CONVEX Assembly-Language User's Guide

Document No. 740-000030-204

Sixth Edition
October 1988

CONVEX Computer Corporation
Richardson, Texas

CONVEX Assembly-Language User's Guide
Order No. DSW-006
Sixth Edition

Printed in the United States of America

Revision Information for
CONVEX Assembly-Language User's Guide

Edition	Document No.	Description
Sixth	740-000030-204	Released with CONVEX UNIX V7.0, October 1988. Includes the following changes: • Chapter 2 — Describes features of CONVEX C200 Series architecture supporting parallel processing — Expands discussion of vector registers — Describes communication registers — Describes new fields in the processor status word (PSW) — Describes data type and memory alignment • Chapter 4 — Describes instruction typing — Describes extended opcodes — Describes operations under mask — Adds discussion of synchronization instructions used to control communication registers — Adds discussion of CPU control instructions used to requests CPUs to join in the computation of a process — Describes *.tdata* and *.tbss* program segments • Chapter 5 — Lists registers supported in CONVEX C200 Series architecture • Chapter 8 — Includes new vector routine — Includes sample parallel program • Appendix A — Includes new instruction set • Appendix B — Includes new instructions
Fifth	740-000030-202	Released with CONVEX UNIX V6.2, April 1988.
4.0	740-000030-201	Released with CONVEX UNIX V6.1, October 1987.
3.0	740-000130-000	Released with CONVEX UNIX V4.0, August 1986.
2.0	740-000130-000	Released with CONVEX UNIX V2.0, July 1985.
1.0	740-000130-000	Initial release with CONVEX UNIX V1.0, Feb. 1985.

Table of Contents

Appendices

List of Tables

List of Figures

Preface

Purpose and Audience

The *CONVEX Assembly-Language User's Guide* provides assembly-language programmers with the basic information needed to write, assemble, and tailor programs for processing on CONVEX supercomputers. Specifically, this guide

- provides pertinent facts about the CONVEX hardware architecture
- describes assembler instruction formats, opcodes, and operations
- describes conventions peculiar to writing assembly-language programs on CONVEX supercomputers
- introduces general procedures for using the assembler
- provides detailed examples of working assembly-language programs
- introduces information helpful to advanced programming

This guide is not a complete source for assembly-language programming on CONVEX supercomputers. Complete reference information for hardware-related issues is in the *CONVEX Architecture Reference*; UNIX commands and utilities are described in the *CONVEX UNIX Programmer's Manual.*

The *CONVEX Assembly-Language User's Guide* addresses users who are experienced programmers wishing to develop assembly-language programs on CONVEX supercomputers and who have the following experience:

- Assembly-language programming experience
- UNIX operating system experience

Organization

To learn fundamental information necessary to writing assembly-language programs on CONVEX supercomputers, read Chapters 1–6. To use the assembler, read Chapter 7. To review topics of interest for advanced programming, read Chapters 8–9, which provide examples of working assembly-language code and discuss higher-level programming.

Specifically, this guide is organized into the following chapters and appendices:

- Chapter 1, "Introduction," presents an overview of the CONVEX assembler, its operating environment, and support tools.
- Chapter 2, "Architecture Overview," describes fundamental information about CONVEX supercomputer architecture as it relates to assembly-language programming. This chapter includes information on addressing modes, memory addressing, the register set, operands, and the processor status word.

- Chapter 3, "Assembler Instruction Format," describes the command-line format and details the specific options of an assembly statement.
- Chapter 4, "Opcodes," discusses the machine operation codes, presents an overview of the instruction set, and describes pseudo-operations and assembler directives.
- Chapter 5, "Operands," discusses operands, including symbolic names, expressions, and addressing modes.
- Chapter 6, "Conventions," includes the calling conventions for the CONVEX assembly language, a brief explanation of linkages and normal register usage, and special conventions for C, FORTRAN, and the math library.
- Chapter 7, "Using the Assembler," tells how to invoke the assembler and describes error messages.
- Chapter 8, "Sample Programs," includes detailed examples of working assembly-language programs, including a block-copy, vector, and parallel program.
- Chapter 9, "Advanced Topics," discusses such techniques as optimizing performance, debugging, using macros and the preprocessor, and writing parallel programs.
- Appendix A describes the CONVEX supercomputer instruction set.
- Appendix B is an alphabetical list of opcodes sorted by name.
- Appendix C tells how to use the *contact*(1) utility to submit problem reports on software and documentation.

Notational Conventions

The following conventions are used in this document:

- Words enclosed in rounded rectangles indicate keyboard keys that you press. For example, RETURN refers to the carriage-return key. Words separated by a hyphen and enclosed in rounded rectangles indicate two keys that you must press simultaneously. For example, CTRL-X indicates that you must press the CTRL key while simultaneously pressing the keyboard ⊗ character key.
- **Boldface** type indicates user-entered information for a computer program. You should enter these command sequences exactly as they appear.
- Brackets ([ ]) designate optional entries. Brackets are also used to distinguish parts of command strings that may be omitted.
- A horizontal ellipsis (. . .) shows repetition of the preceding item(s).
- A vertical ellipsis shows continuation of a sequence where not all of the statements in an example are shown.
- Commands, utilities, and files that are documented in the *CONVEX UNIX Programmer's Manual* are italicized; occurrences that include a number enclosed in parentheses refer to the appropriate section of the manual (for example, *vmstat*(1) means that documentation for the *vmstat* command is located in Section 1 of the *Programmer's Manual*).

- The | symbol is used in command sequences in which you must pick no more than one alternative from a list of command options. For example, in the following command sequence,

  (fp)> **s[et] s[pu-selftest]** = [*d[isable]* | *e[nable]*]

  you must choose either *d[isable]* or *e[nable]*, but you may not choose both.

- The percent symbol (%) signifies the standard C shell user prompt.

Associated Documents

Using this guide successfully may require information not specific to the tasks described herein or not within the scope of this guide. The following documents may answer questions to help you solve problems encountered when working through this book. CONVEX Computer Corporation provides the following related documents:

- *CONVEX Architecture Reference.* This manual describes the architecture and instruction set used by the CONVEX computer system.

- *CONVEX UNIX Programmer's Manual,* Parts I and II. This guide documents the UNIX operating system.

  CONVEX adb (Assembly-Language Debugger User's Guide. This guide describes how to use the *adb* debugger to debug programs at the assembly-language level.

- *CONVEX C Compiler User's Guide.* This guide describes the commands necessary to run C on CONVEX supercomputers.

- *CONVEX Consultant User's Guide.* This guide describes several optional utilities, including the *csd* source-code debugger, the post-mortem dump (*pmd*) utility, and the *gprof, prof,* and *bprof* profilers.

- *CONVEX FORTRAN User's Guide.* This guide describes how to compile, link, debug, and execute programs working in FORTRAN on the CONVEX UNIX operating system.

- *CONVEX Loader User's Guide.* This guide describes how to use the loader for specific applications.

- *CONVEX UNIX Primer* provides an introduction to the CONVEX UNIX operating system. Recommended for novice UNIX users, it describes how to log in and start using the CONVEX system and explains many of the commonly used UNIX commands, the *vi* text editor, and the C shell.

Ordering Documentation

To order the current edition of this or any other CONVEX document, you need to know the exact title or the six-character order number. See the copyright page of this document for its six-character order number. To find the order numbers for other CONVEX documents, see the *CONVEX COMPUTER Price Book* or call the Technical Assistance Center or your local CONVEX office.

To order an edition other than the current edition, you need to know the 12-digit document number. See the title page of this document for its 12-digit document number. To find the document numbers for other CONVEX documents, call the Technical Assistance Center or your local CONVEX office.

To order CONVEX documentation, send requests to

CONVEX Computer Corporation
Customer Service
PO Box 833851
Richardson, TX 75083-3851 USA

Technical Assistance

If you have questions that are not answered in this book, contact the CONVEX Technical Assistance Center (TAC). To contact the TAC, use one of the following phone numbers:

Within the continental U.S.	1(800)952-0379
Outside continental U.S.	Contact local CONVEX office

Reader Response

If you have comments or questions about the contents of this book, please notify the CONVEX documentation department by using the "Reader's Forum" or the "Index Enhancements" form located at the end of this guide.

Chapter 1
Introduction

This chapter briefly introduces background information important to the use of the CONVEX assembler. Specifically, this chapter describes the relationship of the assembler to

- other CONVEX utilities
- CONVEX compilers
- CONVEX loader

The CONVEX assembler operates in conjunction with the CONVEX UNIX operating system, compilers, macro processors, debuggers, loader, object module libraries, and other auxiliary programs. The operating system is documented in the *CONVEX UNIX Programmer's Manual.* Utility programs are documented in the *CONVEX UNIX Utilities User's Guides.*

1.1 Utilities

Utilities typically used with the assembler include the *adb* debugger, the *make* utility, the *rcs* utility, and the *error* utility. The *CONVEX UNIX Programmer's Manual* completely describes these utilities. The *CONVEX adb Debugger User's Guide* further describes the use of *adb.*

1.1.1 Assembly-Language Debugger

The assembly-language debugger, *adb,* traces the execution of assembly-language programs. You can run *adb* on any executable program without reassembly. This debugger is also a useful tool for examining variables, setting breakpoints, and examining core dumps from failed programs.

1.1.2 *make* Utility

The *make* utility assembles source files as required to create target programs.

1.1.3 *rcs* Utility

The revision control system, *rcs,* archives revisions of assembly-language source files.

1.1.4 *error* Utility

The *error* utility merges assembler error messages into the assembler source file, where you can see the errors on the lines at which they occur.

1.2 Compilers

Once compiled, object files from the C compiler, FORTRAN compiler, and assembler can be loaded together for execution as long as compatible calling sequences are used. The FORTRAN compiler and the C compiler invoke the assembler when files that end in *.s* are specified on the compiler command line.

1.3 Loader

You may also invoke the assembler directly as described in Chapter 7. The assembler accepts an assembly-language source file as input and produces an object module as output. You may then invoke the loader to create an executable file.

1.4 Optimum Use of the Assembler

To assist you in making optimum use of the assembler, Chapter 9 discusses scheduling instructions, using caches, debugging, using macros and the preprocessor, and writing parallel assembly-language programs.

Chapter 2
Architecture Overview

This chapter describes general features of CONVEX supercomputer architecture that may help assembly-language programmers make informed programming decisions. Specifically, this chapter defines keywords and discusses the following topics:

- overview of CONVEX supercomputer architecture
- CONVEX register sets
- fields of the processor status word (PSW)
- operand lengths and types
- addressing modes
- data length and memory alignment
- virtual memory addressing

Before continuing in this chapter, familiarize yourself with the keywords listed in the next section.

2.1 Keywords

Process—A collection of one or more instruction streams executing within a single logical address space

Thread—Any single instruction stream executing within a process

Multiprocessing—The creation and scheduling of processes on any subset of CPUs in a system configuration

Processor status word (PSW)—A 32-bit register that contains coded bits indicating the current state of the processor

Stack pointer (SP)—Pointer to the top of the runtime stack, above which the next activation record is allocated

Frame pointer (FP)—Pointer to the last frame pushed on the runtime stack by a subprogram call

Argument pointer (AP)—Pointer to the location on the runtime stack of the first argument passed from the calling routine to the current routine

2.2 CONVEX Architecture

CONVEX supercomputers are uniprocessing and multiprocessing systems that incorporate vector processors within their CPUs. The system architecture combines 64-bit integrated scalar and vector processing with large real and virtual memory, high-performance I/O, UNIX-supported user applications, and flexible system software.

CONVEX supercomputers have several subsystems that can operate in parallel (including outboard I/O processors), in addition to a set of eight general-purpose registers, a set of eight address registers, and a set of eight vector registers (each vector register has a length of 128 elements). The CPU(s) can operate on integer data of lengths 8, 16, 32, and 64 bits as well as floating-point data of lengths 32 and 64 bits. A potentially large physical memory (1 Gbyte) with 4096-byte pages complements the large virtual memory (4 Gbytes).

2.3 Parallel Processing

The CONVEX C200 Series architecture provides the operating system and user with a flexible set of instructions for dynamic CPU allocation, deallocation, and communication. Each CPU in a CONVEX multiprocessor operates independently as a standard CONVEX 64-bit supercomputer. CPUs within a system configured for multiple CPUs share the same physical address space.

The multiprocessor-management hardware tightly couples these CPUs to allow a user application to execute in parallel. Synchronization instructions and CPU control instructions support parallel execution of a process, but do not enforce it. Because these instructions do not enforce parallel execution, you can write code that is independent of the number of CPUs.

The fundamental principle of operation for multiprocessing is that each CPU within a system configuration is responsible for its own scheduling. Each process *posts* the need for another CPU to join in the computation. An idle CPU recognizes the request and responds to it without the operating system intervening.

2.4 CONVEX Register Sets

CONVEX C1 architecture supports the following three general register sets:

- Address registers A0 through A7 (8 × 32 bits)
- Scalar registers S0 through S7 (8 × 64 bits)
- Vector registers V0 through V7 (8 vectors, each 128 elements × 64 bits).

CONVEX C200 Series architecture supports the above registers plus a set of communication registers that allow for synchronization and communication between threads of a process.

These register sets and the processor status word (PSW) are described in the following sections.

2.4.1 Address Registers

The address registers consist of eight 32-bit registers. The address registers do not support floating-point operations or 64-bit integer arithmetic, whereas the scalar and vector registers support all arithmetic operations.

Address register A0 is the stack pointer (SP), A6 is the argument pointer (AP), and A7 is the frame pointer (FP). For more information on the argument pointer, stack pointer, and frame pointer, see Chapter 6, "Conventions."

2.4.2 Scalar Registers

The scalar registers consist of eight 64-bit registers. The scalar registers support all arithmetic operations.

2.4.3 Vector Registers

The vector registers consist of eight vectors, each 128 elements by 64 bits. These registers support all arithmetic operations and allow you to speed up program execution by pipelining repetitive operations. To add 128 pairs of numbers, you need to specify six operations:

- Set vector length register to 128
- Set vector stride
- Vector load—first of each pair
- Vector load—second of each pair
- Vector add
- Vector store—of result

The integrated vector processor processes the operation much more quickly than it processes a loop.

Three special hardware registers support vector processing:

- Vector length—A number from 0 to 128 that indicates how many elements of a vector register are to be used
- Vector stride—The number of bytes between successive memory addresses whose contents are loaded or stored into the vector register
- Vector merge—A 128-bit register that indicates which elements of a vector are to participate in some operations

2.4.4 Communication Registers

The hardware communication registers are a set of high-speed registers used to exchange data between the threads of a process. All threads of a process share the same communication registers.

The C200 Series hardware supports 8 sets of 128 communication registers. Each CPU is linked to a set of communication registers by means of the communication index register (CIR), which is assigned an appropriate value by the operating system.

2.4.4.1 Communication Register Addressing

Each communication register is an addressable 64-bit register. Of the 128 communication registers indexed by the CIR, 64 are reserved for the hardware and operating system and 64 are available for user applications.

Communication register addressing is based on the CIR. That is, communication registers are not addressed as 1024 contiguous registers (8 × 128), but as 8 sets of 128. The 64 communication registers you can access are numbered from 8000 (hex) through 803F (hex).

NOTE

The first 32 user-accessible communication registers are reserved for compilers and runtime libraries. Use communication registers 8020 (hex) through 803F (hex) for assembly-language applications.

2.4.4.2 Hardware Lock Bit

Each communication register is associated with a hardware maintained *lock* (i.e., semaphore) bit. This lock bit synchronizes threads of a process running in parallel by

- checking to see whether or not an event has completed
- checking to see whether or not a communication register has valid data
- controlling *critical regions* of code

This lock bit may be independently manipulated using synchronization instructions (see section 4.4.14, "Synchronization Instructions").

2.4.5 Processor Status Word (PSW)

The processor status word (PSW) is a 32-bit register that contains coded bits indicating the current state of the processor. This register contains flags that enable or disable exception processing and indicate the results of numerical operations. The PSW contains no privileged mode bits. Figure 2-1 illustrates how the bits are arranged in a C100 Series PSW. Figure 2-2 illustrates how the bits are arranged in a C200 Series PSW.

Chapter 3
Assembler Instruction Formats

This chapter illustrates and describes the format of an assembly-language instruction. Specifically, this chapter defines keywords and discusses the following topics:

- assembly-language instruction format
- label field in an assembler instruction
- operation field in an assembler instruction
- operand field in an assembler instruction
- comment field in an assembler instruction
- terminator field in an assembler instruction

Before continuing this chapter, familiarize yourself with the keywords listed in the next section.

3.1 Keywords

Name label—A user-defined symbolic name that allows instructions to refer to a specific address within the program

Instruction mnemonic—A symbolic name for a machine instruction

Assembler directive—A command to the assembler that either alters its operation or causes it to reserve storage for data values

3.2 Instruction Format

An assembly-language program is a sequence of instructions and assembler directives. Appendix A defines the CONVEX instruction set, while Chapter 4 describes the assembler directives.

An instruction consists of five fields: a label field, an operation field, an operand field, a comment field, and a terminator field. Only the terminator field is required. Figure 3-1 shows the instruction format and a sample instruction.

Figure 3–1: Instruction Format

[label]	[operation]	[operand]	[comment]	terminator
looptop:	**ld.w**	**#1,s0**	**;initialize loop counter**	<NL>

The assembler imposes the following input-formatting restrictions:

- No card-column boundary restrictions exist.
- No facility for line continuation exists.
- Fields within the instruction may be separated by any number of spaces and tabs.

All fields in a single statement must appear on the same source line.

3.3 Label Field

Labels allow instructions to refer to specific addresses within the program. The assembler supports two types of labels: name labels and temporary labels. A name label is a user-defined symbolic name. The name "looptop" in Figure 3-1 is a name label. A name label remains defined throughout the source file that contains the label definition. You cannot redefine a name label. Temporary labels have a similar function; temporary labels, however, may be redefined and reused many times within a source file. These labels are typically used for local branches.

3.3.1 Name Labels

Use name labels to define the targets of branch instructions and variable data locations whose addresses are needed throughout a program.

To define a name label, begin an instruction with a symbolic name followed immediately by a colon, as shown in the label field of Figure 3-2. When a name label is defined, it assumes the current value of the assembler location counter.

Figure 3–2: Name Labels Example

```
Label    Operation   Operand     Comment
start:   st.w        #0,count    ;defines the label ''start''
                                 ;and references the label
                                 ;''count''
         .
         .
         .
count:   bs.w        1           ;defines the label ''count''
         .
         .
         .
         jmp         start       ;references the label ''start''
```

3.3.2 Temporary Labels

Use temporary labels to define the instructions that are the targets of branch instructions generated by a program such as a compiler or macroprocessor. Temporary labels can be redefined and reused within a source file and have the form "n$:", where *n* is a 32-bit positive decimal integer. Temporary labels remain defined until the next name label is defined, or until you invoke an assembler directive that changes the value of the current location counter.

Temporary labels eliminate the need for large numbers of unique name labels, as shown in the label field of Figure 3-3.

Figure 3–3: Temporary Labels Example

```
Label    Operation   Operand     Comment
1$:      add.w       #1,a1       ;defines the temporary label ''1$''
         st.w        a1,data
         jmp         1$          ;references the first label ''1$''
next:                            ;next name label undefines ''1$''
         .
         .
         .
1$:      eq.w        #0,a1       ;another (different) definition of ''1$''
         .
         .
         .
         jmp         1$          ;references the second label ''1$''
```

Similar to name labels, temporary labels assume the current value of the location counter when they are defined.

3.4 Operation Field

The second instruction field is the operation field. The operation field contains a mnemonic corresponding to the operation to be performed. Operations are of two types: instruction mnemonics and assembler directives. Following is a brief description of instruction mnemonics and assembler directives.

3.4.1 Instruction Mnemonics

Instruction mnemonics are symbolic names for machine instructions. Including an instruction mnemonic in the operation field of a statement causes the assembler to generate the binary representation for that machine instruction along with correct codes for the operand. For more information on instruction mnemonics, see Chapter 4, "Opcodes."

3.4.2 Assembler Directives

Assembler directives are commands to the assembler that either alter its operation or cause it to reserve storage for data values. Assembler directives do not generate binary instructions. For more information on assembler directives, see Chapter 4, "Opcodes."

3.5 Operand Field

If the operand field is present, it consists of one or more operands separated by commas. The number and type of operands present depend on the operation being performed. For example, the number and type of operands required for the *add* instruction mnemonic depend on the register set being used. To determine the number of operands for a particular instruction, check the instruction descriptions in Appendix A. You can use spaces or tabs between operands, but not within an individual operand.

The operand field generally contains one or more operands that specify a machine register or a memory location using one of the computer addressing modes. For further information, see section 5.3, "Addressing Modes."

3.6 Comment Field

The comment field is optional and typically describes the operations being performed. The comment field begins with a semicolon and ends with a statement terminator (which also ends the statement). A comment may contain any sequence of characters as long as the sequence does not contain one of the termination characters. The assembler ignores the characters in the comment field. Comments may start at any location within the input line.

3.7 Terminator Field

To end an assembly-language statement, use one of three termination characters: newline `<nl>`, form feed `<ff>`, or " ! ". Use the ! character to separate multiple statements entered on a single input line. The terminator field is the only required field in an assembly-language statement.

Chapter 4
Opcodes

This chapter describes machine opcodes and pseudo-operations. Specifically, this chapter defines keywords and discusses the following topics:

- instruction typing
- extended opcodes
- machine opcodes, including
  - memory-reference instructions
  - calculation of effective addresses
  - memory-reference instruction format
  - arithmetic instructions
  - logical instructions
  - data-type specification
  - data-conversion instructions
  - vector-reduction instructions
  - operations under mask
  - program-control instructions
  - span-independent program-control instructions
  - machine-control instructions
  - synchronization instructions
  - CPU control instructions
- pseudo-operations, including descriptions of assembler directives

Before continuing in this chapter, familiarize yourself with the keywords listed in the next section.

4.1 Keywords

Critical region—A region of code that modifies data or computer resources shared by more than one process

Extended opcodes—Term referring to opcodes that have halfword prefixes modifying the behavior of the instruction

Instruction typing—Term referring to the fact that each CONVEX assembly-language instruction is associated with a machine type in the CONVEX family of supercomputers

Operations under mask—Term referring to the ability to include or exclude different vector elements in the instruction depending on whether the vector-merge bit for an element is set to 1 or 0

Prefix opcode—Halfword value that precedes a standard opcode and modifies its behavior

Process—A collection of one or more instruction streams executing within a single logical address space

Thread—Any single instruction stream executing within a process

Thread data—Initialized data that is unshared among all threads comprising a process, but is unique to only that one thread

Thread bss—Uninitialized data that is unshared among all threads comprising a process, but is unique to only that one thread

4.2 Instruction Typing

Not all instructions in the CONVEX assembly-language instruction set run on every machine in the CONVEX family of supercomputers. Each instruction is tagged within the assembler to be of a certain type. The *type* of an instruction specifies the machine(s) that will accept the instruction. The types of all instructions in one assembled file are accumulated in the "flags word" of the executable file's header. The kernel examines these flags to determine whether the machine can or cannot accept the file.

Appendix A, "Instruction Set," lists the CONVEX systems on which each instruction will run. See the "Architecture" field in section A.2, "Instruction-Page Layout."

4.3 Extended Opcodes

An opcode is the sequence of bits in an instruction that determines the operation to be performed. Extended opcodes are one- to three- halfword CONVEX opcodes prefixed with a 16-bit modifier. This modifier is called a *prefix opcode*. The prefix opcode has a bit (called the E bit) set to 1 (true) or 0 (false) that modifies the behavior of the instruction. For example, following are three instructions: a standard opcode (prefixed ST) and two extended opcodes (prefixed E1 and E0).

Mnemonic	Hex	Binary
eq.b Vj,Vk	0x6800	ST 0110100000
eq.b.t Vj,Vk	0x7EF8 0x6800	E1 0110100000
eq.b.f Vj,Vk	0x7EF0 0x6800	E0 0110100000

The two prefix opcodes are defined in hex as 0x7EF8 and 0x7EF0.

The extended opcode format allows for the addition of many new instructions, a majority of which are vector operations. The C200 Series architecture supports these new vector instructions by recognizing *operations under mask*, which employ the extended opcode format. For further information on these instructions, see Section 4.4.10, "Operations Under Mask."

For further information on extended opcodes, see the *CONVEX Architecture Reference.*

4.4 Machine Opcodes

When specified, the operation field of an instruction contains a mnemonic. This mnemonic is the symbolic name for a binary machine instruction. The assembler uses the instruction mnemonic, the operand types, and possibly a data-type specification to determine the appropriate bit pattern required for the instruction.

The CONVEX instruction set is divided into the following categories:

- Memory-reference instructions
- Arithmetic instructions
- Logical instructions
- Data-conversion instructions
- Vector-reduction instructions
- Program-control instructions
- Span-independent program-control instructions
- Machine-control instructions
- Synchronization instructions
- CPU control instructions

The following sections describe these instructions and requisite information.

4.4.1 Memory-Reference Instructions

Use memory-reference instructions to load registers from memory, store the contents of registers in memory, and modify memory. Table 4-1 lists the memory-reference instructions.

Table 4-1: Memory-Reference Instructions

Instruction	Description
ld	Load register from memory
st	Store register into memory
tac *	Test and clear a memory location
tas	Test and set a memory location
psh	Push register onto memory stack
pop	Pop the top element on the stack into a register
ldea	Load the effective address into an address register
pshea	Push the effective address onto the stack
ldvi	Load a vector register with a vector of indices
stvi	Store a vector register with a vector of indices

* C200 Series only

4.4.2 Effective Address Calculation

Memory-reference instructions calculate an effective address by adding the value found in the displacement field of the instruction (see Figure 4-2) to the contents of the A register specified in the instruction. The A register is referenced by the *Aj* field. When no register is specified or when A0 is specified, 0 (instead of A0) is added to the value of the instruction displacement field.

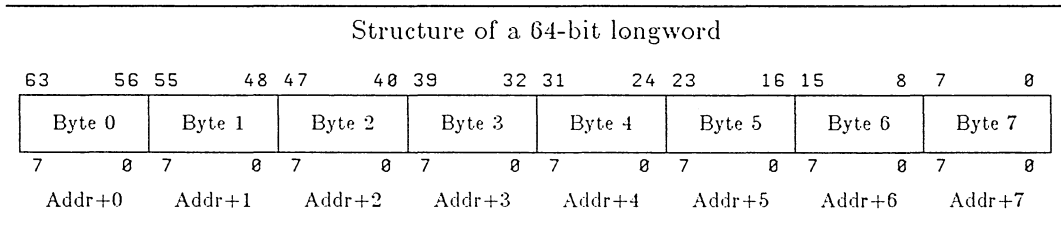
Memory-reference instructions may specify indirect addressing. When indirect addressing is used, the calculated effective address is not a final address, but a reference to a location in memory that contains the final effective address.

Instructions that reference memory are usually 32, 48, or 64 bits long. The difference in length is caused by a difference in the length of the displacement field and the presence or absence of the prefix halfword.

4.4.3 Memory Longword Structure

Figure 4-1 illustrates the ordering of each addressable entity within a 64-bit longword.

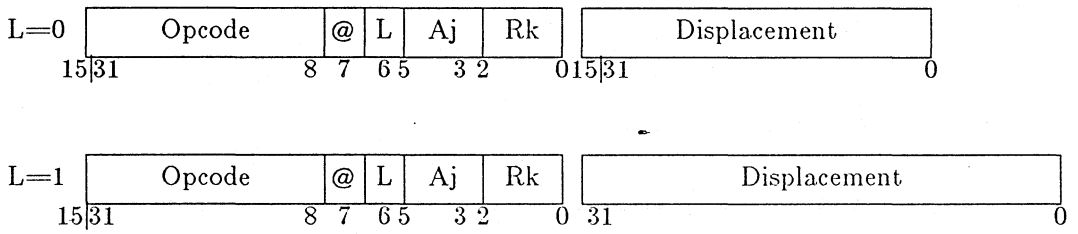
Figure 4–1: Memory Longword Structure



4.4.4 Memory-Reference Instruction Format

Figure 4-2 shows the structure of memory-reference instructions.

Figure 4–2: Memory-Reference Instruction Format



The fields in Figure 4-2 are defined as follows:

Opcode = bits<15|31..8> Specifies which operation to perform on the referenced data, and selects the register class (A, S, or V) referred to by the *Rk* field.

@ = bit<7> Indirection Specifies the presence or absence of indirection. If @=0, no indirection is specified; the effective address is the sum of the contents of *Aj*, plus the displacement. If @=1, indirection is specified; the effective address is the address stored at the location computed by adding the contents of *Aj* to the displacement.

L = bit<6> Length Specifies the length of the displacement field. If L=0, the displacement field is a two's complement 16-bit integer. This field is sign-extended to 32 bits before it is added to the effective address. If L=1, the displacement field is a two's complement 32-bit integer. The assembler automatically generates the instruction length. The length is based on the displacement.

Aj = bits<5..3> Specifies the *A* (address) register used to generate the logical address. If you specify *A0* or if you do not specify a register, the value of 0 is used as the contents of *A0* for absolute addressing. The true contents of *A0* are unused. If you specify *A1-7*, the contents of the specified address register are added to the displacement field to generate the effective target address. If indirection is not specified, the effective address is the target address. If indirection is specified, the effective target address references byte 0 of a 32-bit word that contains the effective address.

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

Displacement This field contains a 16-bit or 32-bit value that is either added to the contents of an *A* register or used directly as a byte address. A 16-bit displacement value is sign-extended to 32 bits before use.

4.4.5 Arithmetic Instructions

Use arithmetic instructions to do arithmetic operations on the contents of machine registers and to store the results of an arithmetic operation in a machine register. Table 4-2 lists the arithmetic instructions.

Table 4-2: Arithmetic Instructions

Instruction	Description
add	Add the operands
sub	Subtract the operand
mul	Multiply the operands
div	Divide the operand
neg	Negate the operand
leu	Compare less than or equal unsigned operands
ltu	Compare less than unsigned operands
le	Compare less than or equal operands
lt	Compare less than operands
eq	Compare equal operands
ne	Compare not equal operands

4.4.6 Logical Instructions

Use logical instructions to do logical operations on the specified register operands. Table 4-3 lists the logical instructions.

Table 4-3: Logical Instructions

Instruction	Description
and	Perform the logical product on the operands
or	Perform the logical sum on the operands
not	Perform the one's complement on the operands
xor	Perform the exclusive-or operation on the operands
lop	Determine bit position of left-most one bit
tzc	Perform a count of trailing zero bits
mov	Copy source operand
shf	Logically shift contents of the register
plc	Count the population (number of ones) in the operand

4.4.7 Data-Type Specification

Many instructions operate on data of a specific width, or size. Most CONVEX instruction mnemonics include a data-type specifier along with the basic instruction name. The width specifier consists of a period followed by a single letter that specifies the desired operand size. For example, *ld.w* is a load (*ld*) instruction whose data size, *.w*, is a 32-bit machine word. (Several data-type specifiers also include a second letter that specifies either the upper or lower half of a 64-bit constant.) Table 4-4 lists the data-type specifiers for CONVEX assembly language.

Table 4-4: Data-Type Specifiers

Specifier	Description
.b	Byte data (8 bits)
.h	Halfword data (16 bits)
.w	Word data (32 bits)
.l	Longword data (64 bits)
.s	Single-precision floating-point data (32 bits)
.d	Double-precision floating-point data (64 bits)
.x	Extended data (128 bits)
.u	Upper word of longword (32 bits)
.lu	Upper word of a 64-bit longword constant
.ll	Lower word of a 64-bit longword constant
.du	Upper word of a 64-bit double-precision floating-point constant
.dl	Lower word of a 64-bit double-precision floating-point constant

4.4.8 Data-Conversion Instructions

Use data-conversion instructions to convert data from one format to another. The CONVEX instruction set does not provide a complete set of conversion combinations, so you may find it necessary to use data-conversion instructions in combinations. Table 4-5 lists the data-conversion instructions.

Table 4-5: Data-Conversion Instructions

Instruction	Description
cvtb.w	Convert a byte to a word
cvth.w	Convert a halfword to a word
cvtw.b	Convert a word to a byte
cvtw.h	Convert a word to a halfword
cvtw.s	Convert a word to single-precision floating-point format
cvtw.l	Convert a word to a longword
cvtl.s	Convert a longword to single-precision floating-point format
cvtl.d	Convert a longword to double-precision floating-point format
cvtl.w	Convert a longword to a word
cvts.w	Convert a single-precision floating-point number to a word
cvts.d	Convert a single-precision floating-point number to double-precision format
cvts.l	Convert a single-precision floating-point number to longword format
cvtd.s	Convert a double-precision floating-point number to single-precision format
cvtd.l	Convert a double-precision floating-point number to longword format

4.4.9 Vector-Reduction Instructions

Use vector-reduction instructions to reduce vector operands to a scalar value. Table 4-6 lists these instructions.

Table 4-6: Vector-Reduction Instructions

Instruction	Description
sum	Perform the summation (sigma) operation on a vector
prod	Perform the product (pi) operation on a vector
all	Perform the logical product (AND) operation on the elements of a vector
any	Perform the logical summation (OR) operation on the elements of a vector
parity	Perform the exclusive-or operation on the elements of a vector
max	Find the maximum element of a vector
min	Find the minimum element of a vector

4.4.10 Operations Under Mask

Most vector instructions have the ability of operating under mask. These instructions use the extended opcode format; that is, they are prefixed with a halfword modifier to operate under true or false mask (see section 4.3, "Extended Opcodes").

Operations under mask work as follows: the vector-merge (VM) register has a bit associated with each vector register element. When an operation is performed under mask, each element is either included or excluded from the operation based on the state of its corresponding VM bit. Operations under mask assume two forms:

True Elements with the VM bit set to 1 are included in the instruction; these instructions have a *.t* suffix.

False Elements with the VM bit set to 0 are included in the instruction; these instructions have a *.f* suffix.

Following are two examples of operations under mask—*add.w.t* and *add.w.f*. In these examples, assume the following initial values:

V0=0 1 2 3 4 5 VL=6

V1=6 7 8 9 2 3 VM=0 1 1 0 0 1

V2=5 5 5 5 5 5

Performing an *add.w.t* V0,V1,V2 produces the following:

V2=5 8 10 5 5 8

Performing an *add.w.f* V0,V1,V2 produces the following:

V2=6 5 5 12 6 5

4.4.11 Program-Control Instructions

Use program-control instructions to call and return from subroutines and to jump or branch to specific locations or labels. Table 4-7 lists these instructions. The term "pc" means program counter.

Table 4-7: Program-Control Instructions

Instruction	Description
call	Call a subroutine in long format
calls	Call a subroutine in short format
callq	Call a subroutine in quick format
rtn	Return from a subroutine
rtnq	Return from a subroutine that was called with a callq instruction
bkpt	Perform a trap for the debugger
pbkpt	Force a process breakpoint
sysc	Perform a system call
jmp	Jump to the specified location
jmpi.f	Jump on interrupt-on false
jmpi.t	Jump on interrupt-on true
jmpa.f	Jump if address carry is false
jmpa.t	Jump if address carry is true
jmps.f	Jump if scalar carry is false
jmps.t	Jump if scalar carry is true
br	Perform a pc-relative branch
bri.f	Perform a pc-relative branch if the interrupt-enable flag is false (off)
bri.t	Perform a pc-relative branch if the interrupt interrupt-enable flag is true (on)
bra.f	Perform a pc-relative branch if the address carry is false
bra.t	Perform a pc-relative branch if the address carry is true
brs.f	Perform a pc-relative branch if the scalar carry is false
brs.t	Perform a pc-relative branch if the scalar carry is true

4.4.12 Span-Independent Program-Control Instructions

Use span-independent program-control instructions to access generic forms of the jump and branch instructions. The assembler resolves span-independent program-control instructions into either jump or branch instructions, depending on the value of the displacement field in the instruction.

Instructions beginning with *jmp* transfer program control to the absolute address specified in the address portion of the instruction. Instructions beginning with *br* determine the address to branch to by adding or subtracting from the current value of the program counter. The quantity added or subtracted is passed to the instruction as an argument.

Span-independent instructions begin with *jbr* and accept absolute addresses, relative addresses, or labels as arguments. If you use labels with a *jbr* instruction, the assembler calculates the relative distance to the location specified and the width of displacement to be used. The assembler also

generates the appropriate form of either *jmp* or *br* when resolving the *jbr* instructions. The assembler changes displacement widths when you change locations or when you modify the program.

Table 4-8 lists these instructions.

