  | Sj  | Vk  |
----------------------------
15         9,8  6,5  3,2  0
```

Operation:

```
    DO 10 a = 0, (VL-1)
        Vk(a) = Vi(a) * Sj
10 CONTINUE
```

PSW:

SIV = Integer Overflow; Integer Only
OV = Exponent Overflow; Floating Point Only
UN = Exponent Underflow; Floating Point Only
RO = Reserved Operand; Floating Point Only

Exceptions:

Integer Overflow
Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

mul.b Vi,Sj,Vk	1110100	Multiply vector/scalar integer byte
mul.h Vi,Sj,Vk	1110101	Multiply vector/scalar integer halfword
mul.w Vi,Sj,Vk	1110110	Multiply vector/scalar integer word
mul.l Vi,Sj,Vk	1110111	Multiply vector/scalar integer longword
mul.s Vi,Sj,Vk	1001100	Multiply vector/scalar single float
mul.d Vi,Sj,Vk	1001101	Multiply vector/scalar double float

Description:

The contents of the scalar register Sj are multiplied with the contents of the vector register Vi, and the vector result is loaded into the vector register Vk. The number of elements multiplied is determined by the value of VL at the time execution begins.

Notes:

DIVIDE VECTOR/SCALAR div.(b|h|w|l|s|d) Vi,Sj,Vk

Purpose:

To divide a vector by a scalar.

Format:

```
----------------------------
|  Opcode     | Vi  | Sj  | Vk  |
----------------------------
15          9,8  6,5  3,2  0
```

Operation:

```
    DO 10 a = 0, (VL-1)
         Vk(a) = Vi(a) / Sj
10 CONTINUE
```

PSW:

```
SIV  = Integer Overflow ! Integer Only
OV = Exponent Overflow  ! Floating Point Only
UN = Exponent Underflow ! Floating Point Only
SDZ = Divide by Zero    ! Integer Only
RO = Reserved Operand   ! Floating Point Only
FDZ = Divide by Zero    ! Floating Point Only
```

Exceptions:

Integer Overflow
Exponent Overflow
Exponent Underflow
Divide by Zero
Reserved operand

Opcode:

div.b Vi,Sj,Vk	1111100	Divide vector/scalar integer byte
div.h Vi,Sj,Vk	1111101	Divide vector/scalar integer halfword
div.w Vi,Sj,Vk	1111110	Divide vector/scalar integer word
div.l Vi,Sj,Vk	1111111	Divide vector/scalar integer longword
div.s Vi,Sj,Vk	1001110	Divide vector/scalar single float
div.d Vi,Sj,Vk	1001111	Divide vector/scalar double float

Description:

The contents of the vector Vi are divided by the scalar Sj, and the vector result is loaded into the vector register Vk. The number of elements divided is determined by the value of VL at the time execution begins.

Notes:

Hold issues: functional unit and register reservation.

AND VECTOR/VECTOR and Vi,Vj,Vk

Purpose:
To AND the contents of two vectors.

Format:

```
-----------------------------
|  Opcode    | Vi  | Vj  | Vk |
-----------------------------
15          9,8  6,5  3,2   0
```

Operation:

```
    DO 10 a = 0, (vl-1)
       Vk(a) = Vi(a) .AND. Vj(a)
10 CONTINUE
```

PSW:

Exceptions:

Opcode:

and Vi,Vj,Vk	1010000 AND two vectors

Description:
The elements of the Vj vector register are ANDed with the elements of the Vi register. The results of the AND are loaded into Vk. The number of ANDs is equal to VL. All 64-bits of each element of a vector register participate in the operation.

Notes:
The contents of one vector register, Vi, can be moved to another vector register, Vk, by specifying Vi and Vj as the same register. Thus, for example, AND V0, V0, V1 moves the contents of V0 to V1.

OR VECTOR/VECTOR

or Vi,Vj,Vk

Purpose:

To OR the contents of two vectors.

Format:

```
 ----------------------------
 |  Opcode    | Vi | Vj | Vk |
 ----------------------------
 15          9,8  6,5  3,2   0
```

Operation:

```
    DO 10 a = 0, (VL-1)
         Vk(a) = Vi(a) .OR. Vj(a)
 10 CONTINUE
```

PSW:

Exceptions:

Opcode:

or Vi,Vj,Vk	1010001 OR two vectors

Description:

The elements of the Vj vector register are ORed with the elements of the Vi register. The results of the OR are loaded into Vk. The number of ORs is equal to VL. All 64 bits of each element of a vector register participate in the operation.

Notes:

EXCLUSIVE OR VECTOR/VECTOR

Purpose:
To Exclusive OR the contents of two vectors.

Format:

```
----------------------------
| Opcode     | Vi | Vj | Vk |
----------------------------
15          9,8  6,5  3,2   0
```

Operation:

```
    DO 10 a = 0,(VL-1)
          Vk(a) = Vi(a) .XOR. Vj(a)
10 CONTINUE
```

PSW:

Exceptions:

Opcode:

xor Vi,Vj,Vk	1010010	Exclusive OR two vectors

Description:
The elements of the Vj vector register are exclusive ORed with the elements of the Vi register. The results of the exclusive OR are loaded into Vk. The number of exclusive ORs is equal to VL. All 64 bits of each element of a vector register participate in the operation.

Notes:

Purpose:

To COMPLEMENT the contents of a vector.

Format:

```
-----------------------------
| Opcode      |  Vj   |  Vk  |
-----------------------------
15          6,5     3,2     0
```

Operation:

```
     DO 10 a = 0, (VL-1)
          Vk(a) = .NOT. Vj(a)
10 CONTINUE
```

PSW:

Exceptions:

Opcode:

not Vj,Vk	0110001011	Complement a vector

Description:

The elements of the Vj vector register are complemented and the results loaded into the Vk vector register. The number of operations is equal to VL. All 64-bits of each element of a vector register participate in the operation.

Notes:

Purpose:
To AND the contents of a vector and a scalar.

Format:
```
----------------------------
|  Opcode    | Vi | Sj | Vk |
----------------------------
15         9,8  6,5  3,2  0
```

Operation:
```
   DO 10 a = 0, (Vl-1)
      Vk(a) = Vi(a) .AND. Sj
10 CONTINUE
```

PSW:

Exceptions:

Opcode:
and Vi,Sj,Vk   1010100 AND vector/scalar

Description:
The elements of the Vi vector register are ANDed with the contents of the Sj register. The results of the AND are loaded into Vk. The number of ANDs is equal to VL. All 64-bits of each element of a vector register participate in the operation.

Notes:

Purpose:
To OR the contents of a vector and a scalar.

Format:

```
----------------------------
| Opcode     | Vi | Sj | Vk |
----------------------------
15          9,8  6,5  3,2  0
```

Operation:

```
   DO 10 a = 0, (VL-1)
        Vk(a) = Vi(a) .OR. Sj
10 CONTINUE
```

PSW:

Exceptions:

Opcode:

or Vi,Sj,Vk	1010101 OR vector/scalar

Description:
The elements of the Vj vector register are ORed with the contents of the Sj register. The results of the OR are loaded into Vk. The number of ORs is equal to VL. All 64 bits of each element of a vector register participate in the operation.

Notes:

EXCLUSIVE OR VECTOR/SCALAR xor Vi,Sj,Vk

Purpose:

To Exclusive OR the contents of a vector and a scalar

Format:

```
-----------------------------
| Opcode     | Vi | Sj | Vk |
-----------------------------
15          9,8  6,5  3,2  0
```

Operation:

```
      DO 10 a = 0,(VL-1)
           Vk(a) = Vi(a) .XOR. Sj
   10 CONTINUE
```

PSW:

Exceptions:

Opcode:

xor Vi,Sj,Vk   1010110 Exclusive OR vector/scalar

Description:

The elements of the Vi vector register are exclusive ORed with the contents of the Sj register. The results of the exclusive OR are loaded into Vk. The number of exclusive ORs is equal to VL. All 64 bits of each element of a vector register participate in the operation.

Notes:

LOGICAL SHIFT VECTOR/SCALAR

Purpose:

To logically shift the contents of a vector register by a scalar register.

Format:

```
-----------------------------
| Opcode      | Sj    | Vk  |
-----------------------------
15           6,5     3,2    0
```

Operation:

```
   DO 10 a = 0, (VL-1)
        Vk(a) = Shift Vk(a) by Sj<7..0>
10 CONTINUE
```

PSW:

Exceptions:

Opcode:

shf Sj,Vk	0110001100	Shift a vector accumulator

Description:

The contents of Vk(i) are shifted according to the contents of Sj. When Sj is positive Vk(i) is shifted left. When Sj is negative Vk(i) is shifted right. Each vector element of Vk until VL is shifted. All 64 bits of Vk(i) participate in the shift. Vacated positions are zero filled. Only Sj<7..0> are used to control the shift. Sj<63..8> are ignored.

Notes:

Arithmetic shifts are implemented using multiplies and divides.

MOVE SCALAR/VECTOR mov Si,Sj,Vk

Purpose:

To move a scalar register to a vector register element.

Format:

```
--------------------------
| Opcode | Si | Sj | Vk |
--------------------------
15      9,8  6,5  3,2   0
```

Operation:

Vk(Sj<6..0>) = Si

PSW:

Exceptions:

Opcode:

mov Si,Sj,Vk	1000001	Move a scalar to a vector element

Description:

The contents of the scalar register Si are loaded into an element of the vector register Vk. The particular Vk(i) is determined from bits<6..0> of Sj. All other bits of Sj are ignored.

Notes:

MOVE VECTOR ELEMENT/SCALAR

Purpose:

To move a vector element into a scalar register.

Format:

```
--------------------------
|  Opcode | Vi  | Sj  | Sk |
--------------------------
15        9,8  6,5  3,2   0
```

Operation:

Sk = Vi(Sj<6..0>)

PSW:

Exceptions:

Opcode:

```
mov Vi,Sj,Sk     1000000 Move a vector element to a scalar
```

Description:

The vector element in the Vi register referenced by bits<6..0> of Sj is loaded in Sk.

Notes:

CHAPTER 14

14 Comparisons/Mask/Merge/Compress Instructions

The instructions in this chapter perform comparisons between vectors and scalars, and perform manipulations using the vector merge (VM) register. The general methodology is for a comparison to produce a bit vector, where a 1 indicates that the comparison is .TRUE., and a 0 indicates that the comparison is .FALSE. This method is similar to the way comparisons with address and scalar registers function. The results of a vector compare (and consequently the contents of the VM register) are used quite differently from manipulations on the A registers, however.

On most machines, many common operations performed on vectors, such as compress, mask, and merge, require the use of branch instructions. In CONVEX machines, these operations are included as primitives in the instruction set. To eliminate the use of branch instructions when manipulating vectors, a full set of primitives is provided that implements compress, mask, and merge operations on vector accumulators. These primitives permit vector operations to be implemented with a sequence of chained vector instructions--as with vector clip, for example, where every element greater than 5 is replaced with 5. Other typical operations using these instruction set primitives include: operations on sparse vectors using a compress operation, the number and location of zero crossings; the number of successful comparisons; sorts, and others.

Three forms of compare operations are provided: .LE., .LT., and .EQ. For the other 3 operations, the following identities hold: .NOT. (V .REL. S) <=> V (.NOT. .REL.) S, where the following relations are complements:

```
.NE. <--> .EQ.
.LE. <--> .GT.
.LT. <--> .GE.
```

14.1 Vector Compares

The two instructions defined in this section are Compare Vector/Vector, (used between two vectors) and Compare Vector/Scalar (used between one vector and one scalar). Note: the instruction Compare Scalar/Scalar, covered in Chapter 10--"Scalar Register Instruction Set"--is used between two scalars. The three Compare instructions are lt, le, and eq:

```
(lt, le, eq)x(b,h,w,l,s,d)
```

There are six different data types: integer 8, 16, 32, and 64, and single and double precision floating point (32 and 64 bit). The only flag affected in the PSW is the RO bit, the Reserved Operand used for Floating Point only.

14.2 Mask/Merge/Compress

The instructions defined in this section are Compress, Merge Vector/Vector, Merge Vector/Scalar, Mask Vector/Vector, and Mask Vector/Scalar:

CPRS MERG mask

The Compress instruction uses the VM register to extract elements selectively from one vector register and place them in another. Either 0's or 1's of VM may be used by specifying the .f or .t (false or true) version of the instruction, respectively. Both Mask and Merge instructions take two input operands and produce a third operand as the result. These operands are referred to as Vi, Rj, and Vk, where Vk is the output. Rj may be either a vector or a scalar register. The Merge and Mask instructions differ only in the way in which the indices are used to create the result vector. For the Mask instruction, element n of Vk is either element n of Vi or element n of Rj. In the case of the Merge instruction, the indices of Vi and Rj are only incremented if that particular register is selected by VM.

These instructions are best described by a few examples. First, a simple rule is presented. Each instruction either has a single, "true" version, or both a true (.t) and a false (.f) version. This facility allows the user to utilize either the 1's of VM (.t) or the 0's (.f). Thus, in the .t case, when the appropriate bit of VM is a 1, the Rj operand is selected. The various combinations of VM, .t, and .f are shown in the following diagram.

```
                 VM
             0        1
    ------------------
    .t   |  Vi       Rj
         |
    .f   |  Rj       Vi
```

14.2.1 Mask/Merge/Compress Examples

Examples of compress, mask, and merge are now presented. The following values are assumed before instruction execution:

Comparisons/Mask/Merge/Compress Instructions

```
VO = 0 1 2 3 4 5

V1 = a b c d e f

VM = 0 1 1 0 0 1

VL = 6

S1 = 8
```

Performing a compress on VO produces the following:

```
1 2 5 = cprs.t VO,V5

0 3 4 = cprs.f VO,V5
```

Performing a mask of VO and V1 produces the following:

```
0 b c 3 4 f = mask.t VO,V1,V5
```

and:

```
a 1 2 d e 5 = mask.t V1,VO,V5
```

and:

```
0 8 8 3 4 8 = mask.t VO,S1,V5
```

Performing a merge of VO and V1 produces the following:

```
0 a b 1 2 c 4 5 6 d e f = merg.t VO,V1,V5
```

where VL = 12, and

```
VM = 0 1 1 0 0 1 0 0 0 1 1 1
```

Performing a merge of VO and S1 produces the following:

```
0 8 8 1 2 8 3 4 5 8 8 8 = merg.t VO,S1,V5
```

or:

```
8 0 1 8 8 2 8 8 8 3 4 5 = merg.f VO,S1,V5
```

where VL = 12, and

```
VM = 0 1 1 0 0 1 0 0 0 1 1 1
```

COMPARE VECTOR/VECTOR (le|lt|eq).(b|h|w|l|s|d|) Vj,Vk

Purpose:

To compare two vectors and load VM

Format:

```
-----------------------------
| Opcode      |  Vj    |  Vk  |
-----------------------------
15           6,5     3,2     0
```

Operation:

```
DO 10    a = 0, (VL-1)
         IF(Vj .OPCODE-TEST. Vk(a)) THEN
                  VM(a) = 1
         ELSE
                  VM(a) = 0
   10    CONTINUE
```

PSW:

R0 = Reserved Operand ; Floating Point Only

Exceptions:

Reserved Operand (floating point only)

Opcode:

le.b Vj,Vk	0110101000	Compare less than or equal byte
lt.b Vj,Vk	0110110000	Compare less than byte
eq.b Vj,Vk	0110100000	Compare equal byte
le.h Vj,Vk	0110101001	Compare less than or equal halfword
lt.h Vj,Vk	0110110001	Compare less than halfword
eq.h Vj,Vk	0110100001	Compare equal halfword
le.w Vj,Vk	0110101010	Compare less than or equal word
lt.w Vj,Vk	0110110010	Compare less than word
eq.w Vj,Vk	0110100010	Compare equal word
le.l Vj,Vk	0110101011	Compare less than or equal longword
lt.l Vj,Vk	0110110011	Compare less than longword
eq.l Vj,Vk	0110100011	Compare equal longword
le.s Vj,Vk	0110011000	Compare less than or equal single
lt.s Vj,Vk	0110011010	Compare less than single
eq.s Vj,Vk	0110010000	Compare equal single
le.d Vj,Vk	0110011001	Compare less than or equal double float
lt.d Vj,Vk	0110011011	Compare less than double float
eq.d Vj,Vk	0110010001	Compare equal double precision

Description:

The elements of the Vj vector register are signed compared with the elements of the Vk register. The results of the compare are loaded into VM. The number of compares is equal to VL. VM(n) is loaded with the result of the compare between Vk(n) and Vj(n). When VL is less than 128, the remaining bit positions of VM are reset to 0.

Notes:

1 There are no unsigned vector compares.
2 The plc instruction can be used to determine the number of successful compares.
3 By removing VM to a scalar register and performing a leading zero count, the index of the first successful compare can be performed.

COMPARE VECTOR/SCALAR (le|lt|eq).(b|h|w|l|s|d|) Sj,Vk

Purpose:

To compare a vector and a scalar and load VM

Format:

```
-----------------------------
| Opcode     |  Sj   |  Vk  |
-----------------------------
15          6,5     3,2     0
```

Operation:

```
DO 10    a = 0,(VL-1)
         IF (Sj .OPCODE-TEST. Vk(a)) THEN
                 VM(a) = 1
         ELSE
                 VM(a) = 0
   10    CONTINUE
```

PSW:

RO = Reserved Operand ; Floating Point Only

Exceptions:

Reserved Operand (floating point only)

Opcode:

le.b Sj,Vk	0110101100	Compare less than or equal byte
lt.b Sj,Vk	0110110100	Compare less than byte
eq.b Sj,Vk	0110100100	Compare equal byte
le.h Sj,Vk	0110101101	Compare less than or equal halfword
lt.h Sj,Vk	0110110101	Compare less than halfword
eq.h Sj,Vk	0110100101	Compare equal halfword
le.w Sj,Vk	0110101110	Compare less than or equal word
lt.w Sj,Vk	0110110110	Compare less than word
eq.w Sj,Vk	0110100110	Compare equal word
le.l Sj,Vk	0110101111	Compare less than or equal longword
lt.l Sj,Vk	0110110111	Compare less than longword
eq.l Sj,Vk	0110100111	Compare equal longword
le.s Sj,Vk	0110011100	Compare less than or equal single
lt.s Sj,Vk	0110011110	Compare less than single
eq.s Sj,Vk	0110010100	Compare equal single
le.d Sj,Vk	0110011101	Compare less than or equal double float
lt.d Sj,Vk	0110011111	Compare less than double float
eq.d Sj,Vk	0110010101	Compare equal double precision

Description:

Sj is signed compared with the elements of the Vk register. The results of the compare are loaded into VM. The number of compares is equal to VL. VM(n) is loaded with the result of the compare between Vk(n) and Sj. When VL is less than 128, the remaining bit positions of VM are reset to 0.

Notes:

There are no unsigned vector compares.

COMPRESS

cprs.(t|f) Vj,Vk

Purpose:

To compress a vector using VM.

Format:

```
----------------------------
| Opcode     | Vj    | Vk  |
----------------------------
15          6,5     3,2     0
```

Operation:

```
Vk = VM Compress Vj
a = 0

DO 10 b = 0,(VL-1)        !cprs.f
   IF (VM(b) .EQ. 0) THEN        !0 for false
           Vk(a) = Vj(b)
           a = a + 1
   END IF
10 CONTINUE

DO 10 b = 0,(VL-1)        !cprs.t
   IF (VM(b) .EQ. 1) THEN        !1 for true
           Vk(a) = Vj(b)
           a = a + 1
   END IF
10 CONTINUE
```

PSW:

Exceptions:

Opcode:

cprs.f Vj,Vk	0110001110	Compress a vector using not VM
cprs.t Vj,Vk	0110001111	Compress a vector using VM

Description:

The vector Vj is compressed using VM. The result is loaded into Vk. The number of elements loaded into Vk is equal to the number of 0/1's in VM, up to VL. Vj and VM are unchanged upon completion of the instruction. All 64-bits of a vector element participate in the compress operation.

Notes:

1 The compress operation provides a useful means to operate on vector elements that satisfy a constraint condition. Since compress is a pipelined operation, the

use of scalar operations or compares and branches to achieve the same result is avoided.

2 The compress operation can be useful for implementing a particular class of binary sorts.

3 The plc VM instruction should be used to determine the number of elements loaded into Vk.

MERGE VECTOR/VECTOR merg.t Vi,Vj,Vk

Purpose:

To MERGE one vector into another.

Format:

```
----------------------------
|  Opcode    | Vi  | Vj  | Vk  |
----------------------------
15          9,8  6,5  3,2   0
```

Operation:

```
a = 0
b = 0

DO 10 e = 0,(VL-1)
        IF (VM(e) .eq. 1) THEN
              Vk(e) = Vj(b)
              b = b + 1
        ELSE
              Vk(e) = Vi(a)
              a = a + 1
        END IF
10 CONTINUE
```

PSW:

Exceptions:

Opcode:

merg.t Vi,Vj,Vk 1000010 Merge vector/vector

Description:

The vectors Vi and Vj are merged into the vector Vk using VM. The number of merge operations is equal to VL. Vi, Vj, and VM are unchanged after the merge operation is completed. The merge operation moves sequential elements from either the Vi or Vj vectors to Vk according to values contain in VM.

Notes:

1 The merge provides a convenient means by which to reassemble operands from two vectors into one vector. Typically, the operands were initially scrambled using a compress operation.

2 Merge using .NOT. VM is equivalent to MERGE with Vi and Vj interchanged.

MERGE VECTOR/SCALAR merg.(t|f) Vi,Sj,Vk

Purpose:

To MERGE a scalar into a vector.

Format:

```
----------------------------
| Opcode    | Vi | Sj | Vk |
----------------------------
15         9,8  6,5  3,2  0
```

Operation:

```
b = 0
IF (MERG.T) THEN
        BIT = 1
ELSE
        BIT = 0
ENDIF
DO 10 a = 0, (VL-1)
        IF (VM(a) .EQ. BIT) THEN
               Vk(a) = Sj
        ELSE
               Vk(a) = Vi(b)
               b = b + 1
        END IF
10 CONTINUE
```

PSW:

Exceptions:

Opcode:

```
merg.t Vi,Sj,Vk 1000110 Merge vector/scalar
merg.f Vi,Sj,Vk 1000100 Merge vector/scalar using not VM
```

Description:

The scalar Sj and the vector Vi are merged into the vector Vk using VM. The number of merge operations is equal to VL. Sj, Vi, and VM are unchanged after the merge operation is completed.

Notes:

The merge provides a convenient means by which to reassemble operands into one vector. Typically, the operands were initially scrambled using a compress operation.

MASK VECTOR/VECTOR mask.t Vi,Vj,Vk

Purpose:

To MASK one vector into another.

Format:

```
----------------------------
|  Opcode    | Vi  | Vj  | Vk |
----------------------------
15          9,8  6,5  3,2   0
```

Operation:

Vk = VM MASK (Vi, Vj)

```
DO 10 a = 0, (VL-1)
        IF (VM(a) .EQ. 1) THEN
                 Vk(a)= Vj(a)
        ELSE
                 Vk(a)= Vi(a)
        END IF
10 CONTINUE
```

PSW:

Exceptions:

Opcode:

mask.t Vi,Vj,Vk 1000011 Mask vector/vector

Description:

The vectors Vi and Vj are masked into the vector Vk using VM. The number of mask operations is equal to VL. Vi, Vj, and VM are unchanged after the mask operation is completed.

Notes:

Rearranging Vi and Vj is equivalent to using .NOT. VM.

MASK VECTOR/SCALAR mask.(t|f) Vi,Sj,Vk

Purpose:

To MASK a scalar into a vector

Format:

```
----------------------------
| Opcode    | Vi | Sj | Vk |
----------------------------
15         9,8  6,5  3,2  0
```

Operation:

```
Vk = VM MASK (Vi, Sj)

        IF  (MASK.T) THEN
                BIT = 1
        ELSE
                BIT =0
        ENDIF

DO 10 a = 0,(VL-1)
        IF (VM(a) .EQ. BIT)  THEN
                Vk(a)= Sj
        ELSE
                Vk(a)= Vi(a)
        END IF
10 CONTINUE
```

PSW:

Exceptions:

Opcode:

```
mask.t Vi,Sj,Vk 1000111 Mask vector/scalar using VM
mask.f Vi,Sj,Vk 1000101 Mask vector/scalar using not VM
```

Description:

The scalar Sj and the vector Vi are masked into the vector Vk using VM. The number of mask operations is equal to VL. Sj, Vi, and VM are unchanged after the mask operation is completed.

Notes:

1 The mask scalar operation provides a convenient means to perform the vector clip operation. Typically, the clip threshold is used to compute a bit vector loaded into VM. The mask scalar operation is then used to load the threshold value into the vector elements whose values exceeded this threshold.

2 IF VM is all 1's, the scalar Sj is extended and loaded into all the elements of Vk, for mask.t. This is a convenient way of loading a scalar into all elements of a vector accumulator.

CHAPTER 15

15 Vector Reduction Instruction Set

Reduction operations reduce a vector to a scalar. There are two inputs to a reduction operator: one input is a scalar register, the other input a vector register. A scalar input is provided so that reduction operations can be performed for vectors greater than 128 elements. Mathematically, reduction operations are the SUM (sum reduction) and PROD (multiply or product reduction). Additional reduction operations are provided to implement the FORTRAN MAX and MIN intrinsics, as well as reduction using logical operators, such as AND(ALL) , OR(ANY), and XOR(PARITY). These latter reduction operations are used for a certain class of pattern recognition algorithms.

For these operations, the scalar register Sk must be initialized to the identity operand of the particular operation. Thus, for add the identity operand is 0, for multiply 1, for and all 1's, for OR all 0's, for XOR all 0's, for MAX the smallest number, and for MIN the maximum number. Other operations are provided to manipulate the VM register. These take the form of reductions on VM to determine the number of 1's or 0's.

15.1 Arithmetic Reductions

The instructions defined in this section are Sum, Product, Max, and Min Vector on the following data types: (b|h|w|l|s|d). The flags in the PSW affected by these instructions are as follows:

SIV = Integer Overflow
OV = Exponent Overflow
UN = Exponent Underflow
RO = Reserved Operand

15.2 Logical Reductions

This section supports the following instructions: OR Reduce Vector (ANY), AND Reduce Vector (ALL), and EXCLUSIVE OR Reduce Vector (PARITY). None of the flags in the PSW are affected by these instructions.

15.3 Population Count Vector

The Population Count Vector instruction affects no flags in the PSW. It returns the number of ones in each element of the input vector.

SUM VECTOR

sum.(b|h|w|l|s|d) Vk

Purpose:

To SUM all the elements of a vector.

Format:

```
--------------------------
|  Opcode        | Vk/Sk  |
--------------------------
15              3,2      0
```

Operation:

```
DO 10 a = 0, (VL-1)
        Sk = Sk + Vk(a)        ! See notes below.
10 CONTINUE
```

PSW:

SIV = Integer Overflow
UN = Exponent Underflow
OV = Exponent Overflow
RO = Reserved Operand

Exceptions:

Integer Overflow
Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

sum.b Vk	0111111000000	Sum a vector of bytes
sum.h Vk	0111111000001	Sum a vector of halfwords
sum.w Vk	0111111000010	Sum a vector of words
sum.l Vk	0111111000011	Sum a vector of longwords
sum.s Vk	0111111010000	Sum a vector of single float
sum.d Vk	0111111010001	Sum a vector of double float

Description:

The sum of the scalar register Sk and all elements of Vk until VL is calculated. The number of elements added is determined by VL, and the result is loaded into Sk.

Notes:

1 The scalar register should be initialized to zero for the first use of the summation instruction.

2 The sequence of the sum executed by the hardware is NOT

identical to the FORTRAN sequence as noted above. Please see the Hardware Reference Manual for the exact sequence of operations.

3 Since this instruction reduces a Vk and Sk into Sk, the assembler will accept either Vk or Sk as the argument to this instruction.

PRODUCT VECTOR prod.(b|h|w|l|s|d) Vk

Purpose:

To obtain the products of all the elements of a vector.

Format:

```
-------------------------
|  Opcode        | Vk/Sk  |
-------------------------
15              3,2      0
```

Operation:

```
   DO 10 a = 0, (VL-1)
        Sk = Sk * Vk(a)            ! See notes below.
10 CONTINUE
```

PSW:

SIV = Integer Overflow
OV = Exponent Overflow
UN = Exponent Underflow
RO = Reserved Operand

Exceptions:

Integer Overflow
Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

prod.b Vk	0111111011000	Multiply reduce a vector of bytes
prod.h Vk	0111111011001	Multiply reduce a vector of halfwords
prod.w Vk	0111111011010	Multiply reduce a vector of words
prod.l Vk	0111111011011	Multiply reduce a vector of longwords
prod.s Vk	0111111010010	Multiply reduce a vector of single flo
prod.d Vk	0111111010011	Multiply reduce a vector of double flo

Description:

The product of the scalar register Sk and all elements of Vk is calculated. The number of elements multiplied is determined by VL, and the result is loaded into Sk.

Notes:

1 The scalar register should be initialized to one for the first use of the multiply reduce instruction.

2 The sequence of the products performed by the hardware

is NOT identical to the FORTRAN sequence as noted above. Please refer to the Hardware Reference Manual for the exact sequence of operations.

3 Since this instruction reduces a Vk and Sk into Sk, the assembler will accept either Vk or Sk as the argument to this instruction.

MAX VECTOR

max.(b|h|w|l|s|d) Vk

Purpose:

To find the maximum element of a vector.

Format:

```
-------------------------
|  Opcode        | Vk/Sk  |
-------------------------
15             3,2      0
```

Operation:

```
DO 10   a = 0, (VL-1)
        IF (Vk(a).GT.Sk)  THEN
                Sk = Vk (a)
        ENDIF
10 CONTINUE
```

PSW:

RO = Reserved Operand

Exceptions:

Reserved Operand; floating point

Opcode:

max.b Vk	0111111001000	Max of a vector of bytes
max.h Vk	0111111001001	Max of a vector of halfwords
max.w Vk	0111111001010	Max of a vector of words
max.l Vk	0111111001011	Max of a vector of longwords
max.s Vk	0111111010100	Max of a vector of single float
max.d Vk	0111111010101	Max of a vector of double float

Description:

The maximum of Sk and all the elements of Vk is determined. The number of elements searched is determined by VL, and the maximum element is loaded into Sk.

Notes:

1 The scalar register should be initialized to the minimum value for the first use of the max instruction.

2 Since this instruction reduces a Vk and Sk into Sk, the assembler will accept either Vk or Sk as the argument to this instruction.

MIN VECTOR min.(b|h|w|l|s|d) Vk

Purpose:

To find the minimum element of a vector.

Format:

```
-------------------------
|  Opcode       | Vk/Sk  |
-------------------------
15              3,2      0
```

Operation:

```
DO 10    a = 0, (VL-1)
         IF (Vk(a) .LT. Sk) THEN
                 Sk = Vk (a)
         ENDIF
10 CONTINUE
```

PSW:

RO = Reserved Operand

Exceptions:

Reserved Operand; floating point only

Opcode:

min.b Vk	0111111001100	Min of a vector of bytes
min.h Vk	0111111001101	Min of a vector of halfwords
min.w Vk	0111111001110	Min of a vector of words
min.l Vk	0111111001111	Min of a vector of longwords
min.s Vk	0111111010110	Min of a vector of single float
min.d Vk	0111111010111	Min of a vector of double float

Description:

The minimum of Sk and all the elements of Vk is determined. The number of elements searched is determined by VL, and the minimum element is loaded into Sk.

Notes:

1 The scalar register should be initialized to the maximum value for the first use of the MIN instruction.

2 Since this instruction reduces Vk and Sk into Sk, the assembler will accept either a Vk or Sk as the argument to this instruction.

AND REDUCE VECTOR

Purpose:

To AND reduce all the elements of a vector.

Format:

```
-------------------------
|  Opcode          | Vk/Sk  |
-------------------------
15              3,2       0
```

Operation:

```
DO 10 a = 0, (VL-1)
        Sk = Sk .AND. Vk(a)
10 CONTINUE
```

PSW:

Exceptions:

Opcode:

all Vk	0111111000100	AND reduce a vector

Description:

The AND of the scalar register Sk and all elements of Vk is calculated. The number of elements ANDed is determined by VL. The result is loaded into Sk. If all the corresponding bits in Sk and Vk are a 1, then a 1 is loaded in the corresponding bit position of the result Sk.

Notes:

1 The scalar register should be initialized to one for the first use of the AND reduce instruction.

2 All 64-bits of each element participate in the reduction operation.

3 Since this instruction reduces Vk and Sk into Sk, the assembler will accept either a Vk or Sk as the argument to this instruction.

Purpose:
To OR reduce all the elements of a vector.

Format:

```
-------------------------
| Opcode        | Vk/Sk  |
-------------------------
15              3,2      0
```

Operation:

```
DO 10 a = 0, (VL-1)
        Sk = Sk .OR. Vk(a)
10 CONTINUE
```

PSW:

Exceptions:

Opcode:

any Vk	0111111000101	OR reduce a vector

Description:
The OR of the scalar register Sk and all elements of Vk is calculated . The number of elements ORed is determined by VL. The result is loaded into Sk. If any of the corresponding bits in Sk or Vk(i) is a 1, then a 1 is loaded into the corresponding bit position in the result loaded into Sk.

Notes:

1 The scalar register should be initialized to zero for the first use of the ANY instruction.

2 All 64-bits of each element participate in the reduction.

3 Since this instruction reduces Vk and Sk into Sk, the assembler will accept either a Vk or Sk as the argument to this instruction.

EXCLUSIVE OR REDUCE VECTOR

Purpose:
To EXCLUSIVE OR reduce all the elements of a vector.

Format:

```
--------------------------
|  Opcode        | Vk/Sk  |
--------------------------
15              3,2      0
```

Operation:

```
DO 10 a = 0, (VL-1)
        Sk = Sk .Exclusive OR. Vk(a)
10 CONTINUE
```

PSW:

Exceptions:

Opcode:

parity Vk	0111111000110	Exclusive OR reduce a vector

Description:
The exclusive OR of the scalar register Sk and all elements of Vk is calculated. The number of elements exclusively ORed is determined by VL. The result is loaded into Sk.

Notes:

1 The scalar register should be initialized to zero for the first use of the reduction instruction.

2 All 64 bits of each vector element participate in the operation.

3 Since this instruction reduces Vk and Sk into Sk, the assembler will accept either a Vk or Sk as the argument to this instruction.

POPULATION COUNT VECTOR plc.t Vj,Vk

Purpose:

To count the number of one's in each vector element.

Format:

```
-----------------------------
| Opcode     | Vj   | Vk   |
-----------------------------
15          6,5    3,2     0
```

Operation:

```
DO 10 a = 0, (VL-1)
        Vk(a) = 0
        DO 10 j = 0, 63
                IF (Vj<b> .EQ. 1)   THEN
                        Vk(a) = Vk(a) + 1
                ENDIF
10 CONTINUE
```

PSW:

Exceptions:

Opcode:

plc.t Vj,Vk	0110001101	Population Count of a Vector

Description:

The number of 1's in each vector element of Vj is loaded in the corresponding element of Vk. The number loaded is zero extended on the left. The number of vector elements is equal to VL.

Notes:

The parity of the population count of each element can be calculated by performing a population count and then by performing a vector and Vi,Sj,Vk with the scalar register loaded with a 1 in the least significant bit, and 0's elsewhere.

CHAPTER 16

16 VL, VS, and VM Instruction Set

This chapter is divided into two sections: VL and VS operators; and VM Operations.

16.1 VL and VS

Included in this section are the following instructions: Move Address/VL; Load VL/Immediate; Move Address/VS; Load VS Immediate; Load VS and VL; Store VS and VL; Move Scalar/VL, and Move Scalar/VS. None of the flags in the PSW are affected by these operations.

16.2 VM Operations

The VM operations include the following: Load VM; Store VM; Population Count VM, and Move VM/Scalar.

MOVE ADDRESS/VL

Purpose:

To move the contents of an address register to the vector length register, VL.

Format:

```
----------------------------
| Opcode              | Ak  |
----------------------------
15                  3,2    0
```

Operation:

```
IF (Ak .GE. 128) THEN ; mov Ak,VL
        VL = 128
ELSE
        IF (Ak .LT. 0) THEN
           VL = 0
        ELSE
           VL = Ak<6..0>
        END IF
END IF

Ak = VL ! mov VL,Ak
```

PSW:

Exceptions:

Opcode:

mov Ak,VL	0111110110011	Move Ak to VL
mov VL,Ak	0111110110010	Move VL to Ak

Description:

The contents of the A register are moved to VL. If the contents of Ak are greater than 128, then 128 is loaded into VL. If the contents of Ak are less than 128, then Ak is loaded into VL. When VL is moved to Ak, Ak<31..7> are loaded with 0.

Notes:

LOAD VL/IMMEDIATE ld.w #N,VL

Purpose:
To load the vector length register with an immediate

Format:

```
-----------------------------  --------------------
| Opcode  |L| 000 |  Ak    |  |        N          |
-----------------------------  --------------------
15,        6,5   3,2       0  31|16               0
```

Operation:

```
IF (Immediate .GE. 128) THEN
        VL = 128
ELSEIF (Aj .LT. 0) THEN
                VL = 0
        ELSE
                VL = Immediate<6..0>
        END IF
END IF
```

PSW:

Exceptions:

Opcode:

ld.w #N,VL	000110000	Load VL with an immediate

Description:
The immediate field is used to load VL. If the immediate field is greater than 128 then 128 is loaded into VL. If the immediate is less than 128 then immediate<6..0> is loaded into VL.

Notes:

MOVE ADDRESS/VS

Purpose:

To move the contents of an address register to/from the vector stride register, VS.

Format:

```
----------------------------
| Opcode             | Ak   |
----------------------------
15                  3,2     0
```

Operation:

```
VS = Ak ! mov Ak,VS
Ak = VS ! mov VS,Ak
```

PSW:

Exceptions:

Opcode:

mov VS,Ak	0111110110000	Move VS to Ak
mov Ak,VS	0111110110001	Move Ak to VS

Description:

The contents of the A register are moved to/from VS.

Notes:

LOAD VS/IMMEDIATE ld.w #N,VS

Purpose:
To load the vector stride register from an immediate

Format:

```
----------------------------  --------------------
| Opcode  |L| 000 |  Ak    |  |        N         |
----------------------------  --------------------
15,        6,5   3,2      0  31|16               0
```

Operation:
VS = Immediate

PSW:

Exceptions:

Opcode:

ld.w #N,VS	000110001	Load VS from an immediate

Description:
The immediate field is loaded into VS.

Notes:

LOAD VS AND VL   ld.l <effa>,VLS

Purpose:

To load the vector stride register and the vector length register from memory

Format:

```
----------------------------    ------------------
| Opcode  |@|L| Aj  | Ak  |    | Displacement  |
----------------------------    ------------------
15        8,7,6,5   3,2   0    (31,15)          0
```

Operation:

```
VS = c(Effective Address<63..31>)  ! VS loaded from higher order 32 bit
VL = c(Effective Address<31..0>)   ! VL loaded from lower order 32 bits
```

PSW:

Exceptions:

Opcode:

```
ld.l <effa>,VLS 000010100       Load VS and VL from memory
```

Description:

The 64 bit operand in memory is fetched. Bits<63..32> are loaded into the vector stride register, VS. Bits<31..0> are loaded into the vector length register, VL.

Notes:

1 VL is unconditionally loaded with the exact contents of memory.

STORE VS AND VL　　　st.l VLS, <effa>

Purpose:
To store the vector stride register and the vector length register to memory

Format:

```
----------------------------    ------------------
| Opcode    |@|L| Aj   | Ak  |    | Displacement  |
----------------------------    ------------------
15          8,7,6,5   3,2   0    (31,15)           0
```

Operation:

```
c(Effective Address<63..31>) = VS ! VS stored into higher order 32 bits
c(Effective Address<31..0>)  = VL ! VL stored into lower order 32 bits
```

PSW:

Exceptions:

Opcode:

st.l VLS,<effa>	000011100	Store VS and VL to memory

Description:
A 64-bit operand is stored in memory. Bits<63..32> are stored from the vector stride register, VS. Bits<31..0> are stored from the vector length register, VL.

Notes:

Purpose:

To move the contents of Sk to VL.

Format:

```
-----------------------------
| Opcode             | Ak   |
-----------------------------
15                  3,2    0
```

Operation:

```
IF (Sk .GE 128) THEN !  mov.w Sk VL
                VL = 128
ELSE
        IF (Sk .LT. 0) THEN
                VL = 0
        ELSE
                VL = Sk<6..0>
        ENDIF
ENDIF
```

PSW:

Opcode:

mov.w Sk,VL	0111110110111	Move Sk to VL

Description:

The least significant 32 bits of Sk (Sk<31..0>) are moved to VL.

Purpose:
    To move the contents of Sk to VS.

Format:

```
----------------------------
|  Opcode           |  Ak   |
----------------------------
15                 3,2     0
```

Operation:
    VS = Sk ! mov.w, Sk, VS

PSW:

Opcode:

mov.w Sk,VS	0111110110101	Move Sk to VS

Description:
    The least significant 32 bits of Sk (Sk<31..0>) are moved to VS.

LOAD VM

ld.x <effa>, VM

Purpose:

To load the VM register from memory

Format:

```
----------------------------     ------------------
| Opcode  |@|L| Aj  | Ak  |      | Displacement   |
----------------------------     ------------------
15         8,7,6,5  3,2   0      (31,15)          0
```

Operation:

VM = c(Effective Address<127..0>) ! Load 128 bits beginning at <effa>

PSW:

Exceptions:

Opcode:

ld.x <effa>, VM	000010110	Load VM from memory

Description:

The 128 bits (16 bytes), beginning at the effective address are loaded into VM.

Notes:

1 VM <127..120> are loaded from the byte referenced by the effective address. VM <7..0> are loaded from the byte referenced by effective address + 15.

STORE VM st.x VM,<effa>

Purpose:
To store the VM register into memory

Format:

```
----------------------------    -----------------
| Opcode  |@|L| Aj  | Ak  |    | Displacement  |
----------------------------    -----------------
15        8,7,6,5  3,2   0     (31,15)        0
```

Operation:
c(Effective Address<127..0>) = VM ! Store 128 bits beginning at Effective Address.

PSW:

Exceptions:

Opcode:

st.x VM,<effa>	000011110	store VM into memory

Description:
VM is stored into the 128 bits (16 bytes), beginning at the effective address.

Notes:

1 VM <127..120> are stored in the byte referenced by the effective address. VM <7..0> are stored in the byte referenced by the effective address + 15.

POPULATION COUNT VM plc. (t|f) VM,Sk

Purpose:

To load the number of ones or zeros in VM into an S register.

Format:

```
--------------------------
|  Opcode          | Sk  |
--------------------------
15                 3,2  0
```

Operation:

```
IF (PLC.T) THEN
        DO 10  a = 0, (VL-1)
                Sk = Sk + VM<a>
        10 CONTINUE
ELSE
        DO 10  a = 0, (VL-1)
                Sk = Sk + .NOT. VM<a>
        10 CONTINUE
ENDIF
```

Exceptions:

Opcode:

plc.f VM,Sk	0111111011100	Load the number of 0's in VM into Sk
plc.t VM,Sk	0111111011101	Load the number of 1's in VM into Sk

Description:

The number of 1's or 0's in VM until VL is loaded into Sk, bits 6..0. All other Sk bits are reset to 0.

Notes:

These instructions typically determine the number of successful compare operations performed. The VM set by the compare could also be used to compress a vector register, and then further processing would use a VL value determined by the plc VM instructions.

MOVE VM/SCALAR mov Sj,VM,Sk

Purpose:
    To move VM to and from a scalar register.

Format:

```
---------------------------
| Opcode          | Sj | Sk |
---------------------------
15               6,5  3,2  0
```

Operation:

```
Sk = VM(Sj)   ! use Sj<0> to index VM<127..64> or VM <63..0>
VM(Sj) = Sk   ! move Sk to VM
```

PSW:

Exceptions:

Opcode:

mov Sj,Sk,VM	0110000100	Load VM(Sj) from Sk.
mov Sj,VM,Sk	0110000101	Load Sk from VM

Description:
    The contents of Sj determine which part of VM is manipulated. If the least significant bit in Sj is a 0, VM<63..0> are manipulated. If Sj is a 1, VM<127..63> are manipulated. Sk is moved to and from VM(Sj). Bits<63..1> of Sj are ignored.

Notes:
    Typically, a move of VM into the Sk scalar register is used to determine all of the elements that satisfy a multi-value logical relation. These logical relations may involve zero crossing algorithms or all elements that are between several boundary conditions (greater than x and less than y).

APPENDIX A
NOTATIONAL CONVENTIONS

This text utilizes the notational conventions listed below:

- o Bit numbering is right to left, 0 through N-1. The most significant numerical bit is N-1, the least significant 0. In essence, the bit numbering represents the binary weight of a position.
- o Bit fields are specified using the following convention:

  REG<15..0>

  where the bit field is REG from bits 15 through 0.
- o Individual bit positions within a register are denoted by specific positions separated by commas. For example, REG<15,4,0> denotes bits 15, 4, and 0 of REG.
- o Byte numbering is from left to right.
- o A bit is a single binary value or entity.
- o A byte is 8 bits.
- o A halfword is 16 bits.
- o A word is 32 bits.
- o A longword is 64 bits.
- o Single precision is a 32 bit floating point word.
- o Double precision is a 64 bit floating point longword.
- o An instruction is a multi-halfword operand.
- o A register is a programmer visible hardware storage element internal to the processor.
- o Main memory or physical memory is the physical storage present in the computer system.
- o Logical or virtual memory or memory is the perceived amount of main memory as seen by the application programmer.
- o The symbol K is an abbreviation for 1,024.
- o The symbol M is an abbreviation for 1,048,576.
- o The symbol G is an abbreviation for 1,073,741,824.
- o TBD means to be determined.
- o A stack is a linked-list group of words. A stack is useful for dynamic allocation and deallocation of memory.
- o A return Block is a collection of registers that is pushed or popped from a stack in response to an instruction or other event.

Where used in the document, the terms "reserved" or "undefined" are meant to convey to the hardware and software engineer what to expect, if anything, from unused fields in registers. The programming of algorithms which are based on the use of undefined or reserved fields is not recommended.

Wherever feasible, the FORTRAN language (F'77 and 8x) will be used as a metalanguage to describe algorithms. All of the proper FORTRAN syntax and semantics will be used. For example, the symbol "G" as defined above is represented by 2**30 in FORTRAN. Comments on a line are indicated with a ";" preceding the comment.

APPENDIX B
OP CODES SORTED BY NUMBER

0x0000	11-7	exit	Error Exit Instruction
0x0100	11-7	jmp <effa>	Jump Always
0x0200	11-7	jmpi.f <effa>	Jump on ION false
0x0300	11-7	jmpi.t <effa>	Jump on ION true
0x0400	11-7	jmpa.f <effa>	Jump on address carry false
0x0500	11-7	jmpa.t <effa>	Jump on address carry true
0x0600	11-7	jmps.f <effa>	Jump on scalar carry false
0x0700	11-7	jmps.t <effa>	Jump on scalar carry true
0x0900	9-26	ldea <effa>,Ak	Load effective address
0x0a00	16-6	ld.l <effa>,VLS	Load VS and VL from memory
0x0b00	16-10	ld.x <effa>, VM	Load VM from memory
0x0c00	9-32	tas <effa>	Test and Set a memory byte
0x0d00	9-27	pshea <effa>	Push effective address
0x0e00	16-7	st.l VLS,<effa>	Store VS and VL to memory
0x0f00	16-11	st.x VM,<effa>	store VM into memory
0x1000	12-20	halt #N,Ak	Halt the central processing unit
0x1008	10-6	ld.d #N,Sk	Load immediate, most significant bits
0x1080	11-16	sysc #r,#g	Perform a system call
0x1088	10-6	ld.du #N,Sk	Load 64 bit floating immed., upper half
0x1088	10-6	ld.lu #N,Sk	Load 64 bit integer immed., upper half
0x1088	10-6	ld.u #N,Sk	Load immediate, upper half
0x1100	9-6	ld.h #N,Ak	Load halfword imm. into Ak
0x1108	10-6	ld.l #N,Sk	Load 64 bit sign extended immediate
0x1180	9-6	ld.w #N,Ak	Load imm. into Ak
0x1188	10-6	ld.dl #N,Sk	Load 64 bit floating immed., lower half
0x1188	10-6	ld.ll #N,Sk	Load 64 bit integer immed., lower half
0x1188	10-6	ld.w #N,Sk	Load a 32 bit immediate
0x1200	9-21	and #N,Ak	AND imm. to addr. reg.
0x1208	10-21	and #N,Sk	AND scalar/immediate
0x1280	9-22	or #N,Ak	OR imm. to addr. reg.
0x1288	10-22	or #N,Sk	OR scalar/immediate
0x1300	9-23	xor #N,AK	Exclusive OR imm. to addr. reg.
0x1308	10-23	xor #N,Sk	Exclusive OR scalar/immediate
0x1380	9-25	shf #N,AK	Logical shift imm. to addr. reg.
0x1388	10-24	shf #N,Sk	Shift Scalar/immediate
0x1400	9-12	add.h #N,Ak	Add imm. address halfword
0x1408	10-17	add.h #N,Sk	Add scalar/immed. integer halfword
0x1480	9-12	add.w #N,Ak	Add imm. address word
0x1488	10-17	add.w #N,Sk	Add scalar/immed. integer word
0x1500	9-13	sub.h #N,Ak	Subtract imm. address halfword
0x1508	10-18	sub.h #N,Sk	Subtract scalar/immed. integer halfword
0x1580	9-13	sub.w #N,Ak	Subtract imm. address word
0x1588	10-18	sub.w #N,Sk	Subtract scalar/immed. integer word
0x1600	9-14	mul.h #N,Ak	Multiply imm. address halfword
0x1608	10-19	mul.h #N,Sk	Multiply scalar/immed. integer halfword
0x1680	9-14	mul.w #N,Ak	Multiply imm. address word
0x1688	10-19	mul.w #N,Sk	Multiply scalar/immed. integer word
0x1700	9-15	div.h #N,Ak	Divide imm. address halfword
0x1708	10-20	div.h #N,Sk	Divide scalar/scalar integer halfword
0x1780	9-15	div.w #N,Ak	Divide imm. address word
0x1788	10-20	div.w #N,Sk	Divide scalar/scalar integer word

0x1800	16-3	ld.w #N,VL	Load VL with an immediate
0x1808	10-17	add.s #N,Sk	Add scalar/immed. single float
0x1880	16-5	ld.w #N,VS	Load VS from an immediate
0x1888	10-18	sub.s #N,Sk	Subtract scalar/immed. single float
0x1908	10-19	mul.s #N,Sk	Multiply scalar/immed. single float
0x1988	10-20	div.s #N,Sk	Divide scalar/scalar single float
0x1a08	10-31	le.s #N,Sk	Compare less than or equal single
0x1a88	10-31	lt.s #N,Sk	Compare less than single
0x1b00	9-41	eq.h #N,Ak	Compare equal halfword
0x1b08	10-31	eq.h #N,Sk	Compare equal halfword
0x1b80	9-41	eq.w #N,Ak	Compare equal word
0x1b88	10-31	eq.w #N,Sk	Compare equal word
0x1c00	9-42	leu.h #N,Ak	Compare unsigned less than halfword
0x1c08	10-33	leu.h #N,Sk	Compare unsigned less than or equal halfword
0x1c80	9-42	leu.w #N,Ak	Compare unsigned less than word
0x1c88	10-33	leu.w #N,Sk	Compare unsigned less than or equal word
0x1d00	9-42	ltu.h #N,Ak	Compare unsigned less than halfword
0x1d08	10-33	ltu.h #N,Sk	Compare unsigned less than halfword
0x1d80	9-43	ltu.w #N,Ak	Compare unsigned less than word
0x1d88	10-33	ltu.w #N,Sk	Compare unsigned less than word
0x1e00	9-41	le.h #N,Ak	Compare less than or equal halfword
0x1e08	10-31	le.h #N,Sk	Compare less than or equal halfword
0x1e80	9-41	le.w #N,Ak	Compare less than or equal word
0x1e88	10-31	le.w #N,Sk	Compare less than or equal word
0x1f00	9-41	lt.h #N,Ak	Compare less than halfword
0x1f08	10-31	lt.h #N,Sk	Compare less than halfword
0x1f80	9-41	lt.w #N,Ak	Compare less than word
0x1f88	10-31	lt.w #N,Sk	Compare less than word
0x2000	11-10	call <effa>	Call a subroutine, make a long frame
0x2100	11-11	calls <effa>	Call a subroutine, make a short frame
0x2200	11-14	callq <effa>	Push the program counter and jump
0x2400	13-18	ste.b Sk,<effa>	Store an extended scalar byte
0x2500	13-18	ste.h Sk,<effa>	Store an extended scalar halfword
0x2600	13-18	ste.s Sk,<effa>	Store an extended scalar single float
0x2600	13-18	ste.w Sk,<effa>	Store an extended scalar word
0x2700	13-18	ste.d Sk,<effa>	Store an extended scalar double float
0x2700	13-18	ste.l Sk,<effa>	Store an extended scalar longword
0x2800	9-4	ld.b <effa>,Ak	Load addr. reg. byte
0x2900	9-4	ld.h <effa>,Ak	Load addr. reg. halfword
0x2a00	9-4	ld.w <effa>,Ak	Load addr. reg. word
0x2c00	9-5	st.b Ak,<effa>	Store addr. reg. byte
0x2d00	9-5	st.h Ak,<effa>	Store addr. reg. halfword
0x2e00	9-5	st.w Ak,<effa>	Store addr. reg. word
0x3000	10-4	ld.b <effa>,Sk	Load scalar byte
0x3100	10-4	ld.h <effa>,Sk	Load scalar halfword
0x3200	10-4	ld.s <effa>,Sk	Load scalar single float
0x3200	10-4	ld.w <effa>,Sk	Load scalar word
0x3300	10-4	ld.d <effa>,Sk	Load scalar double float
0x3300	10-4	ld.l <effa>,Sk	Load scalar longword
0x3400	10-5	st.b Sk,<effa>	Store scalar byte
0x3500	10-5	st.h Sk,<effa>	Store scalar halfword
0x3600	10-5	st.s Sk,<effa>	Store scalar single float
0x3600	10-5	st.w Sk,<effa>	Store scalar word
0x3700	10-5	st.d Sk,<effa>	Store scalar double float

0x3700	10-5	st.l Sk,<effa>	Store scalar longword
0x3800	13-8	ld.b <effa>,Vk	Load vector byte
0x3900	13-8	ld.h <effa>,Vk	Load vector halfword
0x3a00	13-8	ld.s <effa>,Vk	Load vector single float
0x3a00	13-8	ld.w <effa>,Vk	Load vector word
0x3b00	13-8	ld.d <effa>,Vk	Load vector double float
0x3b00	13-8	ld.l <effa>,Vk	Load vector longword
0x3c00	13-10	st.b Vk,<effa>	Store vector byte
0x3d00	13-10	st.h Vk,<effa>	Store vector halfword
0x3e00	13-10	st.s Vk,<effa>	Store vector single float
0x3e00	13-10	st.w Vk,<effa>	Store vector word
0x3f00	13-10	st.d Vk,<effa>	Store vector double float
0x3f00	13-10	st.l Vk,<effa>	Store vector longword
0x4000	9-44	cvtw.b Aj,Ak	Convert word to byte
0x4040	9-44	cvtw.h Aj,Ak	Convert word to halfword
0x4080	9-44	cvtb.w Aj,Ak	Convert byte to word
0x40c0	9-44	cvth.w Aj,AK	Convert half to word
0x4100	10-34	cvtw.b Sj,Sk	Convert word to byte
0x4140	10-34	cvtw.h Sj,Sk	Convert word to halfword
0x4180	10-34	cvtb.w Sj,Sk	Convert byte to word
0x41c0	10-34	cvth.w Sj,Sk	Convert halfword to word
0x4200	10-34	cvtw.s Sj,Sk	Convert word to single float
0x4240	10-34	cvts.w Sj,Sk	Convert single float to word
0x4280	10-34	cvtd.s Sj,Sk	Convert double float to single float
0x42c0	10-34	cvts.d Sj,Sk	Convert single float to double float
0x4300	10-34	cvts.l Sj,Sk	Convert single float to longword
0x4340	10-34	cvtd.l Sj,Sk	Convert double float to longword
0x4380	10-34	cvtl.s Sj,Sk	Convert longword to single float
0x43c0	10-34	cvtl.d Sj,Sk	Convert longword to double float
0x4400	9-33	ldpa Aj,Ak	Load a physical byte address into Ak
0x4440	9-25	shf #n,Ak	Logical shift left short imm.
0x4480	9-6	ld.h #n,Ak	Load short imm. into Ak
0x44c0	9-6	ld.w #n,Ak	Load short imm. into Ak
0x4500	10-34	cvtl.w Sj,Sk	Convert longword to word
0x4540	10-34	cvtw.l Sj,Sk	Convert word to longword
0x4580	10-38	plc.t Sj,Sk	Count the number of 1's in Sj
0x45c0	10-37	tzc Sj,Sk	Count of trailing zeroes in Sj
0x4600	9-37	eq.h Aj,Ak	Compare equal halfword
0x4640	9-37	eq.w Aj,Ak	Compare equal word
0x4680	9-40	eq.h #n,Ak	Compare equal halfword
0x46c0	9-40	eq.w #n,Ak	Compare equal word
0x4700	10-27	eq.b Sj,Sk	Compare equal byte
0x4740	10-27	eq.h Sj,Sk	Compare equal halfword
0x4780	10-27	eq.w Sj,Sk	Compare equal word
0x47c0	10-27	eq.l Sj,Sk	Compare equal longword
0x4800	9-39	leu.h Aj,Ak	Compare unsigned less than or equal halfword
0x4840	9-39	leu.w Aj,Ak	Compare unsigned less or equal than word
0x4880	9-42	leu.h #n,Ak	Compare unsigned less than or equal halfword
0x48c0	9-42	leu.w #n,Ak	Compare unsigned less than or equal word
0x4900	10-29	leu.b Sj,Sk	Compare less than or equal byte
0x4940	10-29	leu.h Sj,Sk	Compare less than or equal halfword
0x4980	10-29	leu.w Sj,Sk	Compare less than or equal word
0x49c0	10-29	leu.l Sj,Sk	Compare less than or equal longword
0x4a00	9-39	ltu.h gt,Aj,Ak	Compare unsigned less than halfword

0x4a40	9-39	ltu.w gt,Aj,Ak	Compare unsigned less than word
0x4a80	9-42	ltu.h #n,Ak	Compare unsigned less than halfword
0x4ac0	9-42	ltu.w #n,Ak	Compare unsigned less than word
0x4b00	10-29	ltu.b Sj,Sk	Compare less than byte
0x4b40	10-29	ltu.h Sj,Sk	Compare less than halfword
0x4b80	10-29	ltu.w Sj,Sk	Compare less than word
0x4bc0	10-29	ltu.l Sj,Sk	Compare less than longword
0x4c00	9-37	le.h Aj,Ak	Compare less than or equal signed halfword
0x4c40	9-37	le.w Aj,AK	Compare less than or equal signed word
0x4c80	9-40	le.h #n,Ak	Compare less than or equal halfword
0x4cc0	9-40	le.w #n,Ak	Compare less than or equal word
0x4d00	10-27	le.b Sj,Sk	Compare less than or equal byte
0x4d40	10-27	le.h Sj,Sk	Compare less than or equal halfword
0x4d80	10-27	le.w Sj,Sk	Compare less than or equal word
0x4dc0	10-27	le.l Sj,Sk	Compare less than or equal longword
0x4e00	9-37	lt.h Aj,Ak	Compare less than signed halfword
0x4e40	9-37	lt.w Aj,Ak	Compare less than signed word
0x4e80	9-40	lt.h #n,Ak	Compare less than halfword
0x4ec0	9-40	lt.w #n,Ak	Compare less than word
0x4f00	10-27	lt.b Sj,Sk	Compare less than byte
0x4f40	10-27	lt.h Sj,Sk	Compare less than halfword
0x4f80	10-27	lt.w Sj,Sk	Compare less than word
0x4fc0	10-27	lt.l Sj,Sk	Compare less than longword
0x5000	9-16	add.w Sj,Ak	Add scalar to addr word
0x5040	9-24	shf Aj,Ak	Shift an address
0x5080	9-29	mov Aj,Ak	Move addr. reg.
0x50c0	10-39	mov Sj,Ak	Move 32 bits of Sj into Ak.
0x5100	10-41	mov.s Sj,Sk	Move scalar register double float
0x5100	10-41	mov.w Sj,Sk	Move scalar register word
0x5140	10-36	shf Sj,Sk	Shift a scalar
0x5180	10-41	mov.d Sj,Sk	Move scalar register single float
0x5180	10-41	mov.l Sj,Sk	Move scalar register longword
0x51c0	10-40	mov Aj,Sk	Move an address to a scalar
0x5200	9-17	and Aj,Ak	AND addr. reg.
0x5240	9-18	or Aj,Ak	OR addr. reg.
0x5280	9-19	xor Aj,Ak	Exclusive OR addr. reg.
0x52c0	9-20	not Aj,Ak	Complement addr. reg.
0x5300	10-13	and Sj,Sk	AND scalar/scalar
0x5340	10-14	or Sj,Sk	OR scalar/scalar
0x5380	10-15	xor Sj,Sk	Exclusive OR scalar/scalar
0x53c0	10-16	not Sj,Sk	Complement scalar/scalar
0x5400	10-27	le.s Sj,Sk	Compare less than or equal single float
0x5440	10-27	le.d Sj,Sk	Compare less than or equal double float
0x5480	10-27	lt.s Sj,Sk	Compare less than single float
0x54c0	10-27	lt.d Sj,Sk	Compare less than double float
0x5500	10-8	add.s Sj,Sk	Add scalar/scalar single float
0x5540	10-8	add.d Sj,Sk	Add scalar/scalar double float
0x5580	10-9	sub.s Sj,Sk	Subtract scalar/scalar single float
0x55c0	10-9	sub.d Sj,Sk	Subtract scalar/scalar double float
0x5600	10-27	eq.s Sj,Sk	Compare equal single float
0x5640	10-28	eq.d Sj,Sk	Compare equal double float
0x5680	9-11	neg.h Aj,Ak	Negate addr. reg. halfword
0x56c0	9-11	neg.w Aj,Ak	Negate addr. reg. word
0x5700	10-10	mul.s Sj,Sk	Multiply scalar/scalar single float

0x5740	10-10	mul.d Sj,Sk	Multiply scalar/scalar double float
0x5780	10-11	div.s Sj,Sk	Divide scalar/scalar single float
0x57c0	10-11	div.d Sj,Sk	Divide scalar/scalar double float
0x5800	9-7	add.h Aj,Ak	Add addr. reg. halfword
0x5840	9-7	add.w Aj,Ak	Add addr. reg. word .
0x5880	9-12	add.h #n,Ak	Add short imm. address halfword
0x58c0	9-12	add.w #n,Ak	Add short imm. address word
0x5900	10-8	add.b Sj,Sk	Add scalar/scalar integer byte
0x5940	10-8	add.h Sj,Sk	Add scalar/scalar integer halfword
0x5980	10-8	add.w Sj,Sk	Add scalar/scalar integer word
0x59c0	10-8	add.l Sj,Sk	Add scalar/scalar integer longword
0x5a00	9-8	sub.h Aj,Ak	Subtract addr. reg. halfword
0x5a40	9-8	sub.w Aj,Ak	Subtract addr. reg. word
0x5a80	9-13	sub.h #n,Ak	Subtract short imm. address halfword
0x5ac0	9-13	sub.w #n,AK	Subtract short imm. address word
0x5b00	10-9	sub.b Sj,Sk	Subtract scalar/scalar integer byte
0x5b40	10-9	sub.h Sj,Sk	Subtract scalar/scalar integer halfword
0x5b80	10-9	sub.w Sj,Sk	Subtract scalar/scalar integer word
0x5bc0	10-9	sub.l Sj,Sk	Subtract scalar/scalar integer longword
0x5c00	9-9	mul.h Aj,Ak	Multiply addr. reg. halfword
0x5c40	9-9	mul.w Aj,AK	Multiply addr. reg. word
0x5c80	9-14	mul.h #n,Ak	Multiply short imm. address halfword
0x5cc0	9-14	mul.w #n,Ak	Multiply short imm. address word
0x5d00	10-10	mul.b Sj,Sk	Multiply scalar/scalar integer byte
0x5d40	10-10	mul.h Sj,Sk	Multiply scalar/scalar integer halfword
0x5d80	10-10	mul.w Sj,Sk	Multiply scalar/scalar integer word
0x5dc0	10-10	mul.l Sj,Sk	Multiply scalar/scalar integer longword
0x5e00	9-10	div.h Aj,Ak	Divide addr. reg. halfword
0x5e40	9-10	div.w Aj,Ak	Divide addr. reg. word
0x5e80	9-15	div.h #n,Ak	Divide short imm. address halfword
0x5ec0	9-15	div.w #n,Ak	Divide short imm. address word
0x5f00	10-11	div.b Sj,Sk	Divide scalar/scalar integer byte
0x5f40	10-11	div.h Sj,Sk	Divide scalar/scalar integer halfword
0x5f80	10-11	div.w Sj,Sk	Divide scalar/scalar integer word
0x5fc0	10-11	div.l Sj,Sk	Divide scalar/scalar integer longword
0x6100	16-13	mov Sj,Sk,VM	Load VM(Sj) from Sk.
0x6140	16-13	mov Sj,VM,Sk	Load Sk from VM
0x62c0	13-34	not Vj,Vk	Complement a vector
0x6300	13-38	shf Sj,Vk	Shift a vector accumulator
0x6340	15-11	plc.t Vj,Vk	Population Count of a Vector
0x6380	14-8	cprs.f Vj,Vk	Compress a vector using not VM
0x63c0	14-8	cprs.t Vj,Vk	Compress a vector using VM
0x6400	14-4	eq.s Vj,Vk	Compare equal single
0x6440	14-5	eq.d Vj,Vk	Compare equal double precision
0x6480	13-25	neg.s Vj,Vk	Negate vector/vector single float
0x64c0	13-25	neg.d Vj,Vk	Negate vector/vector double float
0x6500	14-6	eq.s Sj,Vk	Compare equal single
0x6540	14-7	eq.d Sj,Vk	Compare equal double precision
0x6580	10-12	neg.s Sj,Sk	Negate scalar/scalar single float
0x65c0	10-12	neg.d Sj,Sk	Negate scalar/scalar double float
0x6600	14-4	le.s Vj,Vk	Compare less than or equal single
0x6640	14-4	le.d Vj,Vk	Compare less than or equal double float
0x6680	14-4	lt.s Vj,Vk	Compare less than single
0x66c0	14-4	lt.d Vj,Vk	Compare less than double float

0x6700	14-6	le.s Sj,Vk	Compare less than or equal single
0x6740	14-6	le.d Sj,Vk	Compare less than or equal double float
0x6780	14-6	lt.s Sj,Vk	Compare less than single
0x67c0	14-6	lt.d Sj,Vk	Compare less than double float
0x6800	14-4	eq.b Vj,Vk	Compare equal byte
0x6840	14-4	eq.h Vj,Vk	Compare equal halfword
0x6880	14-4	eq.w Vj,Vk	Compare equal word
0x68c0	14-4	eq.l Vj,Vk	Compare equal longword
0x6900	14-6	eq.b Sj,Vk	Compare equal byte
0x6940	14-6	eq.h Sj,Vk	Compare equal halfword
0x6980	14-6	eq.w Sj,Vk	Compare equal word
0x69c0	14-6	eq.l Sj,Vk	Compare equal longword
0x6a00	14-4	le.b Vj,Vk	Compare less than or equal byte
0x6a40	14-4	le.h Vj,Vk	Compare less than or equal halfword
0x6a80	14-4	le.w Vj,Vk	Compare less than or equal word
0x6ac0	14-4	le.l Vj,Vk	Compare less than or equal longword
0x6b00	14-6	le.b Sj,Vk	Compare less than or equal byte
0x6b40	14-6	le.h Sj,Vk	Compare less than or equal halfword
0x6b80	14-6	le.w Sj,Vk	Compare less than or equal word
0x6bc0	14-6	le.l Sj,Vk	Compare less than or equal longword
0x6c00	14-4	lt.b Vj,Vk	Compare less than byte
0x6c40	14-4	lt.h Vj,Vk	Compare less than halfword
0x6c80	14-4	lt.w Vj,Vk	Compare less than word
0x6cc0	14-4	lt.l Vj,Vk	Compare less than longword
0x6d00	14-6	lt.b Sj,Vk	Compare less than byte
0x6d40	14-6	lt.h Sj,Vk	Compare less than halfword
0x6d80	14-6	lt.w Sj,Vk	Compare less than word
0x6dc0	14-6	lt.l Sj,Vk	Compare less than longword
0x6e00	13-25	neg.b Vj,Vk	Negate vector/vector integer byte
0x6e40	13-25	neg.h Vj,Vk	Negate vector/vector integer halfword
0x6e80	13-25	neg.w Vj,Vk	Negate vector/vector integer word
0x6ec0	13-25	neg.l Vj,Vk	Negate vector/vector integer longword
0x6f00	10-12	neg.b Sj,Sk	Negate scalar/scalar integer byte
0x6f40	10-12	neg.h Sj,Sk	Negate scalar/scalar integer halfword
0x6f80	10-12	neg.w Sj,Sk	Negate scalar/scalar integer word
0x6fc0	10-12	neg.l Sj,Sk	Negate scalar/scalar integer longword
0x7000	11-5	nop	No Operation
0x7100	11-5	br	Branch Always
0x7200	11-5	bri.f	Branch on ION false
0x7300	11-5	bri.t	Branch on ION true
0x7400	11-5	bra.f	Branch on address carry false
0x7500	11-5	bra.t	Branch on address carry true
0x7600	11-5	brs.f	Branch on scalar carry false
0x7700	11-5	brs.t	Branch on scalar carry true
0x7800	13-12	ldvi.b Vj,Vk	Index Load vector byte
0x7840	13-12	ldvi.h Vj,Vk	Index Load vector halfword
0x7880	13-12	ldvi.s Vj,Vk	Index Load vector single float
0x7880	13-12	ldvi.w Vj,Vk	Index Load vector word
0x78c0	13-12	ldvi.d Vj,Vk	Index Load vector double float
0x78c0	13-12	ldvi.l Vj,Vk	Index Load vector longword
0x7E70	12-23	mov Sk,VV	Move scalar to vector valid flag
0x7E78	12-24	tstvv	Test value of vector valid flag
0x7a00	13-14	stvi.b Vk,Vj	Index Store vector byte
0x7a40	13-14	stvi.h Vk,Vj	Index Store vector halfword

0x7a80	13-14	stvi.s Vk,Vj	Index Store vector single float
0x7a80	13-14	stvi.w Vk,Vj	Index Store vector word
0x7ac0	13-14	stvi.d Vk,Vj	Index Store vector double float
0x7ac0	13-14	stvi.l Vk,Vj	Index Store vector longword
0x7b00	13-16	stvi.b Sk,Vj	Scalar Index Store vector byte
0x7b40	13-16	stvi.h Sk,Vj	Scalar Index Store vector halfword
0x7b80	13-16	stvi.s Sk,Vj	Scalar Index Store vector single float
0x7b80	13-16	stvi.w Sk,Vj	Scalar Index Store vector word
0x7bc0	13-16	stvi.d Sk,Vj	Scalar Index Store vector double float
0x7bc0	13-16	stvi.l Sk,Vj	Scalar Index Store vector longword
0x7c00	12-5	ldsdr Ak	Load process SDR's
0x7c08	12-6	ldkdr Ak	Load all 8 SDR's
0x7c20	12-8	patu	Purge the entire ATU
0x7c28	12-9	pate Ak	Purge ATU entry
0x7c30	12-13	pich	Purge the instruction cache
0x7c38	12-14	plch	Purge the logical cache
0x7c40	9-35	mov PSW,Ak	Store the PSW into an addr. reg.
0x7c48	9-36	mov Ak,psw	Load an addr. reg. into the PSW
0x7c50	9-28	mov PC,Ak	Load next PC address
0x7c60	12-16	mov ITR,Sk	Move the itc,itsr,nitc into Sk
0x7c68	12-15	mov Sk,ITR	Load NITC, ITC, ITSR from Sk
0x7c78	12-17	mov Sk,itsr	Load ITSR with a scalar
0x7c80	11-15	rtnq	Pop the program counter and jump
0x7c90	11-12	rtn	Return from subroutine call
0x7ca8	12-10	rtnc	Return from a context block
0x7ca8	16-9	mov.w Sk,VS	Move Sk to VS
0x7cb8	16-8	mov.w Sk,VL	Move Sk to VL
0x7d00	9-30	psh.w Ak	Push an addr. reg.
0x7d10	9-31	pop.w Ak	Pop word into addr. reg.
0x7d20	10-25	psh.w Sk	Push Sk<31..0> onto the stack
0x7d28	10-25	psh.l Sk	Push Sk<63..0> onto the stack.
0x7d30	10-26	pop.w Sk	Pop Sk<31..0> from the stack
0x7d38	10-26	pop.l Sk	Pop Sk<63..0> from the stack.
0x7d40	12-12	eni	Enable interrupts, set ion to 1
0x7d48	12-12	dsi	Disable interrupts,reset ion to 0
0x7d50	11-9	bkpt	Breakpoint
0x7d60	12-19	mski Sk	Mask Out Interrupt
0x7d68	12-18	xmti Sk	Transmit Interrupt
0x7d80	16-4	mov VS,Ak	Move VS to Ak
0x7d88	16-4	mov Ak,VS	Move Ak to VS
0x7d90	16-2	mov VL,Ak	Move VL to Ak
0x7d98	16-2	mov Ak,VL	Move Ak to VL
0x7dc0	12-21	diag Ak	Execute non-standard microcode sequence
0x7e00	15-2	sum.b Vk	Sum a vector of bytes
0x7e08	15-2	sum.h Vk	Sum a vector of halfwords
0x7e10	15-2	sum.w Vk	Sum a vector of words
0x7e18	15-2	sum.l Vk	Sum a vector of longwords
0x7e20	15-8	all Vk	AND reduce a vector
0x7e28	15-9	any Vk	OR reduce a vector
0x7e30	15-10	parity Vk	Exclusive OR reduce a vector
0x7e40	15-6	max.b Vk	Max of a vector of bytes
0x7e48	15-6	max.h Vk	Max of a vector of halfwords
0x7e50	15-6	max.w Vk	Max of a vector of words
0x7e58	15-6	max.l Vk	Max of a vector of longwords

0x7e60	15-7	min.b Vk	Min of a vector of bytes
0x7e68	15-7	min.h Vk	Min of a vector of halfwords
0x7e70	15-7	min.w Vk	Min of a vector of words
0x7e78	15-7	min.l Vk	Min of a vector of longwords
0x7e80	15-2	sum.s Vk	Sum a vector of single float
0x7e88	15-2	sum.d Vk	Sum a vector of double float
0x7e90	15-4	prod.s Vk	Multiply reduce a vector of single float
0x7e98	15-4	prod.d Vk	Multiply reduce a vector of double float
0x7ea0	15-6	max.s Vk	Max of a vector of single float
0x7ea8	15-6	max.d Vk	Max of a vector of double float
0x7eb0	15-7	min.s Vk	Min of a vector of single float
0x7eb8	15-7	min.d Vk	Min of a vector of double float
0x7ec0	15-4	prod.b Vk	Multiply reduce a vector of bytes
0x7ec8	15-4	prod.h Vk	Multiply reduce a vector of halfwords
0x7ed0	15-4	prod.w Vk	Multiply reduce a vector of words
0x7ed8	15-4	prod.l Vk	Multiply reduce a vector of longwords
0x7ee0	16-12	plc.f VM,Sk	Load the number of 0's in VM into Sk
0x7ee8	16-12	plc.t VM,Sk	Load the number of 1's in VM into Sk
0x8000	13-40	mov Vi,Sj,Sk	Move a vector element to a scalar
0x8200	13-39	mov Si,Sj,Vk	Move a scalar to a vector element
0x8400	14-10	merg.t Vi,Vj,Vk	Merge vector/vector
0x8600	14-13	mask.t Vi,Vj,Vk	Mask vector/vector
0x8800	14-12	merg.f Vi,Sj,Vk	Merge vector/scalar using not VM
0x8a00	14-14	mask.f Vi,Sj,Vk	Mask vector/scalar using not VM
0x8c00	14-12	merg.t Vi,Sj,Vk	Merge vector/scalar
0x8e00	14-14	mask.t Vi,Sj,Vk	Mask vector/scalar using VM
0x9000	13-22	mul.s Vi,Vj,Vk	Multiply vector/vector single float
0x9200	13-22	mul.d Vi,Vj,Vk	Multiply vector/vector double float
0x9400	13-23	div.s Vi,Vj,Vk	Divide vector/vector single float
0x9600	13-23	div.d Vi,Vj,Vk	Divide vector/vector double float
0x9800	13-28	mul.s Vi,Sj,Vk	Multiply vector/scalar single float
0x9a00	13-28	mul.d Vi,Sj,Vk	Multiply vector/scalar double float
0x9c00	13-29	div.s Vi,Sj,Vk	Divide vector/scalar single float
0x9e00	13-29	div.d Vi,Sj,Vk	Divide vector/scalar double float
0xa000	13-31	and Vi,Vj,Vk	AND two vectors
0xa200	13-32	or Vi,Vj,Vk	OR two vectors
0xa400	13-33	xor Vi,Vj,Vk	Exclusive OR two vectors
0xa800	13-35	and Vi,Sj,Vk	AND vector/scalar
0xaa00	13-36	or Vi,Sj,Vk	OR vector/scalar
0xac00	13-37	xor Vi,Sj,Vk	Exclusive OR vector/scalar
0xb000	13-20	add.s Vi,Vj,Vk	Add vector/vector single float
0xb200	13-20	add.d Vi,Vj,Vk	Add vector/vector double float
0xb400	13-21	sub.s Vi,Vj,Vk	Subtract vector/vector single float
0xb600	13-21	sub.d Vi,Vj,Vk	Subtract vector/vector double float
0xb800	13-26	add.s Vi,Sj,Vk	Add vector/scalar single float
0xba00	13-26	add.d Vi,Sj,Vk	Add vector/scalar double float
0xbc00	13-27	sub.s Vi,Sj,Vk	Subtract vector/scalar single float
0xbe00	13-27	sub.d Vi,Sj,Vk	Subtract vector/scalar double float
0xc000	13-20	add.b Vi,Vj,Vk	Add vector/vector integer byte
0xc200	13-20	add.h Vi,Vj,Vk	Add vector/vector integer halfword
0xc400	13-20	add.w Vi,Vj,Vk	Add vector/vector integer word
0xc600	13-20	add.l Vi,Vj,Vk	Add vector/vector integer longword
0xc800	13-26	add.b Vi,Sj,Vk	Add vector/scalar integer byte
0xca00	13-26	add.h Vi Sj,Vk	Add vector/scalar integer halfword

0xcc00	13-26	add.w Vi,Sj,Vk	Add vector/scalar integer word
0xce00	13-26	add.l Vi,Sj,Vk	Add vector/scalar integer longword
0xd000	13-21	sub.b Vi,Vj,Vk	Subtract vector/vector integer byte
0xd200	13-21	sub.h Vi,Vj,Vk	Subtract vector/vector integer halfword
0xd400	13-21	sub.w Vi,Vj,Vk	Subtract vector/vector integer word