Table 4-8: Span-Independent Program-Control Instructions

Instruction	Description
jbr	Jump or branch to the specified location
jbrs.f	Jump or branch if the scalar carry is false
jbrs.t	Jump or branch if the scalar carry is true
jbra.f	Jump or branch if the address carry is false
jbra.t	Jump or branch if the address carry is true
jbri.f	Jump or branch if the interrupt-enable flag is false (off)
jbri.t	Jump or branch if the interrupt-enable flag is true (on)

4.4.13 Machine-Control Instructions

Use machine-control instructions to control the various units within the machine. These instructions are used principally by the operating system and are of little use in normal system and application programs. Table 4-9 lists these instructions.

Some of these instructions are *privileged* instructions that cannot be used in user programs. Please refer to the *CONVEX Architecture Reference* for detailed information on privileged instructions.

Table 4-9: Machine-Control Instructions

Instruction	Description
eni	Enable interrupts
dsi	Disable interrupts
patu	Purge the ATU
pate	Purge a single ATU entry
ldsdr	Load a segment-descriptor register
ldkdr	Load the kernel segment-descriptor register
halt	Halt the processor
exit	Error exit of a program
nop	Perform no operation
pich	Purge the instruction cache
plch	Purge the logical cache
xmti	Transmit interrupt
mski	Mask interrupt

4.4.14 Synchronization Instructions

Recall that communication registers allow for communication between threads of a process (see section 2.4.4, "Communication Registers"). These registers control the process flow after a process divides into threads. Synchronization instructions let you manipulate these communication registers to manage the execution of parallel programs.

In particular, two sets of synchronization instructions manage communication registers:

- *lck* (lock) and *ulk* (unlock) instructions
- *snd* (send) and *rcv* (receive) instructions

These synchronization instructions use the address carry bit (C) or scalar carry (SC) bit in the PSW to indicate success or failure. The following sections discuss these instructions in more detail.

4.4.14.1 *lck* and *ulk*

The *lck* and *ulk* instructions use the hardware *lock* bit as a semaphore (see section 2.4.4, "Communication Registers"). The *lck* instruction sets the lock bit. If the lock bit is clear, then C is set. If the lock bit is already set, then C is cleared to indicate failure.

The *ulk* instruction clears the lock bit, and C is set to the previous value of the lock bit.

The *lck* and *ulk* instructions may be used to

- check to see whether or not an event has completed
- define *critical regions* of code

4.4.14.2 *snd* and *rcv*

The *snd* instruction moves data from an A or S register to a communication register and sets C or SC if the communication register is unlocked. If the communication register is locked, *snd* leaves the communication register unchanged and clears C or SC, depending on whether an A or S register was specified.

The *rcv* instruction moves data from a communication register to an A or S register and sets C or SC if the register is unlocked. If the register is locked, then C or SC is cleared.

NOTE

The *snd* and *rcv* instructions automatically handle locking. You do, however, need to supply branch-back instructions to handle instances when *snd* and *rcv* temporarily fail because of the action of another thread.

The *snd* and *rcv* instructions use the hardware lock bit to

- control critical regions of code
- determine whether a communication register has valid data.

A communication register is locked when it has valid data.

Table 4-10 lists these and other synchronization instructions used to manipulate communication registers. For further information on these instructions, see Appendix A, "Instruction Set."

Table 4-10: Synchronization Instructions

Instruction	Description
ulk	Unlock a communication register
lck	Lock a communication register
snd	Send the contents of a register to a communication register
rcv	Receive the contents of a communication register into a register
inc	Increment a communication register by another register and return the new value
sndr	Send the contents of a register to a synchronized resource structure
rcvr	Receive the contents of a synchronized resource structure in memory into a register
popr	Pop a word from a resource structure at effective address into an address register
pshr	Push an address register onto the resource structure at effective address

For an illustration of how these instructions may be used, see Figure 8-3, "Parallel Program" and section 9.6, "Parallel Programming in CONVEX Assembly Language."

4.4.15 CPU Control Instructions

Allocating CPUs in the CONVEX multiprocessor architecture is defined in terms a *fork*. A fork is a request to the CPU. CPU control instructions allow you to

- Post a fork, which requests a CPU to share in the computation of a single process
- Clear a fork
- Force the current CPU back into an idle state
- Force a multithreaded process back into a single thread

CPU control instructions essentially offer two programming schemes. One scheme uses *pfork*, *wfork*, and *cfork* instructions. The other scheme uses *spawn* and *join* instructions. These programming schemes represent different ways of synchronizing resources and allocating available CPU time. For more information on these programming schemes, see section 9.6, "Parallel Programming in CONVEX Assembly Language," and the *CONVEX Architecture Reference*. The following sections briefly describe these instructions.

4.4.15.1 *pfork*, *wfork*, and *cfork*

The *pfork* instruction requests that one additional thread start executing. The new thread

- begins at a PC of the effective address
- uses the stack pointer specified in Ak field of the *pfork* instruction
- inherits the PSW, FP, and AP of the parent thread

The *wfork* instruction causes the thread to terminate and immediately begins executing a *pending* thread. A pending thread is a thread request (in this case, a *pfork* request) that has been posted and is waiting to be accepted and processed by an available CPU. The *wfork* instruction does not, however, clear pending threads. The *cfork* instruction clears pending threads.

4.4.15.2 *spawn* and *join*

The *spawn* instruction requests that as many threads start executing as there are processors available. The new threads

- begin at a PC of the effective address
- use the stack pointer specified in Ak field of the *spawn* instruction
- inherit the PSW, FP, and AP of the parent thread

The *join* instruction reduces the process to a single thread of execution. Threads in a process reach a *join* and terminate their work. Each CPU terminates its thread of execution and either returns to an idle state or continues execution after the *join* as a single-threaded process.

Table 4-11 lists these and other CPU control instructions used to request other CPUs to assist in the computation of a single process.

Table 4-11: CPU Control Instructions

Instruction	Description
spawn	Post the need for as many CPUs as possible to assist in the computation of a process
join	Force all threads created by a process to join at a single execution point
pfork	Post the need for a CPU to assist in the computation of a process
cfork	Clear a fork
wfork	Terminate a thread, idle the current CPU, and look for any posted forks
idle	Attempt to accept a posted fork in the specified communication index register (CIR); if a fork is not posted, idle the current CPU without deallocating the current thread

For further information on these instructions, see Appendix A, "Instruction Set."

4.5 Pseudo-Operations

Pseudo-operations are assembler directives that either alter the assembler's operation or cause the assembler to reserve storage for data values. The syntax of these directives is similar to the instruction set syntax.

This section describes the following directives:

- Program-segment directives
- Initialized- and uninitialized-storage-allocation directives
- Alignment directives
- Floating-point directives
- External symbolic name directives
- Symbol-table directives

4.5.1 Program-Segment Directives

Address space in CONVEX UNIX programs is divided into six sections called program segments. A program segment is block of code identified by user-defined program-segment directives. The following directives identify the type of data included in each respective program segment.

.text	code
.data	initialized data
.tdata	initialized thread data
.bss	uninitialized data
.tbss	uninitialized thread data
.cdata	initialized common block

The assembler begins generating instructions and data in the *.text* segment. The program cannot write into the *.text* segment (read-only data can be stored in this segment). The program may, however, write into the *.data*, *.tdata*, *.bss*, *.tbss*, and *.cdata* segments.

The *.tdata* and *.tbss* directives are specifically designed to create thread data segments with unshared memory. Use these directives when you need to write thread memory to the same virtual address, as in a parallelized loop. Unshared thread memory is managed by the thread identifier (TID) register, which makes threads unique in the translation from virtual to physical memory. Data within *.tdata* and *.tbss* directives are mapped to *.data* and *.bss* when run on a C1 system.

Threads with unshared memory may require separate stack spaces. The behavior of programs that do not have separate stack spaces for different threads is difficult to debug. To create an individual thread stack for uninitialized thread data, use the *.tbss* directive to allocate data space and then assign a thread SP to that space.

The *.cdata* segment differs from the other program segments in that the current program counter for each *.cdata* segment is set back to zero, while the program counter is always incremented for the other segments. You must precede each *.cdata* directive with a *.comm* directive, that allocates the space to be filled by the *.cdata* segment (*.comm* directives are discussed in detail in the section 4.5.5, "External Symbolic Name Directives"). You can use any number of *.cdata* segments in your program, as long as they each follow a *.comm* directive.

The *.cdata* segment generates a fatal warning if it exceeds the space allotted by the *.comm* directive. If *.cdata* exceeds the space allotted by the *.comm* directive, adjust the *.comm* directive to the correct size and reassemble the program.

If multiple *.cdata* directives exists for the same location within one file, the assembler checks to make sure that initialized locations are not reinitialized or partially overwritten. Do not reinitialize variables within a common block or a fatal assembler error will occur. If multiple *.cdata* directives exists among several files, the loader overlays multiple initialized common sections, then issues a warning.

Figure 4-3 shows the use of these program segments.

These program segments can be subdivided into four subsections (0, 1, 2, or 3). Use these subsections to specify code and data in a convenient order. The assembler collects code or data generated in each subsection in the order in which it appears. Format the program-segment directives to specify subsections as follows:

.text *n*
.data *n*
.tdata *n*
.bss *n*
.tbss *n*
.cdata *name*

where *n* is 0, 1, 2, or 3, or an absolute symbolic name whose value is 0, 1, 2, or 3. The subsection default is 0. Subsections are output in increasing numeric order. For the *.cdata* directive, *name* is the name of a common block, and you may use any number of these.

Figure 4–3: Use of Program Segments

```
            .text                           ;current section is .text
;
;  Assembly is taking place in the text
;  section.
;
start:      ld.w #1000,a1
            ld.w data,a2

            .data                           ;current section is .data
            .comm  page,  48                ;allocate space for .cdata
            .cdata page                     ;current section is .cdata
;
;  Set up an initialized data table
;
data:       ds.w 1,2,3,4,5,6,7,8,9,10,11,12

            .bss                            ;current section is .bss
;
;  Set up uninitialized data table
;
udata:      bs.w 1000
;
;  Now back to the text section
;
            .text                           ;current section is .text
;
;  Continue with the program code
            mov a1,a3
```

4.5.2 Storage Directives

To allocate storage, use *bs*, the block-storage directive, and *ds*, the define-storage directive. Note that storage is not automatically aligned. For information on storage alignment, see section 4.5.3, "Alignment Directives."

The following sections describe using the *bs* and *ds* directives.

4.5.2.1 Uninitialized Storage Allocation

When you specify the *bs* directive, the assembler allocates a specified number of storage locations of a particular width. The format of the directive is as follows:

bs.*x* *n*

where *x* is one of the data-type specifiers *b*, *h*, *w*, *l*, *s*, or *d* (see Table 4-4). The single operand *n* represents the number of units of the specified width *x* to be allocated. If the directive has a label, the value assigned to the label is the address of the left-most byte of the first unit. The following example illustrates the use of the *bs* directive.

```
abc:    bs.w    50      ;Allocate 50 words of storage
xyz:    bs.l    10      ;Allocate 10 longwords of storage
```

4.5.2.2 Initialized Storage Allocation

When you specify the *ds* directive, the assembler allocates and initializes storage locations. Within the directive, you specify the number of locations to allocate and their initial values. The format of the directive is as follows:

ds.*x* *operand* [,*operand* [...]]

where *x* is one of the data-type specifiers *b*, *h*, *w*, *l*, *s*, or *d* (see Table 4-4), and *operand* is a numeric constant, symbolic name, character constant, or string constant. Each operand is used, in turn, to initialize items of storage whose size is specified by *x*.

The operands following the *ds* directive define the initial values assigned to the storage locations. The directives that specify either single-precision or double-precision floating-point numbers should specify operands that are floating-point numbers. The *halfword*, *word*, and *longword* directives require integers as operands. The *ds.b* directive accepts either strings or integers as operands, which lets you store messages. The following example illustrates the correct use of the *w*, *s*, and *b* operands with the *ds* directive.

```
ABC:    ds.w    0x1000,200,5        ;define and initialize
                                    ;word storage
xyz:    ds.s    458.3,100.0         ;define and initialize
                                    ;two single-precision
                                    ;storage locations
msg:    ds.b    ''Message'',012,015 ;define a message with
                                    ;a following CR and LF
```

The *ds* directive does not override the characteristics of the program segment within which it is defined. Therefore, using this directive in the *.text* segment allocates and initializes the number of locations specified, while using the *ds* directive within the *.bss* or *.tbss* segment generates an error message. The following examples illustrate the use of this directive.

```
var1:   ds.w    5000    ;allocate a word containing the number 5000
var2:   ds.w    0,0,0   ;allocate 3 words, each 0
```

The initialization arguments may include a repeat factor that initializes blocks of storage with a single value. You can specify the repeat factor as either a numerical value or the value of an absolute local symbolic name. In the latter case, precede the data operand with the symbolic name specified as a repeat factor and enclose the data operand in parentheses The following examples illustrate these two uses of the repeat factor.

```
table:  ds.w 1000(0)      ;allocate 1000 words of zero

foo=    100
tiny:   ds.b foo(0x20)    ;allocate 100 bytes of 0x20
```

4.5.2.3 Strings

A string is a sequence of characters that is delimited by quotation marks. Strings cannot span lines in the source program. Use strings only as arguments to a storage-definition directive. The following list contains examples of valid strings:

```
"CONVEX Assembler"

"A message."

"Error number 20"
```

The assembler allows you to specify escape sequences within strings. These are similar to certain C escape sequences. Table 4-12 lists the valid escape sequences.

Table 4-12: Assembler Escape Sequences

Description	Assembler Notation
Newline	\n
Horizontal tab	\t
Backspace	\b
Carriage return	\r
Form feed	\f
Backslash	\\
Single quote	\'
Double quote	\"
Bit pattern	\ddd

The escape sequence *\ddd* consists of the backslash and one to three octal digits specifying the value of the desired character. The *\0* indicates the character NUL. When the character following a backslash is not one of those specified in Table 4-12, the backslash is ignored.

Assembler strings, unlike C strings, are not automatically terminated with a null byte. You can use null bytes as terminators for string constants by including the *\0* escape as the final character in the string.

4.5.3 Alignment Directives

Code and data in a module may have to be aligned to a specific byte-address multiple within the program address space. For example, all instructions must begin on even address boundaries and certain CONVEX UNIX data structures must begin on 4096-byte boundaries. Also, many memory references operate most efficiently when the referenced data are aligned on 32-bit or 64-bit address boundaries. These alignments, as described below, may be specified explicitly in the assembly-language program. Both the assembler and the loader coordinate to maintain the program alignments that you specify.

The *.align* directive specifies the byte multiple to which any subsequent output in the current program section is aligned. For example:

```
.align   2
```

adjusts the location counter of the current program segment for halfword alignment so that the low-order bit of the location counter is zero. An alignment of 2 bytes at the beginning of the code portions of a module remains throughout the remainder of the module, since all instructions are a multiple of 16 bits in length. Data segments within the program may need to be aligned so that program variables do not begin on inefficient address boundaries. In most cases, you achieve maximum efficiency when data are given an alignment equal to the size of the item in bytes. For example, 2-byte data should be aligned on 2-byte boundaries, and 4-byte data on 4-byte boundaries. However, 8-byte data need only be aligned on 4-byte boundaries. Each segment is initially aligned on an 8-byte boundary.

The assembler handles alignment specifications of 8 or less bytes by padding the output for the current segment with bytes of zeros. The loader automatically preserves alignments of 8 bytes or less, and manages alignments greater than 8 bytes. The assembler passes these

large alignments to the loader for processing. You may specify only one alignment of greater than 8 bytes for each segment of each module. This alignment must appear in the source file before any code or data to be generated in that segment. Once code or data has been generated in a segment, an alignment directive greater than 8 is meaningless.

The *.pad* directive moves the program counter to the next multiple of its argument (assuming an origin of zero). For instance:

```
.pad 1024
```

moves the program counter to the next multiple of 1024. If you pad to greater than a longword (8 bytes), you must use the *.align* directive in conjunction with the *.pad* to make certain the boundary is appropriate for that module. Using the *.align n* directive forces the loader to load that section on a multiple of *n*.

For further information on storage alignments, see the *CONVEX C Compiler User's Guide* and the *CONVEX Loader User's Guide* .

4.5.4 Floating-Point Directives

The *.fpmode* directive controls the translation of floating-point constants within your assembly code. The directive takes one operand, either *ieee* or *native*. Any other operands are invalid and do not change the floating-point mode currently in effect. Following is an example of how to use the *.fpmode* directive:

```
                    .
                    .
                    .
.fpmode ieee                        ;translate following floating-point
                                    ;constants to IEEE format
                    .
                    .
                    .
.data
ds.d 3.415                          ;translated to IEEE format
                    .
                    .
                    .
.fpmode native                      ;translate following floating-point
                                    ;constants to native format
                    .
                    .
                    .
ld.s #1.0,s0                        ;translated to native format
```

The initial floating-point mode of the assembler is determined by the floating-point-format option specified on the command line. The *-fi* option specifies IEEE mode, and the *-fn* option specifies native mode. If no *-f* option is specified, the site default is used.

The assembler, not the *.fpmode* directive, sets the execution mode of the object file produced from your source code. The assembler sets the floating-point mode of the object file based on the floating-point-format command-line option. If you do not specify a floating-point-format command-line option, the assembler uses the site default. The system manager determines the site default when running the *sysgen* program to generate the system. (See Chapter 3 of the *CONVEX System Manager's Guide* for details on system defaults.)

Single-mode programs (all IEEE or all native) never need a *.fpmode* directive; the mode specified by the floating-point-format option determines the translation mode of all floating-point constants in the file. You may use *.fpmode* directives in single-mode programs as long as the directive agrees with the mode specified on the command line. In single-mode programs, a fatal error occurs when the argument to *.fpmode* differs from that given with the floating-point-format option.

It is good practice to state explicitly the intended mode for the resulting object (and executable) files with the floating-point-format option (*-fi* or *-fn*) rather than to make assumptions about the machine on which the file is assembled. The default mode is machine-dependent.

4.5.5 External Symbolic Name Directives

Complete programs often reside in separately assembled modules that the loader later combines into a single program. Two external symbolic name directives (*.globl* and *.comm*) let the assembled modules communicate. These directives define symbolic names that are declared and used in separate modules.

The assembler and loader support external and local symbolic names. External symbolic names are global symbolic names visible to other modules, and are used to resolve external references in other modules. Local symbolic names are invisible to other modules; they are known only within the current assembly object file.

Source programs define the values for external symbolic names by declaring the symbolic name to be global in one object module. Other object modules simply refer to the symbolic name. Any symbolic name that the assembler cannot resolve within a source file is treated as an unresolved external symbolic name.

4.5.5.1 *.globl* Directive

The *.globl* directive is usually used for function or subroutine names. It notifies the assembler that the symbolic names following are declared to be visible in other assembly modules. The format of the *.globl* directive is as follows:

.globl *name1* [,*name2* [,...]]

Symbolic names specified in *.globl* directives are defined in the current module. The loader resolves external references to global symbolic names during the linking process. You may define 20 symbolic names per statement using a comma-separated list—more than that returns the error "Wrong number of operands." The next example illustrates the use of the *.globl* directive.

```
        .globl abc      ;declare the symbolic name ''abc''
                        ;as globally visible
        .text
abc:    sub.w #4,sp     ;routine entry point is global
```

In the preceding example, another assembly-language source file could refer to the symbolic name *abc* without any special declaration. The assembler treats any symbolic names in a source file that are referenced, but not defined, as unresolved external symbolic names.

4.5.5.2 *.comm* Directive

The *.comm* directive is usually used for data shared between program units. Use the *.comm* directive to define external symbolic names of storage areas. This directive reserves storage in the *.bss* or *.tbss* program section, for example, for FORTRAN COMMON areas. The loader resolves the common storage directives by allocating storage in the *.bss* or *.tbss* section. The size of the allocated area is the maximum size specified in any module that contains a definition for the given symbolic name. If more than one *.comm* directive appears in a file, the largest size defined is the size for that common block.

The *.comm* directives may appear anywhere. You must, however, precede a *.cdata* directive with a *.comm* directive. The *.comm* allocates the space filled by *.cdata*. A *.comm/.cdata* pair can appear in one file, and a *.comm* for the same common block can appear in a separate file (e.g., the common block is declared and initialized in one FORTRAN file and referenced in another). When this occurs, the loader determines that the common block does not belong in the *.bss* or *.tbss* section and is really an initialized common block.

The format of the *.comm* directive is as follows:

.comm *name,expression*

where the *expression* used in the directive defines the size of the common area in bytes and must be an absolute constant. *Name* refers to the symbolic name defined as the name of the common storage block. Symbolic names used in *.comm* directives may be redefined in other *.comm* directives. The size of the generated common block is the maximum size declared in any module.

4.5.6 Symbol-Table Directives

The symbol table contains information about all the symbolic names in a program. The assembler maintains the symbol table and writes it to the generated object file. The loader then uses the table to resolve external references and to add symbolic name relocation to instructions and data that refer to relocatable symbolic names.

Three symbol-table directives transfer source language information from compilers to the source-code debugger, *csd*. These directives are as follows:

.stabs *<string constant>,<abs expr>,<abs expr>,<abs expr>,<reloc expr>*

.stabn *<abs expr>,<abs expr>,<abs expr>,<reloc expr>*

.stabd *<abs expr>,<abs expr>,<abs expr>*

In each of the three directives, the first absolute expression (*abs expr*) specifies the type of symbol-table entry to be made. Use the absolute expressions, *abs* and *expr*, to fill in specific descriptive fields within the symbol-table entry. Only the *.stabs* directive enters a string (*string constant*) into the output object file. The *.stabs* and *.stabn* directives enter a relocatable value (*reloc expr*) into the symbol-table entry. The *.stabd* directive implicitly enters the relocatable value into the symbolic name equal to the current location counter of the *.stabd* directive.

Chapter 5
Operands

This chapter describes the component parts of an assembly-language operand. Specifically, this chapter defines keywords and discusses the following topics:

- assembler character set
- operators
- terms
- constants, including numeric constants, character constants, and symbolic names
- expressions, including type propagation in expressions
- symbolic-name assignment statements
- location counter
- addressing modes

Before continuing this chapter, familiarize yourself with the keywords listed in the next section.

5.1 Keywords

Terms—Constants and symbolic names that make an expression

Expressions—Combination of constants and symbolic names joined by operators

Relocatable expressions—expressions that reference addresses that cannot be resolved at assembly time

Absolute expressions—expressions that contain only terms that do not need to be relocated and can be resolved at assembly time

External expressions—expressions that contain at least one external symbolic name

Dot—The . character, which serves as a symbolic name for the value of the current assembler location counter

5.2 Assembler Character Set

The assembler uses a subset of the ASCII character set to represent named entities, numbers, and the operations performed on these language components. The assembler uses the characters shown in the following list:

- The letters A through Z and a through z
- The numerals 0 through 9
- Left and right parentheses ( )
- The operators + - * / & and |
- The operator @ to denote operand indirection
- Commas to separate operands
- Semicolons to identify comment fields
- Colons in label definitions
- The characters . _ and $ in names
- The tilde character ~ as the unary complement operator
- The character = in direct assignment statements
- The space and tab characters as white space
- The line feed, form feed and ! characters for terminators
- The character \ for escape sequences
- The character # for immediate operands

5.2.1 Operators

Operators are characters that join constants and symbolic names to create expressions. Table 5-1 lists the binary and unary operators.

Table 5-1: Binary and Unary Operators

Binary		Unary	
+	Addition	-	Unary minus
-	Subtraction	~	Logical negation
*	Multiplication		
/	Division		
&	Logical AND		
|	Logical OR		

The assembler evaluates expressions from left to right with no operator precedence. Thus, the expression "1+2*3" evaluates to 9, not 7. Unary operators have precedence over binary operators since they are considered part of a term; both terms of a binary operator must be resolved before you can apply the binary operator.

5.2.2 Terms

Terms are the constants and symbolic names that make up an expression. Following are the four types of terms:

- Numeric constants
- Character constants
- Symbolic names
- Numeric constants, character constants, and symbolic names preceded by a unary operator. For example, both sum and -sum are terms. Multiple unary operators are also allowed; e.g., -- A has the same value as A.

5.2.2.1 Numeric Constants

The assembler supports decimal, octal, and hexadecimal integer constants, and floating-point constants in C and FORTRAN notation. The default radix for integers is decimal. A decimal number is a sequence of digits that begins with a digit other than zero. A string of digits that begins with a zero is interpreted as an octal number. A hexadecimal constant begins with the sequence "0x" or "0X", and is followed by a sequence of decimal digits or the uppercase or lowercase letters A through F.

Floating-point constants can be represented either as a sequence of decimal digits followed by a period (followed by additional decimal digits) or in the usual scientific notation. For example:

```
1.25e2 = 125.0 = 0.125E3 = 125. = 125E0
```

A floating-point constant represents either a single-precision (32-bit) or a double-precision (64-bit) value, depending on the context in which it appears.

Numeric constants follow the same rules as those in the C compiler. An integer constant can have a trailing "LL" (long long) to indicate a 64-bit constant. A decimal integer constant that is larger than the largest signed 32-bit integer is treated as a 64-bit constant. An octal or hexadecimal constant that is larger than the largest unsigned 32-bit integer is treated as a 64-bit constant. Note that 32-bit integer constants may have an optional trailing "L" (for long) for compatibility with constants accepted by the C compiler. Specifying all 64 bits of long long constants guarantees no sign-extension problems when converting a 32-bit constant to a 64-bit constant.

5.2.2.2 Character Constants

In addition to numeric constants, the assembler also supports character constants. A character constant is denoted by enclosing the desired character within apostrophes. For example:

'a' represents the character constant for lowercase a and has a value of 0x61.

'A' represents the character constant for uppercase A and has a value of 0x41.

Although the assembler can form character constants for many nonprinting characters, using the direct assignment statement to define a symbolic name for the constant produces more readable programs.

The assembler allows you to specify nonprinting characters using escape sequences in character constants. Escape sequences are described in Table 4-12.

5.2.2.3 Symbolic Names

You can use decimal digits, letters, and the special characters $, . , and _ in symbolic names. (A digit cannot be used as the first character in the name.) All characters in the name are significant. Uppercase and lowercase letters are distinct; for example, the assembler considers the symbolic names "ONE" and "one" to be separate symbolic names.

A symbolic name is declared when the assembler first recognizes it as a symbolic name in the program. A symbolic name is defined when a value is associated with it. A symbolic name used as a label may not be redefined; all other symbolic names can receive a new value.

Symbolic names may be declared

- As the label of a statement
- In a symbolic-name assignment statement
- As an external symbolic name using the *.globl* directive
- As a common symbolic name using the *.comm* directive
- As a local symbolic name

Instruction mnemonics, assembler directive names, and register names are reserved symbolic names. You cannot redefine a reserved symbolic name.

5.2.3 Expressions

Expressions are a combination of constants and symbolic names, joined by operators. The assembler resolves expressions to a signed 64-bit value, then converts the result of the expression to a size appropriate for the destination. An expression can be relocatable, absolute, or external.

5.2.3.1 Relocatable Expressions

Relocatable expressions reference addresses that cannot be resolved at assembly time. Relocatable expressions contain two parts: a constant value and an offset determined by the loader. Relocatable expressions are necessary since the loader determines the addresses of certain symbolic names after assembly is complete. If the assembler encounters one of these symbolic names, the expression is relocatable and the symbolic name value is measured by the origin of the program segment in which it is found.

Symbolic names that cannot be resolved at assembly time include text, data, tdata, bss, and tbss symbolic names. Text, data, tdata, bss, and tbss symbolic names are located in the *.text*, *.data*, *.tdata*, *.bss* and *.tbss* segments of the program, respectively.

Since most of the symbols encountered in a typical module are relocatable, the majority of expressions found in a given module are relocatable. Valid relocatable expressions include the following:

- Expressions that consist of a relocatable term
- Expressions that consist of a relocatable term plus or minus a constant
- Expressions that consist of a relocatable term plus or minus an absolute term

The following examples illustrate both valid and invalid uses of relocatable expressions:

```
sum             relocatable
sum+5           relocatable
sum*2           not relocatable (error)
2-sum           not relocatable (error)
```

5.2.3.2 Absolute Expressions

Absolute expressions contain only terms that do not need to be relocated and can be resolved at assembly time. Since addresses are not the principal terms of these expressions, subsequent manipulation of these terms by the loader is counterproductive. An absolute symbolic name is one whose value is an absolute expression. To prevent loader manipulation, define values for absolute symbolic names using assignment statements.

5.2.3.3 External Expressions

External expressions contain at least one external symbolic name. An external symbolic name is visible from other modules, and its value is accessible to the loader. External symbolic names are not always defined in another module but are visible globally and can be used to resolve external references in other modules.

There are two types of external symbolic names. The first type is called "defined external," and is declared externally using the *.globl* assembler directive. These names are also defined in the current assembly. The loader has access to the value and type of these symbolic names in order to resolve references from other object modules.

The second type of external symbolic name is called "undefined external." These names are not defined in the current assembly. Any symbolic name used in the current source file, but not defined, is assumed to be defined in another module. Similarly, symbolic names specified in a *.globl* directive, but not declared, are assumed to be defined in another module. If you use such a symbolic name, the loader resolves the undefined external references to definitions in other modules.

You define a symbolic name when you give it a value either explicitly (using an assignment statement) or implicitly (using the symbolic name as a statement label).

5.2.4 Type Propagation in Expressions

When you combine operands by expression operators, the resulting type depends on the types of the operands and the operator. The combination rules include the following:

- If both operands are absolute, the result is absolute.
- Addition operations: if one operand is either relocatable or an undefined external, the other operand must be absolute and the result is relocatable.
- Subtraction operations: if the first operand is relocatable, the second operand may be absolute or relocatable. If the second operand is absolute, the result is relocatable. If the second operand is relocatable within the same segment as the first, the result is absolute. If the first operand is external undefined, the second must be absolute. All other combinations are invalid.

5.2.5 Symbolic Name Assignment Statements

A symbolic name assignment statement assigns the value of an expression to a symbolic name. The format of a symbolic name assignment statement is as follows:

symbolic name = *expression*

Following are examples of valid assignments:

```
count = 0x100

init = 0
```

Each statement assigns one value to one symbolic name. You may redefine symbolic names defined by assignment to take the value of the most recent assignment. You may not redefine labels or register symbolic names.

Absolute, relocatable, and external symbolic names adhere to the following rules:

- If the defining expression is absolute, the symbolic name is also absolute. Absolute symbolic names may be treated as constants in subsequent expressions.
- If the expression is relocatable, the symbolic name is also relocatable. Relocatable symbolic names are declared in the same program section as the relocatable symbol in the relocatable expression.
- If the expression contains an external symbolic name, the symbolic name defined by the assignment statement is also considered external. You may define external symbolic names by direct assignment.

5.2.6 Location Counter

The " . " character (called *dot*) is a symbolic name that contains the value of the current assembly location counter. The value of the location counter during assembly is the number of bytes generated in the current subsegment (0, 1, 2 or 3) of the current program segment (text, data, tdata, bss, or tbss). The value of the location counter is always relocatable.

The *dot* symbolic name follows the standard rules of use or assignment. Use reassignment to change the offset of the next instruction or datum. For example

```
xyz = .
```

assigns the symbolic name *xyz* the current value of the assembler location counter, whereas

```
. =. + 10
```

adds 10 to the assembler location counter.

This facility can be used to deposit instructions or data at fixed offsets within a routine, such as in array initialization where the addresses and values of the initialization data may appear in random order (as for DATA statements in FORTRAN). For example, the following two sequences of directives in Table 5-2 produce the same effect; each creates a table holding the values 1, 2, and 3.

Table 5-2: Location-Counter-Directive Sequences

Sequence 1	Sequence 2
`.data`	`.data`
`table:`	`table:`
`ds.w 1`	`.=.+8`
`ds.w 2`	`ds.w 3`
`ds.w 3`	`.=.-8`
	`ds.w 2`
	`.=.-8`
	`ds.w 1`

In Table 5-2, the current value of *dot* is the byte offset address where the next byte generated is deposited. This value changes as soon as any bytes are generated. Thus, under Sequence 2, the lines `.=.+8` and `.=.-8` do not refer to the same offset within the table.

Using this mechanism, there is no way to specify an absolute address in memory when the program executes. You can use this mechanism in conjunction with the *.align* directive to force alignments of data or instructions. Any value assigned to *dot* must not contain any relocatable symbolic name references. Any holes created in the text or data segment by skipping locations are set to zero if no other values are deposited at the locations.

5.3 Addressing Modes

The CONVEX assembly-language instruction set supports the following operand-addressing modes: register mode, immediate mode, and six modes for specifying operands in memory. Because the CONVEX architecture is based on three sets of high-speed registers, most of the instructions use the register mode, which is used for register-to-register operations.

5.3.1 Register Mode

The majority of instructions in the instruction set require one or more register-mode operands that specify which register is used as the operand. The actual machine instruction generated by the assembler depends on both the register set used and the register within that register set.

Table 5-3 lists the register names for the CONVEX supercomputer register set.

Table 5-3: Register Names

Registers	Function
a0, A0, sp, SP	Address registers
a1, A1	
a2, A2	
a3, A3	
a4, A4	
a5, A5	
a6, A6, ap, AP	
a7, A7, fp, FP	
s0, S0	Scalar registers
s1, S1	
s2, S2	
s3, S3	
s4, S4	
s5, S5	
s6, S6	
s7, S7	
v0, V0	Vector registers
v1, V1	
v2, V2	
v3, V3	
v4, V4	
v5, V5	
v6, V6	
v7, V7	
vm, VM	Vector merge register
vmu, VMU	Vector merge register, upper longword
vml, VML	Vector merge register, lower longword
vs, VS	Vector stride register
vl, VL	Vector length register
vls, VLS	Vector length and stride combination
vv, VV	Vector valid register
	Other registers
cir, CIR	Communication index register
cpuid, CPUID	CPU identification
icr, ICR	Interrupt control register
psw, PSW	Processor status word
pc, PC	Program counter
tcpu, TCPU	Target CPU register
tid, TID	Thread identification
toc, TOC	Time of century clock
ttr, TTR	Thread timer

General-register operands are specified by a letter and a number. The letter denotes the register set, whereas the number specifies the register number in the set. The general-register sets for CONVEX supercomputers are the address registers (A), the scalar registers (S), and the vector registers (V). You can specify the register-set letter in either uppercase or lowercase. Each set contains eight registers denoted by the numbers 0–7. The following examples illustrate the correct use of register-mode operands.

a2	Address register 2
A2	Address register 2
S4	Scalar register 4
V0	Vector register 0
v2	Vector register 2

To improve program readability, you can reference the following three address registers with special functions by special names.

SP or sp	Stack pointer (A0)
AP or ap	Argument pointer (A6)
FP or fp	Frame pointer (A7)

The assembler uses reserved words to denote the machine registers. The assembler references each register by the reserved words only; hence, you may not use other symbolic names to denote machine registers.

Special-purpose registers in the machine are used for machine control. Use special reserved words to specify these registers. Table 5-4 lists these special registers. Consult the *CONVEX Architecture Reference* for more information on the function of these registers.

Table 5-4: Machine-Control Registers

Special Registers	Descriptions
CIR or cir	Communication index register
CPUID or cpuid	CPU identification
ICR or icr	Interrupt control register
PSW or psw	Processor status word
PC or pc	Program counter
TCPU or tcpu	Target CPU register
TID or tid	Thread identification
TOC or toc	Time of century
TTR or ttr	Thread timer
VM or vm	Vector merge register
VMU or vmu	Vector merge register, upper longword
VML or vml	Vector merge register, lower longword
VS or vs	Vector stride register
VL or vl	Vector length register
VLS or vls	Vector length and stride combination
VV or vv	Vector valid register

5.3.2 Immediate Addressing Mode

Immediate operands provide a method for referencing data from the program's instruction stream. The assembler syntax used to specify an immediate operand is as follows:

#expression

where *expression* defines the value of the immediate operand in question. The value of the

expression may be either relocatable or absolute. The loader resolves valid relocatable expressions by adding a relocation offset to the value of the absolute portion of the expression. Table 5-5 shows both valid and invalid immediate operands.

Table 5-5: Immediate Operands

Valid Immediate Operands	
#3	The absolute constant 3.
#label-5	The relocatable expression with the value of the label plus the absolute constant -5.
#lab1-lab2	The absolute value that is the difference between the relocatable symbolic names. This value does not result in a relocatable expression because the relocation constants cancel each other in the subtraction.
Invalid Immediate Operands	
#lab1+lab2	Sum of two relocatable expressions.
#lab1*2	This operand is not a relocatable expression because the relocation offset cannot be added to an absolute quantity.
#5-label	This operand is not a relocatable expression because the relocation offset cannot be added to an absolute quantity.

Floating-point values can appear as immediate operands in ".s" and ".d" versions of the instructions that support immediate operands. For example, following is a legal instruction that loads 3.14159 into register s0:

```
ld.s    #3.14159,s0
```

The type of datum placed in the immediate field should agree with the suffix of the instruction operator. For example,

```
ld.w    #3.5,s0
```

generates an error message because *ld.w* is an integer operation. The proper encoding of this instruction is

```
ld.s    #3.5,s0
```

Values of 32 bits or less are assembled directly into the immediate field of the instruction. For values greater than 32 bits, the assembler encodes either the upper or lower 32 bits into the immediate field of the instruction, depending on the width specifier of the opcode.

The pseudo-instructions *ld.du*, *ld.dl*, *ld.lu*, and *ld.ll* simplify the loading of 64-bit immediate operands. For example, the code sequence

```
ld.lu   #0x123456789abcdef, s0
ld.ll   #0x123456789abcdef, s0
```

loads the upper and lower halves of the constant into s0. The *ld.lu* and *ld.ll* instructions discard the lower and upper halves, respectively, of their immediate operand. Similarly, the instructions

```
ld.du   #1.1,s1
ld.dl   #1.1,s1
```

load the double-precision floating-point value 1.1 into s1. These pseudo-instructions generate the CONVEX opcodes (*ld.d*, *ld.u*, and *ld.w*) needed to load the immediate value. The assembler does not check to ensure that these instructions occur in pairs. The assembler converts floating-point constants into either 32- or 64-bit representations, depending on the context.

5.3.3 Memory-Addressing Modes

This section describes the addressing modes used to specify operands in memory. These modes are typically used to load registers from memory or to transfer the contents of registers to memory addresses.

5.3.3.1 Absolute-Addressing Mode

An absolute address in the virtual address space is referenced when a program specifies its location. You can specify this location with an expression that evaluates to the desired address. To achieve this evaluation, include an absolute number and a relocatable portion in the expression. The loader resolves the relocatable portion by adding a relocation constant to the relocatable portion, and then adding in the absolute number. Following are examples of instructions that use absolute addressing modes.

```
tas    1000         ;Test and Set the byte
                    ;at location 1000 in the current segment.

ld.w   data+6,a4    ;a4 ⇐ c(data+6)
                    ;load the value located
                    ;at the sixth through ninth byte
                    ;after the relocatable symbolic name
                    ;''data'' into register a4.
```

5.3.3.2 Register-Deferred Mode

When you use register-deferred mode, the address registers can contain the address of an operand to be used by the instruction. When you use this mode, the assembler expects the specified register to contain a pointer to the operand rather than the operand itself. To specify this addressing mode, enclose in parentheses the address register that contains the pointer. The following examples illustrate the use of this mode.

```
ld.w (a3),a4   ;a4⇐c(a3)
               ;load the contents of the 4 bytes
               ;pointed to by address register
               ;a3 into address register a4.

ld.b (a3),s4   ;s4⇐c(a3)
               ;load the byte pointed to by
               ;address register a3 into
               ;scalar register s4.
```

5.3.3.3 Indexed Mode

The indexed mode adds an offset or base to the contents of the specified address register, and the result is used as a pointer to the operand. To form this addressing mode, precede a deferred-register specifier with an expression, as in the following examples.

```
ld.w   100(a3),a4    ;a4⇐c(a3+100)
                     ;The contents of address
                     ;register a3 and the value
                     ;100 are added to form a
                     ;pointer to the 4-byte source
                     ;operand.

st.w   s5,blue(a4)   ;c(blue+a4)⇐s5
                     ;The contents of scalar register
                     ;s5 are stored in the memory
                     ;location that is pointed to by
                     ;the sum of the value of absolute
                     ;symbolic name ''blue'' and the contents of
                     ;address register a4.
```

5.3.3.4 Indirect-Absolute Mode

This mode enables you to use a single level of indirection when specifying an absolute address. An expression specifies the absolute address that contains the operand pointer. To specify this mode, include the @ character before an expression. The following examples illustrate the use of this addressing mode.

```
ld.w   @1000,a4      ;a4⇐c(c(1000))
                     ;The 4-byte memory location 1000
                     ;contains the address of a 4-byte
                     ;operand that is to be loaded into
                     ;address register a4.

st.b   s4,@dpointer  ;c(c(dpointer))⇐s4
                     ;Store the byte contained in
                     ;scalar register s4 into the memory
                     ;location pointed to by the 4 bytes
                     ;at memory location specified by
                     ;''dpointer''.
```

5.3.3.5 Indirect-Deferred Mode

This mode adds another level of indirection to the register-deferred mode. To specify this mode, prefix the @ character to a deferred-register operand enclosed within parentheses. The deferred-register operand contains the address of a pointer to the desired operand. The following examples illustrate the use of this addressing mode.

```
ld.w   @(a4),a5   ;a5⇐c(c(a4))
                  ;Address register a4 contains the
                  ;address of 4 bytes that specify
                  ;the address of the desired word
                  ;operand.

st.b   s5,@(a5)   ;c(c(a5))⇐s5
                  ;Register a5 contains the address of
                  ;4 bytes that specify the
                  ;address of the location where the
                  ;byte contained in s5 is stored.
```

5.3.3.6 Indirect-Indexed Mode

To denote indirect-indexed mode, precede an indexed operand with the @ character. When you specify this mode, the assembler adds the value of the register and the value of the absolute index expression to form the address of a pointer to the desired operand. Use of this mode is illustrated by the following examples.

```
ld.w   @sum(a3),a4   ;a4=c(c(sum+a3))
                     ;The value of the absolute symbolic name
                     ;''sum'' and the contents of address
                     ;register a3 are added together to
                     ;form the address of a pointer to the
                     ;4-byte operand to be loaded into
                     ;address register a4.

st.b   s5,@8(a4)     ;c(c(8+a4))⇐s5
                     ;The byte contained in register
                     ;s5 is stored in the memory
                     ;location pointed to by the pointer
                     ;that is stored at the address that
                     ;the sum of register a4 and 8.
```

Chapter 6
Conventions

This chapter describes conventions you should be familiar with when using the assembler. Specifically, this chapter defines keywords and discusses the following topics:

- address registers, including the stack pointer, argument pointer, and frame pointer
- linkages
- runtime-stack layout
- general calling conventions
- C calling conventions
- FORTRAN calling conventions

Before continuing this chapter, familiarize yourself with the keywords listed in the next section.

6.1 Keywords

Processor status word (PSW)—A 32-bit register that contains coded bits indicating the current state of the processor

Program counter (PC)—A 32-bit register that contains the address of the next instruction to be executed

6.2 Special Address Registers

Some address registers are used for special functions by the hardware and the compilers to implement subprogram calls. The address registers A0, A6, and A7 are used as the stack-pointer (SP), argument-pointer (AP), and frame-pointer (FP) registers, respectively.

6.2.1 Stack Pointer

The *psh* and *pop* machine instructions manipulate data on the top of a runtime stack by incrementing and decrementing the SP. Whereas the other address registers can be used as index registers from load and store instructions, SP cannot. The hardware uses a zero in the index register field of a load or store instruction to indicate that no index register is to be used. This encoding prevents the use of the stack pointer (A0) as an index register. Compilers generate automatic (local) storage on top of the runtime stack by directly adjusting SP.

6.2.2 Argument Pointer

By convention, the code generated by the CONVEX compilers uses A6 as an argument-pointer register. Before calling a subprogram (subroutine or function), the generated code loads into AP the base address of the argument list to be passed to the called routine. This argument list may be located either on the runtime stack or anywhere in the program address space. When possible, compilers build packets of subprogram arguments at compile time to increase runtime efficiency. The called routine references the arguments passed to it by using AP as an index register.

6.2.3 Frame Pointer

The call instructions (*call* and *calls*) and the return instructions (*rtn and rtnc*) use register A7 as a frame pointer in order to maintain linkage information for subprogram calls. Each *call* or *calls* instruction causes the contents of several registers to be saved on the top of the runtime stack (determined by the contents of SP), and the value of FP is updated to point at the saved context. The compiler-generated code uses the FP as an index register to refer to automatic storage. The program can find the previous routine context blocks by retrieving subsequent frame pointers out of the saved context block pointed to by the current frame pointer.

A function uses scalar register S0 to return its result value to the calling routine. The return (*rtn*) instruction does not alter the value of S0, so the return value is available to the calling routine.

6.2.4 Compiler-Generated Code

Other than the SP, AP, and FP registers, the compiler-generated code makes no other assumptions about registers being preserved across procedure calls. The CONVEX C compiler (*cc*) uses four scalar registers (S4, S5, S6, and S7) to maintain often-used values (register variables). During the process of calling a subprogram, the code generated by *cc* explicitly stores any of these four registers currently in use and restores them on return. The CONVEX vectorizing C compiler (*vc*) ignores register variables. The CONVEX FORTRAN compiler (*fc*) explicitly saves whatever intermediate information exists in registers before calling a subprogram.

6.3 Linkages

The interprocedural call instructions *call*, *calls*, and *callq* store information on and retrieve information from the runtime stack. Each of the three instructions push current state information on the runtime stack and branch to a destination address contained within the instruction. For the return, the instructions keep track of the called routine's virtual address.

6.3.1 *callq*

The *callq* instruction, or fast call, pushes the current value of the program-counter register on the runtime stack. The value in the stack-pointer register is decreased by four to account for the pushed value. None of the other registers is saved by the *callq* instruction.

To return, the routine executes a *rtnq* instruction, which removes the program-counter (PC) value (the top word of the runtime stack), and places the return address into the PC where execution continues after the subroutine call. The CONVEX C and FORTRAN compilers never use the *callq* instruction for calling user-written functions; *callq* is used only for calling short library routines that perform housekeeping functions.

6.3.2 *calls*

The *calls* instruction is the most frequently used instruction for interprocedural calling. Like the *callq* instruction, the *calls* instruction pushes the value of the return address onto the runtime stack. It also pushes the value of the processor status word, the value of the frame-pointer register (A7), and the value of the argument-pointer register (A6).

After these four words have been pushed on the stack, the frame-pointer register (A7) is updated to contain the current value of the stack pointer. The called routine can then access the last stack frame on the runtime stack using the frame-pointer register. To return control to its caller, the called routine executes a *rtn* instruction.

6.3.3 *call*

Like the *calls* instruction, the *call* instruction saves the current values of the program counter, processor status word, frame pointer, and argument pointer on the runtime stack. The *call* instruction also saves the current values of the other A and S registers, except A0 and S0.

The CONVEX C and FORTRAN compilers currently do not generate any *call* instructions, but use the faster *calls* instruction instead.

6.4 General Calling Conventions

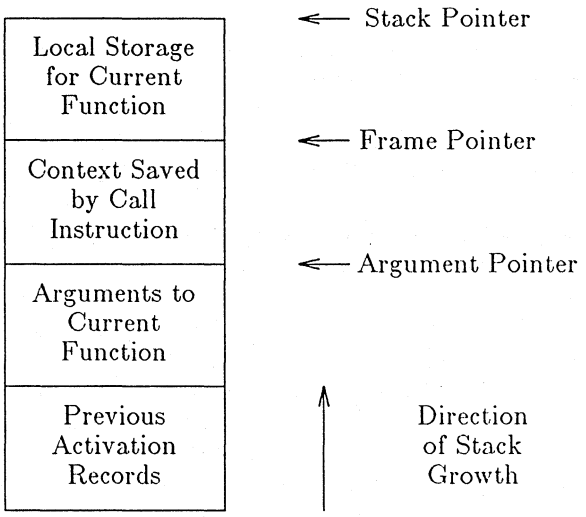
This section describes the general layout of the activation records used with the CONVEX C functions and FORTRAN functions and subroutines. When function or subroutine calls are executed, the current state of several hardware registers used by the calling routine must be preserved. The contents of these registers are pushed onto the runtime stack as a part of an activation record. The called function may then alter the machine registers as it runs. The hardware, as part of the return to the original calling routine, restores the old values of the saved registers.

The CONVEX compilers do not preserve the value of every register across a function call. Only those registers required to maintain the state of the runtime stack are preserved. Called routines that can restore the frame-pointer register to its original state are allowed to modify any register passed to them.

6.4.1 Function or Subroutine Stack Layout

Figure 6-1 illustrates the top of the runtime stack. Although the stack grows downward in the address space, this diagram shows the stack as growing upward on the page. The stack-pointer register contains the address of the topmost location on the runtime stack. The frame-pointer register contains the address of the last frame pushed on the runtime stack by a *call* or *calls* instruction. The argument-pointer register contains the address of the arguments passed to the current routine.

Figure 6–1: Top of the Runtime Stack



6.4.2 Function or Subroutine Calling Sequence

When a function is called, the compiler-generated code follows one of the following two sequences:

Sequence 1

1. Pushes the values of the arguments to the function onto the runtime stack in reverse order.
2. Updates the argument-pointer register. The updated register should point to the first argument in the argument list. (The first argument in the list is the last one pushed.)
3. Pushes an additional word. This word should contain the number of arguments passed.
4. Calls the routine with a *calls* instruction.

Executing the *calls* instruction places a stack frame on the runtime stack. The stack frame contains the current values of the program counter (return address), the current value of the

processor status word, the old value of the frame pointer, and the current value of the argument pointer.

Sequence 2

For FORTRAN, the arguments are precompiled when possible, so the calling sequence is reduced to the following:

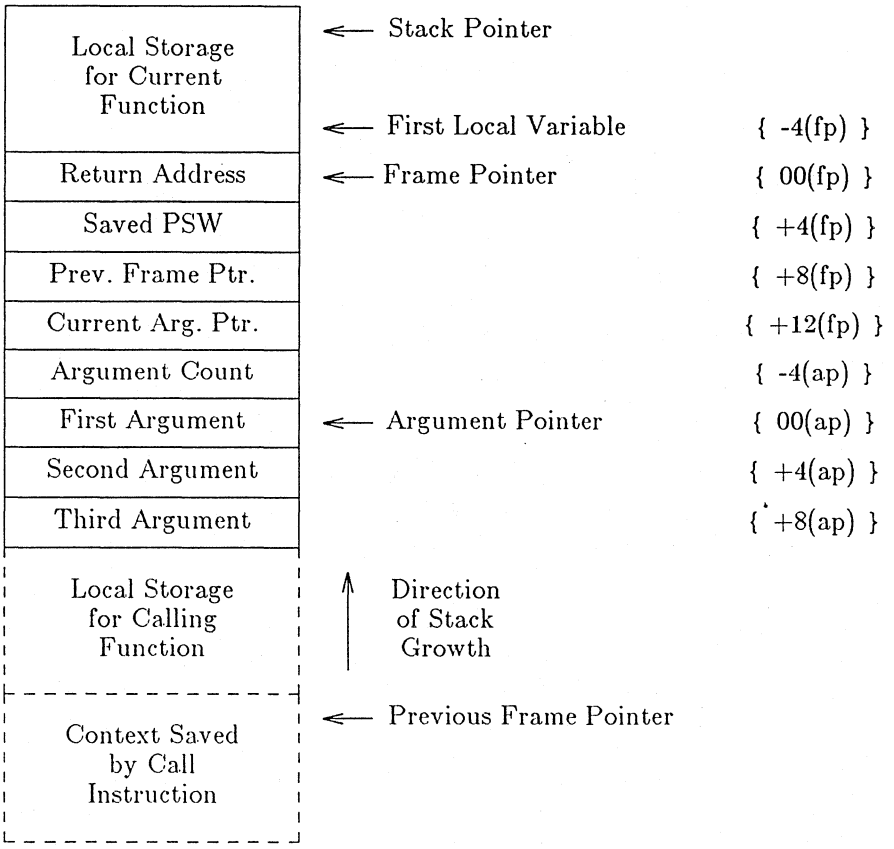
1. Loads the address of the argument packet into the argument pointer
2. Calls the subroutine (using the *calls* instruction)

The conventions that apply to function calls are listed below.

- The called routine can allocate storage for local variables on top of the runtime stack. No stack references in CONVEX C code are made relative to the top of the runtime stack. Storage allocated on the stack by called routines is automatically deallocated when the function returns.
- The called routine need not preserve the contents of any register except the frame pointer. The called routine uses the current value of the argument-pointer register to access the arguments passed to the routine by its parent.
- The frame-pointer register points to the context block pushed by the caller when calling the child routine. The called child routine references the local storage it has allocated on the runtime stack by referencing negative offsets from the frame pointer.
- The called routine references arguments passed to it by its parent by referencing positive offsets from the argument-pointer register: 0(ap) corresponds to the first argument, 4(ap) corresponds to the second argument, and so on. The word with an address of "-4" relative to the argument pointer contains a count of the number of arguments passed to the routine.

Figure 6-2 illustrates the layout of the stack as seen by a routine after it has been called, and after it has allocated some storage for local variables on the top of the runtime stack. The stack is shown as a series of 32-bit words.

Figure 6–2: Stack Layout



Called routines return to their parents by

1. Placing a return value in register S0
2. Executing the *rtn* instruction

When the computer executes the *rtn* instruction, automatic storage allocated by the called routine is automatically deallocated. This instruction also restores the PSW register and the frame pointer to their previous states, and then returns control to the location immediately following the *calls* instruction that called the routine. After control returns to the parent, the stack-pointer register points to the location that contains the pushed argument count. The parent routine adds a fixed offset to the value in the stack pointer to remove the argument count and any pushed arguments. The value added is the total number of bytes pushed prior to the call. Finally, before the parent can access any of its own arguments, it must reload its own argument-pointer register from the current frame on the stack. This value is offset +12 bytes from the recently restored frame pointer.

6.5 C Calling Conventions

This section includes basic information on the C calling conventions. For more information, see the *CONVEX C Compiler User's Guide* or the *CONVEX Vector C Compiler User's Guide.*

6.5.1 Code Generated for Function Calls

The following is a section of sample CONVEX assembly-language code for a function call. All C functions are called in this manner. Only a few code-support routines are called using other mechanisms, which the compiler generates on a case-by-case basis. Figure 6-3 illustrates the call to a C routine with child (a, b, c).

Figure 6–3: Calling a C Subroutine

```
_parent

        psh.w   c3              ; value of rightmost argument
        psh.w   b2              ; value of second argument
        psh.w   a1              ; value of first argument
        mov     sp,ap           ; arg pointer points at first arg
        pshea   3               ; push count of 3 of args passed
        calls   _child          ; use ''calls'' to call function
        add.w   16,sp           ; remove bytes pushed for args
        ld.w    12(fp),ap       ; reload our argument pointer
                                ;
                                ; the code shown below is used in
                                ; the called child function:
                                ;
        .globl  _child          ; called function
_child:
        sub.w   20,sp           ; allocate bytes for local variables
        ld.w    0(ap),s0        ; load the value of the first arg
                                ; assuming 32-bit size
        ld.w    -4(ap),s1       ; load count of the args passed
        st.w    s0,-4(fp)       ; first local int is at -4(fp)
        sub.w   s0,s0           ; value returned in s0
        rtn                     ; return to caller
```

6.5.2 Function Names

Function names and global variables produced by the compiler in object code are unrestricted in length. An underscore character is prefixed to each global variable.

6.5.3 Function Arguments and Return Values

C arguments are passed using the "call-by-value" method, in which the values of arguments, rather than their addresses, are passed. C programs may pass addresses as arguments to functions by using pointer variables. Structures in C are also passed by value. All of the elements of the structure are pushed onto the runtime stack prior to the call. The calling process accelerates when structure pointers, rather than the structures themselves, are passed. Results returned from functions are returned in scalar register S0. Functions that return structures as results place the function result in a static area in the data segment and return a pointer to that area in S0.

6.6 FORTRAN Calling Conventions

This section includes some basic information on FORTRAN calling conventions. For more detailed information, see the *CONVEX FORTRAN User's Guide.*

6.6.1 Code Generated for Function Calls

Figure 6-4 is an example of FORTRAN, assembler, and C code to call a FORTRAN subroutine.

Figure 6–4: Calling a FORTRAN Subroutine

```
call sub        (a,b,c)         ;FORTRAN call with three real arguments

                                ; equivalent assembler calling sequence
pshea   c                       ; push the third argument's address
pshea   b                       ; push the second argument's address
pshea   a                       ; push the first argument's address
mov     sp,ap                   ; set up the ap
pshea   3                       ; push number of words in argument list
calls   _sub_                   ; call the subroutine
ld.w    12(fp)ap                ; restore ap
add.w   #16,sp                  ; restore sp

sub_ (&a,&b,&c);                /* equivalent C call */
```

Note: The arguments are pushed in reverse because the stack grows toward low memory addresses. Thus, the last argument pushed ends up at the top of the list. This way, trailing arguments can be omitted while function arguments are still uniformly referenced.

6.6.2 Subprogram Names

On UNIX systems, the name of a common block or a FORTRAN subprogram has an underscore appended to it by the compiler to distinguish it from a C procedure or external variable with the same user-assigned name. FORTRAN library procedure names have embedded underscores to avoid clashes with user-assigned subroutine names.

6.6.3 Function Arguments and Return Values

A function of type integer, logical, real, or double precision returns the corresponding type in scalar register S0. A complex or double-complex function is mapped to a FORTRAN subroutine with an additional initial argument pointing to where the return value is to be stored. Table 6-1 illustrates a complex function and its FORTRAN subroutine.

Table 6-1: Complex Function and FORTRAN Subroutine

Complex FORTRAN Function	Subroutine Equivalent
`complex function f(...)`	`subroutine f(temp,...)`

A character-valued function is equivalent to a FORTRAN subroutine with two extra initial arguments: a data address and a length. Table 6-2 illustrates a character-valued function, its C equivalent, and how to invoke its C equivalent.

Table 6-2: Character-Valued Function and Subroutine Equivalent

Character-Valued FORTRAN Function	Subroutine Equivalent
`character*15 function g(...)`	`subroutine g(temp,call by value(15),...)`

6.6.4 Subroutine Arguments and Return Values

Subroutines are invoked as if they are integer-valued functions whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the function, but are used to do an indexed branch in the calling procedure. If the subroutine has no entry points with alternate return arguments, the returned value is undefined. The statement

```
call nret(*1, *2, *3)
```

is treated exactly as if it were the computed *goto*

```
goto (1, 2, 3), nret( )
```

All FORTRAN arguments are passed by address. Also, for every argument that is of type character, an argument giving the length of the value is passed. (The string lengths are *word* quantities passed by value.) The order of arguments is as follows:

1. Extra arguments for complex and character functions
2. Address for each datum or function
3. A *word* for each character or procedure argument

Chapter 7
Using the Assembler

This chapter describes general procedures for using the assembler. Specifically, this chapter discusses the following topics:

- invoking the assembler
- selecting floating-point format
- redirecting assembler object output
- suppressing warning messages
- generating a source listing
- error messages

7.1 Invoking the Assembler

To invoke the assembler directly from the shell, use the following command sequence:

as *filename*

where *filename* is the name of a previously created assembly-language program that ends in *.s*. You do not have to specify the *.s* extension in the argument because the assembler assumes that the filename ends in a lowercase *.s* (see *abc* in the screen example below). Specify only one argument per *as* command.

The next screen example illustrates the use of the *as* command to produce object files from three assembly-language programs named *abc.s*, *abd.s*, and *abl.asm.s*, respectively. User input is in bold type, and the *ls* command lists the object files created by the assembler and the original source files.

```
%as abc
%as abd.s
%as abl.asm.s
%ls
abc.o   abd.o   abl.asm.o
abc.s   abd.s   abl.asm.s
```

The *as* command in this screen example calls the assembler to produce object files from the source files *abc.s*, *abd.s*, and *abl.asm.s*. The assembler writes the object file produced as output to a file with the same name as the source file, but with an extension of *.o* instead of *.s*. In this example, the assembler produces object files named *abc.o*, *abd.o*, and *abl.asm.o*.

7.1.1 Selecting Floating-Point Format

You can specify the format to which floating-point constants are translated using the floating-point format options on the *as* command line as follows:

as -f*format filename*

where the formats are

-fn Specifies that floating-point constants are translated to native format. If you specify this option and then try to use a *.fpmode ieee* directive in your code, you receive a message similar to the following:

```
"filename", line 51: illegal '.fpmode' directive for native mode program.
```

-fi Specifies that floating-point constants are translated to IEEE format. If you specify this option and then try to use a *.fpmode native* directive in your code, you receive a message similar to the following:

```
"filename", line 51: illegal '.fpmode' directive for ieee mode program.
```

CAUTION

You must have IEEE hardware for the assembler to create IEEE object files. If you run the assembler and specify IEEE format when your site does not have the appropriate hardware to support it, you receive a message stating that IEEE hardware is unavailable, and the assembler exits.

If you do not specify a floating-point format option, the site default mode is used.

7.1.2 Redirecting Assembler Object Output

To direct object output to other destinations, use the *-o* option. For example, the command sequence

```
as -o xyz.o abc
```

creates an object file named *xyz.o* from the source file *abc.s*.

7.1.3 Suppressing Warning Messages

To suppress warning messages issued by the assembler, use the *-w* option. Warning messages include the following:

- `Text segment .align > 2 and <= 8 not allowed, changed to .align 2.`
- `A0, A6 and A7 are not general-purpose registers.`
- `Symbol too long`
- `After code/data generated in segment, .align>8 is meaningless (and ignored)`

7.1.4 Generating a Source Listing

To generate a source listing on the standard output stream, use the *-l* option. This listing includes the following four fields of information (illustrated in the screen example below):

First field	The source-line number within the input file.
Second field	The value of the location counter (*dot*) within the current program segment
Third field	The assembled instruction as it appears in memory
Fourth field	The current input source statement

The following screen example illustrates the use of the *-l* option to generate a source listing with these four fields of information. User input is in bold type. Note that the fourth field (the input source statement) consists of two columns of information.

```
%as -l abl.asm.s
000001  00000000                        .text
000002  00000000  114889abcdef          ld.l    #0x123456789abcdef, s0
000003  00000006  300103e8              ld.b    1000, s1
000004  0000000a  32010000              ld.w    0, s1
```

To paginate the program listing for printing, use the *pr* (print file) command. To link the assembler output with the loader program, use *ld* (link editor). See the *pr*(1) and *ld*(1) man pages for more information.

7.2 Error Messages

When the assembler detects errors, it sends error messages to the standard error stream *stderr*. This error stream is displayed on your terminal unless you redirect it. Each error message includes the filename, line number, and a description of the error. Following is a typical error message:

```
"file.s", line 13: Invalid op-code
```

When the assembler detects an error, it ignores the remainder of the statement and begins processing on the next line.

Chapter 8
Sample Programs

This chapter illustrates and explains sample assembly-language code. Specifically, this chapter defines keywords and describes examples of code to do the following:

- copy a block of storage from one location in memory to another
- execute a vector routine
- execute a parallel program

Before continuing in this chapter, familiarize yourself with the keywords listed in the next section.

8.1 Keywords

Program segment—A block of code identified by a program-segment directive

Process—A collection of one or more instruction streams executing within a single logical address space

Multiprocessing—The creation and scheduling of processes on any subset of CPUs in a system configuration

Thread—Any single instruction stream executing within a process

8.2 Code to Copy Block of Memory

Figure 8-1 is a sample CONVEX assembly-language program to copy a block of memory from one location to another.

Figure 8–1: Copy Block of Memory

```
1       ;
2       ; _Bcopy -- block memory copy
3       ;
4       ; This routine copies a block of storage from one location in memory
5       ; to another.
6       ;
7       ; The calling sequence is:
8       ;
9       ;       psh.w   byte-count-value
10      ;       psh.w   destination-address
11      ;       psh.w   source-address
12      ;       mov     sp,ap
13      ;       pshea   #3
14      ;       calls   _Bcopy
15      ;       add.w   #16,sp
16      ;
17      ; No registers are preserved
18
19              .text
20              .globl  _Bcopy
21
22      _Bcopy:
23              ld.w    4(ap),a1        ; a1 <== dest
24              ld.w    0(ap),a2        ; a2 <== source
25              ld.w    8(ap),a3        ; a3 <== count
26
27      lcopy:  leu.w   #8,a3           ; copy any longwords?
28              jbra.f  wtest           ; no--try words
29              ld.l    (a2),s0         ; yes--load a longword
30              st.l    s0,(a1)         ; store a longword
31              add.w   #8,a1           ; bump destination pointer
32              add.w   #8,a2           ; bump source pointer
33              sub.w   #8,a3           ; decrement count
34              jbr     lcopy           ; try again
35
36      wtest:
37              leu.w   #4,a3           ; can copy one word?
38              jbra.f  htest           ; no--try halfword
39              ld.w    (a2),s0         ; yes--load a word
40              st.w    s0,(a1)         ; store a word
41              add.w   #4,a1           ; bump destination pointer
42              add.w   #4,a2           ; bump source pointer
43              sub.w   #4,a3           ; decrement count
44
45      htest:
46              leu.w   #2,a3           ; can copy one halfword?
47              jbra.f  btest           ; no--try bytes
48              ld.h    (a2),s0         ; yes--load a halfword
49              st.h    s0,(a1)         ; store a halfword
50              add.w   #2,a1           ; bump destination pointer
51              add.w   #2,a2           ; bump source pointer
52              sub.w   #2,a3           ; decrement count
53
54      btest:
```

Characters following a semicolon on a line are comments, and are ignored by the assembler. Lines 1–17 are comments that describe the function of the routine.

Line 19: the `.text` directive specifies that any instructions or data that follow are to be treated as program instructions. The assembler places program instructions in write-protected memory when the program is executed.

Line 20 the `.globl` directive specifies that the symbolic name `_Bcopy` is visible to other routines in the program. Use `.globl` to create symbolic names that are visible to all the modules in a program. By convention, the C compiler creates global symbolic names with a leading underscore character.

Line 22 the `_Bcopy:` label supplies the location used as the relocatable value for the global symbolic name specified in the `.globl` directive.

Lines 23–25 load the values of the three arguments into address (A) registers.

Lines 27–35 contain a loop of 8-byte load instructions and 8-byte store instructions that move longwords from the source address to the destination address. The counter in register `a3` is decremented by 8 during each pass through the loop (see line 33), and the pointers are incremented by 8 (see lines 31–32). The `leu` instruction in line 27 is an unsigned, less-than-or-equal-to comparison. The `jbra.f` instruction in line 28 is a conditional branch instruction that exits the loop.

Lines 36–43 along with lines 45–52, and 54–58 (on the next page) copy a 4-byte, 2-byte, and 1-byte parcel, respectively, to decrement the byte count to zero.

Line 55 continues on the next page.

Figure 8-1: Copy Block of Memory, continued

```
55              leu.w   #1,a3           ; one last byte ?
56              jbra.f  done            ; no--finish up
57              ld.b    (a2),s0         ; yes--load a byte
58              st.b    s0,(a1)         ; store a byte
59
60      done:
61              rtn                     ; return
62
63              .comm   src, 24         ; allocate 24 byte space called "src"
64              .cdata  src             ; switch sections to "src"
65              ds.w    1               ; declare a word - assign the value 1
66              ds.w    2               ; assign the value 2 to another word
67              ds.w    3               ; assign a word the value 3
68
69              .comm   dest, 24        ; allocate 24 bytes to "dest"
70              .cdata  dest            ; switch sections to "dest"
71              ds.w    6(0)            ; assign six words the value zero
72
73      .       .text                   ; switch to the text section
74              .globl  _main           ; make "_main" a global symbol
75
76      _main:  pshea   #24             ; push the constant 24 - arg 3
77              pshea   dest            ; push the address of dest - arg 2
78              pshea   src             ; push the address of src - arg 1
79              mov     sp, ap          ; set the argument pointer
80              pshea   #3              ; push the count of arguments
81              calls   _Bcopy          ; copy 24 bytes from src to dest
82              add.w   #16, sp         ; remove the arguments from the stack
83              ld.w    12(fp), ap      ; reset the argument pointer
84              rtn                     ; return
85
86              .cdata  src             ; switch to the section "src"
87              .=.+12                  ; move over the initialized space
88              ds.w    4               ; the fourth word of "src"
89              ds.w    5               ; initialize the fifth word to 5
90              ds.w    6               ; initialize the sixth word to 6
91
```

Lines 60–61 — the `rtn` instruction returns program control to the caller of `_Bcopy`.

Lines 63–90 — illustrate how `_Bcopy` is called and how `.comm` and `.cdata` directives are used.

Line 63–67 — along with lines 86–90 create and initialize a block of memory named `src` that is 24 bytes long. The initialized values are words containing the integers 1–6. In line 87, the `.=.+12` gets around the initialized storage space in the first `.cdata src` directive on line 64. Without moving the PC in this way, the data would be reinitialized.

Lines 69–71 — create another block of memory, named `dest`, that is also 24 bytes long and initialized to zero.

Line 73 — switches back to the `.text` segment. The code that follows this line is placed after the `rtn` instruction on line 61.

Line 74 — makes `_main` a global symbol corresponding to a C program named `main`.

Lines 76–81 — push a set of arguments onto the stack. They are pushed onto the stack in backward order; the C call is `_Bcopy (src, dest, 24)`. Line 79 sets the argument pointer to point to the argument packet. Line 80 pushes the argument count onto the stack for use by routines with variable-length argument lists. The argument count is unused by `_Bcopy`, but could be found at location `-4(ap)` if needed. Line 81 calls `_Bcopy`, pushing a short call frame onto the stack.

Line 82 — removes the arguments and argument count from the stack.

Line 83 — restores the argument pointer, `ap`, from the call frame created when `_main` was called. This is unused here, but could be used in a longer procedure.

Line 84 — returns program control to the caller of _main.

8.3 Vector Routine

Figure 8-2 is a sample assembly-language program to do an elementary vector operation. This code is essentially the SAXPY subprogram in the VECLIB library.

The FORTRAN call statement for this subprogram is as follows:

```
CALL SAXPY (n, a, x, incx, y, incy)
```

See the *CONVEX VECLIB User's Guide* for further details.

Figure 8–2: Vector Routine

```
1       ;---    subroutine saxpy (n, a, x,incx, y,incy)
2       ;
3       ;       saxpy computes a real*4 linked triad.
4       ;
5       ;---    Fortran call.
6       ;
7       ;       This is an optimized Convex Assembly-Language
8       ;       implementation of the following Fortran subprogram:
9       ;
10      ;               subroutine saxpy (n, a, x,incx, y,incy)
11      ;               real*4 x(*),y(*)
12      ;               if ( n .le. 0 ) return
13      ;               if ( a .eq. 0:0 ) return
14      ;               ix = 1
15      ;               iy = 1
16      ;               if ( incx .lt. 0 ) ix = 1 - (n-1) * incx
17      ;               if ( incy .lt. 0 ) iy = 1 - (n-1) * incy
18      ;               do 10 i = 1, n
19      ;               y(iy) = a * x(ix) + y(iy)
20      ;               ix= ix + incx
21      ;               iy= iy + incy
22      ;        10     continue
23      ;               return
24      ;               end
25
26              .globl  _saxpy_
27
28      _saxpy_:                        ; entry point
29
30              ld.w    @ 0(ap),s0      ; fetch n
31              ld.s    @ 4(ap),s1      ; fetch a
32              ld.w    @12(ap),a2      ; fetch incx
33              ld.w    @20(ap),a4      ; fetch incy
34              ld.w      8(ap),a1      ; fetch )x
35              ld.w     16(ap),a3      ; fetch )y
36
37              lt.w    #0,s0           ; check n
38              xor     s3,s3
39              brs.f   exit            ; exit immediately if n <= 0
40
41              eq.s    s3,s1           ; a : 0.0
42              mov     a2,s2
43              mov     a4,s4
44              shf     #2,a2           ; byte stride for x
45              shf     #2,a4           ; byte stride for y
46              brs.t   exit            ; exit immediately if a = 0.0
47
48              le.w    #0,s2           ; 0 : incx
49              mov.w   s0,a5           ; n
50              shf     #9,s2           ; segment stride for x
51              brs.t   sax1            ; if incx >= 0
52              mul.w   a2,a5           ; move to other end of x vector
53              add.w   a2,a1
54              sub.w   a5,a1
```

Characters following a semicolon on a line are comments, and are ignored by the assembler. Lines 1–25 are comments that describe the function of the routine.

Line 26 is a `.globl` assembler directive specifying that the symbolic name `_saxpy_` is visible to other routines in the program. By convention, the FORTRAN compiler appends underscore characters to both ends of subprogram names.