0xd600	13-21	sub.l Vi,Vj,Vk	Subtract vector/vector integer longword
0xd800	13-27	sub.b Vi,Sj,Vk	Subtract vector/scalar integer byte
0xda00	13-27	sub.h Vi,Sj,Vk	Subtract vector/scalar integer halfword
0xdc00	13-27	sub.w Vi,Sj,Vk	Subtract vector/scalar integer word
0xde00	13-27	sub.l Vi,Sj,Vk	Subtract vector/scalar integer longword
0xe000	13-22	mul.b Vi,Vj,Vk	Multiply vector/vector integer byte
0xe200	13-22	mul.h Vi,Vj,Vk	Multiply vector/vector integer halfword
0xe400	13-22	mul.w Vi,Vj,Vk	Multiply vector/vector integer word
0xe600	13-22	mul.l Vi,Vj,Vk	Multiply vector/vector integer longword
0xe800	13-28	mul.b Vi,Sj,Vk	Multiply vector/scalar integer byte
0xea00	13-28	mul.h Vi,Sj,Vk	Multiply vector/scalar integer halfword
0xec00	13-28	mul.w Vi,Sj,Vk	Multiply vector/scalar integer word
0xee00	13-28	mul.l Vi,Sj,Vk	Multiply vector/scalar integer longword
0xf000	13-23	div.b Vi,Vj,Vk	Divide vector/vector integer byte
0xf200	13-23	div.h Vi,Vj,Vk	Divide vector/vector integer halfword
0xf400	13-23	div.w Vi,Vj,Vk	Divide vector/vector integer word
0xf600	13-23	div.l Vi,Vj,Vk	Divide vector/vector integer longword
0xf800	13-29	div.b Vi,Sj,Vk	Divide vector/scalar integer byte
0xfa00	13-29	div.h Vi,Sj,Vk	Divide vector/scalar integer halfword
0xfc00	13-29	div.w Vi,Sj,Vk	Divide vector/scalar integer word
0xfe00	13-29	div.l Vi,Sj,Vk	Divide vector/scalar integer longword

APPENDIX C
OP CODES SORTED BY NAME

0x5900	10-8	add.b Sj,Sk	Add scalar/scalar integer byte
0xc800	13-26	add.b Vi,Sj,Vk	Add vector/scalar integer byte
0xc000	13-20	add.b Vi,Vj,Vk	Add vector/vector integer byte
0x5540	10-8	add.d Sj,Sk	Add scalar/scalar double float
0xba00	13-26	add.d Vi,Sj,Vk	Add vector/scalar double float
0xb200	13-20	add.d Vi,Vj,Vk	Add vector/vector double float
0x1400	9-12	add.h #N,Ak	Add imm. address halfword
0x1408	10-17	add.h #N,Sk	Add scalar/immed. integer halfword
0x5880	9-12	add.h #n,Ak	Add short imm. address halfword
0x5800	9-7	add.h Aj,Ak	Add addr. reg. halfword
0x5940	10-8	add.h Sj,Sk	Add scalar/scalar integer halfword
0xca00	13-26	add.h Vi Sj,Vk	Add vector/scalar integer halfword
0xc200	13-20	add.h Vi,Vj,Vk	Add vector/vector integer halfword
0x59c0	10-8	add.l Sj,Sk	Add scalar/scalar integer longword
0xce00	13-26	add.l Vi,Sj,Vk	Add vector/scalar integer longword
0xc600	13-20	add.l Vi,Vj,Vk	Add vector/vector integer longword
0x1808	10-17	add.s #N,Sk	Add scalar/immed. single float
0x5500	10-8	add.s Sj,Sk	Add scalar/scalar single float
0xb800	13-26	add.s Vi,Sj,Vk	Add vector/scalar single float
0xb000	13-20	add.s Vi,Vj,Vk	Add vector/vector single float
0x1480	9-12	add.w #N,Ak	Add imm. address word
0x1488	10-17	add.w #N,Sk	Add scalar/immed. integer word
0x58c0	9-12	add.w #n,Ak	Add short imm. address word
0x5840	9-7	add.w Aj,Ak	Add addr. reg. word
0x5000	9-16	add.w Sj,Ak	Add scalar to addr word
0x5980	10-8	add.w Sj,Sk	Add scalar/scalar integer word
0xcc00	13-26	add.w Vi,Sj,Vk	Add vector/scalar integer word
0xc400	13-20	add.w Vi,Vj,Vk	Add vector/vector integer word
0x7e20	15-8	all Vk	AND reduce a vector
0x1200	9-21	and #N,Ak	AND imm. to addr. reg.
0x1208	10-21	and #N,Sk	AND scalar/immediate
0x5200	9-17	and Aj,Ak	AND addr. reg.
0x5300	10-13	and Sj,Sk	AND scalar/scalar
0xa800	13-35	and Vi,Sj,Vk	AND vector/scalar
0xa000	13-31	and Vi,Vj,Vk	AND two vectors
0x7e28	15-9	any Vk	OR reduce a vector
0x7d50	11-9	bkpt	Breakpoint
0x7100	11-5	br	Branch Always
0x7400	11-5	bra.f	Branch on address carry false
0x7500	11-5	bra.t	Branch on address carry true
0x7200	11-5	bri.f	Branch on ION false
0x7300	11-5	bri.t	Branch on ION true
0x7600	11-5	brs.f	Branch on scalar carry false
0x7700	11-5	brs.t	Branch on scalar carry true
0x2000	11-10	call <effa>	Call a subroutine, make a long frame
0x2200	11-14	callq <effa>	Push the program counter and jump
0x2100	11-11	calls <effa>	Call a subroutine, make a short frame
0x6380	14-8	cprs.f Vj,Vk	Compress a vector using not VM
0x63c0	14-8	cprs.t Vj,Vk	Compress a vector using VM
0x4080	9-44	cvtb.w Aj,Ak	Convert byte to word
0x4180	10-34	cvtb.w Sj,Sk	Convert byte to word

0x4340	10-34	cvtd.l Sj,Sk	Convert double float to longword
0x4280	10-34	cvtd.s Sj,Sk	Convert double float to single float
0x40c0	9-44	cvth.w Aj,AK	Convert half to word
0x41c0	10-34	cvth.w Sj,Sk	Convert halfword to word
0x43c0	10-34	cvtl.d Sj,Sk	Convert longword to double float
0x4380	10-34	cvtl.s Sj,Sk	Convert longword to single float
0x4500	10-34	cvtl.w Sj,Sk	Convert longword to word
0x42c0	10-34	cvts.d Sj,Sk	Convert single float to double float
0x4300	10-34	cvts.l Sj,Sk	Convert single float to longword
0x4240	10-34	cvts.w Sj,Sk	Convert single float to word
0x4000	9-44	cvtw.b Aj,Ak	Convert word to byte
0x4100	10-34	cvtw.b Sj,Sk	Convert word to byte
0x4040	9-44	cvtw.h Aj,Ak	Convert word to halfword
0x4140	10-34	cvtw.h Sj,Sk	Convert word to halfword
0x4540	10-34	cvtw.l Sj,Sk	Convert word to longword
0x4200	10-34	cvtw.s Sj,Sk	Convert word to single float
0x7dc0	12-21	diag Ak	Execute non-standard microcode sequence
0x5f00	10-11	div.b Sj,Sk	Divide scalar/scalar integer byte
0xf800	13-29	div.b Vi,Sj,Vk	Divide vector/scalar integer byte
0xf000	13-23	div.b Vi,Vj,Vk	Divide vector/vector integer byte
0x57c0	10-11	div.d Sj,Sk	Divide scalar/scalar double float
0x9e00	13-29	div.d Vi,Sj,Vk	Divide vector/scalar double float
0x9600	13-23	div.d Vi,Vj,Vk	Divide vector/vector double float
0x1700	9-15	div.h #N,Ak	Divide imm. address halfword
0x1708	10-20	div.h #N,Sk	Divide scalar/scalar integer halfword
0x5e80	9-15	div.h #n,Ak	Divide short imm. address halfword
0x5e00	9-10	div.h Aj,Ak	Divide addr. reg. halfword
0x5f40	10-11	div.h Sj,Sk	Divide scalar/scalar integer halfword
0xfa00	13-29	div.h Vi,Sj,Vk	Divide vector/scalar integer halfword
0xf200	13-23	div.h Vi,Vj,Vk	Divide vector/vector integer halfword
0x5fc0	10-11	div.l Sj,Sk	Divide scalar/scalar integer longword
0xfe00	13-29	div.l Vi,Sj,Vk	Divide vector/scalar integer longword
0xf600	13-23	div.l Vi,Vj,Vk	Divide vector/vector integer longword
0x1988	10-20	div.s #N,Sk	Divide scalar/scalar single float
0x5780	10-11	div.s Sj,Sk	Divide scalar/scalar single float
0x9c00	13-29	div.s Vi,Sj,Vk	Divide vector/scalar single float
0x9400	13-23	div.s Vi,Vj,Vk	Divide vector/vector single float
0x1780	9-15	div.w #N,Ak	Divide imm. address word
0x1788	10-20	div.w #N,Sk	Divide scalar/scalar integer word
0x5ec0	9-15	div.w #n,Ak	Divide short imm. address word
0x5e40	9-10	div.w Aj,Ak	Divide addr. reg. word
0x5f80	10-11	div.w Sj,Sk	Divide scalar/scalar integer word
0xfc00	13-29	div.w Vi,Sj,Vk	Divide vector/scalar integer word
0xf400	13-23	div.w Vi,Vj,Vk	Divide vector/vector integer word
0x7d48	12-12	dsi	Disable interrupts,reset ion to 0
0x7d40	12-12	eni	Enable interrupts, set ion to 1
0x4700	10-27	eq.b Sj,Sk	Compare equal byte
0x6900	14-6	eq.b Sj,Vk	Compare equal byte
0x6800	14-4	eq.b Vj,Vk	Compare equal byte
0x5640	10-28	eq.d Sj,Sk	Compare equal double float
0x6540	14-7	eq.d Sj,Vk	Compare equal double precision
0x6440	14-5	eq.d Vj,Vk	Compare equal double precision
0x1b00	9-41	eq.h #N,Ak	Compare equal halfword
0x1b08	10-31	eq.h #N,Sk	Compare equal halfword

0x4680	9-40	eq.h #n,Ak	Compare equal halfword
0x4600	9-37	eq.h Aj,Ak	Compare equal halfword
0x4740	10-27	eq.h Sj,Sk	Compare equal halfword
0x6940	14-6	eq.h Sj,Vk	Compare equal halfword
0x6840	14-4	eq.h Vj,Vk	Compare equal halfword
0x47c0	10-27	eq.l Sj,Sk	Compare equal longword
0x69c0	14-6	eq.l Sj,Vk	Compare equal longword
0x68c0	14-4	eq.l Vj,Vk	Compare equal longword
0x5600	10-27	eq.s Sj,Sk	Compare equal single float
0x6500	14-6	eq.s Sj,Vk	Compare equal single
0x6400	14-4	eq.s Vj,Vk	Compare equal single
0x1b80	9-41	eq.w #N,Ak	Compare equal word
0x1b88	10-31	eq.w #N,Sk	Compare equal word
0x46c0	9-40	eq.w #n,Ak	Compare equal word
0x4640	9-37	eq.w Aj,Ak	Compare equal word
0x4780	10-27	eq.w Sj,Sk	Compare equal word
0x6980	14-6	eq.w Sj,Vk	Compare equal word
0x6880	14-4	eq.w Vj,Vk	Compare equal word
0x0000	11-7	exit	Error Exit Instruction
0x1000	12-20	halt #N,Ak	Halt the central processing unit
0x0100	11-7	jmp <effa>	Jump Always
0x0400	11-7	jmpa.f <effa>	Jump on address carry false
0x0500	11-7	jmpa.t <effa>	Jump on address carry true
0x0200	11-7	jmpi.f <effa>	Jump on ION false
0x0300	11-7	jmpi.t <effa>	Jump on ION true
0x0600	11-7	jmps.f <effa>	Jump on scalar carry false
0x0700	11-7	jmps.t <effa>	Jump on scalar carry true
0x2800	9-4	ld.b <effa>,Ak	Load addr. reg. byte
0x3000	10-4	ld.b <effa>,Sk	Load scalar byte
0x3800	13-8	ld.b <effa>,Vk	Load vector byte
0x1008	10-6	ld.d #N,Sk	Load immediate, most significant bits
0x3300	10-4	ld.d <effa>,Sk	Load scalar double float
0x3b00	13-8	ld.d <effa>,Vk	Load vector double float
0x1188	10-6	ld.dl #N,Sk	Load 64 bit floating immed., lower half
0x1088	10-6	ld.du #N,Sk	Load 64 bit floating immed., upper half
0x1100	9-6	ld.h #N,Ak	Load halfword imm. into Ak
0x4480	9-6	ld.h #n,Ak	Load short imm. into Ak
0x2900	9-4	ld.h <effa>,Ak	Load addr. reg. halfword
0x3100	10-4	ld.h <effa>,Sk	Load scalar halfword
0x3900	13-8	ld.h <effa>,Vk	Load vector halfword
0x1108	10-6	ld.l #N,Sk	Load 64 bit sign extended immediate
0x3300	10-4	ld.l <effa>,Sk	Load scalar longword
0x0a00	16-6	ld.l <effa>,VLS	Load VS and VL from memory
0x3b00	13-8	ld.l <effa>,Vk	Load vector longword
0x1188	10-6	ld.ll #N,Sk	Load 64 bit integer immed., lower half
0x1088	10-6	ld.lu #N,Sk	Load 64 bit integer immed., upper half
0x3200	10-4	ld.s <effa>,Sk	Load scalar single float
0x3a00	13-8	ld.s <effa>,Vk	Load vector single float
0x1088	10-6	ld.u #N,Sk	Load immediate, upper half
0x1180	9-6	ld.w #N,Ak	Load imm. into Ak
0x1188	10-6	ld.w #N,Sk	Load a 32 bit immediate
0x1800	16-3	ld.w #N,VL	Load VL with an immediate
0x1880	16-5	ld.w #N,VS	Load VS from an immediate
0x44c0	9-6	ld.w #n,Ak	Load short imm. into Ak

0x2a00	9-4	ld.w <effa>,Ak	Load addr. reg. word
0x3200	10-4	ld.w <effa>,Sk	Load scalar word
0x3a00	13-8	ld.w <effa>,Vk	Load vector word
0x0b00	16-10	ld.x <effa>, VM	Load VM from memory
0x0900	9-26	ldea <effa>,Ak	Load effective address
0x7c08	12-6	ldkdr Ak	Load all 8 SDR's
0x4400	9-33	ldpa Aj,Ak	Load a physical byte address into Ak
0x7c00	12-5	ldsdr Ak	Load process SDR's
0x7800	13-12	ldvi.b Vj,Vk	Index Load vector byte
0x78c0	13-12	ldvi.d Vj,Vk	Index Load vector double float
0x7840	13-12	ldvi.h Vj,Vk	Index Load vector halfword
0x78c0	13-12	ldvi.l Vj,Vk	Index Load vector longword
0x7880	13-12	ldvi.s Vj,Vk	Index Load vector single float
0x7880	13-12	ldvi.w Vj,Vk	Index Load vector word
0x4d00	10-27	le.b Sj,Sk	Compare less than or equal byte
0x6b00	14-6	le.b Sj,Vk	Compare less than or equal byte
0x6a00	14-4	le.b Vj,Vk	Compare less than or equal byte
0x5440	10-27	le.d Sj,Sk	Compare less than or equal double float
0x6740	14-6	le.d Sj,Vk	Compare less than or equal double float
0x6640	14-4	le.d Vj,Vk	Compare less than or equal double float
0x1e00	9-41	le.h #N,Ak	Compare less than or equal halfword
0x1e08	10-31	le.h #N,Sk	Compare less than or equal halfword
0x4c80	9-40	le.h #n,Ak	Compare less than or equal halfword
0x4c00	9-37	le.h Aj,Ak	Compare less than or equal signed halfword
0x4d40	10-27	le.h Sj,Sk	Compare less than or equal halfword
0x6b40	14-6	le.h Sj,Vk	Compare less than or equal halfword
0x6a40	14-4	le.h Vj,Vk	Compare less than or equal halfword
0x4dc0	10-27	le.l Sj,Sk	Compare less than or equal longword
0x6bc0	14-6	le.l Sj,Vk	Compare less than or equal longword
0x6ac0	14-4	le.l Vj,Vk	Compare less than or equal longword
0x1a08	10-31	le.s #N,Sk	Compare less than or equal single
0x5400	10-27	le.s Sj,Sk	Compare less than or equal single float
0x6700	14-6	le.s Sj,Vk	Compare less than or equal single
0x6600	14-4	le.s Vj,Vk	Compare less than or equal single
0x1e80	9-41	le.w #N,Ak	Compare less than or equal word
0x1e88	10-31	le.w #N,Sk	Compare less than or equal word
0x4cc0	9-40	le.w #n,Ak	Compare less than or equal word
0x4c40	9-37	le.w Aj,AK	Compare less than or equal signed word
0x4d80	10-27	le.w Sj,Sk	Compare less than or equal word
0x6b80	14-6	le.w Sj,Vk	Compare less than or equal word
0x6a80	14-4	le.w Vj,Vk	Compare less than or equal word
0x4900	10-29	leu.b Sj,Sk	Compare less than or equal byte
0x1c00	9-42	leu.h #N,Ak	Compare unsigned less than halfword
0x1c08	10-33	leu.h #N,Sk	Compare unsigned less than or equal halfword
0x4880	9-42	leu.h #n,Ak	Compare unsigned less than or equal halfword
0x4800	9-39	leu.h Aj,Ak	Compare unsigned less than or equal halfword
0x4940	10-29	leu.h Sj,Sk	Compare less than or equal halfword
0x49c0	10-29	leu.l Sj,Sk	Compare less than or equal longword
0x1c80	9-42	leu.w #N,Ak	Compare unsigned less than word
0x1c88	10-33	leu.w #N,Sk	Compare unsigned less than or equal word
0x48c0	9-42	leu.w #n,Ak	Compare unsigned less than or equal word
0x4840	9-39	leu.w Aj,Ak	Compare unsigned less or equal than word
0x4980	10-29	leu.w Sj,Sk	Compare less than or equal word
0x4f00	10-27	lt.b Sj,Sk	Compare less than byte

0x6d00	14-6	lt.b Sj,Vk	Compare less than byte
0x6c00	14-4	lt.b Vj,Vk	Compare less than byte
0x54c0	10-27	lt.d Sj,Sk	Compare less than double float
0x67c0	14-6	lt.d Sj,Vk	Compare less than double float
0x66c0	14-4	lt.d Vj,Vk	Compare less than double float
0x1f00	9-41	lt.h #N,Ak	Compare less than halfword
0x1f08	10-31	lt.h #N,Sk	Compare less than halfword
0x4e80	9-40	lt.h #n,Ak	Compare less than halfword
0x4e00	9-37	lt.h Aj,Ak	Compare less than signed halfword
0x4f40	10-27	lt.h Sj,Sk	Compare less than halfword
0x6d40	14-6	lt.h Sj,Vk	Compare less than halfword
0x6c40	14-4	lt.h Vj,Vk	Compare less than halfword
0x4fc0	10-27	lt.l Sj,Sk	Compare less than longword
0x6dc0	14-6	lt.l Sj,Vk	Compare less than longword
0x6cc0	14-4	lt.l Vj,Vk	Compare less than longword
0x1a88	10-31	lt.s #N,Sk	Compare less than single
0x5480	10-27	lt.s Sj,Sk	Compare less than single float
0x6780	14-6	lt.s Sj,Vk	Compare less than single
0x6680	14-4	lt.s Vj,Vk	Compare less than single
0x1f80	9-41	lt.w #N,Ak	Compare less than word
0x1f88	10-31	lt.w #N,Sk	Compare less than word
0x4ec0	9-40	lt.w #n,Ak	Compare less than word
0x4e40	9-37	lt.w Aj,Ak	Compare less than signed word
0x4f80	10-27	lt.w Sj,Sk	Compare less than word
0x6d80	14-6	lt.w Sj,Vk	Compare less than word
0x6c80	14-4	lt.w Vj,Vk	Compare less than word
0x4b00	10-29	ltu.b Sj,Sk	Compare less than byte
0x1d00	9-42	ltu.h #N,Ak	Compare unsigned less than halfword
0x1d08	10-33	ltu.h #N,Sk	Compare unsigned less than halfword
0x4a80	9-42	ltu.h #n,Ak	Compare unsigned less than halfword
0x4b40	10-29	ltu.h Sj,Sk	Compare less than halfword
0x4a00	9-39	ltu.h gt,Aj,Ak	Compare unsigned less than halfword
0x4bc0	10-29	ltu.l Sj,Sk	Compare less than longword
0x1d80	9-43	ltu.w #N,Ak	Compare unsigned less than word
0x1d88	10-33	ltu.w #N,Sk	Compare unsigned less than word
0x4ac0	9-42	ltu.w #n,Ak	Compare unsigned less than word
0x4b80	10-29	ltu.w Sj,Sk	Compare less than word
0x4a40	9-39	ltu.w gt,Aj,Ak	Compare unsigned less than word
0x8a00	14-14	mask.f Vi,Sj,Vk	Mask vector/scalar using not VM
0x8e00	14-14	mask.t Vi,Sj,Vk	Mask vector/scalar using VM
0x8600	14-13	mask.t Vi,Vj,Vk	Mask vector/vector
0x7e40	15-6	max.b Vk	Max of a vector of bytes
0x7ea8	15-6	max.d Vk	Max of a vector of double float
0x7e48	15-6	max.h Vk	Max of a vector of halfwords
0x7e58	15-6	max.l Vk	Max of a vector of longwords
0x7ea0	15-6	max.s Vk	Max of a vector of single float
0x7e50	15-6	max.w Vk	Max of a vector of words
0x8800	14-12	merg.f Vi,Sj,Vk	Merge vector/scalar using not VM
0x8c00	14-12	merg.t Vi,Sj,Vk	Merge vector/scalar
0x8400	14-10	merg.t Vi,Vj,Vk	Merge vector/vector
0x7e60	15-7	min.b Vk	Min of a vector of bytes
0x7eb8	15-7	min.d Vk	Min of a vector of double float
0x7e68	15-7	min.h Vk	Min of a vector of halfwords
0x7e78	15-7	min.l Vk	Min of a vector of longwords

0x7eb0	15-7	min.s Vk	Min of a vector of single float
0x7e70	15-7	min.w Vk	Min of a vector of words
0x5080	9-29	mov Aj,Ak	Move addr. reg.
0x51c0	10-40	mov Aj,Sk	Move an address to a scalar
0x7d98	16-2	mov Ak,VL	Move Ak to VL
0x7d88	16-4	mov Ak,VS	Move Ak to VS
0x7c48	9-36	mov Ak,psw	Load an addr. reg. into the PSW
0x7c60	12-16	mov ITR,Sk	Move the itc,itsr,nitc into Sk
0x7c50	9-28	mov PC,Ak	Load next PC address
0x7c40	9-35	mov PSW,Ak	Store the PSW into an addr. reg.
0x8200	13-39	mov Si,Sj,Vk	Move a scalar to a vector element
0x50c0	10-39	mov Sj,Ak	Move 32 bits of Sj into Ak.
0x6100	16-13	mov Sj,Sk,VM	Load VM(Sj) from Sk.
0x6140	16-13	mov Sj,VM,Sk	Load Sk from VM
0x7c68	12-15	mov Sk,ITR	Load NITC, ITC, ITSR from Sk
0x7E70	12-23	mov Sk,VV	Move scalar to vector valid flag
0x7c78	12-17	mov Sk,itsr	Load ITSR with a scalar
0x7d90	16-2	mov VL,Ak	Move VL to Ak
0x7d80	16-4	mov VS,Ak	Move VS to Ak
0x8000	13-40	mov Vi,Sj,Sk	Move a vector element to a scalar
0x5180	10-41	mov.d Sj,Sk	Move scalar register single float
0x5180	10-41	mov.l Sj,Sk	Move scalar register longword
0x5100	10-41	mov.s Sj,Sk	Move scalar register double float
0x5100	10-41	mov.w Sj,Sk	Move scalar register word
0x7cb8	16-8	mov.w Sk,VL	Move Sk to VL
0x7ca8	16-9	mov.w Sk,VS	Move Sk to VS
0x7d60	12-19	mski Sk	Mask Out Interrupt
0x5d00	10-10	mul.b Sj,Sk	Multiply scalar/scalar integer byte
0xe800	13-28	mul.b Vi,Sj,Vk	Multiply vector/scalar integer byte
0xe000	13-22	mul.b Vi,Vj,Vk	Multiply vector/vector integer byte
0x5740	10-10	mul.d Sj,Sk	Multiply scalar/scalar double float
0x9a00	13-28	mul.d Vi,Sj,Vk	Multiply vector/scalar double float
0x9200	13-22	mul.d Vi,Vj,Vk	Multiply vector/vector double float
0x1600	9-14	mul.h #N,Ak	Multiply imm. address halfword
0x1608	10-19	mul.h #N,Sk	Multiply scalar/immed. integer halfword
0x5c80	9-14	mul.h #n,Ak	Multiply short imm. address halfword
0x5c00	9-9	mul.h Aj,Ak	Multiply addr. reg. halfword
0x5d40	10-10	mul.h Sj,Sk	Multiply scalar/scalar integer halfword
0xea00	13-28	mul.h Vi,Sj,Vk	Multiply vector/scalar integer halfword
0xe200	13-22	mul.h Vi,Vj,Vk	Multiply vector/vector integer halfword
0x5dc0	10-10	mul.l Sj,Sk	Multiply scalar/scalar integer longword
0xee00	13-28	mul.l Vi,Sj,Vk	Multiply vector/scalar integer longword
0xe600	13-22	mul.l Vi,Vj,Vk	Multiply vector/vector integer longword
0x1908	10-19	mul.s #N,Sk	Multiply scalar/immed. single float
0x5700	10-10	mul.s Sj,Sk	Multiply scalar/scalar single float
0x9800	13-28	mul.s Vi,Sj,Vk	Multiply vector/scalar single float
0x9000	13-22	mul.s Vi,Vj,Vk	Multiply vector/vector single float
0x1680	9-14	mul.w #N,Ak	Multiply imm. address word
0x1688	10-19	mul.w #N,Sk	Multiply scalar/immed. integer word
0x5cc0	9-14	mul.w #n,Ak	Multiply short imm. address word
0x5c40	9-9	mul.w Aj,AK	Multiply addr. reg. word
0x5d80	10-10	mul.w Sj,Sk	Multiply scalar/scalar integer word
0xec00	13-28	mul.w Vi,Sj,Vk	Multiply vector/scalar integer word
0xe400	13-22	mul.w Vi,Vj,Vk	Multiply vector/vector integer word

0x6f00	10-12	neg.b Sj,Sk	Negate scalar/scalar integer byte
0x6e00	13-25	neg.b Vj,Vk	Negate vector/vector integer byte
0x65c0	10-12	neg.d Sj,Sk	Negate scalar/scalar double float
0x64c0	13-25	neg.d Vj,Vk	Negate vector/vector double float
0x5680	9-11	neg.h Aj,Ak	Negate addr. reg. halfword
0x6f40	10-12	neg.h Sj,Sk	Negate scalar/scalar integer halfword
0x6e40	13-25	neg.h Vj,Vk	Negate vector/vector integer halfword
0x6fc0	10-12	neg.l Sj,Sk	Negate scalar/scalar integer longword
0x6ec0	13-25	neg.l Vj,Vk	Negate vector/vector integer longword
0x6580	10-12	neg.s Sj,Sk	Negate scalar/scalar single float
0x6480	13-25	neg.s Vj,Vk	Negate vector/vector single float
0x56c0	9-11	neg.w Aj,Ak	Negate addr. reg. word
0x6f80	10-12	neg.w Sj,Sk	Negate scalar/scalar integer word
0x6e80	13-25	neg.w Vj,Vk	Negate vector/vector integer word
0x7000	11-5	nop	No Operation
0x52c0	9-20	not Aj,Ak	Complement addr. reg.
0x53c0	10-16	not Sj,Sk	Complement scalar/scalar
0x62c0	13-34	not Vj,Vk	Complement a vector
0x1280	9-22	or #N,Ak	OR imm. to addr. reg.
0x1288	10-22	or #N,Sk	OR scalar/immediate
0x5240	9-18	or Aj,Ak	OR addr. reg.
0x5340	10-14	or Sj,Sk	OR scalar/scalar
0xaa00	13-36	or Vi,Sj,Vk	OR vector/scalar
0xa200	13-32	or Vi,Vj,Vk	OR two vectors
0x7e30	15-10	parity Vk	Exclusive OR reduce a vector
0x7c28	12-9	pate Ak	Purge ATU entry
0x7c20	12-8	patu	Purge the entire ATU
0x7c30	12-13	pich	Purge the instruction cache
0x7ee0	16-12	plc.f VM,Sk	Load the number of 0's in VM into Sk
0x4580	10-38	plc.t Sj,Sk	Count the number of 1's in Sj
0x7ee8	16-12	plc.t VM,Sk	Load the number of 1's in VM into Sk
0x6340	15-11	plc.t Vj,Vk	Population Count of a Vector
0x7c38	12-14	plch	Purge the logical cache
0x7d38	10-26	pop.l Sk	Pop Sk<63..0> from the stack.
0x7d10	9-31	pop.w Ak	Pop word into addr. reg.
0x7d30	10-26	pop.w Sk	Pop Sk<31..0> from the stack
0x7ec0	15-4	prod.b Vk	Multiply reduce a vector of bytes
0x7e98	15-4	prod.d Vk	Multiply reduce a vector of double float
0x7ec8	15-4	prod.h Vk	Multiply reduce a vector of halfwords
0x7ed8	15-4	prod.l Vk	Multiply reduce a vector of longwords
0x7e90	15-4	prod.s Vk	Multiply reduce a vector of single float
0x7ed0	15-4	prod.w Vk	Multiply reduce a vector of words
0x7d28	10-25	psh.l Sk	Push Sk<63..0> onto the stack.
0x7d00	9-30	psh.w Ak	Push an addr. reg.
0x7d20	10-25	psh.w Sk	Push Sk<31..0> onto the stack
0x0d00	9-27	pshea <effa>	Push effective address
0x7c90	11-12	rtn	Return from subroutine call
0x7ca8	12-10	rtnc	Return from a context block
0x7c80	11-15	rtnq	Pop the program counter and jump
0x1380	9-25	shf #N,AK	Logical shift imm. to addr. reg.
0x1388	10-24	shf #N,Sk	Shift Scalar/immediate
0x4440	9-25	shf #n,Ak	Logical shift left short imm.
0x5040	9-24	shf Aj,Ak	Shift an address
0x5140	10-36	shf Sj,Sk	Shift a scalar

0x6300	13-38	shf Sj,Vk	Shift a vector accumulator
0x2c00	9-5	st.b Ak,<effa>	Store addr. reg. byte
0x3400	10-5	st.b Sk,<effa>	Store scalar byte
0x3c00	13-10	st.b Vk,<effa>	Store vector byte
0x3700	10-5	st.d Sk,<effa>	Store scalar double float
0x3f00	13-10	st.d Vk,<effa>	Store vector double float
0x2d00	9-5	st.h Ak,<effa>	Store addr. reg. halfword
0x3500	10-5	st.h Sk,<effa>	Store scalar halfword
0x3d00	13-10	st.h Vk,<effa>	Store vector halfword
0x3700	10-5	st.l Sk,<effa>	Store scalar longword
0x0e00	16-7	st.l VLS,<effa>	Store VS and VL to memory
0x3f00	13-10	st.l Vk,<effa>	Store vector longword
0x3600	10-5	st.s Sk,<effa>	Store scalar single float
0x3e00	13-10	st.s Vk,<effa>	Store vector single float
0x2e00	9-5	st.w Ak,<effa>	Store addr. reg. word
0x3600	10-5	st.w Sk,<effa>	Store scalar word
0x3e00	13-10	st.w Vk,<effa>	Store vector word
0x0f00	16-11	st.x VM,<effa>	store VM into memory
0x2400	13-18	ste.b Sk,<effa>	Store an extended scalar byte
0x2700	13-18	ste.d Sk,<effa>	Store an extended scalar double float
0x2500	13-18	ste.h Sk,<effa>	Store an extended scalar halfword
0x2700	13-18	ste.l Sk,<effa>	Store an extended scalar longword
0x2600	13-18	ste.s Sk,<effa>	Store an extended scalar single float
0x2600	13-18	ste.w Sk,<effa>	Store an extended scalar word
0x7b00	13-16	stvi.b Sk,Vj	Scalar Index Store vector byte
0x7a00	13-14	stvi.b Vk,Vj	Index Store vector byte
0x7bc0	13-16	stvi.d Sk,Vj	Scalar Index Store vector double float
0x7ac0	13-14	stvi.d Vk,Vj	Index Store vector double float
0x7b40	13-16	stvi.h Sk,Vj	Scalar Index Store vector halfword
0x7a40	13-14	stvi.h Vk,Vj	Index Store vector halfword
0x7bc0	13-16	stvi.l Sk,Vj	Scalar Index Store vector longword
0x7ac0	13-14	stvi.l Vk,Vj	Index Store vector longword
0x7b80	13-16	stvi.s Sk,Vj	Scalar Index Store vector single float
0x7a80	13-14	stvi.s Vk,Vj	Index Store vector single float
0x7b80	13-16	stvi.w Sk,Vj	Scalar Index Store vector word
0x7a80	13-14	stvi.w Vk,Vj	Index Store vector word
0x5b00	10-9	sub.b Sj,Sk	Subtract scalar/scalar integer byte
0xd800	13-27	sub.b Vi,Sj,Vk	Subtract vector/scalar integer byte
0xd000	13-21	sub.b Vi,Vj,Vk	Subtract vector/vector integer byte
0x55c0	10-9	sub.d Sj,Sk	Subtract scalar/scalar double float
0xbe00	13-27	sub.d Vi,Sj,Vk	Subtract vector/scalar double float
0xb600	13-21	sub.d Vi,Vj,Vk	Subtract vector/vector double float
0x1500	9-13	sub.h #N,Ak	Subtract imm. address halfword
0x1508	10-18	sub.h #N,Sk	Subtract scalar/immed. integer halfword
0x5a80	9-13	sub.h #n,Ak	Subtract short imm. address halfword
0x5a00	9-8	sub.h Aj,Ak	Subtract addr. reg. halfword
0x5b40	10-9	sub.h Sj,Sk	Subtract scalar/scalar integer halfword
0xda00	13-27	sub.h Vi,Sj,Vk	Subtract vector/scalar integer halfword
0xd200	13-21	sub.h Vi,Vj,Vk	Subtract vector/vector integer halfword
0x5bc0	10-9	sub.l Sj,Sk	Subtract scalar/scalar integer longword
0xde00	13-27	sub.l Vi,Sj,Vk	Subtract vector/scalar integer longword
0xd600	13-21	sub.l Vi,Vj,Vk	Subtract vector/vector integer longword
0x1888	10-18	sub.s #N,Sk	Subtract scalar/immed. single float
0x5580	10-9	sub.s Sj,Sk	Subtract scalar/scalar single float

0xbc00	13-27	sub.s Vi,Sj,Vk	Subtract vector/scalar single float
0xb400	13-21	sub.s Vi,Vj,Vk	Subtract vector/vector single float
0x1580	9-13	sub.w #N,Ak	Subtract imm. address word
0x1588	10-18	sub.w #N,Sk	Subtract scalar/immed. integer word
0x5ac0	9-13	sub.w #n,AK	Subtract short imm. address word
0x5a40	9-8	sub.w Aj,Ak	Subtract addr. reg. word
0x5b80	10-9	sub.w Sj,Sk	Subtract scalar/scalar integer word
0xdc00	13-27	sub.w Vi,Sj,Vk	Subtract vector/scalar integer word
0xd400	13-21	sub.w Vi,Vj,Vk	Subtract vector/vector integer word
0x7e00	15-2	sum.b Vk	Sum a vector of bytes
0x7e88	15-2	sum.d Vk	Sum a vector of double float
0x7e08	15-2	sum.h Vk	Sum a vector of halfwords
0x7e18	15-2	sum.l Vk	Sum a vector of longwords
0x7e80	15-2	sum.s Vk	Sum a vector of single float
0x7e10	15-2	sum.w Vk	Sum a vector of words
0x1080	11-16	sysc #r,#g	Perform a system call
0x0c00	9-32	tas <effa>	Test and Set a memory byte
0x7E78	12-24	tstvv	Test value of vector valid flag
0x45c0	10-37	tzc Sj,Sk	Count of trailing zeroes in Sj
0x7d68	12-18	xmti Sk	Transmit Interrupt
0x1300	9-23	xor #N,AK	Exclusive OR imm. to addr. reg.
0x1308	10-23	xor #N,Sk	Exclusive OR scalar/immediate
0x5280	9-19	xor Aj,Ak	Exclusive OR addr. reg.
0x5380	10-15	xor Sj,Sk	Exclusive OR scalar/scalar
0xac00	13-37	xor Vi,Sj,Vk	Exclusive OR vector/scalar
0xa400	13-33	xor Vi,Vj,Vk	Exclusive OR two vectors

APPENDIX D
FLOATING POINT ALGORITHMS

OVERVIEW

This appendix details the floating point algorithms used by the CONVEX instruction set. These algorithms involve rounding, sequencing of operations, and other considerations.

The CONVEX format for a double precision floating point operand in memory is:

```
 -----------------------------------------------------------------------
 | S |  Exponent   |                      Fraction                     |
 -----------------------------------------------------------------------
  63,62          52,51                                                 0
 BYTE 0                                                           BYTE 7
```

The CONVEX format for a single precision floating point operand in memory is:

```
                 ---------------------------------------
                 | S | Exponent |       Fraction       |
                 ---------------------------------------
                  31,30       23,22                    0
                 BYTE 0                           BYTE 3
```

ADD/SUBTRACT

The fraction of the floating point operands are expanded internally as follows:

1 A 1 is appended to the higher bit position of the fraction.

2 Two guard bits are appended to the right of the least significant fraction bit. These bits are referred to as G and R in that order.

3 A sticky bit is appended to the right of the two guard bits. The sticky or S bit is the OR of all bits to the right of the R bit.

4 An additional bit is appended to the higher fraction, the V bit, for overflow.

Thus the internal floating point format is:

The initial values of the V,G,R, and S bits are all 0.

5 The exponents of the two fractions are compared. The fraction of the smaller exponent is shifted right an amount equal to the absolute difference of the exponents. All right shifted bits are shifted through the G, R, and S bits.

6 Any binary 1's shifted past the 2 guard bits are remembered in S.

7 If any of the input operands were zero (Sign is 0, and Exponent is 0) no shifting occurs.

8 If any of the input operands was reserved (Sign is 1, and Exponent is 0), no shifting occurs, a reserved operand exception occurs, and the output of the ADD/SUB is a reserved operand.

9 Otherwise, the two fractions are algebraically added/subtracted according to the sign and opcode.

10 The result is normalized. If V became 1, the intermediate result is right shifted one bit position. R is ORed with S. If a generated subtract was performed, the intermediate result is left shifted until a normalized intermediate result is obtained. S need not participate in the left shifts; zero or S may be shifted into R from the right. G is loaded with R. S is always unchanged.

11 The intermediate normalized result is rounded as follows:

G	R	S	Round Performed (to LSB)
0	0	0	Add 0
0	0	1	Add 0
0	1	0	Add 0
0	1	1	Add 0
1	0	0	Add LSB of fraction - round to nearest even.
1	0	1	Add 1
1	1	0	Add 1
1	1	1	Add 1

12 The rounded intermediate result is normalized again and the exponent is adjusted if necessary to yield the final result.

<u>MULTIPLY</u>

Multiplication of 2 normalized floating point numbers produces an intermediate result that is either normalized or at most requires one left shift. The steps for multiplication are as follows:

1 If either of the two operands is a reserved operand, the result is a reserved operand.

2 If either of the two operands is zero (Sign is 0, and Exponent is 0), then the result is a true zero (Sign, Exponent, and fraction of all 0).

3 Otherwise. add the exponents, keeping an extra bit of precision to account for a normalization shift that could correct an exponent overflow.

4 Multiply the two fractions right to left.

5 Maintain the G, R, and S bits during intermediate calculations.

6 Post normalize if required.

7 Round the intermediate result per step 11 in the description in ADD/SUB.

8 The rounded intermediate result is normalized again and the exponent adjusted if necessary to yield the final result.

<u>DIVIDE</u>

Division of 2 normalized floating point numbers produces an intermediate result that is normalized.

1 If either of the two operands is a reserved operand, the result is a reserved operand.

2 If the divisor is zero, then the result is a reserved operand. Also the FDZ bit in the PSW is set to 1.

3 Subtract the exponents producing the result exponent.

4 Divide the numerator by the denominator mantissa until a normalized result is obtained. A (n+2) bit quotient is generated where n is the length of the mantissa of the operands. The two additional quotient bits represent the G and R bits. The state of the S bit is implementation specific. The S bit may always be assumed to be 0, or may represent the OR of some portion if not the entire remainder.

5 Round the intermediate result per step 11 in the description of ADD/SUB.

6 The rounded intermediate result is normalized again and the exponent adjusted if necessary to yield the final result.

CONVERSIONS

Converting from float to fix: always round toward 0 (truncate). This conversion obeys the FORTRAN rounding convention.

Converting from fix to float: properly normalize the integer, then truncate to the appropriate mantissa size.

Please note that rounding from float to fix can be achieved by first adding .5 to the floating point operand and then by executing the float to fix instruction. Thus, RND (3.4) equals TRUNCATE (3.9) = 3 and RND (3.5) equals TRUNCATE (4.0) = 4.

APPENDIX E
ASSEMBLER NOTATION

Introduction

The CONVEX instruction set has eight possible operand addressing modes: register mode, immediate operands and six modes for specifying operands in memory. Since the CONVEX architecture is based on three sets of high speed registers, most of the instructions are limited to register operands only. Each addressing mode is described in detail in the following sections.

Instruction Format

An assembly language program is a sequence of instructions. These instructions may be machine instructions that will be translated into machine language instructions, or they may be directives to the assembler. Both types of instructions follow the same basic format, as follows.

The instruction is composed of five fields: label, mnemonic, operand list, comment, and the terminator. The only field that is required is the terminator field (exclamation point or newline). The format for an instruction is shown below:

```
[label:] mnemonic [operand list] [;comment] terminator
```

Sample code:

```
Arg = 4
                  .
                  .
                  .

loop:        eq.w    #0,s0
             jmps.t  exit
             add.w   #1,s0
             ld.w    arg(a2),s1
             add.w   s1,s0
             br      loop

exit:
                  .
                  .
                  .
```

Addressing Modes

Register Mode

The majority of the instructions in the instruction set requires one or more register mode operands. A register mode operand specifies the register set and the particular register in the set to be used as the operand. The actual machine instruction generated by the assembler will depend on both the register set used and the register within that register set.

General register operands are specified by a letter and a number. The letter denotes the register set, and the number specifies the register number in the set. The general register sets for the machine are the address registers (A), the scalar registers (S), and the vector registers (V). The register set letter can be specified in either upper case or lower case. Each register set contains eight registers denoted by the numbers zero to seven.

To aid in program readability, three of the address registers can also be referenced by special names. These are the Stack Pointer (SP), Argument Pointer (AP), and Frame Pointer (FP).

The assembler itself uses reserved words to denote the registers in the machine. Each register is referenced by the reserved words only; hence additional symbols may not be defined to denote machine registers.

There are also ten special purpose registers--items 8 through 16 listed in the Register Syntax Summary Table below--in the machine that are used for machine control. These registers are specified by reserved words.

<u>Immediate Operands</u>

Immediate operands provide a method for referencing data in the program's instruction stream. The assembler syntax for specifying an immediate operand is written as "#" followed by the expression that defines the value of the immediate operand in question.

<u>Memory Addressing Modes</u>

In addition to the register and immediate addressing modes, there are six modes for specifying operands in memory. In general, these modes are used for loading registers from memory and for storing the contents of a register in memory.

<u>Absolute Addressing Mode</u>

A program can reference an absolute address in the virtual address space by specifying the location desired. This specification can be done by using an expression that evaluates to the address of the location desired:

```
        ld.w  addr,reg
```

<u>Register Deferred Mode</u>

Via the register deferred mode, the address registers can contain the address of an operand to be used by the instruction. Thus, the register contains a pointer to the operand rather than the operand itself. This addressing mode is specified by enclosing the address register that contains the pointer in parentheses:

```
        ld.w  (INDEX reg), reg
```

<u>Indexed Mode</u>

The indexed mode adds an offset or base to the contents of the address register specified and uses the result as a pointer to the operand. This addressing mode is formed by preceding a deferred register specifier with an expression:

```
        ld.w  addr (INDEX reg), reg
```

<u>Indirect Absolute Mode</u>

This mode provides for one level of indirection from an absolute address.. The operand pointer is located at the absolute address specified by an expression. This addressing mode is specified by preceding an expression for the absolute location by the character "@".