Line 28 defines the global symbolic name as the value of the current location counter. Since no program-section directive occurs before this line, the assembler begins using the `.text 0` section.

Lines 30–33 load the values of the non-array arguments into scalar and address registers. These quantities are maintained in the registers throughout the routine for efficiency. The `@` character is used to specify indirect addressing for FORTRAN calls by address.

Lines 34–35 load the addresses of the two array arguments into address registers. The unbalanced closing parentheses in the comment fields of these statements means "the address of."

Lines 37, 39, and 98
perform the `if` test in the comment at line 12.

Lines 38, 41, 46, and 98
perform the `if` test in the comment at line 13. The two instructions on lines 38 and 41 are used instead of the single instruction `eq.s #0.0,s1`, so as to treat negative zero in IEEE floating-point mode as zero.

Lines 42–45 are indexing initialization for the strip-mining loop. These instructions are placed between the floating-point comparison instruction on line 41 and the conditional branch instruction on line 46 so their execution can occur in parallel with the comparison.

Lines 48–54 perform the `if` test and dependent statement in the comment at line 17. This code, however, does not determine the value of variable `ix`. Instead, it determines the address of array element `x(ix)`.

Line 55 continues on the next page.

Figure 8-2: Vector Routine, continued

```
55
56      sax1:   mov.w   s0,VL           ; load first segment of x(i)
57              mov.w   a2,VS
58              ld.s    0(a1),v0
59              add.w   s2,a1           ; next )x
60              mul.s   v0,s1,v1        ; a * x(i)
61
62              le.w    #0,s4           ; 0 : incy
63              mov.w   s0,a5           ; n
64              shf     #9,s4           ; segment stride for y
65              brs.t   sax2            ; if incy >= 0
66              mul.w   a4,a5           ; move to other end of y vector
67              add.w   a4,a3
68              sub.w   a5,a3
69
70      sax2:   mov.w   a4,VS           ; load next segment of y(i)
71              ld.s    0(a3),v2
72              add.s   v1,v2,v3        ; new y(i)
73
74              sub.w   #128,s0         ; advance length
75              st.s    v3,0(a3)        ; store new y(i)
76              lt.w    #0,s0           ; check length
77              add.w   s4,a3           ; next )y
78              brs.f   exit            ; exit if done
79
80      ;---    linked triad loop.
81
82      sax3:   mov.w   s0,VL           ; load next segment of x(i)
83              mov.w   a2,VS
84              ld.s    0(a1),v0
85              add.w   s2,a1           ; next )x
86              mul.s   v0,s1,v1        ; a * x(i)
87
88              mov.w   a4,VS           ; load next segment of y(i)
89              ld.s    0(a3),v2
90              add.s   v1,v2,v3        ; new y(i)
91
92              sub.w   #128,s0         ; advance loop
93              st.s    v3,0(a3)        ; store new y(i)
94              lt.w    #0,s0           ; check loop
95              add.w   s4,a3           ; next )y
96              brs.t   sax3            ; loop if not done
97
98      exit:   rtn                     ; return to caller
99
100     ;---    end of _saxpy_
```

Lines 56–60 begin the vectorized processing of the loop in the comment lines 18–22. Lines 56–58 load the first strip of the x vector from memory. Line 60 performs the scalar-times-vector multiplication for the strip. The vector load will chain into the vector multiply since the result register of the load is an operand register for the multiply and since there are no functional unit or register reservations. If the value of argument n does not exceed 128, the entire x vector is loaded and multiplied; otherwise, the processing is restricted to the first 128 elements of x. Line 59 adds 128 × incx to the register containing the address of the x vector; the resulting register value is the address of x(ix) when loop index i = 129. If n exceeds 128, this value is used in the loop in lines 82–96; otherwise, it is not used.

Lines 62–68 perform the if test and dependent statement in the comment at line 18. This is similar to the processing of lines 48–54. Putting this code here instead of before line 56 allows its execution to occur in parallel with lines 56–60. Furthermore, since lines 56–60 do not depend on lines 62–68, arranging the code as shown allows the vector operations to begin sooner. Once vector operations begin, the remainder of the execution time is dominated by the vector processing, and almost all scalar processing is "free."

Lines 70–78 complete the vectorized processing for the first strip, begun in lines 56–60. Lines 70–71 load the first strip of the y vector. Line 71 chains into the vector add in line 72, and fills the first VL elements of vector register V3 with a*x(ix) + y(iy). Line 75 stores V3 back into the y vector in memory. Line 77 adds 128 × incy to the register containing the address of the y vector. Lines 74, 76, and 78 check to see if n exceeds 128. If not, line 78 branches to the return instruction in line 98.

Lines 82–96 form the strip-mine loop. Each iteration of this loop, except the last one, corresponds to 128 iterations of the DO-loop in lines 18–22. The last iteration processes the remaining 1 to 128 DO-loop iterations. The processing is identical to that in lines 56–60 and 70–78, except that the branch condition in line 96 is inverted from that in line 78 to repeat the strip-mine loop instead of to skip it.

Line 98 corresponds to the return statement in comment line 23.

8.4 Sample Parallel Program

Figure 8-3 is a sample C program that points to assembly-language code to execute row computations in parallel. Lines 1–90 make up the C program. The assembly-language code begins on line 93.

Figure 8–3: Parallel Program

```
1       #include <sys/time.h>
2       #include <sys/resource.h>
3       struct rusage rusage;
4       struct timeval tm1,tm2,tm3,tm4;
5       long long t1,t2,t3,t4;
6       int n, pcnt, d1,d2;
7       double f1,f2,z1,z2;
8
9       #define ARRSIZE 512
10      double  a[ARRSIZE][ARRSIZE],b[ARRSIZE][ARRSIZE],c[ARRSIZE][ARRSIZE];
11
12      long long gettoc();
13      int multrow();
14
15      main()
16      {   int i,j;
17
18      /* initialize a, b */
19      for (i = 0; i < ARRSIZE; i++)
20              for (j = 0; j < ARRSIZE; j++)
21              {   a[i][j] = i+j;
22              b[i][j] = i*j;
23              }
24
25      /* Print headers */
26          printf("   n                    wall clock    cpu time");
27          printf("      mflops   seq/par ratio 0);
28          for (n = 16; n <= ARRSIZE; n *= 2)  /* let n = 16, 32, ... ARRSIZE */
29          {
30          /* execute row computations in parallel */
31              gettime(&t1,&tm1);
32              pcnt = exfn(multrow,n);
33              gettime(&t2,&tm2);
34
35          /* execute row computations sequentially */
36              gettime(&t3,&tm3);
37              for(i = 1; i <= n; i++) multrow(i,n);
38              gettime(&t4,&tm4);
39
40          /* compute wall clock and cpu delta times */
41          /* compute mflop rates, print results for current n */
42              d1 = deltatime(&tm1,&tm2);  d2 = deltatime(&tm3,&tm4);
43              z1 = (double)(t2-t1);  z2 = (double)(t4-t3);
44              f1 = n*n*n*2/z1;       f2 = n*n*n*2/z2;
45              printf("%4d parallel (%d) %8d usecs  %8d usecs   %5.1f0,
46                     n,pcnt,(int)(t2-t1),d1,f1);
47              printf("%4d sequential   %8d usecs  %8d usecs   %5.1f   %4.2f0,
48                     n,(int)(t4-t3),d2,f2, z2/z1);
49          }
50      }
51
52
```

Figure 8-3: Parallel Program, continued

```
53
54      gettime(t,tm)
55      /* set t = current time of century clock */
56      /* set tm = current exact user cpu usage */
57      struct timeval *tm;
58      long long *t;
59      {
60          getrusage(RUSAGE_SELF,&rusage);
61          *tm = rusage.ru_exutime;
62          *t = gettoc();
63      }
64
65      deltatime(t1,t2)
66      /* compute difference between two time value times, return single int        */
67      /* overflows when difference is greater than 2**31 usec (about 2000 seconds) */
68      struct timeval *t1,*t2;
69      {   return ((t2->tv_usec - t1->tv_usec) + (t2->tv_sec - t1->tv_sec) * 1000000);
70      }
71
72
73
74
75
76      multrow(i,n)
77      /* compute single row of matrix multiply */
78      /* use static matrices a,b,c */
79      int i,n;
80      {
81          int j,k;
82          i--;        /* adjust i to be 0 - (n-1) */
83
84          for (j = 0; j < n; j++) c[i][j]  = 0;
85
86          for (k = 0; k < n; k++)
87              for (j = 0; j < n; j++)
88                  c[i][j] += a[i][k]*b[k][j];
89
90      }
91
```

Figure 8-3: Parallel Program, continued

```
92
93      ;       exfn(func,n) --
94      ;       Spawns as many parallel threads as possible. Each one is
95      ;       given a separate stack of exfnssize bytes. Each thread obtains
96      ;       the next available index and executes (*func)(index,n).  When
97      ;       no more indexes are available, each thread executes a join
98      ;       instruction.  The join terminates all but the last thread
99      ;       so that exfn returns single threaded. The number of times the
100     ;       stack is incremented is used to determine the number of active
101     ;       threads during the computation.  This value is the return
102     ;       value of exfn.
103     ;
104     ; Registers x8000 - x801f are reserved by compiler and runtime systems.
105     STKOFF  =       0x8020
106     INDEX   =       0x8021
107             .data
108             .globl  _exfnssize      ; exfnssize is made global so that it
109                                     ; may be changed by the caller if a
110                                     ; longer stack is required.
111             .align 4                ; align word on word boundary
112     _exfnssize:     ds.w    0x10000
113             .text
114             .globl  _exfn
115
116     _exfn:
117             ulk     STKOFF          ; initialize STKOFF as stack offset
118             ulk     INDEX           ; initialize INDEX as index count
119             sub.w   a1,a1           ;
120             snd.w   a1,STKOFF       ; place a zero in STKOFF, set lock bit
121             snd.w   a1,INDEX        ; place a zero in INDEX, set lock bit
122             spawn   L1,fp           ; spawn other threads
123
124     L1:
125             ld.w    _exfnssize,a1   ; increment sp by stack size for thread
126             inc.w   STKOFF,a1
127             jbra.f  L1
128             sub.w   a1,sp           ; set individual stack for each thread
129     L2:
130             ld.w    4(ap),a2        ; a2 = n
131             ld.w    0(ap),a3        ; a3 = function address
132
```

Characters following a semicolon on a line are comments ignored by the assembler.

Lines 1–90 make up a C program that computes comparative execution speeds of a row computation. Line 32 points to the assembly-language code to execute the computation in parallel. Lines 76–90 compute a single row per invocation.

Lines 93–103 begin the assembly-language code and include comments that describe the function of the routine.

Lines 117–121 initialize two communication registers for use in synchronizing access to common data between the threads. The `ulk` instruction clears the hardware lock bit for a particular communication register so that the `snd` operation will not block. The `snd` instruction places an initial value in the communication register and locks the register. This locking is necessary so that the `inc` instruction used in lines 126 and 135 will not block.

Line 122 starts as many threads as there are processors available. All have identical `fp`, `ap`, and `sp` values.

Lines 124–128 increment the `STKOFF` communication register by `exfnssize` (stack size) so that each thread will have a different `sp`. The `inc` instruction automatically synchronizes access to `STKOFF`. Note that `jbra.f L1` includes an extra instruction. The purpose of this apparent inefficiency is to prevent the automatic "potential deadlock detection" hardware trap from coming into play. In this code, `STKOFF` is known to have the lock bit clear only during another `inc.w` instruction; the `inc.w` instruction will succeed within a very small number of tries.

Line 129 is the start of the main loop for `index=1, n`. `INDEX` is stored in a communication register, and each access to it is synchronized by means of `inc.w` instructions. Thus, each thread will pick up the appropriate next value on each pass through the loop. When all values have been obtained, each thread executes a `join` instruction. The `join` instruction on line 149 causes all threads except the last one to terminate. The last thread uses the `STKOFF` value to determine the number of threads that were active and returns that value in `s0`.

Lines 129–132 fetch arguments from the shared portion of the stack.

Line 133 continues on the next page.

Figure 8-3: Parallel Program, continued

```
133     L3:
134             ld.w    #1,a1           ; compute next index, place in a1
135             inc.w   INDEX,a1
136             jbra.f  L3
137             le.w    a1,a2
138             jbra.f  L4              ;if index > n, then (branch to exit)
139
140             psh.w   a2              ; push n
141             psh.w   a1              ; push index
142             mov     sp,ap           ; set argument pointer
143             pshea   #2              ; set argument count
144             calls   (a3)            ; call func(a,b,c,index,n)
145             add.w   #12,sp          ; pop arguments from stack
146             ld.w    12(fp),ap       ; restore argument pointer to original value
147             br      L2
148     L4:
149             join                    ; now do joins until single threaded again
150
151             get.l   STKOFF,s0       ; return thread count as curiosity
152             ld.w    _exfnssize,s1
153             div.w   s1,s0
154             rtn                     ; will restore correct sp for calling routine
155
156
157
158     ;
159     ;       gettoc - return the time of century register
160     ;                without overhead of syscall
161     ;       toc register is only available on C200 Series processors
162             .globl  _gettoc
163     _gettoc:
164             mov     toc,s0
165             rtn
166
```

Lines 133–138 keep the current INDEX in a communication register, allowing automatic synchronization with the inc.w instruction. As in Line 127, an extra instruction is included when the inc.w fails because of the other threads accessing the same communication register at the same time. This extra instruction prevents deadlock detection traps from invoking the kernel to reschedule the thread.

Lines 140–147 set up the function call. After the call, ap is restored, as registers might have been modified by the called subroutine.

Line 149 is the point a thread reaches only when index > n, so that all calls to func have been started. Each thread will terminate with the join, except the last one. The join instruction also prevents any more threads from being spawned. The last thread computes the thread count based on the value of the STKOFF communication register.

Chapter 9
Advanced Topics

This chapter describes general information and procedures for tailoring your assembly-language code. Specifically, this chapter defines keywords and discusses the following topics:

- coding techniques
- using macros and the preprocessor
- optimizing performance
- using *adb* and *csd* to debug programs
- writing parallel programs

Before continuing in this chapter, familiarize yourself with the keywords listed in the next section.

9.1 Keywords

Process—A collection of one or more instruction streams executing within a single logical address space

Multiprocessing—The creation and scheduling of processes on any subset of CPUs in a system configuration

Thread—Any single instruction stream executing within a process

9.2 Coding Techniques

Knowing when to write code in assembly language is an important consideration in programming. The profilers available in the *Consultant* debugging software (an optional product) are *prof* and *gprof*. These profiling tools can help identify CPU-intensive routines. To profile a program, specify the *-p* or *-pg* compiler option. After the program completes, run *prof* or *gprof*.

Generally, code your original algorithm in a high-level language and get your program to run completely (if slowly) before attempting to write the code in assembly language. It is only worthwhile to forego the ease of maintenance and coding provided by a high-level language if the code is very slow. Assembly code is machine dependent and not portable; it can, however, be fast code.

9.2.1 High-Level Language Processors

Using a high-level language processor, such as C or FORTRAN, is a time-saving way to generate assembly-language code. To use *cc*, *vc*, or *fc* to generate assembly-language code, specify the *-S* option. The *-S* option causes assembly-language code to be written to a file with the same name as the compiler source file but with an extension of *.s*.

To optimize the code, specify one of the *-O* (optimization) options. The *-O* options tell the compiler to call the optimizer.

To further improve the code, you can manually tune the *.s* file, which contains working code that can be hand optimized instead of generated from scratch.

9.2.2 Coding Hints

Often, no matter how much improvement is made in the code for an algorithm, the program still runs too slowly. When this happens, search for a better algorithm rather than attempt to enhance an existing one. The book *The Elements of Programming Style* by Kernighan and Plauger has several hints on how to restructure programs for greater speed.

When writing assembly-language code, follow the normal methods of subroutine linkage and register use. Do not hesitate, though, to use the registers cleverly (for example, instead of local variables) and reuse them as necessary in code that does not require a variable to be available throughout the extent of the code (for example, a loop variable can often be reused).

To embed assembly-language statements in C code instead of writing an entire routine in assembly language, use the *asm* directive; the *asm* syntax is as follows:

```
;asm ("dsi");          /* disable interrupts */
for (i = 0; i < 10; i++)
{
        if (interr[i])
                blast[i] = ON;
}
;asm ("eni");          /* enable interrupts */
```

The extra semicolon before the *asm* is good coding practice that forces the directive to work in all cases—even when it follows an *if* statement. If the semicolon is omitted before the *asm* statement that directly follows an *if* statement, the embedded assembly-language statements may be in the wrong part of the conditional.

9.3 Using Macros and the Preprocessor

To make your assembly-language code more readable or to help you tailor your code to a particular application, use macros and the preprocessor.

The *m4* macro processor provides a collection of built-in macros and allows you to define new macros. To define a new macro, enter

define (*macro_name, macro_text*)

where *macro_name* is an alphanumeric string that begins with a numeral (underscore _ counts as a numeral), and *macro_text* is any text that contains balanced parentheses. Subsequent occurrences of *macro_name* are replaced with *macro_text.*

The assembler does not provide a macro capability, but the C preprocessor does. To use the standard C preprocessor for macros and defined constants, enter

cc -E *name*.**c** > *name*.**s**

where *name* is the name of your assembly-language file. Then invoke the assembler.

9.4 Optimizing Performance

Understanding and using the basic concurrency of the CONVEX supercomputers, plus its pipelining, chaining, overlapping, and parallelizing abilities, can greatly increase the performance of the assembler.

During processing, the CPU passes instructions between the multiple asynchronous processing units. These units are interconnected through high-speed, 64-bit busses and can operate concurrently. Pipelining occurs when the CPU breaks an instruction into separate parts so that the parts are processed simultaneously by the different processing units.

The processing units also have caches that store information and can be used to speed up processing. Once an instruction has retrieved an address from main memory, the address resides in a cache within a processing unit. By ordering assembly-language instructions so that the assembler can retrieve addresses from the cache instead of having to go to main memory, you can cut down on processing time.

9.4.1 Optimizing Scalar Code

When optimizing scalar code, you can use caches more effectively if you order the instructions so that all loads are done before any operations, and all operations are done before any store instructions.

9.4.2 Optimizing Vector Code

When optimizing vector code, remember that there are three fully pipelined functional units that operate concurrently. The first unit is for add, subtract, and logical operations. The second is for load and store functions. The third is for multiply and divide operations. Vector edit functions operate in the second unit in the CONVEX C1 architecture and in the third unit on the CONVEX C200 Series architecture. These units can operate concurrently on three streams of independent data or can be chained together to operate on one data stream.

To optimize the code, then, rearrange instructions so that intrinsics that use the same functional unit are not together. For example, if you need to do both multiply and divide intrinsics, do not put the instructions in sequence. Split the multiply and divide functions with other functions, like add, load, or store.

To speed up the assembler further, do not put three references to a specific vector register bank in two sequential instructions. For example

```
add.d V0,V1,V1
mul.d V1,V3,V4
```

are processed sequentially instead of simultaneously because there are three references to V1. If, instead, the two instructions are

```
add.d V0,V1,V2
mul.d V2,V3,V4
```

the instructions are processed simultaneously. This process is called vector chaining. The functional unit that performs vector operations and scalar floating-point operations is separate from the functional unit that performs operations with the address of registers. This distinction permits overlapped execution of the instructions. For example, scalar loads and vector compares are executed concurrently.

If you are trying to optimize code that has been called by the FORTRAN compiler, be careful not to modify argument packets. Some of the argument packets may be precompiled. Since you cannot tell whether an argument packet has been precompiled, follow the general rule of not modifying any argument packets.

9.5 Debugging With *adb* and *csd*

The *adb* and *csd* debuggers operate on CONVEX supercomputers. The *csd* debugger is not helpful, however, when debugging assembly-language code.

The *adb* debugger is an object-code debugger that requires no special support from the loader or compilers. To use *adb*, enter

adb [**-w**] [*objfil* [*corfil*]]

where *objfil* is an executable CONVEX UNIX file and *corfil* is a core-image file produced by the operating system when a job terminates abnormally. The *-w* option creates both *objfil* and *corfil* (if necessary) and opens them for reading and writing so that *adb* can modify files.

9.6 Parallel Programming in CONVEX Assembly Language

This section addresses commonly used parallel constructs and discusses managing the parallel execution of the threads of a process. Before continuing, you should be familiar with communication registers, synchronization instructions, and CPU control instructions (see sections 2.4.4, 4.4.14, and 4.4.15, respectively, to review these topics). Refer to Figure 8-3 for an illustration of working parallel code. Specifically, this section describes

- Processes and threads
- Instructions to begin and end thread execution
- Communication between threads
- Thread memory management

9.6.1 Processes and Threads

CONVEX UNIX recognizes two types of data flow: processes and threads. A process, in the traditional UNIX sense of the word, is the execution of a program and associated data. To parallelize at the process level requires that you use system calls such as *fork* and *exec*. A thread is a single stream of execution within a process. A single process may create many threads, each sharing a common memory and kernel environment. To parallelize at the thread level requires that you use one of two programming schemes. One scheme uses *pfork*, *wfork*, and *cfork* instructions. The other scheme uses *spawn* and *join* instructions. (For further information on these instructions, see section 4.4.15, "CPU Control Instructions.")

These programming schemes represent different ways of synchronizing resources and allocating available CPU time. The following sections specifically address how to use these instructions to begin and end thread execution.

9.6.2 Creating Threads

The *pfork* (post a fork) and *spawn* instructions divide a process into threads for execution. The *pfork* instruction requests that one additional thread start executing. It is a dynamic, on-call request for an additional thread. The *spawn* instruction requests that as many threads start executing as there are processors available. It is a static, one-time request.

9.6.2.1 Processor Scheduling

The *pfork* and *spawn* instructions take advantage of the CONVEX architecture's automatic self-allocating processors (ASAP). ASAP works as follows: if a processor is idle when it encounters a *pfork* or *spawn* request, a new thread is created and starts executing. If a processor is occupied, then the request for an additional thread remains *pending*, meaning that it remains posted until a processor becomes available or the request is cleared. If a thread request is already pending when either a *pfork* or *spawn* is executed, then the operation fails, and the address-carry bit (C) in the PSW is reset to zero.

NOTE

Do not mix *spawn* and *pfork* instructions without proper synchronization.

9.6.2.2 Special Registers

When a thread is started, the contents of its registers are undefined except for the argument pointer (AP), stack pointer (SP), frame pointer (FP), and program counter (PC). A thread assumes the following values of these registers:

- AP (usually A6) is the same as the thread that initiates the *pfork* or *spawn*
- FP (usually A7) is the same as the thread that initiates the *pfork* or *spawn*
- SP is initially the value of FP
- PC is the instruction address; *pfork* and *spawn* instructions include fields for an address and an A register, as illustrated in the following example:

```
spawn   L1,fp
```

9.6.3 Ending Threads

Two instructions return a process to a single thread: *wfork* (wait for a fork) and *join*. The *wfork* instruction causes the thread to terminate, and waits for a pending thread. It does not, however, clear pending thread requests. The *wfork* instruction works as follows: if the thread belongs to a process that has a request for a new thread posted, then the new thread is created. If the thread belongs to a process that does not have a request for a new thread, but does have active threads on other processors, then another process is scheduled for the current processor.

CAUTION

If the current thread is the only active thread in the process and no other request for a new thread is pending, then the kernel receives a deadlock-detection interrupt, and the process is terminated.

Deadlock occurs when the threads of a process are waiting for a resource that is unavailable. The last thread of a process should never encounter a *wfork*, because *wfork* by definition looks for pending thread requests. Be sure that at least one thread will remain active before specifying a *wfork* instruction.

Use the *cfork* (clear a fork) instruction with *wfork* to clear any pending thread requests generated by a *pfork* or *spawn*.

The *join* instruction works as follows: each thread executes a *join* after it has been processed. The *join* instruction clears any pending thread requests for the current process. If more than one thread is active when a join is executed, then the CPU terminates the current thread. Terminating the thread relinquishes and deallocates the CPU. If the current thread is the last active thread of the current process, then the process continues with the next instruction. After all threads have executed a *join*, the process will only have one thread running.

9.6.4 Communicating Between Threads

A group of instructions included in the C200 Series instruction set work specifically to synchronize activities between threads of a process and the related communication registers (see section 4.4.14, "Synchronization Instructions").

Each communication register has a lock bit that may be used to synchronize access to the register. The *lck* instruction sets the lock bit. If the lock bit is clear, then C is set. If the lock bit is already set, then C is cleared to indicate failure. The *ulk* instruction clears the lock bit, and C is set to the previous value of the lock bit.

Generally speaking, you should follow a synchronization instruction with a *jbra.f* or *jbrs.f* instruction to branch back and repeat the synchronization instruction until it succeeds. These instructions use the address-carry bit (C) or scalar-carry bit (SC) to indicate success or failure.

9.6.5 Thread Memory Management

A thread's memory map is by default the same as that of the initiating thread, so all threads of a process share the same memory. A process or thread, however, may have either *shared* or *unshared* memory.

9.6.5.1 Shared Memory

Shared memory means that more than one thread may use the same logical address to read or write the same physical location in memory. Thread data segments are shared by default.

9.6.5.2 Unshared Memory

Unshared memory means that each thread wants to use the same logical address to access different physical locations in memory. Two registers on each CPU assist in managing unshared thread memory: the communication index register (CIR) and the thread identifier (TID).

CIR — serves as the primary point of reference between the CPU, a CONVEX UNIX process, and the communication registers. Remember that the C200 Series hardware supports 8 sets of 128 communication registers. Each CPU is linked to a set of communication registers by means of the CIR, which is assigned an appropriate value by the operating system. When a process divides into threads for execution, the CIR tracks which subset of communication registers is being used by the CPU.

TID — subdivides a process into individual pages of physical thread memory. TID is usually used with unshared memory because it makes each thread unique in the translation from logical to physical memory.

9.6.5.3 Thread Data Segments and Thread Stacks

Thread data segments are shared by default. To create data segments with unshared memory, use the assembler directives *.tdata* and *.tbss*. These directives work like *.data* and *.bss*, except that address space is *private*. Thread private means that each thread gets a private copy of variables and data declared in *.tdata* and *.tbss*.

Threads with unshared memory require separate stack spaces. The behavior of programs that do not have separate stack spaces for different threads is difficult to debug. To create individual thread stacks, use the *.tbss* directive to allocate data space and then assign a thread SP to that space.

9.7 Further Reference

For more information on multithreaded parallel programs, see the *CONVEX Architecture Reference*. For information on debugging multithreaded parallel programs, see the *CONVEX adb (Assembly-Language Debugger) User's Guide*.

A

Instruction Set

This appendix describes the CONVEX instruction set. A CONVEX instruction may be one of three lengths: one, two, or three halfwords. These lengths are equivalent to instructions that are 16, 32, or 48 bits long, respectively. Even though the fundamental unit of address is the byte, instructions are addressed on a halfword boundary. All instructions begin on even byte boundaries. Bit <0> of the Program Counter (PC) is never interpreted.

A.1 Notational Conventions

The following notation conventions describe the contents of memory in the assembly-language syntax:

- **Effective Address** – the effective memory address of an operand
- **c(effective address)** = **Sk** – the contents of the *Sk* register replace the contents of the memory location specified by the effective address.
- **Sk** = **c(effective address)** – the contents of the memory location specified by the effective address replace the contents of register *Sk*.
- | – specifies alternation. Thus, (a|b) means a or b.
- ! – comment delimiter. All text to the right of the ! is an comment relative to the metalanguage on that line.

A.2 Instruction Page Layout

Each instruction page conforms to the following page layout:

Purpose: The purpose or intent of the instruction

Architecture: The processors that implement the instruction

Format: The physical format of the instruction, including field locations and use. For extended opcodes, the prefix (0x7ef0 or 0x7ef8) is included, with the E bit (bit <3> of the prefix) used to distinguish between extended-0 space and extended-1 space. Some instructions have variants that are standard and extended. For example, *cvtw.s Sj,Sk* is a nonprefixed opcode that appears on the same page as *cvtw.d Sj,Sk*, which is a prefixed opcode. In this case, the prefix should be ignored. The opcode section described below denotes whether an opcode is standard (nonprefixed) or extended (prefixed).

Operation: The C metalanguage description of the instruction

Exceptions: A list of exceptions that are detected. If a trap is enabled, a trap occurs. For all instructions that reference memory, exceptions related to address translation (such as page faults or protection violations) can occur.

Opcode: A listing of the instruction mnemonic, hex opcode, opcode space mnemonic, binary opcode, effect on PSW, and opcode description. The binary representation shown for each opcode is 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 five columns. The first column is the opcode name. The second column contains the hexadecimal encoding of the opcode. The third column contains a two-character mnemonic that describes in which part of "opcode space" the instruction resides and the binary encoding of the opcode. Three opcode spaces are defined:

- ST — Standard opcode space, i.e., no prefix. Any prefix found in the "Format" section described above should be ignored.
- E0 — Extended-0 space, with prefix 0x7EF0
- E1 — Extended-1 space, with prefix 0x7EF8

The fourth column gives the names of the PSW bits that can be affected by the instruction.

The fifth column is the opcode function. For example, for the following instruction,

Mnemonic	Hex	Binary	PSW	Description
add.w Aj,Ak	0x5840	ST 0101100001	C,AIV	Add address registers word

add.w Aj,Ak is the opcode name and the assembly language syntax; 5840 is the hexadecimal opcode representation not including the prefix opcode; ST means that this instruction does not have a prefix opcode, and 0101100001 is the binary opcode representation. The C and AIV bits in the PSW are affected by this instruction. The phrase "Add address registers word" is a brief description of the opcode function.

Description: A description, in English, of the functions performed by the instruction

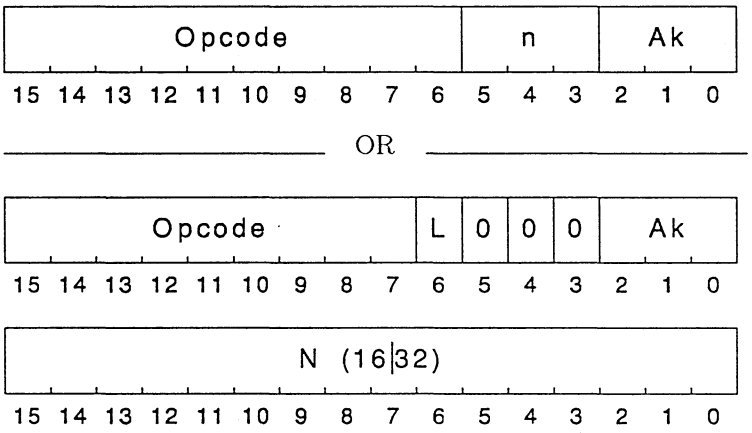
Notes: A list of notes that may be of interest when using this or other appropriate instructions

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

Purpose: To add an immediate field to the contents of an address register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Ak = Ak + Immediate;
```

Exceptions: (h|w): Integer Overflow

Opcode:

Mnemonic	Hex	Binary	PSW	Description
add.h #n,Ak	5880	ST 0101100010	C,AIV	Add short immediate address halfword
add.w #n,Ak	58C0	ST 0101100011	C,AIV	Add short immediate address word
add.h #N,Ak	1400	ST 0001010000	C,AIV	Add immediate address halfword
add.w #N,Ak	1480	ST 0001010001	C,AIV	Add immediate address word

Description: The sum of the (sign-extended) immediate field and the contents of address register Ak replace the contents of Ak.

Notes: Sign extension does not occur for the 3 bits of the short immediate form.

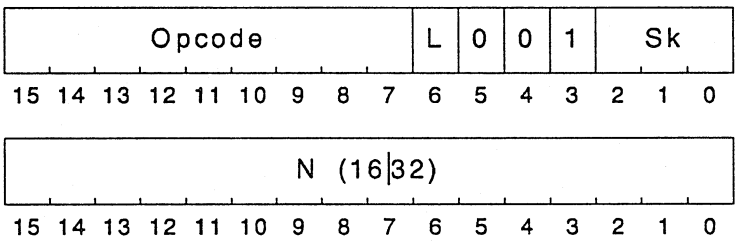
add.(h|w|s) #N,Sk

ADD SCALAR/IMMEDIATE

Purpose: To add an immediate field to the contents of a scalar register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Sk = Sk + Immediate;
```

Exceptions:

(h|w): Integer Overflow

(s): Reserved Operand
Exponent Overflow
Exponent Underflow

Opcode:

Mnemonic	Hex	Binary	PSW	Description
add.h #N,Sk	1408	ST 000101000	SIV,SC	Add scalar/immediate integer halfword
add.w #N,Sk	1488	ST 000101001	SIV,SC	Add scalar/immediate integer word
add.s #N,Sk	1808	ST 000110000	OV,UN,RO	Add scalar/immediate single float

Description: The sum of the (sign-extended) immediate field and the contents of scalar register Sk replace the contents of Sk.

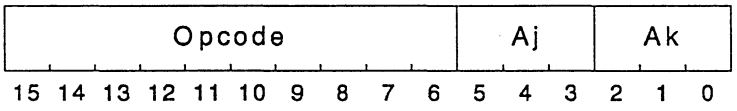
Notes: None

ADD ADDRESS/ADDRESS — add.(h|w) Aj,Ak

Purpose: To add the contents of an address register to the contents of an address register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Ak = Ak + Aj;
```

Exceptions: (h|w): Integer Overflow

Opcode:

Mnemonic	Hex	Binary	PSW	Description
add.h Aj,Ak	5800	ST 0101100000	C,AIV	Add address register halfword
add.w Aj,Ak	5840	ST 0101100001	C,AIV	Add address register word

Description: The sum of the contents of address registers Aj and Ak replaces the contents of Ak.

Notes: None

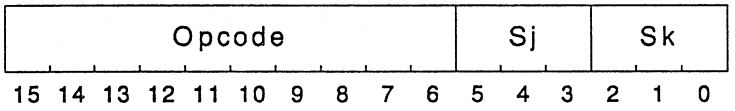
add.(b|h|w|l|s|d) Sj,Sk

ADD SCALAR/SCALAR

Purpose: To add the contents of a scalar register to the contents of a scalar register

Architecture: **C100 Series, C200 Series**

Format:



Operation: `Sk = Sk + Sj;`

Exceptions:

(b|h|w|l): Integer Overflow

(s|d): Reserved Operand
Exponent Overflow
Exponent Underflow

Opcode:

Mnemonic	Hex	Binary	PSW	Description
add.b Sj,Sk	5900	ST 0101100100	SIV,SC	Add scalar/scalar integer byte
add.h Sj,Sk	5940	ST 0101100101	SIV,SC	Add scalar/scalar integer halfword
add.w Sj,Sk	5980	ST 0101100110	SIV,SC	Add scalar/scalar integer word
add.l Sj,Sk	59C0	ST 0101100111	SIV,SC	Add scalar/scalar integer longword
add.s Sj,Sk	5500	ST 0101010100	OV,UN,RO	Add scalar/scalar single float
add.d Sj,Sk	5540	ST 0101010101	OV,UN,RO	Add scalar/scalar double float

Description: The sum of the contents of scalar registers Sj and Sk replaces the contents of Sk.

Notes: None

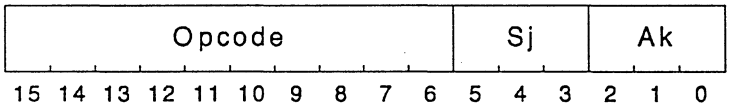
ADD SCALAR/ADDRESS

add.w Sj,Ak

Purpose: To add the contents of a scalar register to the contents of an address register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Ak = Ak + Sj;
```

Exceptions: Integer Overflow

Opcode:

Mnemonic	Hex	Binary	PSW	Description
add.w Sj,Ak	5000	ST 0101000000	C,AIV	Add scalar to address word

Description: The sum of the contents of address register Ak and the least significant 32 bits of scalar register Sj replace the contents of Ak.

Notes: None

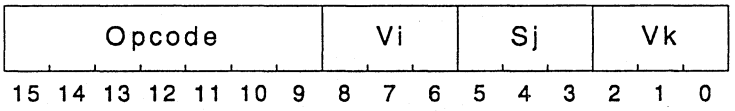
add.(b|h|w|l|s|d) Vi,Sj,Vk

ADD VECTOR/SCALAR

Purpose: To add the contents of a scalar register to the elements of a vector register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
for (a = 0; a < VL; a++) {
    Vk[a] = Vi[a] + Sj;
}
```

Exceptions:

(b|h|w|l): Integer Overflow

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
add.b Vi,Sj,Vk	C800	ST 1100100	SIV	Add vector/scalar integer byte
add.h Vi Sj,Vk	CA00	ST 1100101	SIV	Add vector/scalar integer halfword
add.w Vi,Sj,Vk	CC00	ST 1100110	SIV	Add vector/scalar integer word
add.l Vi,Sj,Vk	CE00	ST 1100111	SIV	Add vector/scalar integer longword
add.s Vi,Sj,Vk	B800	ST 1011100	OV,UN,RO	Add vector/scalar single float
add.d Vi,Sj,Vk	BA00	ST 1011101	OV,UN,RO	Add vector/scalar double float

Description: The sum of the contents of the scalar register Sj and contents of the corresponding element of Vi replace the first VL elements of the vector register Vk.

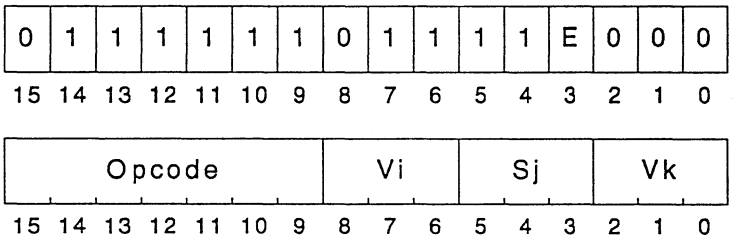
Notes: None

ADD VECTOR/SCALAR MASKED — add.(b|h|w|l|s|d).(t|f) Vi,Sj,Vk

Purpose: To add a scalar to a vector under control of the Vector Merge (VM) register

Architecture: **C200 Series only**

Format:



Operation:

```
switch (E) { /* prefix bit<3> */
      case TRUE:  /* .t */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 1) {        /* if VM<a> is TRUE */
                  Vk[a] = Vi[a] + Sj;
              }
          }       /* end of for loop */
          break;  /* go to end of switch */
      case FALSE: /* .f */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 0) {        /* if VM<a> is FALSE */
                  Vk[a] = Vi[a] + Sj;
              }
          }       /* end of for loop */
          break;  /* go to end of switch */
} /* end of switch */
```

Exceptions:

(b|h|w|l): Integer Overflow

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
add.b.t Vi,Sj,Vk	C800	E1 1100100	SIV	Add vector/scalar byte (VM)
add.b.f Vi,Sj,Vk	C800	E0 1100100	SIV	Add vector/scalar byte (!VM)
add.h.t Vi Sj,Vk	CA00	E1 1100101	SIV	Add vector/scalar halfword (VM)
add.h.f Vi Sj,Vk	CA00	E0 1100101	SIV	Add vector/scalar halfword (!VM)
add.w.t Vi,Sj,Vk	CC00	E1 1100110	SIV	Add vector/scalar word (VM)
add.w.f Vi,Sj,Vk	CC00	E0 1100110	SIV	Add vector/scalar word (!VM)
add.l.t Vi,Sj,Vk	CE00	E1 1100111	SIV	Add vector/scalar longword (VM)
add.l.f Vi,Sj,Vk	CE00	E0 1100111	SIV	Add vector/scalar longword (!VM)
add.s.t Vi,Sj,Vk	B800	E1 1011100	OV,UN,RO	Add vector/scalar single (VM)
add.s.f Vi,Sj,Vk	B800	E0 1011100	OV,UN,RO	Add vector/scalar single (!VM)
add.d.t Vi,Sj,Vk	BA00	E1 1011101	OV,UN,RO	Add vector/scalar double (VM)
add.d.f Vi,Sj,Vk	BA00	E0 1011101	OV,UN,RO	Add vector/scalar double (!VM)

Description: Each of the contents of the first VL elements of vector register Vk is replaced by the sum of the contents of scalar register Sj and the contents of the corresponding element of Vi if the corresponding VM bit is set (clear for *.f*).

Notes: None

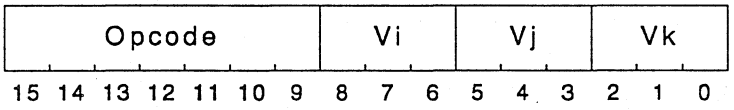
add.(b|h|w|l|s|d) Vi,Vj,Vk

ADD VECTOR/VECTOR

Purpose: To add the elements of two vector registers

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      Vk[a] = Vi[a] + Vj[a];
}
```

Exceptions:

(b|h|w|l): Integer Overflow

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
add.b Vi,Vj,Vk	C000	ST 1100000	SIV	Add vector/vector integer byte
add.h Vi,Vj,Vk	C200	ST 1100001	SIV	Add vector/vector integer halfword
add.w Vi,Vj,Vk	C400	ST 1100010	SIV	Add vector/vector integer word
add.l Vi,Vj,Vk	C600	ST 1100011	SIV	Add vector/vector integer longword
add.s Vi,Vj,Vk	B000	ST 1011000	OV,UN,RO	Add vector/vector single float
add.d Vi,Vj,Vk	B200	ST 1011001	OV,UN,RO	Add vector/vector double float

Description: Each of the contents of the first VL elements of vector register Vk is replaced by the sum of the contents of the corresponding elements of Vi and Vj.

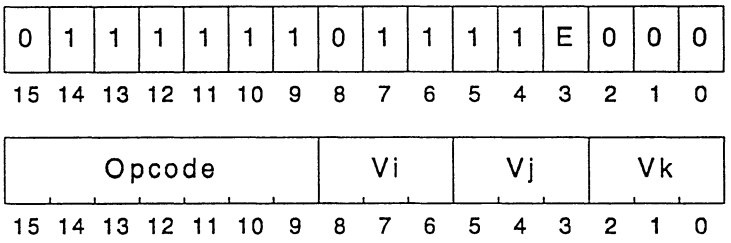
Notes: None

ADD VECTOR/VECTOR MASKED — add.(b|h|w|l|s|d).(t|f) Vi,Vj,Vk

Purpose: To add two vectors under control of the Vector Merge (VM) register

Architecture: C200 Series only

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
      case TRUE:   /*  .t */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 1) {        /* if VM<a> is TRUE */
                  Vk[a] = Vi[a] + Vj[a];
              }
          }        /* end of for loop */
          break;   /* go to end of switch */
      case FALSE:  /*  .f */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 0) {        /* if VM<a> is FALSE */
                  Vk[a] = Vi[a] + Vj[a];
              }
          }        /* end of for loop */
          break;   /* go to end of switch */
} /* end of switch */
```

Exceptions:

(b|h|w|l): Integer Overflow

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
add.b.t Vi,Vj,Vk	C000	E1 1100000	SIV	Add vector/vector byte (VM)
add.b.f Vi,Vj,Vk	C000	E0 1100000	SIV	Add vector/vector byte (!VM)
add.h.t Vi,Vj,Vk	C200	E1 1100001	SIV	Add vector/vector halfword (VM)
add.h.f Vi,Vj,Vk	C200	E0 1100001	SIV	Add vector/vector halfword (!VM)
add.w.t Vi,Vj,Vk	C400	E1 1100010	SIV	Add vector/vector word (VM)
add.w.f Vi,Vj,Vk	C400	E0 1100010	SIV	Add vector/vector word (!VM)
add.l.t Vi,Vj,Vk	C600	E1 1100011	SIV	Add vector/vector longword (VM)
add.l.f Vi,Vj,Vk	C600	E0 1100011	SIV	Add vector/vector longword (!VM)
add.s.t Vi,Vj,Vk	B000	E1 1011000	OV,UN,RO	Add vector/vector single (VM)
add.s.f Vi,Vj,Vk	B000	E0 1011000	OV,UN,RO	Add vector/vector single (!VM)
add.d.t Vi,Vj,Vk	B200	E1 1011001	OV,UN,RO	Add vector/vector double (VM)
add.d.f Vi,Vj,Vk	B200	E0 1011001	OV,UN,RO	Add vector/vector double (!VM)

Description: Each of the contents of the first VL elements of vector register Vk is replaced by the sum of contents of the corresponding elements of Vi and Vj if the corresponding VM bit is set (clear for *.f*).

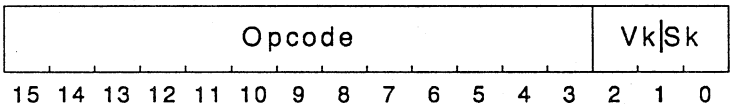
Notes: None

all (Vk|Sk) — AND REDUCE VECTOR

Purpose: To AND reduce all the elements of a vector

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      Sk = Sk & Vk[a];
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
all Vk	7E20	ST 0111111000100	None	AND reduce a vector
all Sk	7E20	ST 0111111000100	None	AND reduce a vector

Description: The bit-wise AND of all 64 bits of the contents of scalar register Sk and the contents of each of the first VL elements of vector register Vk replace the contents of Sk.

Notes:

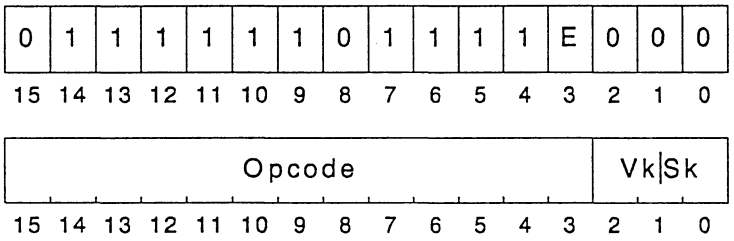
1. Initialize the scalar register properly for the first use of the AND reduce instruction (probably to 0xFFFFFFFF (word) or 0xFFFFFFFFFFFFFFFF (longword)).
2. Either Vk or Sk may be used as a valid argument to this instruction. This instruction operates in distinct matched vector and scalar register pairs: (V0,S0), (V1,S1), (V2,S2), (V3,S3), (V4,S4), (V5,S5), (V6,S6), (V7,S7).

AND REDUCE VECTOR MASKED — all.(t|f) (Vk|Sk)

Purpose: To AND reduce the elements of a subset of a vector register under control of the Vector Merge (VM) register

Architecture: **C200 Series only**

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
      case TRUE:   /*  .t */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 1) {        /* if VM<a> is TRUE */
                  Sk = Sk & Vk[a];
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
      case FALSE:  /*  .t */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 0) {        /* if VM<a> is FALSE */
                  Sk = Sk & Vk[a];
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
all.t Vk	7E20	E1 0111111000100	None	AND reduce a vector (VM)
all.t Sk	7E20	E1 0111111000100	None	AND reduce a vector (VM)
all.f Vk	7E20	E0 0111111000100	None	AND reduce a vector (!VM)
all.f Sk	7E20	E0 0111111000100	None	AND reduce a vector (!VM)

Description: The logical AND of all 64 bits of the contents of scalar register Sk and the contents of each one of the first VL elements of vector register Vk replaces the contents of Sk. Only elements of Vk with corresponding vector merge (VM) bit set (clear for *.f*) participate in the reduction.

Notes:

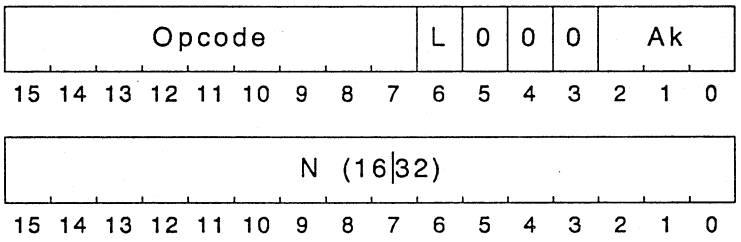
1. Initialize the scalar register properly for the first use of the AND reduce instruction (probably to 0xFFFFFFFF or 0xFFFFFFFFFFFFFFFF).
2. Either Vk or Sk may be used as a valid argument to this instruction. This instruction operates in distinct matched vector and scalar register pairs: (V0,S0), (V1,S1), (V2,S2), (V3,S3), (V4,S4), (V5,S5), (V6,S6), (V7,S7).

and #N,Ak — AND ADDRESS/IMMEDIATE

Purpose: To AND an immediate field to the contents of an address register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Ak = Ak & Immediate;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
and #N,Ak	1200	ST 000100100	None	AND immediate to address register

Description: The bit-wise AND of the (sign-extended) immediate field and the contents of address register Ak replace the contents of Ak. The most significant 32 bits of *Sk* are not affected.

Notes: None

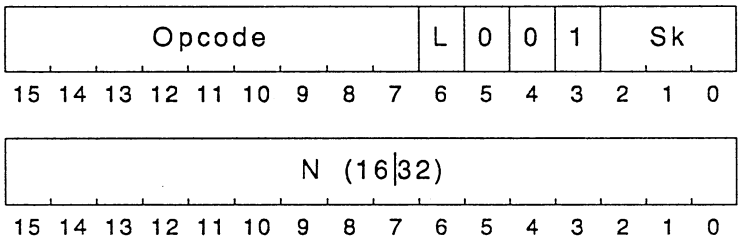
AND SCALAR/IMMEDIATE

and #N,Sk

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

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Sk = Sk & Immediate;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
and #N,Sk	1208	ST 000100100	None	AND scalar/immediate

Description: The bit-wise AND of the (sign-extended) immediate field and the least significant 32 bits of scalar register Sk replace the least significant 32 bits of Sk. The most significant 32 bits of Sk are not affected.

Notes: None

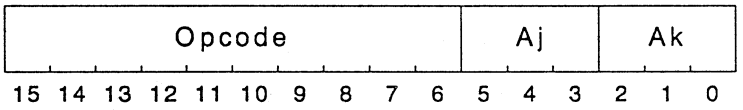
and Aj,Ak

AND ADDRESS/ADDRESS

Purpose: To AND the contents of two address registers

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Ak = Ak & Aj;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
and Aj,Ak	5200	ST 0101001000	None	AND address register

Description: The bit-wise AND of the contents of address registers Aj and Ak replaces the contents of Ak.

Notes: None

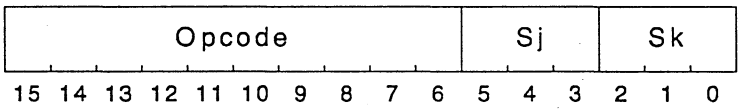
AND SCALAR/SCALAR

and Sj,Sk

Purpose: To AND the contents of two scalar registers

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Sk = Sk & Sj;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
and Sj,Sk	5300	ST 0101001100	None	AND scalar/scalar

Description: The bit-wise AND of the contents of scalar registers Sj and Sk replaces the contents of Sk.

Notes: None

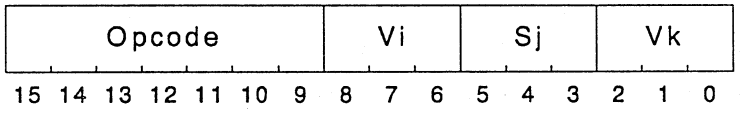
and Vi,Sj,Vk

AND VECTOR/SCALAR

Purpose: To AND the elements of a vector register and the contents of a scalar register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      Vk[a] = Vi[a] & Sj;
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
and Vi,Sj,Vk	A800	ST 1010100	None	AND vector/scalar

Description: Each of the contents of the first VL elements of vector register Vk is replaced by the logical AND of the contents of scalar register Sj and the contents of the corresponding element of Vi.

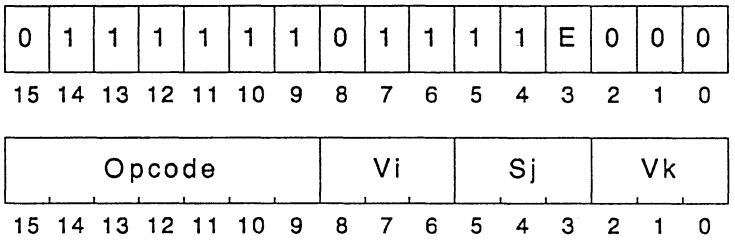
Notes: None

AND VECTOR/SCALAR MASKED — and.(t|f) Vi,Sj,Vk

Purpose: To AND the contents of a vector and a scalar under control of the Vector Merge (VM) register

Architecture: **C200 Series only**

Format:



Operation:

```
switch (E) { /* prefix bit<3> */
     case TRUE:  /* .t */
         for (a = 0; a < VL; a++) {
             if (VM<a> == 1) {        /* if VM<a> is TRUE */
                 Vk[a] = Vi[a] & Sj;
             }
         }   /* end of for loop */
         break;  /* go to end of switch */
     case FALSE: /* .f */
         for (a = 0; a < VL; a++) {
             if (VM<a> == 0) {        /* if VM<a> is FALSE */
                 Vk[a] = Vi[a] & Sj;
             }
         }   /* end of for loop */
         break;  /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
and.t Vi,Sj,Vk	A800	E1 1010100	None	AND vector/scalar (VM)
and.f Vi,Sj,Vk	A800	E0 1010100	None	AND vector/scalar (!VM)

Description: Each of the contents of the first VL elements of vector register Vk is replaced by the logical AND of the contents of scalar register Sj and the contents of the corresponding element of Vi if the corresponding vector merge (VM) bit is set (clear for *.f*).

Notes: None

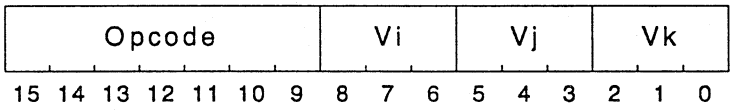
and Vi,Vj,Vk

AND VECTOR/VECTOR

Purpose: To AND the elements of two vector registers

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      Vk[a] = Vi[a] & Vj[a];
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
and Vi,Vj,Vk	A000	ST 1010000	None	AND two vectors

Description: The logical AND of the contents of corresponding elements of Vi and Vj replaces the first VL elements of vector register Vk.

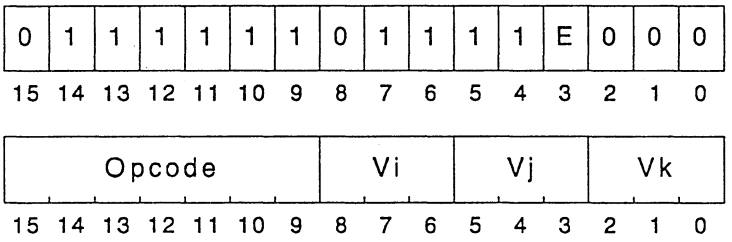
Notes: Copy the contents of vector register Vi to Vj with the instruction *and Vi, Vi, Vj*.

AND VECTOR/VECTOR MASKED — and.(t|f) Vi,Vj,Vk

Purpose: To AND the contents of two vectors under control of the Vector Merge (VM) register

Architecture: **C200 Series only**

Format:



Operation:

```
switch (E) { /* prefix bit<3> */
     case TRUE:  /* .t */
         for (a = 0; a < VL; a++) {
             if (VM<a> == 1) {        /* if VM<a> is TRUE */
                 Vk[a] = Vi[a] & Vj[a];
             }
         }  /* end of for loop */
         break;  /* go to end of switch */
     case FALSE: /* .f */
         for (a = 0; a < VL; a++) {
             if (VM<a> == 0) {        /* if VM<a> is FALSE */
                 Vk[a] = Vi[a] & Vj[a];
             }
         }  /* end of for loop */
         break;  /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
and.t Vi,Vj,Vk	A000	E1 1010000	None	AND two vectors (VM)
and.f Vi,Vj,Vk	A000	E0 1010000	None	AND two vectors (!VM)

Description: The contents of each of the first VL elements of vector register Vk is replaced by the logical AND of the contents of corresponding elements of Vi and Vj if the corresponding VM bit is set (clear for *.f*).

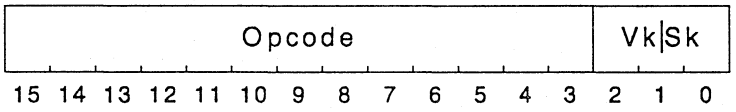
Notes: None

any (Vk|Sk) — OR REDUCE VECTOR

Purpose: To OR reduce all the elements of a vector register

Architecture: C100 Series, C200 Series

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      Sk = Sk & Vk[a];
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
any Vk	7E28	ST 0111111000101	None	OR reduce a vector
any Sk	7E28	ST 0111111000101	None	OR reduce a vector

Description: The bit-wise OR of the contents of scalar register Sk and the contents of each of the first VL elements of vector register Vk replace the contents of Sk.

Notes:

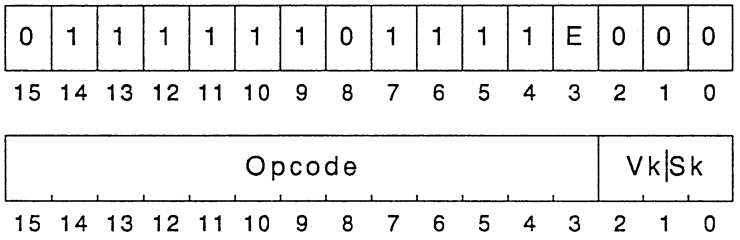
1. Initialize the scalar register properly for the first use of the OR reduce instruction (probably to 0).
2. Either Vk or Sk may be used as a valid argument to this instruction. This instruction operates in distinct matched vector and scalar register pairs: (V0,S0), (V1,S1), (V2,S2), (V3,S3), (V4,S4), (V5,S5), (V6,S6), (V7,S7).

OR REDUCE VECTOR MASKED — any.(t|f) (Vk|Sk)

Purpose: To OR reduce the elements of a subset of a vector register under control of the Vector Merge (VM) register

Architecture: C200 Series only

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
      case TRUE:   /* .t */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 1) {        /* if VM<a> is TRUE */
                  Sk = Sk | Vk[a];
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
      case FALSE:  /* .f */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 0) {        /* if VM<a> is FALSE */
                  Sk = Sk | Vk[a];
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
any.t Vk	7E28	E1 0111111000101	None	OR reduce a vector (VM)
any.t Sk	7E28	E1 0111111000101	None	OR reduce a vector (VM)
any.f Vk	7E28	E0 0111111000101	None	OR reduce a vector (!VM)
any.f Sk	7E28	E0 0111111000101	None	OR reduce a vector (!VM)

Description: The logical OR of the contents of scalar register Sk and the contents of each of the first VL elements of vector register Vk replace the contents of Sk. Only elements of Vk with corresponding VM bit set (clear for *.f*) participate in the reduction.

Notes:

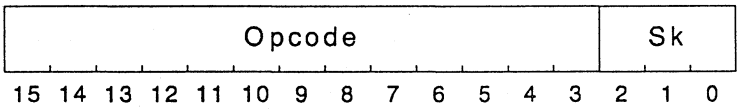
1. Initialize the scalar register properly for the first use of the OR reduce instruction (probably to 0).
2. Either Vk or Sk may be used as a valid argument to this instruction. This instruction operates in distinct matched vector and scalar register pairs: (V0,S0), (V1,S1), (V2,S2), (V3,S3), (V4,S4), (V5,S5), (V6,S6), (V7,S7).

atan.(s|d) Sk — ARC TANGENT

Purpose: To compute the trigonometric arc-tangent of the contents of a scalar register.

Architecture: **C200 Series only**

Format:



Operation:

```
Sk = atan(Sk);
```

Exceptions: (s|d): Floating Instrinsic Error
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
atan.s Sk	7DF0	ST 0111110111110	RO,FIN,IEC	Arc-tangent of a single float
atan.d Sk	7DF8	ST 0111110111111	RO,FIN,IEC	Arc-tangent of a double float

Description: The trigonometric arc-tangent of the contents of Sk replaces the contents of Sk.

Notes:

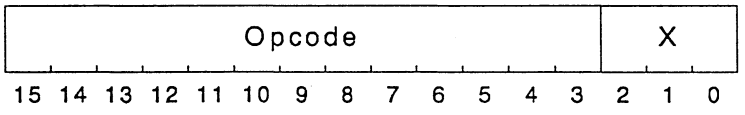
1. The result in an angle in radians between $-\pi/2$ and $\pi/2$.
2. Intrinsic traps go through the same trap handler as other arithmetic traps (RO, FDZ, UN, etc.). If PSW <FUE> and/or PSW <FE> are set and and intrinsic traps are not (INE clear), these bits must be examined to determine the type of the current trap.
3. When PSW<FIN> is set, the PSW<IEC> bits contain a code that specifies the type of error encountered by the intrinsic instruction. Refer to the *CONVEX Architecture Reference*, "Register Set" chapter, "Program Status Word — C200 Series" section for more information on the PSW<IEC> error codes and arithmetic trap conditions.

BREAKPOINT bkpt

Purpose: To jump to a debugger via a breakpoint call

Architecture: **C100 Series, C200 Series**

Format:



Operation: `Perform an extended call to the address contained in the word at address 0000 0050 in page 0 of the current ring.`

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
bkpt	7D50	ST 0111110101010	See note 1	Breakpoint

Description: The *bkpt* instruction executes a subroutine call to the address contained in the word found at address 0000 0050 in the current ring. This call pushes an extended return block (which includes all A and S registers) onto the user stack. The Program Counter (PC) saved in the return block references the instruction immediately following the *bkpt* instruction.

Notes:

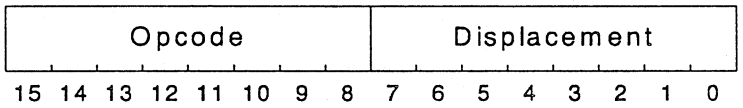
1. The following PSW bits are affected:
   C, AIV, ADZ, SC, SIV, SDZ, OV, UN, FDZ, RO = 0
   FRL = 00
2. Because the length of the *bkpt* instruction is one halfword, it can replace any instruction in the CONVEX C100 Series and C200 Series architecture.
3. The X field is ignored.

br — BRANCH ON PSW BIT

Purpose: To perform a short Program Counter (PC) relative branch, possibly depending on some condition

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
if (condition == TRUE) {
     PC = PC_of_branch + (2 * sign_extended_displacement).
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
br	7100	ST 01110001	None	Branch always
bri.f	7200	ST 01110010	None	Branch on ION false[5]
bri.t	7300	ST 01110011	None	Branch on ION true[5]
bra.f	7400	ST 01110100	None	Branch on address carry false
bra.t	7500	ST 01110101	None	Branch on address carry true
brs.f	7600	ST 01110110	None	Branch on scalar carry false
brs.t	7700	ST 01110111	None	Branch on scalar carry true

Description: If the specified condition is asserted, the sum of twice the sign-extended, 8-bit displacement and the contents of the PC (which still contains the address of the branch instruction) replace the PC. If the tested condition is not asserted, then the next sequential instruction is executed.

Notes:

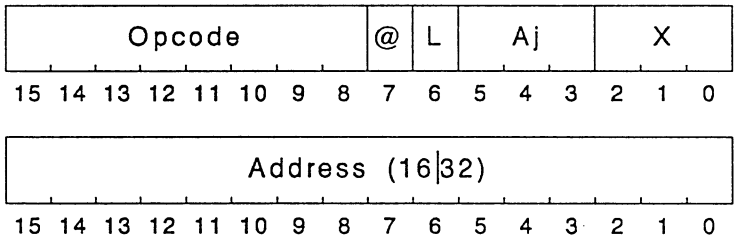
1. These instructions are used for short PC relative branches. All branches are restricted to the current ring.
2. Additional instructions exist to jump to any arbitrary instruction. Refer to the *jmp* instruction.
3. The range of the branch instruction is +127 to −128 *halfwords* (+254 to −256 bytes).
4. The displacement value is calculated automatically from the relative location of a referenced label when the object code is generated from a higher level language.
5. The *bri.(t | f)* instructions should not be used with the C200 Series architecture. The ION flag is asynchronous to the processor. Refer to the notes for the *dsi* and *eni* instructions.

CALL A SUBROUTINE — call <effa>

Purpose: To call a subroutine, creating a long call frame

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
psw[FRL] = 01;          /* long frame */
push(S1); push(S2); push(S3); push(S4); push(S5); push(S6); push(S7);
push(A1); push(A2); push(A3); push(A4); push(A5); push(A6); push(A7);
push(PSW);
push(next_instruction_address);
psw[FRL] = 0; psw[C]  = 0; psw[SC] =  0; psw[AIV] = 0; psw[ADZ] = 0;
psw[UN]  = 0; psw[OV] = 0; psw[FDZ] = 0; psw[RO]  = 0; psw[SIV] = 0;
psw[SDZ] = 0; psw[FIN] = 0;
SP = SP - 92;
A7 = A0;
PC = Effective_Address;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
call <effa>	2000	ST 00100000	See note 1	Call a subroutine, long frame

Description: This instruction pushes a description of the current stack frame onto the stack and then creates a new stack frame. The long call saves all A and S registers except A0 and S0. The Frame Length (FRL) bits in the saved Processor Status Word (PSW) indicate the frame size created in order that the stack can be "unwound" correctly.

A0 and A7 both reference the new top of stack; no other registers are changed. The effective address of the call instruction replaces the value of the Program Counter (PC).

The trap-enable bits of the PSW propagate from the caller to the callee. If the caller has floating-point overflow traps enabled, the callee also has floating-point traps enabled. The status bits of the callee's PSW are reset to 0.

Notes:

1. The following PSW bits are affected:
   C, SC, AIV, ADZ, FRL, UN, OV, FDZ, RO, SIV, SDZ, FIN = 0
2. In typical usage, a *sub.w #N,SP* creates a local area for variables.
3. Software convention dictates that argument references usually be positive displacements from the argument pointer.
4. Software convention dictates that S0 typically holds return values of functions.
5. The FRL bits in the PSW register itself are always reset to 0. To determine the type of frame created on the stack, the FRL bits of the PSW in the saved frame must be examined (the PSW is a constant distance from the top of stack).
6. The *call* instruction is restricted to be within the current ring. That is, if the current ring is 4, the most significant bit of the effective address is ignored. Otherwise, the most significant 3 bits of the effective address are ignored.

7. Before the PSW is pushed on the stack, all existing concurrent processing is completed. This ensures that all exception condition flags accurately reflect the state of the CPU.
8. The X field is ignored.

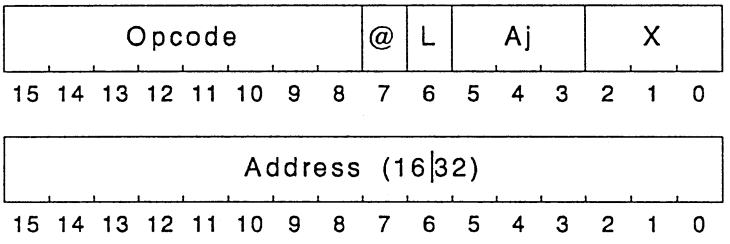
PUSH PC and JUMP

callq <effa>

Purpose: To push the Program Counter (PC) onto the stack and jump

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
push(next_instruction_address);
PC = Effective_Address;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
callq <effa>	2200	ST 00100010	None	Push the PC and jump

Description: The address of the instruction immediately following the *callq* instruction is pushed onto the stack. The effective address replaces the contents of the PC.

Notes:

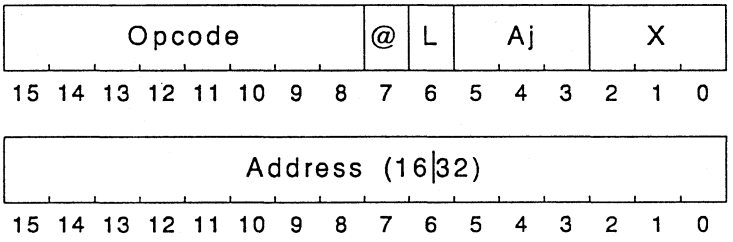
1. This instruction is a fast subroutine call for use when the current address context need not be altered. The *rtnq* instruction is a return from a routine invoked with the *callq* instruction.
2. The *callq* instruction is restricted to be within the current ring. That is, if the current ring is 4, the most significant bit of the effective address is ignored. Otherwise, the most significant 3 bits of the effective address are ignored.
3. The X field is ignored.

calls <effa> — CALL A SUBROUTINE

Purpose: To call a subroutine, creating a short call frame

Architecture: C100 Series, C200 Series

Format:



Operation:

```
PSW[FRL] = 11;          /* short frame */
push(A1); push(A2); push(A3); push(A4); push(A5); push(A6); push(A7);
push(PSW);
push(next_instruction_address);
psw[FRL] = 0;  psw[C]   = 0;  psw[SC]  = 0;  psw[AIV] = 0;  psw[ADZ] = 0;
psw[UN]  = 0;  psw[OV]  = 0;  psw[FDZ] = 0;  psw[RO]  = 0;  psw[SIV] = 0;
psw[SDZ] = 0;  psw[FIN] = 0;
A0 = A0 - 16;
A7 = A0;
PC = Effective_Address;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
calls <effa>	2100	ST 00100001	See note 1	Call a subroutine, short frame

Description: This instruction pushes a description of the current stack frame onto the stack and then creates a new stack frame. The short call saves only registers A6 and A7 (the frame and argument pointers). The Frame Length (FRL) bits in the saved Processor Status Word (PSW) indicate the frame size created in order that the stack can be "unwound" correctly.

A0 and A7 both reference the new top of stack; no other registers are changed. The effective address of the call instruction replaces the value of the Program Counter (PC).

The trap-enable bits of the PSW propagate from the caller to the callee. If the caller has floating-point overflow traps enabled, the callee also has floating-point traps enabled. The status bits of the callee's PSW are reset to 0.

Notes:

1. The following PSW bits are affected:
   C, SC, AIV, ADZ, FRL, UN, OV, FDZ, RO, SIV, SDZ, FIN = 0
2. In typical usage, a *sub.w #N,SP* creates a local area for variables.
3. Software convention dictates that argument references usually be positive displacements from the argument pointer.
4. Software convention dictates that S0 typically holds return values of functions.
5. The FRL bits in the PSW register itself are always reset to 0. To determine the type of frame created on the stack, the FRL bits of the PSW in the saved frame must be examined (the PSW is a constant distance from the top of stack).
6. The *calls* instruction is restricted to be within the current ring. That is, if the current ring is 4, the most significant bit of the effective address is ignored. Otherwise, the most significant 3 bits of the effective address are ignored.