```
        ld.w  @addr, reg
```

<u>Indirect Deferred Mode</u>

This mode provides an additional level of indirection over the deferred mode. The register specified contains the address of the pointer to the desired operand. The character "@", followed by a deferred register operand, specifies this addressing mode.

```
        ld.w @ (INDEX reg), reg
```

<u>Indirect Indexed Mode</u>

Indirect indexed mode is denoted by preceding an indexed operand by the character "@". In this mode, the value of the register and the value of the index expression are added together to form the address of a pointer to the desired operand:

```
        ld.w  @ addr (INDEX reg), reg
```

Table E-1: Register Syntax Summary

Symbolic	Assembled Mode
1. A or a	Address Register Set
2. S or s	Scalar Register Set
3. V or v	Vector Register Set
4. SP or sp	Stack Pointer (A0)
5. AP or ap	Argument Pointer (A6)
6. FP or fp	Frame Pointer (A7)
7. VS or vs	Vector Stride Register
8. VL or vl	Vector Length Register
9. VLS or vls	Vector Stride and Length Combination
10. PSW or psw	Processor Status Word
11. PC or pc	Program Counter
12. ITR or itr	Interval Timer Register
13. ITSR or itsr	Interval Timer Status Register

Table E-2: Addressing Modes Syntax Summary

Symbolic	Assembled Mode
1. #n	immediate
2. n	absolute or PC relative
3. R	register
4. (Rn)	register deferred mode
5. n(Rn)	indexed mode
6. @n	indirect absolute mode
7. @(Rn)	indirect deferred mode
8. @n(Rn)	indirect indexed mode

APPENDIX F
GLOSSARY OF TECHNICAL TERMS USED IN THIS MANUAL

Access Mode
: Any of the five processor access modes in which software executes. On the CONVEX system, processor access modes are: (in order from most to least privileged and protected): kernel (mode 0), executive (mode 1), supervisor (mode 2), agent (mode 3) and user (mode 4). The operating system uses access modes to define protection levels for software executing in the context of a process.

Accumulator
: A hardware register. This register contains the results of arithmetic and logical operations.

Address
: User assigned number used by the operating system to identify a storage location.

Addressing mode
: How the effective address of an instruction operand is calculated using the general registers.

Address space
: Address space, either physical or virtual, available to a process.

Address Translation Fault
: An exception that results from a PTE violation or a non-resident page.

Address Translation Unit
: The address translation unit (ATU) translates logical addresses to physical addresses and stores them in a cache. Thus the ATU is an address cache which accelerates the generation of physical addresses.

Architecture
: The conceptual structure and functional behavior of the system.

Argument Pointer
: An address register specifically dedicated to point to the subroutine argument portion of a program. This program portion can either be in the stack or in part of logical memory pre-allocated by the compiler.

Arrays
: An ordered structure of operands of the same data type. The structure of an array is defined as: length, rank or dimension, stride, and data type.

Base-level interrupt
: An interrupt which occurs when the kernel stack is the process stack. A base-level interrupt is thus an interrupt which occurs when no other interrupts are pending or currently being processed.

Bit
: A binary digit.

Bit complement
Exchanging 0's and 1's in the binary representation of a number (also known as 1's complement).

Bootstrap
The procedure by which a program is initiated the first time. Typically a bootstrap is performed when power is first applied to the processor.

Branch
A class of instructions used to transfer control of a program, specifically relative to the Program Counter.

Breakpoint
An instruction which aids in the debugging of a program. In particular, a breakpoint is a particular location in a program that one would desire to determine the various values of programmer-defined variables.

Byte (b)
A byte is a number of contiguous bits starting on an addressable byte boundary. In CONVEX machines, a byte is eight bits.

C
The systems programming language of the UNIX operating system.

Cache
(See Logical, Physical, Instruction).

Cache memory
A small, high-speed memory placed between main memory and the processor and transparent to the user. CONVEX computers contain many separate caches.

Cache purge
The act of invalidating or removing entries in a cache memory.

Central processing unit
The central processing unit is the portion of a CONVEX machine which recognizes and executes the instruction set.

Chaining
Chaining is the ability to overlap vector operations in the central processing unit. For instance, in the case of a vector load followed by a vector add, the add may be started as soon as the first operands are available rather than waiting for the load to complete.

Compiler
Software tool used to compile a high-level language (e.g., Fortran) into assembly code.

Context (processor)
The entire, current state of the machine associated with the executing process.

Data type
: The way in which bits are grouped and interpreted. For processor instructions, the data type identifies the size of the operand and the significance of the bits in the operand.

Destination
: The operand specified in an instruction which receives the result of the operation.

Displacement
: A derived 32-bit value used to indicate the distance in bytes that the referenced datum is relative to some base value. This base value can either be 0 or the contents of an address register. Please note that 16-bit displacement values are sign extended to 32 bits.

Double (d)
: A double precision floating point number, stored in 64 bits.

Exception
: An exception is a hardware-detected event which disrupts the running of a program, process, or system.

Fault
: An exception, which, while halting the instruction, leaves the registers and memory in a consistent state. The instruction can often resume its course when the cause of the fault is corrected.

Flag
: A 1-bit operand that is generally used to indicate the results of an operation. The results are in the form true or false.

Floating point
: A numerical representation. A floating point operand has a sign (positive or negative, an exponent, and a fraction). The fraction is a fractional representation. The exponent is the value used to produce a power of two scale factor that is subsequently used to multiply the fractions to produce an unsigned value.

FORTRAN
: High-level software language mainly used for scientific applications.

Fraction
: A part of a floating point number. The fraction is the unsigned fractional part that denotes the magnitude of the operand.

Frame
: See Page Frame, Stack Frame

Function unit
: A function unit is a part of the central processing unit (CPU) which performs a set of operations on quantities stored in registers.

Gate array
: A structure that is used by the ring protection mechanism. The gate

array defines the entry points from a lower privileged ring to a higher privileged ring.

Gather
: Loading a vector register using another vector of indices instruction. See the ldvi instruction.

Guard bit
: A bit to the right (the least significant bit positions) of a floating point fraction. The guard bit is used in intermediate calculations using floating point operands.

Halfword (h)
: Two bytes (16 bits)

HUFFMAN's encoding
: A binary encoding that results in the densest packing of information.

Icache
: See Instruction Cache.

Immediates
: Operands which are contained within the instruction stream.

Indexing
: The process of adding a displacement to the contents of an address register.

Indirection
: The process of obtaining the address of an operand by first referencing a word contained within memory.

Instruction
: Used by the programmer to direct operations on the systems' register set and memory.

Instruction cache (ICACHE)
: The I-Cache contains the most recently accessed instructions. The I-Cache accelerates the decoding of instructions. This permits the simultaneous decoding on one instruction with the execution of another instruction.

Interrupt
: An occurrence other than an exception which changes the normal flow of instruction execution. An interrupt originates from hardware, such as an I/O device.

Interval timer
: A privileged register. The interval timer is used to generate an interrupt based on the passage of a period of time.

Kernel
: A part of the operating system that resides in ring 0. The kernel typically manages process creation and deletion, scheduling, and other

high level, system wide features.

Linker
A software tool. The linker "links" together separate software modules into one monolithic module.

Loads
A class of instructions which move data from memory to a register.

Locality of reference
An attribute of a memory reference pattern. Locality of reference refers to the likelihood of an address of a memory reference being numerically close to a recent memory reference address, or the likelihood of a subsequent memory reference being identical to a previous memory reference within a given period of time.

Logical address
Logical address space is that space seen by the application programmer.

Logical cache
The logical cache is a cache that is accessed with logical addresses for fast retrieval of data. It resides in the central processing unit.

Logical memory
Logical or virtual memory is that memory seen by the programmer. The logical memory of a CONVEX computer is 4 Gigabytes.

Longword (1)
Eight bytes (64 bits), the largest integer data type directly supported by hardware in the CONVEX-1.

LSI (Language Specific Information)
The area in the stack that is created as part of a subroutine call. It is langauge dependent and may be zero.

Machine exceptions
Machine exceptions include fatal errors in the system which cannot be handled by the operating system. (See Exception).

Main memory
See Physical Memory.

Maskable interrupt
An interrupt that is masked out. That is, an interrupt that the operating system wishes, at this time, not to respond to.

Memory management
The hardware and software features which control page mapping and protection.

Microcode

A control program that resides within the central processing unit. Microcode also refers to firmware, providing the necessary control that maps assembly language instructions onto processor hardware.

Modified bit
A bit within the central processing unit. The modified bit records all valid write references to pageframes. The modified bit is used by the operating system for memory management.

Negate
An instruction which performs a 2's complement.

Normalization
The process of left shifting a fraction until the leading bit is a 1.

Opcode
The code or sequence of bits in an instruction which determines the operation to be performed.

Operand
A register or memory location referenced by an instruction.

Orthogonality
A characteristic that pertains to the relationship of instructions and the operands they manipulate. An instruction set is orthogonal if one can change one property without having to change other related properties.

Packets
A group of related items. A packet may refer to the subroutine arguments or to a group of bytes that is transmitted over a network.

Page
A page is the unit of logical memory controlled by the memory management algorithms. In CONVEX-1, a page is 4 K (4096) contiguous bytes.

Pagefault
An exception caused by a reference to a valid non-existent page.

Page Frame
A page frame is the unit of physical (main) memory in which pages are placed. Associated with each pageframe are referenced and modified bits to aid in memory management.

Page Table Entry (PTE)
An entry in a page table. A PTE is a word. A PTE contains various flags and fields that are used in the translation of logical to physical addresses. Address translation uses two levels of page table indexing. The first level page table is referenced using bits 28 through 22 of a logical address. This is called the Index.1 field. The second level page table is referenced using bits 21 through 12 of a logical address. This is called the Index.2 field. See Figure 5-4.

Physical address

Hardware-identified address in physical (main) memory consisting of a page frame number and the number of a byte within the page.

Physical cache
The physical cache provides rapid access to recently used physical memory data items.

Pipelining
A technique used to construct high performance processors. Pipelining provides a means by which multiple operations occur concurrently.

Porting
Moving software from one type of machine to another.

Priority
An ordering of events. Priority is applied to protection levels as well as I/O interrupt levels.

Privileged instruction
An instruction used by the operating system or privileged systems programs. It must execute in ring 0, or an exception occurs.

Process
A process is the fundamental unit of program which is managed by the job scheduler.

Process exceptions
Process exceptions belong to the currently running process and may be handled with an exception handler in that process. The exception handler is in the current ring of execution. (See Exception).

Protection
A mechanism provided by hardware and software. Protection is used to ensure that one user is protected from another user or to ensure that a user does not perform an unsafe computation.

Processor Status Word (PSW)
A word that contains control flags. The PSW is used to control and indicate the state of various computations and sequences within the processor.

Push
The act of storing an operand on the stack.

Queue
A data structure in which entries are made at one end and deletions at the other. Often referred to as first-in first-out or FIFO.

Quotient
The result of a division operation.

Read
A memory operation in which the contents of a memory location are accessed and passed to another part of the machine.

Recursion
An arithmetic operation that uses the output of a calculation as the input of the same calculation.

Reduced Instruction Set Computer (RISC)
An architectural concept that applies to the definition of the instruction set of a processor. A RISC instruction set is an orthogonal instruction set that is easy to decode in hardware and for which a compiler can generate highly optimized code.

Reductions
An arithmetic operation that performs a transformation on an array that produces a scalar result.

Register
A hardware entity that is used to contain addresses, operands, and status.

Reservation
Reservation is the process of managing the various function units in the central processing unit. A reservation table is used to record the current status and availability of the function units.

Reset
The process of establishing a known state in a machine register.

Rings
A ring is the unit of logical memory used for protection purposes. There are five rings in CONVEX machines: four for system level usage and one for users. The system rings (Ring0-Ring3) each correspond to one Segment of logical memory, while the user ring (Ring4) contains four segments.

Ring Maximization
Ring maximization is the mechanism used to enforce protection in the logical address space.

Round bit
One of the two guard bits used in the intermediate representation of a floating point number.

Rounding
The process of transforming the intermediate representation of a floating point number to the memory representation. Unbiased rounding uses the round, guard, and sticky bits to determine the exact nature of this transformation. Truncation (as used in converting floating point to fixed point integer) does not use the round, guard, or sticky bits.

Runtime
A software module. A runtime is a software module that is referenced as a procedure. A runtime represents a required function that is not directly supported by the hardware, but it is required by the software.

Scatter
Storing a vector register using another vector of indices. See the stvi instruction.

Segmented ALU
A logic design technique that permits multiple arithmetic operations of the same type to be pipelined.

Segment
The segment is the basic partition of the logical memory space. A segment is 512 megabytes.

Segment descriptor register
Each segment of virtual memory has a segment descriptor register associated with it. Each SDR contains information pertinent to the access and mapping of virtual addresses.

Shift
A class of instructions used to shift the contents of a register right or left.

Single (s)
A single precision floating point number stored in 32 bits.

Source
A register or memory location used as an input to a CONVEX instruction.

Spatial reference
An attribute of a memory reference pattern. Spatial reference pertains to the likelihood of a subsequent memory reference address being numerically close to a previous address.

Stack
A data structure in which the last item entered is the first to be removed. Also referred to as last-in first-out (LIFO). In particular, stacks are used by the Call and Return instructions.

Sticky bit
A bit used in the intermediate calculations of floating point operands. The sticky bit remembers if any binary 1's were shifted out during an alignment or partial product operation.

Stores
A class of instructions used to move the contents of registers to memory.

Subroutine
A software module. A subroutine is a frequently used program that is called from various places in a program.

System exceptions
System exceptions cannot be handled by the current process; they require intervention by the kernel executing in ring 0. (See

Exception).

Trace of instruction execution
The process of tracking the execution of every instruction of a program.

Trap
An out of sequence branch due to the occurrence of an abnormal condition. Typically, this condition is a result of unexpected arithmetic results. (See Exception.)

Trojan Horse Pointer
The Trojan Horse Pointer is an addresss that is passed from one ring to another as part of a system call. In particular, this passed pointer references the more privileged ring as contrasted to the less privileged ring. This is unexpected and undesirable.

True Zero
A floating point number with zero sign bit, zero exponent, and zero fraction.

Unbiased rounding
The process of interpreting the round, guard, and sticky bits. Unbiased rounding, as contrasted to biased rounding, rounds to even in the event that the intermediate floating point result is exactly midway between two floating point representations.

UNIX
An operating system.

Unsigned
A value that is always positive.

Valid bit
A bit used in the control of caches. The valid bit is used to determine if a cache entry contains an entry that can be used.

Valid reference
A valid reference meets two requirements: first, the PTE must be valid (bit 31=1), and second, the type of access being made (Read, Write, or Execute) must be allowed by the appropriate protection bit (bits <3..1> of the PTE).

Vector
An array with one dimension.

Virtual address space
See Logical Address Space.

Word
Four bytes (32 bits)--the fundamental width of items in the CONVEX family of computers.

Working set

That portion of a user's program that is currently in physical memory. Typically the working set is much smaller that the user program.

Write
A memory operation in which a memory location is updated with new data.

Zero
In floating point number representations, zero is represented by a zero sign bit and zero exponent.

INDEX