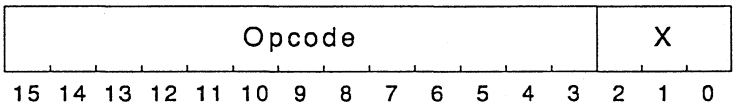
7. Before the PSW is pushed on the stack, all existing concurrent processing is completed. This ensures that all exception condition flags accurately reflect the state of the CPU.
8. The X field is ignored.

cfork — CLEAR FORK

Purpose: To clear a fork

Architecture: C200 Series only

Format:



Operation:

```
if (C = rcv(forkposted))  {   /* PSW<C> = 1 if rcv() succeeds */
      rcv(forklck);
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
cfork	7C88	ST 0111110010	C	Clear a fork

Description: Clear the hardware communication fork event registers in the current CIR. The *cfork* instruction clears any outstanding forks in the hardware communication registers in the current CIR. If no fork was outstanding, *cfork* returns C = 0; otherwise, if a fork was cleared, *cfork* returns C = 1.

Notes:

1. The *cfork* instruction should not be used with the *spawn* instruction. The *cfork* instruction should only be used to clear a fork that was posted using a *pfork* instruction.
2. The X field is unused.

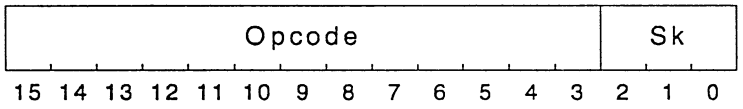
COSINE

cos.(s|d) Sk

Purpose: To compute the trigonometric cosine of the contents of a scalar register

Architecture: **C200 Series only**

Format:



Operation:

```
Sk = cos(Sk);
```

Exceptions: (s|d): Reserved Operand
Floating Instrinsic Error

Opcode:

Mnemonic	Hex	Binary	PSW	Description
cos.s Sk	7CE0	ST 0111110011100	RO,FIN,IEC	Cosine of a single-precision number
cos.d Sk	7CE8	ST 0111110011101	RO,FIN,IEC	Cosine of a double-precision number

Description: The cosine of the contents of Sk replaces the contents of Sk.

Notes:

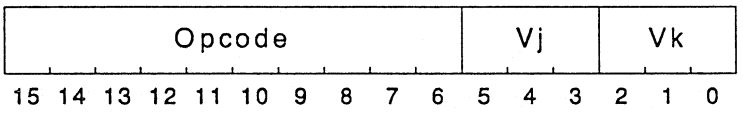
1. The input operand is interpreted as an angle in radians.
2. Intrinsic traps go through the same trap handler as other arithmetic traps (RO, FDZ, UN, etc.). If PSW <FUE> and/or PSW <FE> are set and and intrinsic traps are not (INE clear), these bits must be examined to determine the type of the current trap.
3. When PSW<FIN> is set, the PSW<IEC> bits contain a code that specifies the type of error encountered by the intrinsic instruction. Refer to the *CONVEX Architecture Reference*, "Register Set" chapter, "Program Status Word — C200 Series" section for more information on the PSW<IEC> error codes and arithmetic trap conditions.

cprs.(t|f) Vj,Vk — COMPRESS VECTOR

Purpose: To compress a vector using the Vector Merge (VM) register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
a = 0;
switch (E) {  /* prefix bit<3> */
     case TRUE:   /*  t */
          for (b = 0; b < VL; b++) {
               if (VM<b> == 1) {    /* if VM<b> is TRUE */
                    Vk[a] = Vj[b];
                    a = a + 1;
               }
          }   /* end of for loop */
          break;   /* go to end of switch */
     case FALSE:  /*  f */
          for (b = 0; b < VL; b++) {
               if (VM<b> == 0) {    /* if VM<b> is FALSE */
                    Vk[a] = Vj[b];
                    a = a + 1;
               }
          }   /* end of for loop */
          break;   /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
cprs.f Vj,Vk	6380	ST 0110001110	None	Compress a vector (!VM)
cprs.t Vj,Vk	63C0	ST 0110001111	None	Compress a vector (VM)

Description: The compress instructions copy all 64 bits of those elements from the source vector register Vj that are selected (or not selected, for *cprs.f*) by the VM register into contiguous elements at the beginning of the destination vector register, Vk. The number of elements copied to Vk is equal to the number of 1's (or 0's for *cprs.f*) in VM.

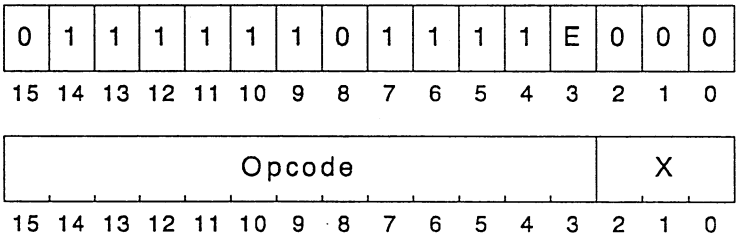
Notes: The *plc VM* instruction calculates the new length of Vk.

CPU TIMER SYNCHRONIZE GLOBAL — ctrsg

Purpose: To update the CPU timer registers in the entire complex to the current time

Architecture: **C200 Series only**

Format:



Operation:

```
for (i = 0; i <= maxcpuid; i++) {
    CIR.cpu_execution_clock[PC,i] += accrued_time;
}
```

Exceptions: Ring Violation (Privileged Instruction)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
ctrsg	7C38	E0 0111110000	None	Move scalar to CPU timer

Description: The CPU timer registers are usually only updated by hardware on ring crossings and process creation or termination. This instruction causes the CPU timer registers on all CPUs in the complex to be updated immediately.

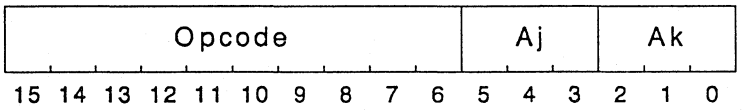
Notes: The X field is ignored.

cvt Aj,Ak — CONVERT INTEGER ADDRESS

Purpose: To convert the integer contents of an address register to an integer of different precision

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Ak = Convert(Aj); /* according to the opcode */
```

Exceptions: (b|h): Integer Overflow

Opcode:

Mnemonic	Hex	Binary	PSW	Description
cvtw.b Aj,Ak	4000	ST 0100000000	AIV	Convert word to byte
cvtw.h Aj,Ak	4040	ST 0100000001	AIV	Convert word to halfword
cvtb.w Aj,Ak	4080	ST 0100000010	None	Convert byte to word
cvth.w Aj,Ak	40C0	ST 0100000011	None	Convert halfword to word

Description: The converted, possibly sign-extended, contents of the source address register, Aj, replace the contents of Ak. Conversions to smaller data sizes can cause the overflow exception.

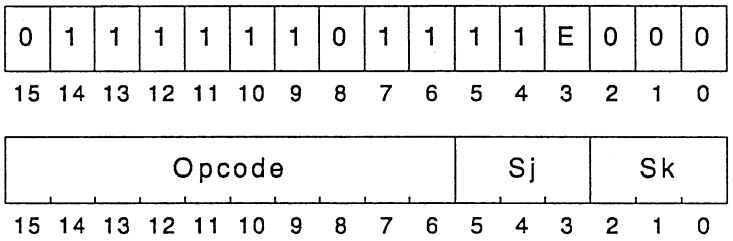
Notes:

1. Implement halfword-to-byte conversions by using *cvth.w* followed by *cvtw.b*. Implement byte-to-halfword conversions with *cvtb.w* followed by *cvtw.h*.
2. Specify unsigned conversions using logical *and* instructions.

Purpose: To convert the contents of one scalar register to a value of different precision or type

Architecture: C100 Series[6d], C200 Series

Format:



Operation:

```
Sk = Convert(Sj); /* according to the opcode */
```

Exceptions:

(b|h|w|l): Integer Overflow

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
cvtw.s Sj,Sk	4200	ST 0100001000	None	Convert word to single float
cvts.w Sj,Sk	4240	ST 0100001001	RO,SIV	Convert single float to word
cvtd.s Sj,Sk	4280	ST 0100001010	RO,SIV,OV,UN	Convert double float to single float
cvts.d Sj,Sk	42C0	ST 0100001011	RO,SIV	Convert single float to double float
cvtw.b Sj,Sk	4100	ST 0100000100	SIV	Convert word to byte
cvtw.h Sj,Sk	4140	ST 0100000101	SIV	Convert word to halfword
cvtb.w Sj,Sk	4180	ST 0100000110	None	Convert byte to word
cvth.w Sj,Sk	41C0	ST 0100000111	None	Convert halfword to word
cvts.l Sj,Sk	4300	ST 0100001100	RO,SIV	Convert single float to longword
cvtd.l Sj,Sk	4340	ST 0100001101	RO,SIV	Convert double float to longword
cvtl.s Sj,Sk	4380	ST 0100001110	None	Convert longword to single float
cvtl.d Sj,Sk	43C0	ST 0100001111	None	Convert longword to double float
cvtl.w Sj,Sk	4500	ST 0100010100	SIV	Convert longword to word
cvtw.l Sj,Sk	4540	ST 0100010101	None	Convert word to longword
cvtw.d Sj,Sk	4500	E0 0100010100	None	Convert word to double float
cvtd.w Sj,Sk	4540	E0 0100010101	RO,SIV	Convert double float to word

Description: The converted contents of the source scalar register, Sj, replace the contents of Sk. Conversions to smaller data sizes can cause the underflow or overflow exceptions. Conversions from floating-point representation to integer use truncation (rounding toward 0) as the rounding algorithm.

Notes:

1. Implement halfword-to-byte conversions by using *cvth.w* followed by *cvtw.b*. Implement byte-to-halfword conversions with *cvtb.w* followed by *cvtw.h*.
2. Specify unsigned integer conversions using logical AND instructions.
3. Convert instructions modify only the bits of the specified precision of the destination operand; all other bits are unchanged.
4. If an input operand is a floating-point reserved operand, then the destination is unchanged and both the RO and SIV flags are set to 1.

5. Truncation from float to fixed follows the FORTRAN standard: -5.9 is truncated to -5; 5.9 is truncated to 5.
6. Conversion from integer to floating-point types is performed thus:

C100, C200 Series

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. If there are any lesser significant bits of the normalized fixed-point number that cannot be contained in the fraction, round the fraction based on these lesser significant bits.

c. Conversion from double float to single float (*cvtd.s*) can cause underflow. If the exponent of the double precision input operand is smaller than the minimum range of single precision, then an underflow is indicated . The result returned is true zero.

C100 Series only

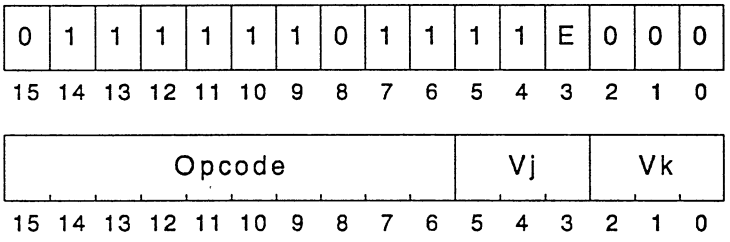
d. Execution of the *cvtw.d Sj, Sk* and *cvtd.w Sj, Sk* instructions in the C100 Series architecture will result in an unimplemented instruction trap.

CONVERT VECTOR — cvt Vj,Vk

Purpose: To convert the contents of one vector register to a value of different precision or type

Architecture: C200 Series only

Format:



Operation:

```
for (a = 0; a < VL; a++) {
     Vk[a] = Convert(Vj[a]);      /* according to the opcode */
}
```

Exceptions:

(b|h|w|l): Integer Overflow

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
cvtw.s Vj,Vk	7900	ST 0111100100	None	Convert word to single float
cvts.w Vj,Vk	7940	ST 0111100101	RO,SIV	Convert single float to word
cvtd.s Vj,Vk	6000	ST 0110000000	RO,SIV,OV,UN	Convert double float to single float
cvts.d Vj,Vk	6040	ST 0110000001	RO,SIV	Convert single float to double float
cvtw.b Vj,Vk	4000	E0 0100000000	SIV	Convert word to byte
cvtw.h Vj,Vk	4040	E0 0100000001	SIV	Convert word to halfword
cvtb.w Vj,Vk	4080	E0 0100000010	None	Convert byte to word
cvth.w Vj,Vk	40C0	E0 0100000011	None	Convert halfword to word
cvts.l Vj,Vk	4200	E0 0100001000	RO,SIV	Convert single float to longword
cvtd.l Vj,Vk	60C0	ST 0110000011	RO,SIV	Convert double float to longword
cvtl.s Vj,Vk	4280	E0 0100001010	None	Convert longword to single float
cvtl.d Vj,Vk	6080	ST 0110000010	None	Convert longword to double float
cvtl.w Vj,Vk	79C0	ST 0111100110	SIV	Convert longword to word
cvtw.l Vj,Vk	7980	ST 0111100111	None	Convert word to longword
cvtw.d Vj,Vk	42C0	E0 0100001011	None	Convert word to double float
cvtd.w Vj,Vk	4240	E0 0100001001	RO,SIV	Convert double to word float

Description: The converted, possibly sign-extended, contents of the source vector register, Vj, replace the contents of Vk. Conversions to smaller data sizes can cause the underflow and overflow exceptions. Conversions from floating-point representation to integer use truncation (rounding towards 0) as the rounding algorithm.

Notes:

1. Implement halfword-to-byte conversions by using *cvth.w* followed by *cvtw.b*; implement byte-to-halfword conversions with *cvtb.w* followed by *cvtw.h*.
2. Specify unsigned conversions using logical AND instructions.
3. Convert instructions modify only the bits of the specified precision of the destination operand; all other bits are unchanged.

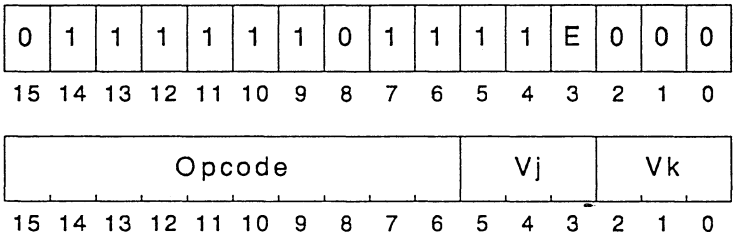
4. If an input operand is a floating-point reserved operand, then the destination is unchanged and both the RO and SIV flags are set to 1.
5. Truncation from float to fix follows the FORTRAN standard: –5.9 is truncated to –5; 5.9 is truncated to 5.
6. Conversion from integer to floating-point types is performed thus:
   a. The fixed-point number is normalized.
   b. In the event that the integer has more bits of significance than the mantissa size of the float output, rounding of the floating-point output value to the closest representable value (round to even in case of a tie) will occur.

CONVERT VECTOR MASKED — cvt.(t|f) Vj,Vk

Purpose: To convert the contents of one vector register to a value of different precision or type under control of the Vector Merge (VM) register

Architecture: **C200 Series only**

Format:



Operation:

```
switch (E) { /* prefix bit<3> */
     case TRUE:  /* .t */
          for (a = 0; a < VL; a++) {
               if (VM<a> == 1) {        /* if VM<a> is TRUE */
                    Vk[a] = Convert(Vj[a]);  /* according to the opcode */
               }
          }   /* end of for loop */
          break;  /* go to end of switch */
     case FALSE: /* .f */
          for (a = 0; a < VL; a++) {
               if (VM<a> == 0) {        /* if VM<a> is FALSE */
                    Vk[a] = Convert(Vj[a]);  /* according to the opcode */
               }
          }   /* end of for loop */
          break;  /* go to end of switch */
} /* end of switch */
```

Exceptions:

(b|h|w|l): Integer Overflow

(s|d): Exponent Underflow
Exponent Overflow
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
cvtw.s.t Vj,Vk	7900	E1 0111100100	None	Convert word to single (VM)
cvtw.s.f Vj,Vk	7900	E0 0111100100	None	Convert word to single (!VM)
cvts.w.t Vj,Vk	7940	E1 0111100101	RO,SIV	Convert single to word (VM)
cvts.w.f Vj,Vk	7940	E0 0111100101	RO,SIV	Convert single to word (!VM)
cvtd.s.t Vj,Vk	6000	E1 0110000000	RO,SIV,OV,UN	Convert double to single (VM)
cvtd.s.f Vj,Vk	6000	E0 0110000000	RO,SIV,OV,UN	Convert double to single (!VM)
cvts.d.t Vj,Vk	6040	E1 0110000001	RO,SIV	Convert single to double (VM)
cvts.d.f Vj,Vk	6040	E0 0110000001	RO,SIV	Convert single to double (!VM)
cvtw.b.t Vj,Vk	4100	E1 0100000100	SIV	Convert word to byte (VM)
cvtw.b.f Vj,Vk	4100	E0 0100000100	SIV	Convert word to byte (!VM)
cvtw.h.t Vj,Vk	4140	E1 0100000101	SIV	Convert word to halfword (VM)
cvtw.h.f Vj,Vk	4140	E0 0100000101	SIV	Convert word to halfword (!VM)
cvtb.w.t Vj,Vk	4180	E1 0100000110	None	Convert byte to word (VM)
cvtb.w.f Vj,Vk	4180	E0 0100000110	None	Convert byte to word (!VM)
cvth.w.t Vj,Vk	41C0	E1 0100000111	None	Convert halfword to word (VM)
cvth.w.f Vj,Vk	41C0	E0 0100000111	None	Convert halfword to word (!VM)
cvts.l.t Vj,Vk	4300	E1 0100001100	RO,SIV	Convert single to longword (VM)
cvts.l.f Vj,Vk	4300	E0 0100001100	RO,SIV	Convert single to longword (!VM)
cvtd.l.t Vj,Vk	60C0	E1 0110000011	RO,SIV	Convert double to longword (VM)

cvtd.l.f Vj,Vk	60C0	E0 0110000011	RO,SIV	Convert double to longword (!VM)
cvtl.s.t Vj,Vk	4380	E1 0100001110	None	Convert longword to single (VM)
cvtl.s.f Vj,Vk	4380	E0 0100001110	None	Convert longword to single (!VM)
cvtl.d.t Vj,Vk	6080	E1 0110000010	None	Convert longword to double (VM)
cvtl.d.f Vj,Vk	6080	E0 0110000010	None	Convert longword to double (!VM)
cvtl.w.t Vj,Vk	79C0	E1 0111100110	SIV	Convert longword to word (VM)
cvtl.w.f Vj,Vk	79C0	E0 0111100110	SIV	Convert longword to word (!VM)
cvtw.l.t Vj,Vk	7980	E1 0111100111	None	Convert word to longword (VM)
cvtw.l.f Vj,Vk	7980	E0 0111100111	None	Convert word to longword (!VM)
cvtw.d.t Vj,Vk	43C0	E1 0100001111	None	Convert word to double (VM)
cvtw.d.f Vj,Vk	43C0	E0 0100001111	None	Convert word to double (!VM)
cvtd.w.t Vj,Vk	4340	E1 0100001101	SIV	Convert double to word (VM)
cvtd.w.f Vj,Vk	4340	E0 0100001101	SIV	Convert double to word (!VM)

Description: The converted, possibly sign-extended, contents of the source vector register, Vj, replace the contents of Vk if the corresponding vector merge (VM) bit is set (clear for *.f*). Conversions to smaller data sizes can cause the underflow or overflow exceptions. Conversions from floating-point representation to integer use truncation (rounding toward 0) as the rounding algorithm.

Notes:

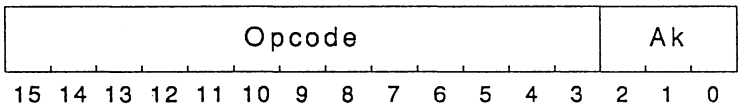
1. Implement halfword-to-byte conversions by using *cvth.w* followed by *cvtw.b*; implement byte-to-halfword conversions with *cvtb.w* followed by *cvtw.h*.
2. Specify unsigned conversions using logical *and* instructions.
3. Convert instructions modify only the bits of the specified precision of the destination operand; all other bits are unchanged.
4. If an input operand is a floating-point reserved operand, then the destination is unchanged and both the RO and SIV flags are set to 1.
5. Truncation from float to fix follows the FORTRAN standard: -5.9 is truncated to -5; 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. In the event that the integer has more bits of significance than the mantissa size of the float output, rounding of the floating-point output value to the closest representable value (round to even in case of a tie) will occur.

EXECUTE DIAGNOSTIC MICROCODE — diag Ak

Purpose: To execute a desired sequence of nonstandard microcode

Architecture: C100 Series, C200 Series

Format:



Operation: `Execute microcode sequence pointed to by the contents of Ak`

Exceptions: Undefined Opcode
Ring Violation (Privileged Instruction [class=8, qualifier=0])

Opcode:

Mnemonic	Hex	Binary	PSW	Description
diag Ak	7DC0	ST 0111110111000	None	Execute nonstandard microcode sequence

Description: The *diag* instruction invokes one of a set of privileged operations specified by address register Ak. An illegal opcode trap occurs if the subopcodes specified by Ak are not supported by this instruction.

The *diag* instructions use the contents of A5 as an address, if one is needed. S0 is the source or destination of data, if one is needed.

Subcodes implemented on the C1, C120, C201, C202, C210, C220, C230, C240 CPUs include:

Ak	Operation
1	Disable logical cache (C1, C120) or data cache (C201, C202, C210, C220, C230, C240)
2	Enable logical cache (C1, C120) or data cache (C201, C202, C210, C220, C230, C240)
7	Pull hard error
11	Store SDR(0–7) at (A5)
12	Enable forced faults a. C1, C120 — Based on scan control b. C201, C202, C210, C220, C230, C240 — On all references
13	Disable forced faults
14	Flush physical cache (C1, C120) or purge data cache (C201, C202, C210, C220, C230, C240)
15	Store halfword at physical address
16	Load halfword from physical address
17	Enable halt from outer rings
18	Disable halt from outer rings
19	Enable diagnostic loop on halt
20	Disable diagnostic loop on halt
642	Read virtual, no ring check
643	Write virtual, no ring check

Subcodes implemented on the C1 and C120 CPUs ***only*** include:

Ak	Operation
5	Store address translation cache at (A5)
6	Load address translation cache from (A5)
640	Read scratch RAM
641	Write scratch RAM

Subcodes implemented on the C201, C202, C210, C220, C230, and C240 CPUs ***only*** include:

Ak	Operation
23	Force hits on data cache (disabled by #1)
25	Store word at physical address
26	Load word from physical address
27	Enable forced faults on all but instruction fetches (accomplished via scan ring control on the C1 and C120 processors)
28	Write communication modified bit covered by virtual communication address in A5 with data in S0<63>
29	Read communication modified bit covered by virtual communication address in A5 into S0<63>
30	*get.l* longword from communication register at address register A5 into scalar register S0 with no protection checking
31	*put.l* longword from scalar register S0 into communication register at address register A5 with no protection checking
32	Clear hardware delta timer
33	Update current ring/CPU/CIR execution timer, and return the new value in scalar register Sk. Update thread timer, return new value in scalar register S1, and clear hardware delta timer.
34	Read the thread timer with no update based on the hardware delta timer. This subcode does not update TTR or CPU execution timer.

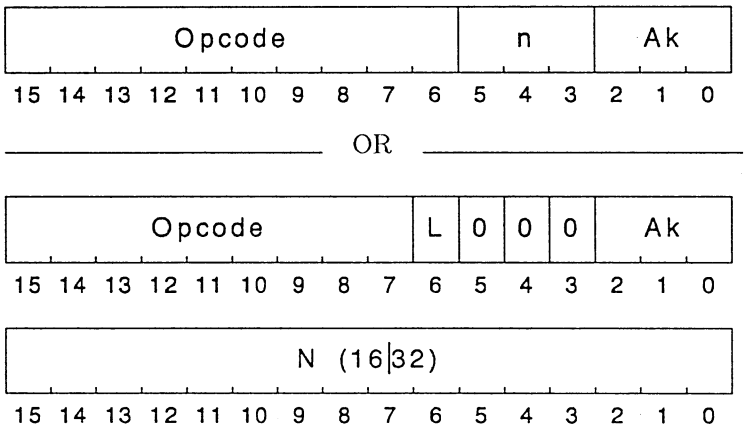
Notes: This instruction is used by diagnostics to reference internal processor registers not accessible to the user program and is specific to each processor implementation.

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

Purpose: To divide the contents of an address register by an immediate

Architecture: C100 Series, C200 Series

Format:



Operation:

```
Ak = Ak / Immediate;
```

Exceptions: (h|w): Integer Overflow
Integer Divide by Zero

Opcode:

Mnemonic	Hex	Binary	PSW	Description
div.h #n,Ak	5E80	ST 0101111010	AIV,ADZ	Divide short immediate address halfword
div.w #n,Ak	5EC0	ST 0101111011	AIV,ADZ	Divide short immediate address word
div.h #N,Ak	1700	ST 000101110	AIV,ADZ	Divide immediate address halfword
div.w #N,Ak	1780	ST 000101111	AIV,ADZ	Divide immediate address word

Description: The quotient of the contents of address register Ak divided by the (sign-extended) immediate operand replaces the contents of Ak.

Notes:

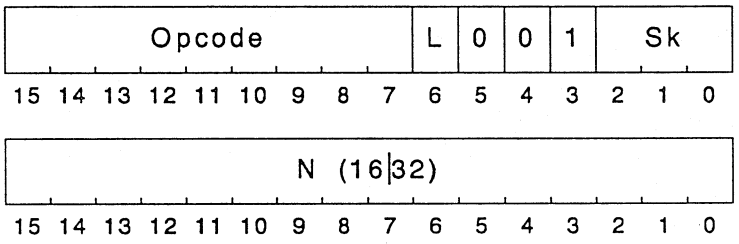
1. Integer overflow occurs if the largest negative number is divided by −1.
2. For divide by 0, the result is the original dividend.
3. Sign extension does not occur for the three bits of the short immediate form.

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

Purpose: To divide the contents of a scalar register by an immediate

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Sk = Sk / Immediate;
```

Exceptions:

(h|w): Integer Overflow
Integer Divide by Zero

(s): Exponent Overflow
Exponent Underflow
Reserved Operand
Floating Divide by Zero

Opcode:

Mnemonic	Hex	Binary	PSW	Description
div.h #N,Sk	1708	ST 000101110	SIV,SDZ	Divide scalar/scalar integer halfword
div.w #N,Sk	1788	ST 000101111	SIV,SDZ	Divide scalar/scalar integer word
div.s #N,Sk	1988	ST 000110011	OV,UN,RO,FDZ	Divide scalar/scalar single float

Description: The quotient of the contents of scalar register Sk divided by the (sign-extended) immediate operand replaces the contents of Sk.

Notes:

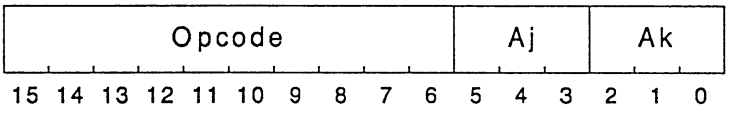
1. Integer overflow occurs if the largest negative number is divided by −1.
2. For divide by zero, the result is the original dividend.

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

Purpose: To divide the contents of one address register by the contents of another address register

Architecture: C100 Series, C200 Series

Format:



Operation:

```
Ak = Ak / Aj;
```

Exceptions: (h|w): Integer Overflow
Address Divide by Zero

Opcode:

Mnemonic	Hex	Binary	PSW	Description
div.h Aj,Ak	5E00	ST 0101111000	AIV,ADZ	Divide address register halfword
div.w Aj,Ak	5E40	ST 0101111001	AIV,ADZ	Divide address register word

Description: The quotient of the contents of address register Ak divided by the contents of Aj replaces the contents of Ak.

Notes:

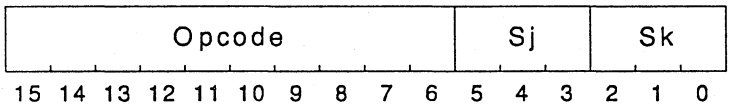
1. Integer overflow occurs if the largest negative number is divided by –1.
2. For address divide by zero, the result is the original dividend.

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

Purpose: To divide the contents of a scalar register by the contents of another scalar register

Architecture: **C100 Series, C200 Series**

Format:



Operation: `Sk = Sk / Sj;`

Exceptions:

(b|h|w|l): Integer Overflow
Integer Divide by Zero

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand
Floating Divide by Zero

Opcode:

Mnemonic	Hex	Binary	PSW	Description
div.b Sj,Sk	5F00	ST 0101111100	SIV,SDZ	Divide scalar/scalar integer byte
div.h Sj,Sk	5F40	ST 0101111101	SIV,SDZ	Divide scalar/scalar integer halfword
div.w Sj,Sk	5F80	ST 0101111110	SIV,SDZ	Divide scalar/scalar integer word
div.l Sj,Sk	5FC0	ST 0101111111	SIV,SDZ	Divide scalar/scalar integer longword
div.s Sj,Sk	5780	ST 0101011110	OV,UN,RO,FDZ	Divide scalar/scalar single float
div.d Sj,Sk	57C0	ST 0101011111	OV,UN,RO,FDZ	Divide scalar/scalar double float

Description: The quotient of the contents of scalar register Sk divided by the contents of Sj replaces the contents of Sk.

Notes:

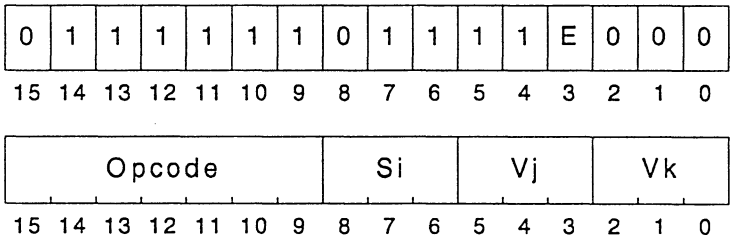
1. Integer overflow occurs if the largest negative number is divided by -1.
2. For divide by zero (floating point), the result is a reserved operand. For integer divide by zero, the original dividend is returned.

REVERSE DIVIDE SCALAR/VECTOR — div.(s|d) Si,Vj,Vk

Purpose: To divide a scalar by each element of a vector

Architecture: C200 Series only

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      Vk[a] = Si / Vj[a];
}
```

Exceptions: (s|d):
- Exponent Overflow
- Exponent Underflow
- Reserved Operand
- Floating Divide by Zero

Opcode:

Mnemonic	Hex	Binary	PSW	Description
div.s Si,Vj,Vk	8400	E0 1000010	OV,UN,RO,FDZ	Divide scalar/vector single float
div.d Si,Vj,Vk	8600	E0 1000011	OV,UN,RO,FDZ	Divide scalar/vector double float

Description: The contents of each of the first VL elements of vector register Vk are replaced by the evaluation of the contents of scalar register Si divided by the contents of the corresponding element of Vj.

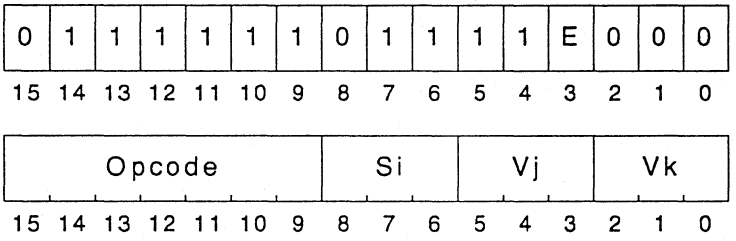
Notes: None

div.(s|d).(t|f) Si,Vj,Vk — REVERSE DIVIDE SCALAR/VECTOR MASKED

Purpose: To divide a scalar by each element of a vector under control of the Vector Merge (VM) register

Architecture: C200 Series only

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
     case TRUE:   /*  t */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 1) {        /* if VM<a> is TRUE */
                  Vk[a] = Si / Vj[a];
              }
          }   /* end of for loop */
          break;   /* go to end of switch */
     case FALSE:  /*  f */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 0) {        /* if VM<a> is FALSE */
                  Vk[a] = Si / Vj[a];
              }
          }   /* end of for loop */
          break;   /* go to end of switch */
} /* end of switch */
```

Exceptions: (s|d): Exponent Overflow
Exponent Underflow
Reserved Operand
Floating Divide by Zero

Opcode:

Mnemonic	Hex	Binary	PSW	Description
div.s.t Si,Vj,Vk	8C00	E1 1000110	OV,UN,RO,FDZ	Divide scalar/vector single (VM)
div.s.f Si,Vj,Vk	8C00	E0 1000110	OV,UN,RO,FDZ	Divide scalar/vector single (!VM)
div.d.t Si,Vj,Vk	8E00	E1 1000111	OV,UN,RO,FDZ	Divide scalar/vector double (VM)
div.d.f Si,Vj,Vk	8E00	E0 1000111	OV,UN,RO,FDZ	Divide scalar/vector double (!VM)

Description: The contents of each of the first VL elements of vector register Vk are replaced by the evaluation of the contents of scalar register Si divided by the contents of the corresponding element of Vj if the corresponding VM bit is set (clear for *.f*).

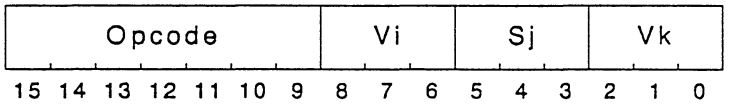
Notes: None

DIVIDE VECTOR/SCALAR — div.(b|h|w|l|s|d) Vi,Sj,Vk

Purpose: To divide the elements of a vector register by the contents of a scalar register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      Vk[a] = Vi[a] / Sj.
}
```

Exceptions:

(b|h|w|l): Integer Overflow
Integer Divide by Zero

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand
Floating Divide by Zero

Opcode:

Mnemonic	Hex	Binary	PSW	Description
div.b Vi,Sj,Vk	F800	ST 1111100	SIV,SDZ	Divide vector/scalar integer byte
div.h Vi,Sj,Vk	FA00	ST 1111101	SIV,SDZ	Divide vector/scalar integer halfword
div.w Vi,Sj,Vk	FC00	ST 1111110	SIV,SDZ	Divide vector/scalar integer word
div.l Vi,Sj,Vk	FE00	ST 1111111	SIV,SDZ	Divide vector/scalar integer longword
div.s Vi,Sj,Vk	9C00	ST 1001110	OV,UN,RO,FDZ	Divide vector/scalar single float
div.d Vi,Sj,Vk	9E00	ST 1001111	OV,UN,RO,FDZ	Divide vector/scalar double float

Description: The quotient of the contents of the corresponding element of Vi divided by the contents of scalar register Sj replaces the contents of each of the first VL elements of vector register Vk.

Notes:

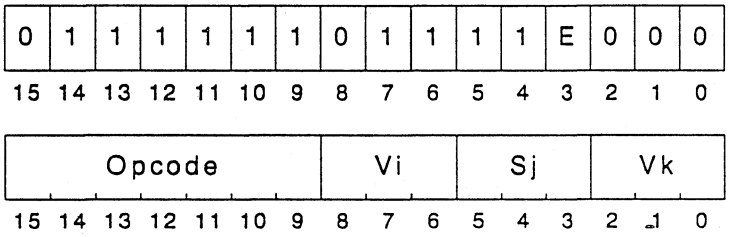
1. Integer overflow occurs if the largest negative number is divided by -1.
2. Integer divide-by-zero returns the original dividend. Floating-point divide-by-zero returns a reserved operand.

div.(b|h|w|l|s|d).(t|f) Vi,Sj,Vk — DIVIDE VECTOR/SCALAR MASKED

Purpose: To divide a vector by a scalar under control of the Vector Merge (VM) register

Architecture: **C200 Series only**

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
      case TRUE:   /*  .t */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 1) {        /* if VM<a> is TRUE */
                  Vk[a] = Vi[a] / Sj;
              }
          }   /* end of for loop */
          break;   /* go to end of switch */
      case FALSE:  /*  .f */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 0) {        /* if VM<a> is FALSE */
                  Vk[a] = Vi[a] / Sj;
              }
          }   /* end of for loop */
          break;   /* go to end of switch */
} /* end of switch */
```

Exceptions:

(b|h|w|l): Integer Overflow
Integer Divide by Zero

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand
Floating Divide by Zero

Opcode:

Mnemonic	Hex	Binary	PSW	Description
div.b.t Vi,Sj,Vk	F800	E1 1111100	SIV,SDZ	Divide vector/scalar byte (VM)
div.b.f Vi,Sj,Vk	F800	E0 1111100	SIV,SDZ	Divide vector/scalar byte (!VM)
div.h.t Vi,Sj,Vk	FA00	E1 1111101	SIV,SDZ	Divide vector/scalar halfword (VM)
div.h.f Vi,Sj,Vk	FA00	E0 1111101	SIV,SDZ	Divide vector/scalar halfword (!VM)
div.w.t Vi,Sj,Vk	FC00	E1 1111110	SIV,SDZ	Divide vector/scalar word (VM)
div.w.f Vi,Sj,Vk	FC00	E0 1111110	SIV,SDZ	Divide vector/scalar word (!VM)
div.l.t Vi,Sj,Vk	FE00	E1 1111111	SIV,SDZ	Divide vector/scalar longword (VM)
div.l.f Vi,Sj,Vk	FE00	E0 1111111	SIV,SDZ	Divide vector/scalar longword (!VM)
div.s.t Vi,Sj,Vk	9C00	E1 1001110	OV,UN,RO,FDZ	Divide vector/scalar single (VM)
div.s.f Vi,Sj,Vk	9C00	E1 1001110	OV,UN,RO,FDZ	Divide vector/scalar single (!VM)
div.d.t Vi,Sj,Vk	9E00	E0 1001111	OV,UN,RO,FDZ	Divide vector/scalar double (VM)
div.d.f Vi,Sj,Vk	9E00	E1 1001111	OV,UN,RO,FDZ	Divide vector/scalar double (!VM)

Description: The contents of each of the first VL elements of vector register Vk are replaced by the quotient of the contents of the corresponding element of Vi divided by the contents of scalar register Sj if the corresponding VM bit is set (clear for *.f*).

Notes:

1. Integer overflow occurs if the largest negative number is divided by –1.

2. Integer divide-by-zero returns the original dividend. Floating-point divide-by-zero returns a reserved operand.

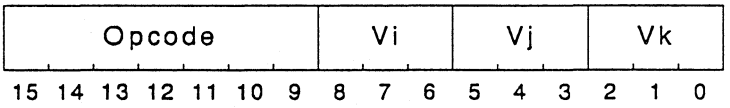
div.(b|h|w|l|s|d) Vi,Vj,Vk

DIVIDE VECTOR/VECTOR

Purpose: To divide the elements of two vector registers

Architecture: C100 Series, C200 Series

Format:



Operation:

```
for (a = 0; a < VL; a++) {
    Vk[a] = Vi[a] / Vj[a];
}
```

Exceptions:

(b|h|w|l): Integer Overflow
Integer Divide by Zero

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand
Floating Divide by Zero

Opcode:

Mnemonic	Hex	Binary	PSW	Description
div.b Vi,Vj,Vk	F000	ST 1111000	SIV,SDZ	Divide vector/vector integer byte
div.h Vi,Vj,Vk	F200	ST 1111001	SIV,SDZ	Divide vector/vector integer halfword
div.w Vi,Vj,Vk	F400	ST 1111010	SIV,SDZ	Divide vector/vector integer word
div.l Vi,Vj,Vk	F600	ST 1111011	SIV,SDZ	Divide vector/vector integer longword
div.s Vi,Vj,Vk	9400	ST 1001010	OV,UN,RO,FDZ	Divide vector/vector single float
div.d Vi,Vj,Vk	9600	ST 1001011	OV,UN,RO,FDZ	Divide vector/vector double float

Description: The quotient of the contents of the corresponding element of Vi divided by the contents of the corresponding element of Vj replaces the contents of each of the first VL elements of vector register Vk.

Notes:

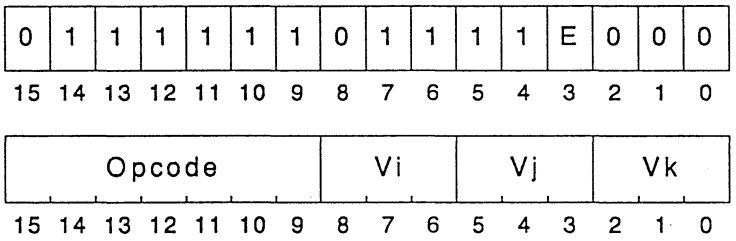
1. Integer overflow occurs if the largest negative number is divided by -1.
2. Integer divide-by-zero returns the original dividend. Floating-point divide-by-zero returns a reserved operand.

DIVIDE VECTOR/VECTOR MASKED — div.(b|h|w|l|s|d).(t|f) Vi,Vj,Vk

Purpose: To divide two vectors under control of the Vector Merge (VM) register

Architecture: C200 Series only

Format:



Operation:

```
switch (E) { /* prefix bit<3> */
     case TRUE:  /* .t */
         for (a = 0; a < VL; a++) {
             if (VM<a> == 1) {        /* if VM<a> is TRUE */
                 Vk[a] = Vi[a] / Vj[a];
             }
         }   /* end of for loop */
         break;  /* go to end of switch */
     case FALSE: /* .f */
         for (a = 0; a < VL; a++) {
             if (VM<a> == 0) {        /* if VM<a> is FALSE */
                 Vk[a] = Vi[a] / Vj[a];
             }
         }   /* end of for loop */
         break;  /* go to end of switch */
} /* end of switch */
```

Exceptions:

(b|h|w|l): Integer Overflow
Integer Divide by Zero

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand
Floating Divide by Zero

Opcode:

Mnemonic	Hex	Binary	PSW	Description
div.b.t Vi,Vj,Vk	F000	E1 1111000	SIV,SDZ	Divide byte vectors (VM)
div.b.f Vi,Vj,Vk	F000	E0 1111000	SIV,SDZ	Divide byte vectors (!VM)
div.h.t Vi,Vj,Vk	F200	E1 1111001	SIV,SDZ	Divide halfword vectors (VM)
div.h.f Vi,Vj,Vk	F200	E0 1111001	SIV,SDZ	Divide halfword vectors (!VM)
div.w.t Vi,Vj,Vk	F400	E1 1111011	SIV,SDZ	Divide word vectors (VM)
div.w.f Vi,Vj,Vk	F400	E0 1111011	SIV,SDZ	Divide word vectors (!VM)
div.l.t Vi,Vj,Vk	F600	E1 1111011	SIV,SDZ	Divide longword vectors (VM)
div.l.f Vi,Vj,Vk	F600	E0 1111011	SIV,SDZ	Divide longword vectors (!VM)
div.s.t Vi,Vj,Vk	9400	E1 1001010	OV,UN,RO,FDZ	Divide single vectors (VM)
div.s.f Vi,Vj,Vk	9400	E0 1001010	OV,UN,RO,FDZ	Divide single vectors (!VM)
div.d.t Vi,Vj,Vk	9600	E1 1001011	OV,UN,RO,FDZ	Divide double vectors (VM)
div.d.f Vi,Vj,Vk	9600	E0 1001011	OV,UN,RO,FDZ	Divide double vectors (!VM)

Description: The contents of each of the first VL elements of vector register Vk are replaced by the quotient of the contents of the corresponding element of Vi divided by the contents of the corresponding element of Vj if the corresponding VM bit is set (clear for *.f*).

Notes:

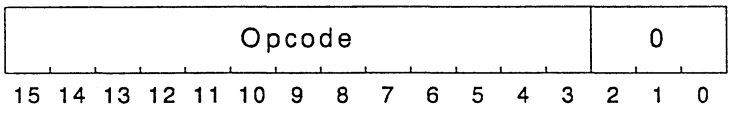
1. Integer overflow occurs if the largest negative number is divided by –1.
2. Integer divide-by-zero returns the original dividend. Floating-point divide-by-zero returns a reserved operand.

DISABLE INTERRUPTS — dsi

Purpose: To disable interrupts

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
C = ION;
ION = 0;  /* dsi - effectively tac(ION) */
```

Exceptions: Ring Violation (Privileged Instruction)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
dsi	7D48	ST 0111110101001	C	Disable interrupts; reset ION to 0

Description: The *dsi* instruction atomically sets the ION flag to 0, and returns the previous state of ION in PSW<C>. Interrupts are disabled when the ION flag is 0.

Notes:

1. The *jmpi.f, jmpi.t, bri.f,* and *bri.t* instructions test the value of ION. Software should avoid using these instructions in the C200 Series architecture due to the asynchronous nature of the interrupt system hardware.
2. To ensure that the *dsi* instruction disables interrupts in the C200 Series architecture, software should use the following code sequence:

   ```
   foo:  dsi
         bra.f foo
   ```

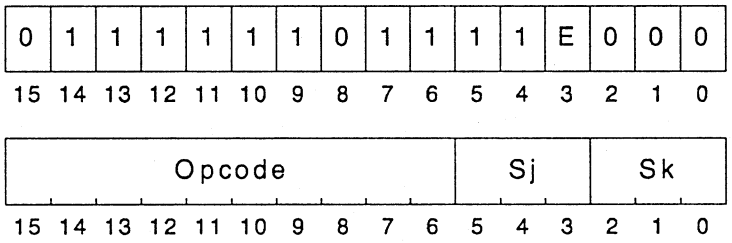
   This requires that software be aware of the state of ION, i.e., software does not try to disable interrupts if it has already disabled them.
3. The ION flag enables or disables all CPUs virtual channel interrupts.
4. This is an atomic instruction.
5. The ION flag is the global semaphore for all other interrupt hardware. It must be reset to zero via *dsi* before any interrupt hardware (local and global enables, interrupt mode, or target CPU) is modified.

enag Sj,Sk — ENABLE GLOBAL INTERRUPT

Purpose: To enable or mask interrupts from being accepted by all CPUs in the complex

Architecture: **C200 Series only**

Format:



Operation:

```
Sk = Global_Interrupt_Enable;
Global_Interrupt_Enable = Sj;
```

Exceptions: Ring Violation (Privileged Instruction)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
enag Sj,Sk	4480	E0 0100010000	None	Enable all global CPU interrupts

Description: The *enag* instruction returns the current value of the global interrupt enable in register Sk and loads into the global interrupt enable with the value contained in Sj. If the bit in the register is a 1, the interrupt channel is enabled. If it is a 0, the channel is masked out.

Notes:

1. Interrupts must be disabled successfully before the execution of this instruction.
2. Each bit in the global interrupt enable corresponds to a virtual channel to which the CPU responds. Refer to the *CONVEX Architecture Reference*, "Exceptions and Interrupts" chapter, for details on the interrupt hardware.
3. The global interrupt enable is used to control interrupt delivery for all the entire Complex.

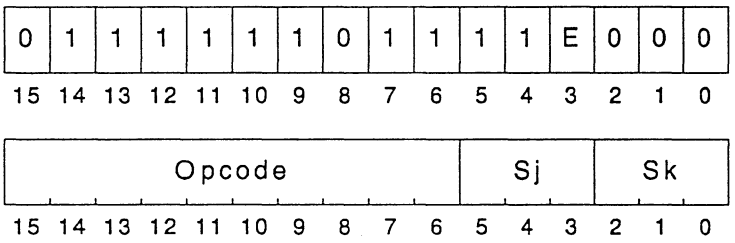
ENABLE LOCAL INTERRUPT

enal Sj,Sk

Purpose: To enable or mask interrupts from being accepted by the local CPU

Architecture: C200 Series only

Format:



Operation:

```
Sk = Local_Interrupt_Enable;
Local_Interrupt_Enable = Sj;
```

Exceptions: Ring Violation (Privileged Instruction)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
enal Sj,Sk	4400	E0 0100010000	None	Enable local CPU interrupt

Description: The *enal* instruction returns the current value of the local CPU interrupt enable in register Sk and loads into the local CPU interrupt enable the value contained in Sj. If the bit in the register is a 1, the interrupt channel is enabled. If it is 0, the channel is masked out.

Notes:

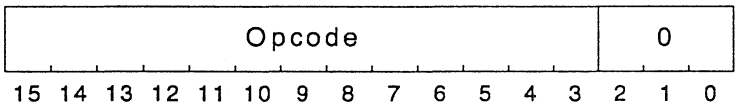
1. Interrupts must be disabled successfully before the execution of this instruction.
2. Each bit in the local interrupt enable corresponds to a virtual channel to which the CPU responds. Refer to the *CONVEX Architecture Reference*, "Exceptions and Interrupts" chapter, for details concerning the interrupt hardware.
3. The local interrupt enable is used to control interrupt delivery for the local CPU.

eni — ENABLE INTERRUPTS

Purpose: To enable interrupts

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
C = ION;
ION = 1;  /* eni - effectively ''tas(ION)'' */
```

Exceptions: Ring Violation (Privileged Instruction)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
eni	7D40	ST 0111110101000	C	Enable interrupts; set ION to 1

Description: The *eni* instruction sets the ION flag to 1 and returns the previous state of ION in the PSW<C> flag. Interrupts are enabled when the ION flag is 1.

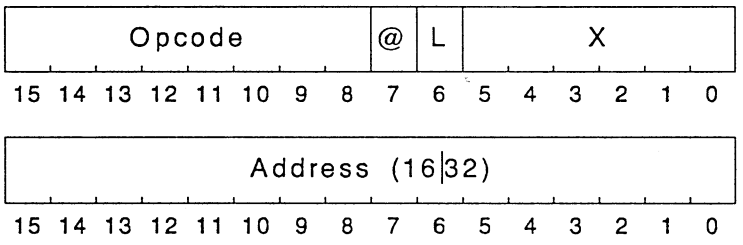
Notes:

1. The *jmpi.f, jmpi.t, bri.f,* and *bri.t* instructions test the value of ION. (Software should avoid using these instructions in the C200 Series architecture due to the asynchronous nature of the interrupt system hardware.)
2. The CPU always executes the next sequential instruction, even though interrupts may be pending when the *eni* instruction is encountered.
3. The ION flag enables or disables all CPUs virtual channel interrupts.
4. The ION flag is the global semaphore for all other interrupt hardware. It must be reset to zero via *dsi* before any interrupt hardware (local and global enables, interrupt mode, or target CPU) is modified.

Purpose: To exit as a result of an error

Architecture: C100 Series, C200 Series

Format:



Operation:

```
psw[FRL] = 01;          /* push a long frame */
push(S1); push(S2); push(S3); push(S4); push(S5); push(S6); push(S7);
push(A1); push(A2); push(A3); push(A4); push(A5); push(A6); push(A7);
push(PSW);
push(next_instruction_address);
A5 = 0;
(Enter system exception handler);
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
exit	0000	ST 00000000	None	Error exit instruction

Description: The error exit instruction causes a system exception of code 0. Refer to the *CONVEX Architecture Reference*, "Exceptions and Interrupts" chapter, for a description of system exceptions.

Notes:

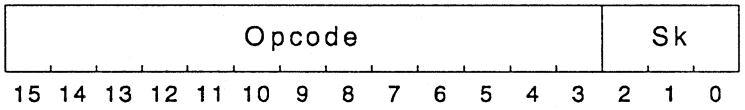
1. The error exit instruction opcode is 0; it can detect transfers to data.
2. In the error exit instruction, @ bit <7> is not interpreted. Indirection never occurs for the error exit instruction.
3. The X field is ignored.

exp.(s|d) Sk

Purpose: To compute e^{Sk}, when e is the base of natural logarithms, i.e., $e = 2.718...$

Architecture: **C200 Series only**

Format:



Operation:

```
Sk = exp(Sk);
```

Exceptions: (s|d): Reserved Operand
Floating Instrinsic Error

Opcode:

Mnemonic	Hex	Binary	PSW	Description
exp.s Sk	7CB0	ST 0111110000110	RO,FIN,IEC	Exponent of a single float
exp.d Sk	7CB8	ST 0111110000111	RO,FIN,IEC	Exponent of a double float

Description: The value of e to the power of the contents of Sk replaces the contents of Sk.

Notes:

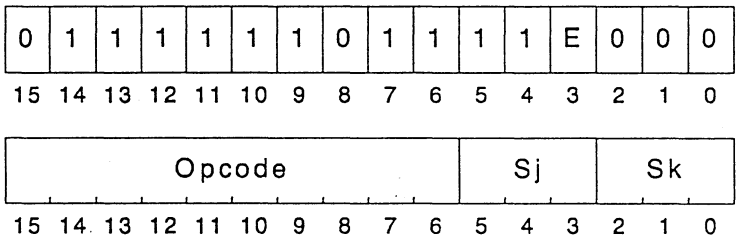
1. Intrinsic traps go through the same trap handler as other arithmetic traps (RO, FDZ, UN, etc.). If PSW <FUE> and/or PSW <FE> are set and and intrinsic traps are not (INE clear), these bits must be examined to determine the type of the current trap.
2. When PSW<FIN> is set, the PSW<IEC> bits contain a code that specifies the type of error encountered by the intrinsic instruction. Refer to the *CONVEX Architecture Reference*, "Register Set" chapter, "Program Status Word — C200 Series" section, for more information on the PSW<IEC> error codes and arithmetic trap conditions.

INTEGERIZE FLOATING SCALAR — frint.(s|d) Sj,Sk

Purpose: To integerize (remove the fractional part from) the contents of a scalar register, with the result being the same type (*.s* or *.d*) as the input.

Architecture: C200 Series only

Format:



Operation:

```
switch (FLOAT_LENGTH) {
      case SINGLE: /*  .s */
          Sk = aint(Sj);   /* truncate fraction from single float */
          break;
      case DOUBLE: /*  .d */
          Sk = dint(Sj);   /* truncate fraction from double float */
          break;
} /* end of switch */
```

Exceptions: (s|d): Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
frint.s Sj,Sk	4700	E0 0100011100	RO	Integerize float single scalar
frint.d Sj,Sk	4740	E0 0100011101	RO	Integerize float double scalar

Description: The integer portion of Sj replaces Sk.

Notes:

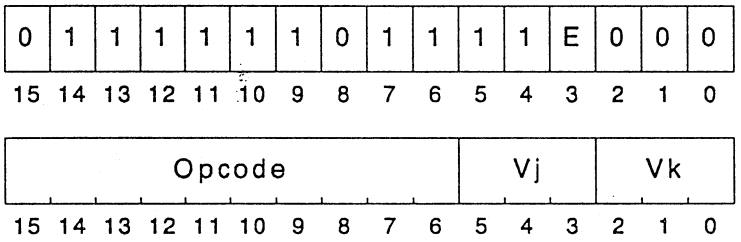
1. If the input is less than 1.0 and greater than −1.0, the result is 0.0.
2. If the input has no fractional part, the output is equal to the input.
3. Use the *cvt* instruction to convert from floating-to-integer type.

frint.(s|d) Vj,Vk — INTEGERIZE FLOATING VECTOR

Purpose: To integerize (remove the fractional part from) the contents of a floating point vector, with the result is being the same type (*.s* or *.d*) as the input.

Architecture: **C200 Series only**

Format:



Operation:

```
switch (FLOAT_LENGTH) {
     case SINGLE: /* .s */
          Vk[a] = aint(Vj[a]); /* truncate fraction from single */
          break;
     case DOUBLE: /* .d */
          Vk[a] = dint(Vj[a]); /*  truncate fraction from double */
          break;
} /* end of switch */
```

Exceptions: (s|d): Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
frint.s Vj,Vk	5880	E0 0101100010	RO	Integerize float single vector
frint.d Vj,Vk	58C0	E0 0101100011	RO	Integerize float double vector

Description: The integer portion of the contents of each of the first VL elements of vector register Vj replaces the corresponding elements of Vk.

Notes:

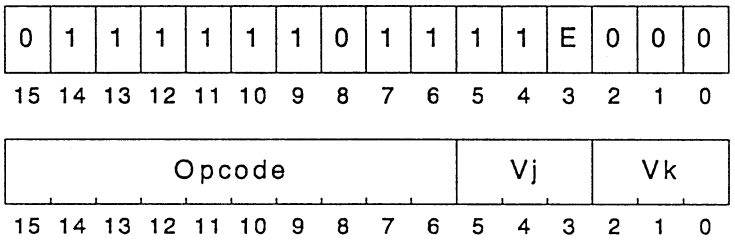
1. If the input is less than 1.0 and greater than –1.0, the result is 0.0.
2. If the input has no fractional part, the output is equal to the input.
3. Use the *cvt* instruction to convert from floating-to-integer type.

INTEGERIZE FLOATING VECTOR MASKED — frint.(s|d).(t|f) Vj,Vk

Purpose: To integerize (remove the fractional part from) the contents of a floating point vector under control of the Vector Merge (VM) register, with the result being the same type (*.s* or *.d*) as the input

Architecture: C200 Series only

Format:



Operation:

```
switch (E) { /* prefix bit<3> */
      case TRUE:  /* .t */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 1) {        /* if VM<a> is TRUE */
                  switch (FLOAT_LENGTH) {
                      case SINGLE: /* .s */
                          Vk[a] = aint(Vj[a]);
                          break;
                      case DOUBLE: /* .d */
                          Vk[a] = dint(Vj[a]);
                          break;
                  } /* end of switch */
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
      case FALSE: /* .f */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 0) {        /* if VM<a> is FALSE */
                  switch (FLOAT_LENGTH) {
                      case SINGLE: /* .s */
                          Vk[a] = aint(Vj[a]);
                          break;
                      case DOUBLE: /* .d */
                          Vk[a] = dint(Vj[a]);
                          break;
                  } /* end of switch */
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
} /* end of switch */
```

Exceptions: (s|d): Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
frint.s.t Vj,Vk	5980	E1 0101100110	RO	Integerize single vector (VM)
frint.s.f Vj,Vk	5980	E0 0101100110	RO	Integerize single vector (!VM)
frint.d.t Vj,Vk	59C0	E1 0101100111	RO	Integerize double vector (VM)
frint.d.f Vj,Vk	59C0	E0 0101100111	RO	Integerize double vector (!VM)

Description: The integer portion of the contents of each of the first VL elements of vector register Vj replaces the corresponding elements of Vk if the corresponding VM bit is set (clear for *.f*).

Notes:

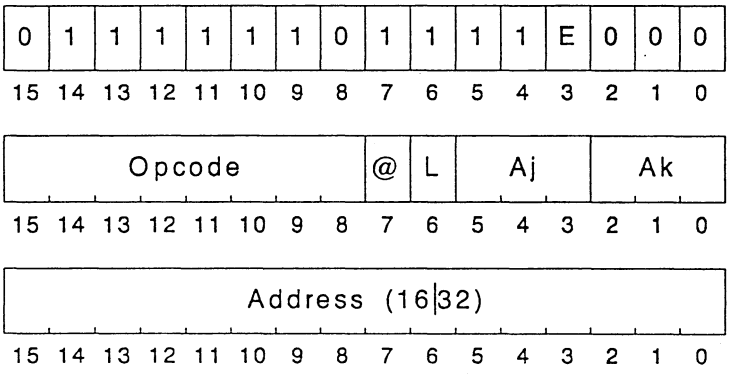
1. If the input is less than 1.0 and greater than –1.0, the result is 0.0.
2. If the input has no fractional part, the output is equal to the input.
3. Use the *cvt* instruction to convert from floating-to-integer type.

GET COMMUNICATION/ADDRESS — get.w <Ceffa>,Ak

Purpose: To get the contents of a communication register into an address register

Architecture: C200 Series only

Format:



Operation:

```
Ak = c(Ceffa)<31..0>;
```

Exceptions: Ring Violation (Invalid Communication Register Address)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
get.w <Ceffa>,Ak	2A00	E0 0010101000	CAT	Get communication/address

Description: A word of data, bits <31..0>, are moved from c(Ceffa) to Ak. The lock bit, L(Ceffa), is not modified.

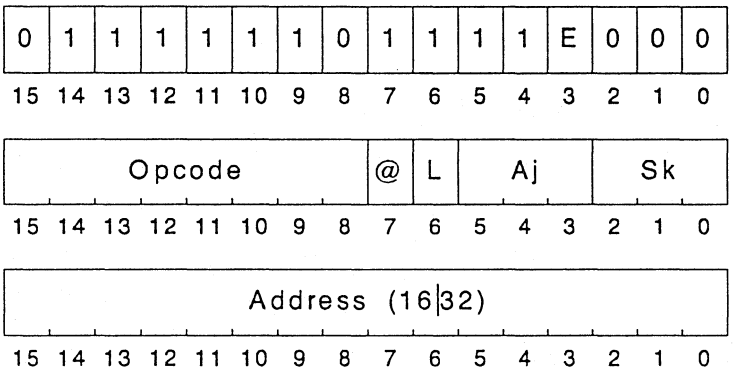
Notes: None

get.l <Ceffa>,Sk

Purpose: To get the contents of a communication register into a scalar register

Architecture: C200 Series only

Format:



Operation:

```
Sk = c(Ceffa);
```

Exceptions: Ring Violation (Invalid Communication Register Address)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
get.l <Ceffa>,Sk	3200	E0 0011001000	CAT	Get communication/scalar

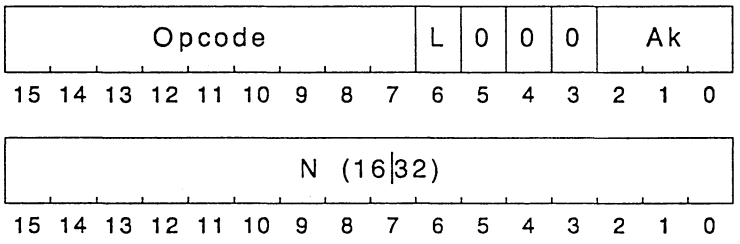
Description: A longword of data is moved from c(Ceffa) to Sk. The lock bit, L(Ceffa), is not modified.

Notes: None

Purpose: To halt the Central Processing Unit (CPU)

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Ak = Immediate;
(halt CPU);
```

Exceptions: Ring Violation (Privileged Instruction)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
halt #N,Ak	1000	ST 000100000	None	Halt the CPU

Description: The (sign-extended) immediate field replaces the contents of address register Ak. The CPU is then halted. Further action is machine-implementation dependent.

Notes: This instruction is typically used for diagnostic and debugging purposes.

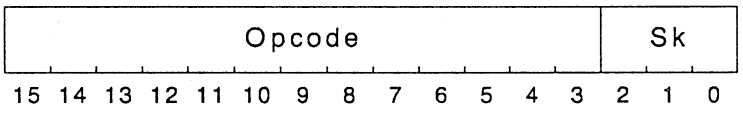
idle Sk

IDLE THE CPU

Purpose: Attempt to accept a posted fork in the specified CIR. If one is not posted, idle the current CPU without deallocating the current thread.

Architecture: **C200 Series only**

Format:



Operation:

```
CIR = Sk;
if (rcv(threadcount)) {
      if (rcv(forkposted)) {
          switch (fork.type) {
              case PFORKED:
                  (take fork in this CIR);
                  ulk(forklck);        /* clear fork */
                  break;
              case SPAWNED:
                  (take fork in this CIR);
                  lck(forkposted); /* repost fork */
                  break;
              case STOPPED:
                  lck(forkposted); /* repost fork */
                  lck(threadcount);    /* ignore if (rcv()) */
                  (idle the CPU);
                  break;
          }   /* end of switch /*
      } else {/* no fork in CIR Sk */
          snd(threadcount);
          (idle the CPU);
      }   /* end if rcv(forkposted) */
} else {
      (idle the CPU);
}     /* end if rcv(threadcount) */
```

Exceptions: Ring Violation (Privileged Instruction)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
idle Sk	7C58	ST 0111110001	None	Idle the CPU

Description: The idle instruction attempts to find a posted fork in the CIR specified in Sk; if a posted fork exists, a new thread is allocated in CIR Sk and the fork is taken directly. In other words, the instruction stream branches with a switch in CIR and TID. The CPU need not be idled. If no fork exists in the specified CIR, the CPU is idled and forks are accepted from other CIRs. The specific actions of an idle CPU are described in the *CONVEX Architecture Reference*, "Multiprocessor Management" chapter.

Notes:

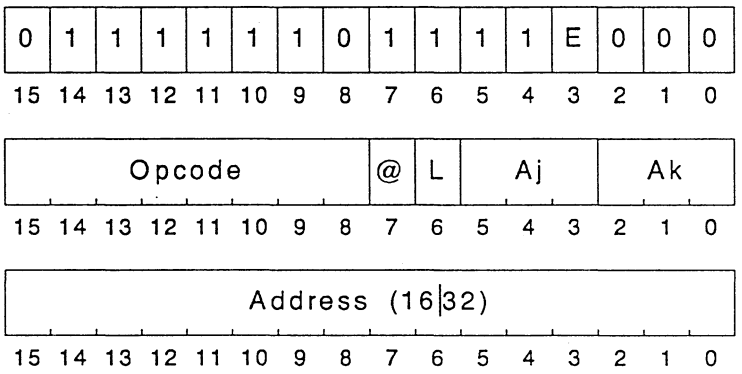
1. This instruction may be traced, i.e., if the TTC or TIT bits in the PSW are set and the CPU attempts to go idle, an instruction trace trap will occur.
2. The current thread before the execution of the idle instruction is not deallocated in the current CIR before any CIR or TID change; software must deallocate this where necessary. The current thread is not deallocated in order to always keep the interrupt thread allocated.

INCREMENT COMMUNICATION/ADDRESS — inc.w <Ceffa>,Ak

Purpose: To increment a communication register by an address register and return the new value

Architecture: C200 Series only

Format:



Operation:

```
if (C = rcv (Ceffa,temp)) {   /* PSW<C> = 1 if rcv() succeeds */
     Ak = temp + Ak;          /* temp was returned by the rcv() */
     snd (Ak,Ceffa);
}
```

Exceptions: Ring Violation (Invalid Communication Register Address)
Deadlock Exception

Opcode:

Mnemonic	Hex	Binary	PSW	Description
inc.w <Ceffa>,Ak	2D00	E0 0010110100	C,AIV,CAT	Increment communication/address

Description: The *inc* instruction atomically increments a communication register by Ak and returns the value in Ak. Status as to the success or failure of the *inc* is returned in the address carry. The hardware lock bit is considered a "valid" bit, i.e., if the bit is one there is valid data to be incremented.

Notes:

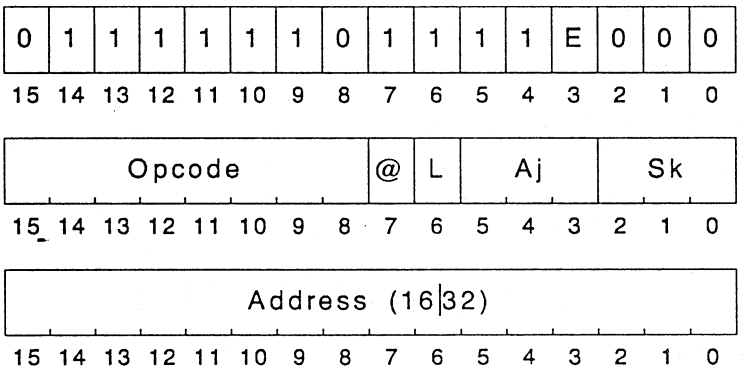
1. Address Carry (C) is affected as described by the preceding operation pseudocode, i.e., C is the status, not the carry-out, of the *inc* operation. Ak is unchanged if the *inc* operation fails (C = 0).
2. This instruction is atomic.
3. The memory dual of this instruction is *incr.w <effa>,Ak*.

inc.l <Ceffa>,Sk — INCREMENT COMMUNICATION/SCALAR

Purpose: To increment a communication register by a scalar register and return the new value

Architecture: **C200 Series only**

Format:



Operation:

```
if (SC = rcv (Ceffa,temp)) {  /* PSW<SC> = 1 if rcv() succeeds */
     Sk = temp + Sk;      /* temp was returned by the rcv() */
     snd (Sk,Ceffa);
}
```

Exceptions: Ring Violation (Invalid Communication Register Address)
Deadlock Exception

Opcode:

Mnemonic	Hex	Binary	PSW	Description
inc.l <Ceffa>,Sk	2D00	E0 0010110100	SC,SIV,CAT	Increment communication/scalar

Description: The *inc* instruction atomically increments a communication register by Sk and returns the value in Sk. Status as to the success or failure of the *inc* is returned in the scalar carry. The hardware lock bit is considered a "valid" bit, i.e., if the bit is 1, there is valid data to be incremented.

Notes:

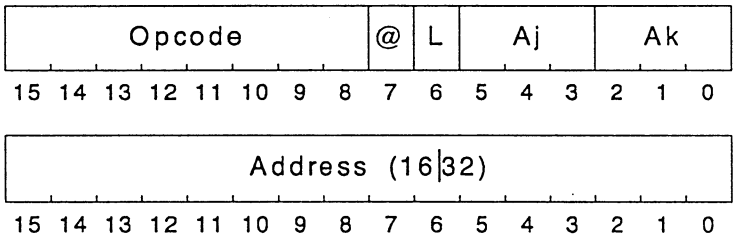
1. Scalar Carry (SC) is affected as described by the preceding operation pseudocode, i.e., SC is the status, not the carry-out, of the *inc* operation. The scalar register Sk is unchanged if the *inc* operation fails (SC = 0).
2. This instruction is atomic.
3. The memory dual of this instruction is *incr.l <effa>,Sk*.

INCREMENT RESOURCE/ADDRESS — incr.w <effa>,Ak

Purpose: To increment the data field of a resource structure by an address register and return the new value

Architecture: C200 Series only

Format:



Operation:

```
msync;
if (tas(effa.lock)) {
      if (effa.valid == 0xff) {
          c(effa.data) += Ak;
          Ak = c(effa.data);
          C = 1;
      } else {
          C = 0;  /* fail - no valid data (uninitialized) */
      }       /* end if effa.valid */
      msync;
      tac(effa.lock);
} else {
      C = 0;       /* fail - structure in transition */
}             /* end if tas() */
```

Exceptions: Deadlock Exception

Opcode:

Mnemonic	Hex	Binary	PSW	Description
incr.w <effa>,Ak	2B00	ST 0010101100	C	Increment resource structure data

Description: The *incr* instruction atomically increments the data word of a resource structure, if the resource structure data word is valid, and returns the value in Ak. Status as to the success or failure of the *incr* is returned in the address carry.

Notes:

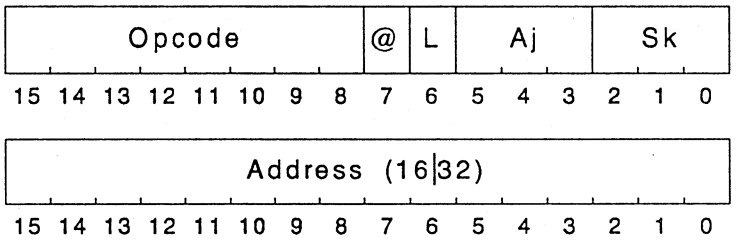
1. Address Carry (C) is affected as described by the preceding operation pseudocode, i.e., C is the status, not the carry-out, of the *incr* operation. Ak is unchanged if the *incr* operation fails (C = 0).
2. This instruction is atomic.
3. The communication register dual of this instruction is *inc.w <Ceffa>, Ak.*

incr.l <effa>,Sk — INCREMENT RESOURCE/SCALAR

Purpose: To increment the data field of a longword resource structure by a scalar register and return the new value

Architecture: **C200 Series only**

Format:



Operation:

```
msync;
if (tas(effa.lock)) {
     if (effa.valid == 0xFF) {
          c(effa.data) += Sk;
          Sk = c(effa.data);
          SC = 1;
     } else {
          SC = 0;      /* fail - no valid data (uninitialized) */
     }           /* end if effa.valid */
     msync;
     tac(effa.lock);
} else {
     SC = 0;          /* fail - structure in transition */
}               /* end if tas() */
```

Exceptions: Deadlock Exception

Opcode:

Mnemonic	Hex	Binary	PSW	Description
incr.l <effa>,Sk	2F00	ST 0010111100	SC	Increment long resource structure

Description: The *incr.l* instruction atomically increments the data longword of a resource structure, if the resource structure data is valid, and returns the value in Sk. Status as to the success or failure of the *incr* is returned in the Scalar Carry (SC).

Notes:

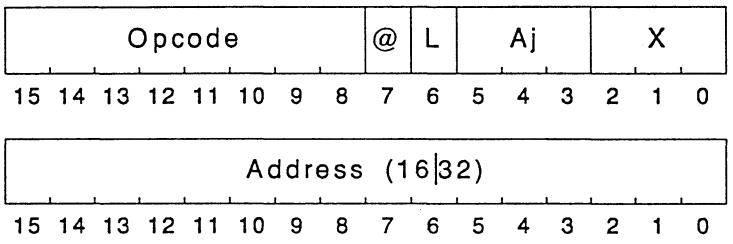
1. Scalar Carry (SC) is affected as described by the preceding operation pseudocode, i.e., SC is the status, not the carry-out, of the *incr* operation. The scalar register Sk is unchanged if the *incr* operation fails (SC = 0).
2. This instruction is atomic.
3. The communication register dual of this instruction is *inc.l <Ceffa>, Sk*.

JUMP ON PSW BIT — jmp <effa>

Purpose: To jump to an address conditionally or unconditionally

Architecture: C100 Series, C200 Series[2]

Format:



Operation:

```
if (Opcode_Cond) {
     PC = Effective_Address;
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
jmp <effa>	0100	ST 00000001	None	Jump always
jmpi.f <effa>	0200	ST 00000010	None	Jump on ION false
jmpi.t <effa>	0300	ST 00000011	None	Jump on ION true
jmpa.f <effa>	0400	ST 00000100	None	Jump on address carry false
jmpa.t <effa>	0500	ST 00000101	None	Jump on address carry true
jmps.f <effa>	0600	ST 00000110	None	Jump on scalar carry false
jmps.t <effa>	0700	ST 00000111	None	Jump on scalar carry true

Description: If the specified condition is asserted, the instruction's effective address replaces the contents of the Program Counter (PC).

Notes:

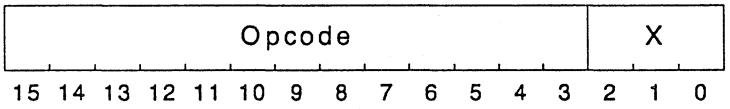
1. All jumps are restricted to be within the current ring. That is, if the current ring is 4, the most significant bit of the effective address is ignored. Otherwise, the most significant 3 bits of the effective address are ignored.
2. The *jmpi.(t | f)* instructions should not be used in the C200 Series architecture. The ION flag is asynchronous to the processor. Refer to the Notes sections of the *dsi* and *eni* instruction definitions.
3. The X field is ignored.

join — JOIN ALL THREADS

Purpose: To force all threads created by a process to join at a single execution point

Architecture: **C200 Series only**

Format:



Operation:

```
msync;
fork.type = STOPPED;
if (!rcv(threadcount)) {      /* loop until rcv() succeeds,*/
     (restart the instruction);   /* ...accepting interrupts */
}
if (threadcount == 1) {
     cfork;
     snd(threadcount);
} else {
     (enter the CPU idle loop);
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
join	7CA0	ST 0111110010	None	Join all threads

Description: The *join* instruction marks any outstanding fork event "STOPPED" in the hardware communication registers in the current CIR. This signals other idle CPUs that the process is joining and the fork should not be taken. If the current thread is not the last thread (this thread is not the last to reach the *join* in the instruction stream), the CPU is relinquished. If the current thread is the last thread (i.e., all other threads have already reached the *join*), execution continues sequentially after the *join*. The actions of an idle CPU are described in the *CONVEX Architecture Reference*, "Multiprocessor Management" chapter.

Notes:

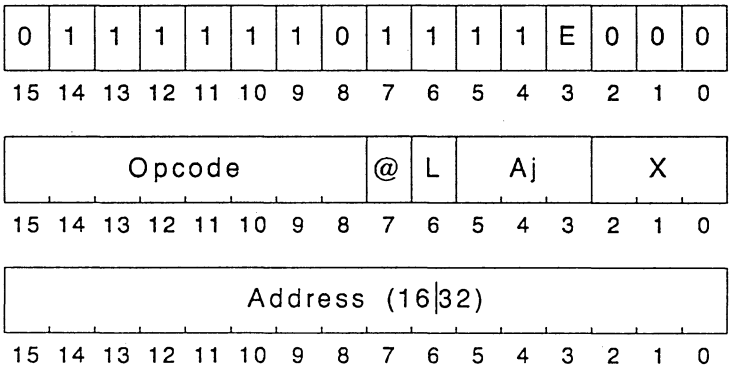
1. Programs that mix usage of *join* and *wfork* must properly synchronize execution of these instructions to ensure *wfork* last thread termination deadlocks do not occur.
2. This instruction may be traced, i.e., if the *trace thread concurrency* (TTC) bit or *thread initialization trap* (TIT) bit in the PSW is set, and the CPU becomes idle and attempts to take another fork, an instruction trace trap will occur.
3. The X field is unused.

LOCK COMMUNICATION REGISTER — lck <Ceffa>

Purpose: To lock a communication register

Architecture: C200 Series only

Format:



Operation:

```
if (L(Ceffa) == 0) {
     L(Ceffa) = 1;
     C = 1;
} else {
     C = 0;
}
```

Exceptions: Ring Violation (Invalid Communication Register Address)
Deadlock Exception

Opcode:

Mnemonic	Hex	Binary	PSW	Description
lck <Ceffa>	0200	E0 0010000000	C,CAT	Lock communication register

Description: If the lock bit for the addressed communication register is set, the communication registers are not modified and "fail" status (C = 0) is returned. If the lock bit for the addressed communication register is cleared, the lock bit is set, and "success" status (C = 1) is returned.

Notes:

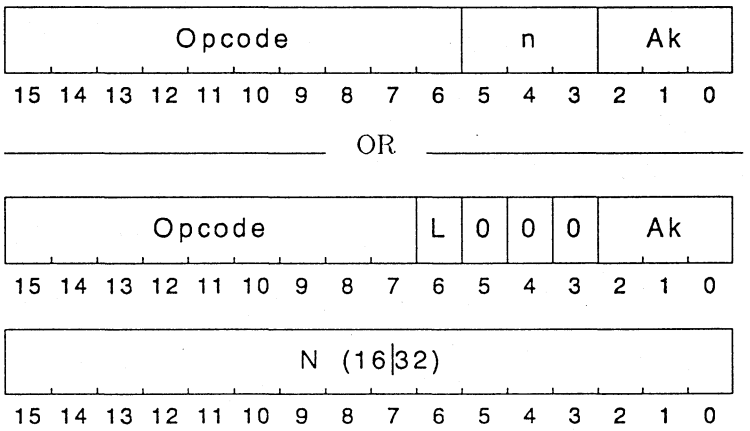
1. Address Carry (C) is affected as described by the preceding operation pseudocode.
2. This is an atomic instruction.
3. The X field is unused.

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

Purpose: To load an address register with an immediate operand

Architecture: C100 Series, C200 Series

Format:



Operation:

```
Ak = Immediate;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
ld.h #N,Ak	1100	ST 000100010	None	Load halfword immediate into Ak
ld.w #N,Ak	1180	ST 000100011	None	Load immediate into Ak
ld.h #n,Ak	4480	ST 0100010010	None	Load short immediate into Ak
ld.w #n,Ak	44C0	ST 0100010011	None	Load short immediate into Ak

Description: The (sign-extended) immediate field replaces the contents of address register Ak.

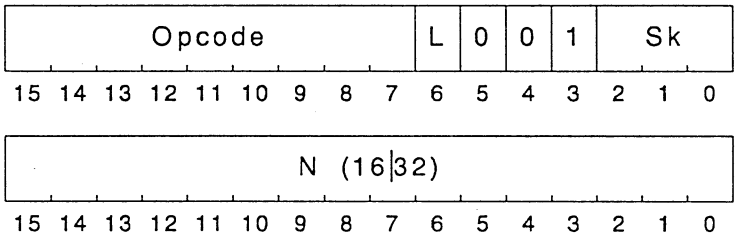
Notes: Sign extension does not occur for the 3 bits of the short immediate form.

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

Purpose: To load an immediate value into a scalar register

Architecture: C100 Series, C200 Series

Format:



Operation:

```
switch (IMMEDIATE_TYPE) {
      case TYPE_D:     /* ld.d */
          Sk<63..32> = Immediate;
          Sk<31..0> = 0;
          break;
      case TYPE_L: /* ld.l */
          Sk<63..32> = Extended_Sign_of_Immediate;
          Sk<31..0> = Immediate;
          break;
      case TYPE_W: /* ld.w  */
      case TYPE_S: /* ld.s  */
      case TYPE_DL:    /* ld.dl */
      case TYPE_LL:    /* ld.ll */
          Sk<31..0> = Immediate;
          break;
      case TYPE_U: /* ld.u  */
      case TYPE_DL:    /* ld.du */
      case TYPE_LL:    /* ld.lu */
          Sk<63..32> = Immediate;
          break;
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
ld.d #N,Sk	1008	ST 000100000	None	Load double float immediate upper 32 bits
ld.l #N,Sk	1108	ST 000100010	None	Load 32-bit immediate sign-extended to 64 bits
ld.w #N,Sk	1188	ST 000100011	None	Load a 32-bit immediate
ld.s #N,Sk	1188	ST 000100011	None	Load a single float immediate
ld.u #N,Sk	1088	ST 000100001	None	Load immediate, upper half
ld.du #N,Sk	1088	ST 000100001	None	Load 64-bit floating immediate, upper half
ld.dl #N,Sk	1188	ST 000100011	None	Load 64-bit floating immediate, lower half
ld.lu #N,Sk	1088	ST 000100001	None	Load 64-bit integer immediate, upper half
ld.ll #N,Sk	1188	ST 000100011	None	Load 64-bit integer immediate, lower half

Description: The (sign-extended) immediate field replaces the contents of scalar register Sk.

Notes:

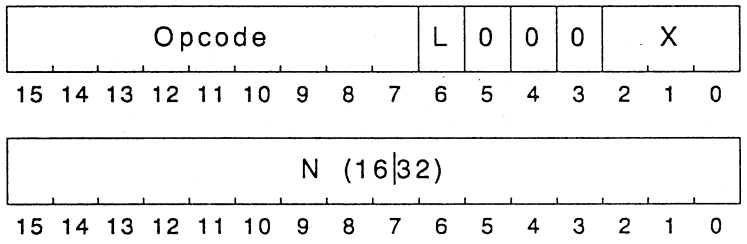
1. The *ld.d #N,Sk* will clear the lower half of Sk.
2. Load a full 64-bit value by performing a *ld.d #N,Sk* followed by a *ld.w #N,Sk*.
3. The forms using *.du*, *.lu*, *.dl*, and *.ll* are mnemonic for "double upper," "long upper," "double lower," and "long lower," respectively. Use them for loading 64-bit values.
4. The *ld.s #N,Sk* is syntactically synonomous to *ld.w #N,Sk*, as shown above.

ld.w #N,VL — LOAD VL/IMMEDIATE

Purpose: To load the Vector Length (VL) register with an immediate

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
if (Immediate >= 128) {
     VL = 128;
} else {
     if (Immediate <= 0) {
          VL = 0;
     } else {
          VL = Immediate;
     }
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
ld.w #N,VL	1800	ST 0001100000	None	Load VL with an immediate

Description: The (sign-extended) immediate field, range limited to [0..128], replaces the contents of the VL register.

Notes: The X field is ignored.

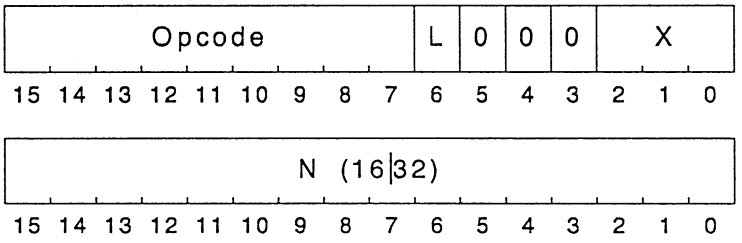
LOAD VS/IMMEDIATE

ld.w #N,VS

Purpose: To load the Vector Stride (VS) register from an immediate

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
VS = Immediate;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
ld.w #N,VS	1880	ST 0001100001	None	Load VS from an immediate

Description: The (sign-extended) immediate field replaces the contents of the 32-bit VS register.

Notes: The X field is ignored.

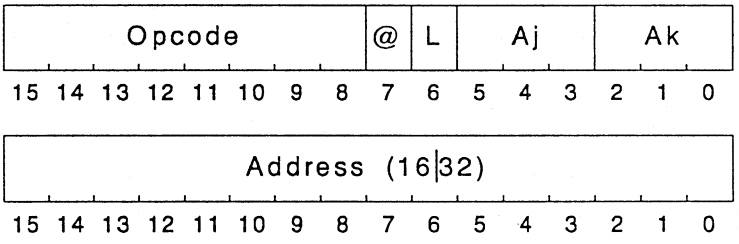
ld.(b|h|w) <effa>,Ak

LOAD ADDRESS REGISTER

Purpose: To load memory contents into an address register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Ak = c(Effective_Address);
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
ld.b <effa>,Ak	2800	ST 00101000	None	Load address register byte
ld.h <effa>,Ak	2900	ST 00101001	None	Load address register halfword
ld.w <effa>,Ak	2A00	ST 00101010	None	Load address register word

Description: The operand referenced by the effective address replaces the contents of the address register Ak.

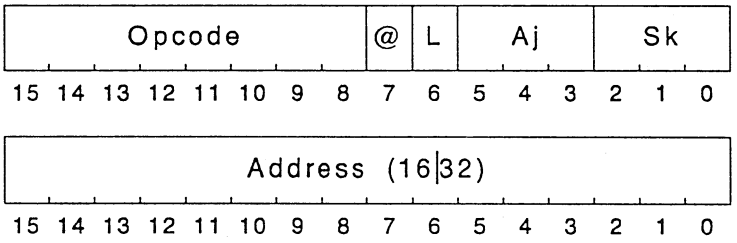
Notes: Byte operands replace only bits <7..0> of the address register Ak. Halfword operands replace bits <15..0> of the address register Ak.

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

Purpose: To load memory contents into a scalar accumulator

Architecture: C100 Series, C200 Series

Format:



Operation:

```
Sk = c(Effective_Address);
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
ld.b <effa>,Sk	3000	ST 00110000	None	Load scalar byte
ld.h <effa>,Sk	3100	ST 00110001	None	Load scalar halfword
ld.w <effa>,Sk	3200	ST 00110010	None	Load scalar word
ld.l <effa>,Sk	3300	ST 00110011	None	Load scalar longword
ld.s <effa>,Sk	3200	ST 00110010	None	Load scalar single float
ld.d <effa>,Sk	3300	ST 00110011	None	Load scalar double float

Description: The operand referenced by the effective address replaces the contents of scalar register Sk.

Notes:

1. Byte operands replace only bits <7..0> of the specified S register. Halfword operands replace only bits <15..0> of the specified S register. Word operands replace only bits <31..0> of the specified S register.

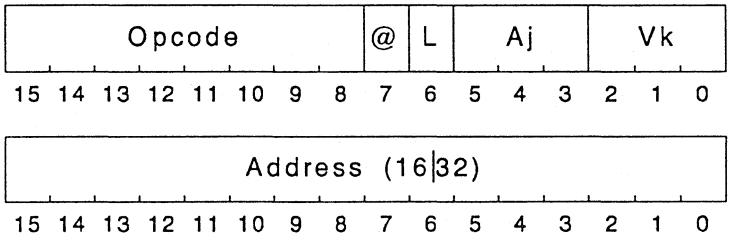
2. The *.s* and *.d* forms rename the *.w* and *.l* forms, respectively, for convenience.

ld.(b|h|w|l|s|d) <effa>,Vk — LOAD VECTOR REGISTER

Purpose: To load memory contents into a vector accumulator

Architecture: C100 Series, C200 Series

Format:



Operation:

```
temp = Effective_Address;
for (a = 0; a < VL; a++) {
     Vk[a] = c(temp); /* according to type */
     temp = temp + VS;
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
ld.b <effa>,Vk	3800	ST 001110000	None	Load vector byte
ld.h <effa>,Vk	3900	ST 001110010	None	Load vector halfword
ld.w <effa>,Vk	3A00	ST 001110100	None	Load vector word
ld.l <effa>,Vk	3B00	ST 001110110	None	Load vector longword
ld.s <effa>,Vk	3A00	ST 001110100	None	Load vector single float
ld.d <effa>,Vk	3B00	ST 001110110	None	Load vector double float

Description: VL elements of a vector stored in memory replace the first VL elements of the specified vector accumulator Vk. The effective address formed by evaluating the @, L, Aj, and address fields references the first element to be loaded. Successive elements come from addresses formed by incrementing this effective address by the contents of the Vector Stride (VS) register. The signed value contained in VS is the distance in bytes.

Notes:

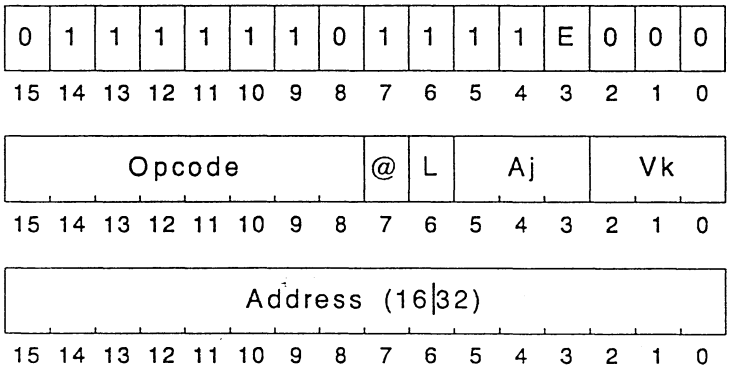
1. The *.s* and *.d* forms rename the *.w* and *.l* forms, respectively, for convenience.
2. If the absolute value of VS is nonzero, but less than the size of the elements being loaded, then results are unpredictable.

LOAD VECTOR REGISTER MASKED — ld.(b|h|w|l|s|d).(t|f) <effa>,Vk

Purpose: To load a vector into a vector accumulator under control of the Vector Merge (VM) register

Architecture: C200 Series only

Format:



Operation:

```
temp = Effective_Address;
switch (E) {  /* prefix bit<3> */
     case TRUE:   /*  t */
          for (a = 0; a < VL; a++) {
               if (VM<a> == 1) {        /* if VM<a> is TRUE */
                    Vk[a] = c(temp); /* according to type */
               }
               temp = temp + VS;
          }   /* end of for loop */
          break;  /* go to end of switch */
     case FALSE:  /*  f */
          for (a = 0; a < VL; a++) {
               if (VM<a> == 0) {        /* if VM<a> is FALSE */
                    Vk[a] = c(temp); /* according to type */
               }
               temp = temp + VS;
          }   /* end of for loop */
          break;  /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
ld.b.t <effa>,Vk	3800	E1 001110000	None	Load vector byte (VM)
ld.b.f <effa>,Vk	3800	E0 001110000	None	Load vector byte (!VM)
ld.h.t <effa>,Vk	3900	E1 001110010	None	Load vector halfword (VM)
ld.h.f <effa>,Vk	3900	E0 001110010	None	Load vector halfword (!VM)
ld.w.t <effa>,Vk	3A00	E1 001110100	None	Load vector word (VM)
ld.w.f <effa>,Vk	3A00	E0 001110100	None	Load vector word (!VM)
ld.l.t <effa>,Vk	3B00	E1 001110110	None	Load vector longword (VM)
ld.l.f <effa>,Vk	3B00	E0 001110110	None	Load vector longword (!VM)
ld.s.t <effa>,Vk	3A00	E1 001110100	None	Load vector single float (VM)
ld.s.f <effa>,Vk	3A00	E0 001110100	None	Load vector single float (!VM)
ld.d.t <effa>,Vk	3B00	E1 001110110	None	Load vector double float (VM)
ld.d.f <effa>,Vk	3B00	E0 001110110	None	Load vector double float (!VM)

Description: VL elements of a vector stored in memory replace the first VL elements of the specified vector accumulator Vk, if the corresponding VM bit is set (clear for *.f*). The effective address formed by evaluating the L, @, Aj, and address fields references the first element to be loaded. Successive candidate elements come from addresses

formed by incrementing this effective address by the contents of the vector stride register, VS. The VS signed value is the distance in bytes.

Notes:

1. The *.s.(t|f)* and *.d.(t|f)* forms rename the *.w.(t|f)* and *.l.(t|f)* forms for convenience.
2. If the absolute value of VS is nonzero, but less than the size of the elements being loaded, then results are unpredictable.

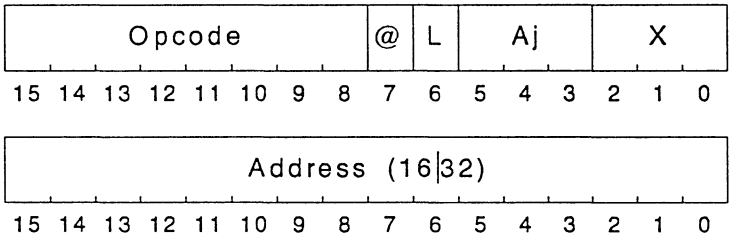
LOAD VS AND VL

ld.l <effa>,VLS

Purpose: To load the Vector Stride (VS) register and the Vector Length (VL) register from memory

Architecture: C100 Series, C200 Series

Format:



Operation:

```
VS = c(Effective_Address)<63..32>;
VL = c(Effective_Address)<31..0>;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
ld.l <effa>,VLS	0A00	ST 000010100	None	Load VS and VL from memory

Description: The most significant 32 bits of the contents of the effective address replace the contents of the VS register.

The least significant 32 bits of the contents of the effective address replace the contents of the VL register, bracketed to the range [0..128].

Notes: The X field is ignored.

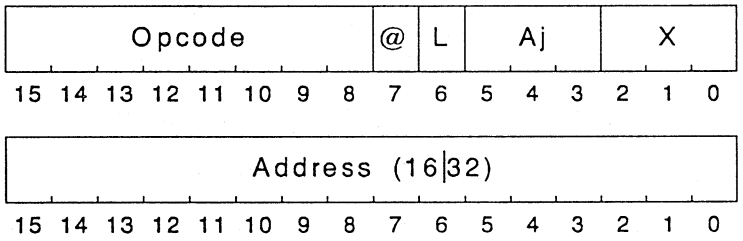
ld.x <effa>, VM

LOAD VM

Purpose: To load the Vector Merge (VM) register from memory

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
VM = c(Effective_Address)<127  0>;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
ld.x <effa>, VM	0B00	ST 00001011	None	Load VM from memory

Description: The 128 bits (16 bytes) beginning at the effective address replace the value of the VM register.

Notes:

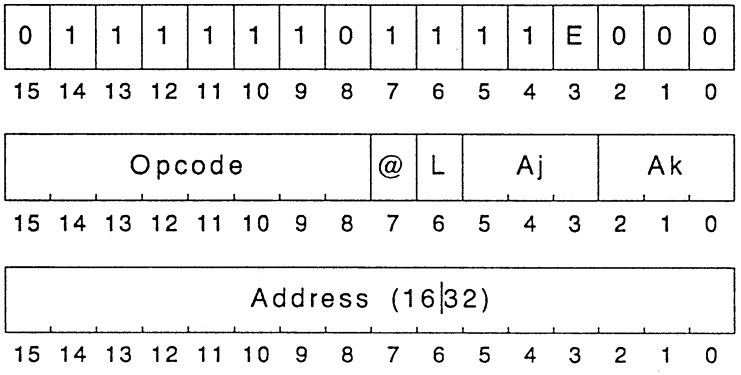
1. VM bits <127..120> are loaded from the byte referenced by the effective address.
2. VM bits <7..0> are loaded from the byte referenced by effective address plus 15.
3. The X field is ignored.

LOAD COMMUNICATION REGISTERS — ldcmr <effa>,Ak

Purpose: Load a specified set of communication registers from memory

Architecture: C200 Series only

Format:



Operation:

```
/* For the CIR specified in Ak: */
if (ring 0 comm register valid bit == 1) {/* memory copy */
     (Load the hardware communication registers and lock bits);
     (Load the ring 0 communication registers and lock bits);
}
if (ring 4 comm register valid bit == 1) {/* memory copy */
     (Load the ring 4 communication registers and lock bits);
}
ring 0 comm register modified bit = 0;
ring 4 comm register modified bit = 0;
```

Exceptions: Ring Violation (Privileged Instruction)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
ldcmr <effa>,Ak	0600	E0 0000011000	None	Load communication registers

Description: The communication register set index contained in Ak specifies a communication register set that is loaded from memory.

The communication register valid bits in memory are checked, and if nonzero the associated communication register lock bit longwords and communication registers are loaded. The communication register modified bits are then zeroed. The memory map that defines the exact format of the memory data is implementation-specific. Refer to the *CONVEX Architecture Reference*, "Multiprocessor Management" and "Implementation-Specific Features" chapters, for details.

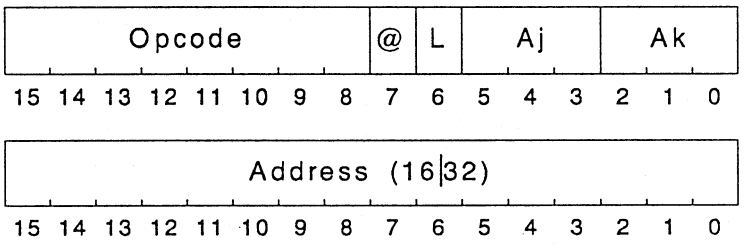
Notes: Prior to the communication registers in memory are the register set modified bit longword, and enough longword locations to hold the lock bits. A full longword contains 64 lock bits.

ldea <effa>,Ak — LOAD EFFECTIVE ADDRESS/ADDRESS

Purpose: To load an address register with a calculated effective address

Architecture: C100 Series, C200 Series

Format:



Operation:

```
Ak = Effective_Address;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
ldea <effa>,Ak	0900	ST 00001001	None	Load effective address

Description: The value of the effective address determined by evaluating the @, L, Aj, and address fields replaces the contents of address register Ak.

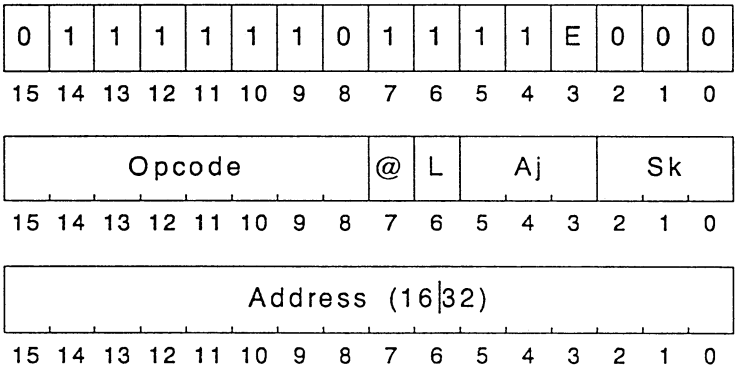
Notes: No ring violation occurs if the developed effective address references an inner ring, or if it is an invalid address. A ring violation is determined only if the developed effective address is actually used as a virtual address in a subsequent operation.

LOAD EFFECTIVE ADDRESS/SCALAR — ldea <effa>,Sk

Purpose: To load a scalar register with a calculated effective address

Architecture: C200 Series only

Format:



Operation:

```
Sk<31..0> = Effective_Address;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
ldea <effa>,Sk	0400	E0 000001000	None	Load effective address/scalar

Description: The value of the effective address determined by evaluating the L, @, Aj, and address fields replaces the contents of the least significant word of scalar register Sk. The upper word of Sk is unaffected.

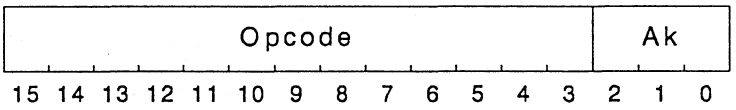
Notes: No ring violation occurs if the developed effective address references an inner ring, or if it is an invalid address. A ring violation is determined only if the developed effective address is actually used as a virtual address in a subsequent operation.

ldkdr Ak — LOAD KERNEL SDRs

Purpose: To load values in all eight Segment Descriptor Registers (SDRs)

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
SDR(0,1,...,7) = c(Ak)(0,1,...,7);
```

Exceptions: Ring Violation (Privileged Instruction)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
ldkdr Ak	7C08	ST 0111110000001	None	Load all eight SDRs

Description: Address register Ak contains the effective address of an 8-word table. The contents of this table's first word entry replace SDR0. Successive words replace higher numbered SDRs.

Notes:

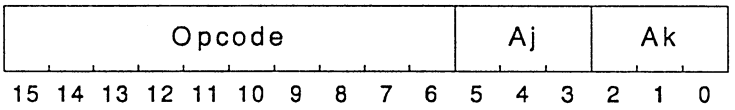
1. The data that are to be loaded into the Ring 0 (kernel) SDRs of a C1 or C120 CPU must be both resident- and word-aligned on a 32-bit boundary or a machine exception results. If the data to be loaded crosses a page boundary, a machine exception results.
2. The execution environment is implemented somewhat differently, depending on the CPU.
   a. When this instruction is executed on a C100 Series architecture CPU, the addressing environment must be physical, i.e., address translation must be disabled, or a machine exception results.
   b. Since the C200 Series architecture CPU always has address translation enabled, the only safe method to execute this instruction is to use the same SDR0 that is already loaded.
3. After loading all eight entries into SDR <0..7>, the C100 Series architecture CPU purges the ATU, logical, and instruction caches and enables logical address translation.
4. After loading all eight entries into SDR <0..7>, the C200 Series architecture CPU purges the ATU only.

LOAD PHYSICAL ADDRESS — ldpa Aj,Ak

Purpose: To convert a logical address to a physical address and return the result to an address register

Architecture: C100 Series, C200 Series

Format:



Operation:

```
/* The logical address contained in address register Aj is converted into a
 * physical address that replaces the contents of Ak.  The CPU validates the
 * address in Aj in the following sequence:
 */

     if (ring_maximization_violation(Aj)) {
         A5 = 0x00000801;
         Aj = 0;
         Ak = 0;
         C  = 1;
         exit;
     }
     if (SDR[segment[Aj]].valid == 0) {
         A5 = 0x00000c04;
         Aj = 0;
         Ak = 0;
         C  = 1;
         exit;
     }
     if (pte1[Aj].valid == 0) {
         A5 = 0x00000c05;
         Aj = phyaddr[pte1[Aj]];
         Ak = pte1[Aj];
         C  = 1
         exit;
     }
     if (pte1[Aj].resident == 0) {
         A5 = 0x00001000;
         Aj = phyaddr[pte1[Aj]];
         Ak = pte1[Aj];
         C  = 1;
         exit;
     }
     if (pte2[Aj].valid == 0) {
         A5 = 0x00000c06;
         Aj = phyaddr[pte2[Aj]];
         Ak = pte2[Aj];
         C  = 1;
         exit;
     }

/*
 * Determine if the memory page is an UNSHARED memory page
 * of a multiple-CPU processor
 */

     if ( (pte2[Aj].LT == 1) && (Architecture = C200) ) {
         if (pte2[Aj].resident == 0) {
             A5 = 0x00001002;
             Aj = phyaddr[pte2[Aj]];
             Ak = pte2[Aj];
             C  = 1;
```

```
            exit;
        }
        if (pteT[Aj].valid) == 0) {
            A5 = 0x00000c08;
            Aj = phyaddr(pteT(Aj,TID));
            Ak = pteT(Aj,TID);
            C  = 1;
            exit;
        }
        if (pteT[Aj,TID].resident) == 0) {
            A5 = 0x00001001;
            Aj = phyaddr[pteT[Aj,TID]];
            Ak = pteT[Aj,TID];
            C  = 1;
            exit;
        }
        A5 = phyaddr[data];
        Aj = phyaddr[pteT[Aj,TID]];
        Ak = pteT[Aj,TID];
        C  = 0;
        exit;

/*
 * Determine if the memory page is a uniprocessor data page OR
 * a SHARED multiprocessor data page
 */

    } else {
        if (pte2[Aj].resident == 0) {
            A5 = 0x00001001;
            Aj = phyaddr[pte2[Aj]];
            Ak = pte2[Aj];
            C  = 1;
            exit;
        }
        A5 = phyaddr[data];
        Aj = phyaddr[pte2[Aj]];
        Ak = pte2[Aj];
        C  = 0;
        exit;
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
ldpa Aj,Ak	4400	ST 0100010000	See note 1	Load a physical byte address into Ak

Description: The contents of Aj are assumed to be a logical address that is translated to its equivalent physical address. If the logical address is valid, the physical address replaces the contents of address register A5, and the Carry (C) bit in the PSW is reset to 0. If the logical address is invalid, error information replaces the contents of A5, and C is set to 1. In either case, the physical address of the last Page Table Entry (PTE) replaces the contents of Aj, and the PTE itself replaces the contents of Ak.

The error information placed in A5 is consistent with system exception conditions detailed in the *CONVEX Architecture Reference*, "Exceptions and Interrupts" chapter.

Notes: 1. For the PSW, C = 1 (if invalid reference) and C = 0 (if valid reference)

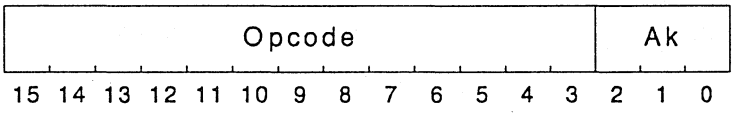
2. This instruction can also be used to determine whether or not a page is resident in physical memory.
3. No access checks, i.e., read, write, or execute, are performed.

ldsdr Ak — LOAD PROCESS SDRs

Purpose: To load values from memory into the process Segment Descriptor Registers (SDRs)

Architecture: C100 Series, C200 Series

Format:



Operation:

```
SDR(1,2,...,7) = c(Ak)(1,2,...,7);
```

Exceptions: Ring Violation (Privileged Instruction)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
ldsdr Ak	7C00	ST 0111110000000	None	Load process SDRs

Description: Address register Ak contains the effective address of a 7-word table. The contents of the first table entry replace the contents of SDR <1>. Successive words replace the contents of higher numbered SDRs.

Notes:

1. The data to be loaded into the SDRs 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. After the load:

C100 Series only

a. a C100 Series architecture CPU purges the ATU, logical cache, and instruction cache of any entries associated with segments 1–7.

C200 Series only

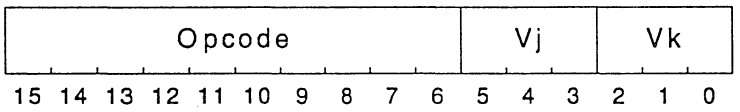
b. a C200 Series architecture CPU purges the ATU only.

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 ("vector gather")

Architecture: C100 Series, C200 Series

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      addr = A5 + Vj[a]<31..0>;
      Vk[a] = c(addr); /* according to type */
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
ldvi.b Vj,Vk	7800	ST 0111100000	None	Index load vector byte
ldvi.h Vj,Vk	7840	ST 0111100001	None	Index load vector halfword
ldvi.w Vj,Vk	7880	ST 0111100010	None	Index load vector word
ldvi.l Vj,Vk	78C0	ST 0111100011	None	Index load vector longword
ldvi.s Vj,Vk	7880	ST 0111100010	None	Index load vector single float
ldvi.d Vj,Vk	78C0	ST 0111100011	None	Index load vector double float

Description: The bottom 32 bits of each of the first VL elements of vector register Vj specify a set of address offsets. The sum of each of these offsets and the contents of address register A5 yield a set of addresses that specify the sources of operands which replace the contents of the first VL elements of Vk.

Notes:

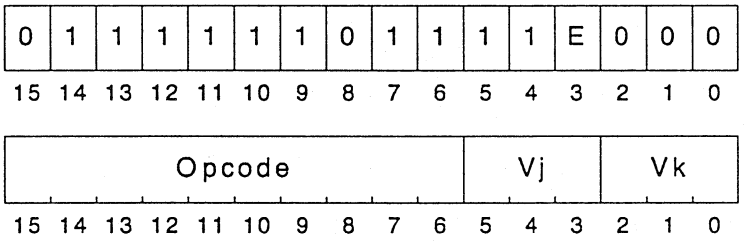
1. An *ldea* instruction typically loads A5 before execution of this instruction.
2. If the distance between successive elements is less than the precision of an element, unpredictable actions occur.
3. The *.s* and *.d* forms rename the *.w* and *.l* forms, respectively, for convenience.
4. In Vk, only the bits specified by the precision of the instruction are changed.

ldvi.(b|h|w|l|s|d).(t|f) Vj,Vk — LOAD VECTOR/VECTOR INDEX MASKED

Purpose: To load a vector into a vector accumulator using a vector of indices under control of the Vector Merge (VM) register ("vector gather")

Architecture: C200 Series only

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
     case TRUE:   /*  .t */
          for (a = 0; a < VL; a++) {
               if (VM<a> == 1) {        /* if VM<a> is TRUE */
                    addr = A5 + Vj[a]<31..0>;
                    Vk[a] = c(addr); /* according to type */
               }
          }   /* end of for loop */
          break;   /* go to end of switch */
     case FALSE:  /*  .f */
          for (a = 0; a < VL; a++) {
               if (VM<a> == 0) {        /* if VM<a> is FALSE */
                    addr = A5 + Vj[a]<31..0>;
                    Vk[a] = c(addr); /* according to type */
               }
          }   /* end of for loop */
          break;   /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
ldvi.b.t Vj,Vk	7800	E1 0111100000	None	Index load vector byte (VM)
ldvi.b.f Vj,Vk	7800	E0 0111100000	None	Index load vector byte (!VM)
ldvi.h.t Vj,Vk	7840	E1 0111100001	None	Index load vector halfword (VM)
ldvi.h.f Vj,Vk	7840	E0 0111100001	None	Index load vector halfword (!VM)
ldvi.w.t Vj,Vk	7880	E1 0111100010	None	Index load vector word (VM)
ldvi.w.f Vj,Vk	7880	E0 0111100010	None	Index load vector word (!VM)
ldvi.l.t Vj,Vk	78C0	E1 0111100011	None	Index load vector longword (VM)
ldvi.l.f Vj,Vk	78C0	E0 0111100011	None	Index load vector longword (!VM)
ldvi.s.t Vj,Vk	7880	E1 0111100010	None	Index load vector single (VM)
ldvi.s.f Vj,Vk	7880	E0 0111100010	None	Index load vector single (!VM)
ldvi.d.t Vj,Vk	78C0	E1 0111100011	None	Index load vector double (VM)
ldvi.d.f Vj,Vk	78C0	E0 0111100011	None	Index load vector double (!VM)

Description: The bottom 32 bits of each of the first VL elements of vector register Vj specify a set of address offsets. The sum of each of these offsets and the contents of address register A5 yield a set of addresses that specify the sources of operands which replace the contents of the first VL elements of Vk, if the corresponding VM bit is set (clear for *.f*).

Notes: 1. An *ldea* instruction typically loads A5 before execution of this instruction.

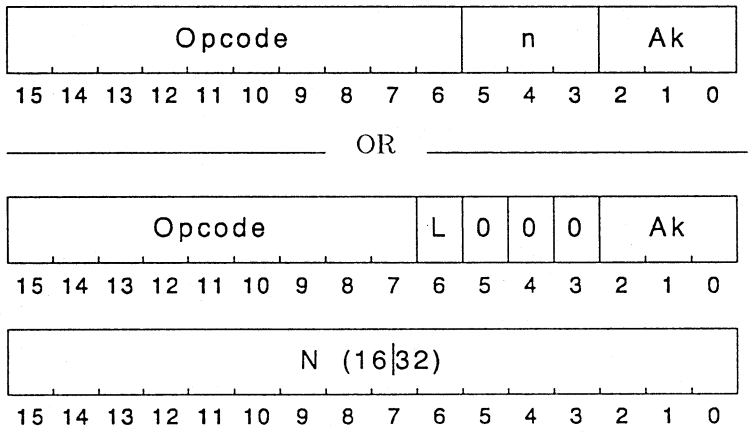
2. If the distance between successive elements is less than the precision of an element, unpredictable actions occur.
3. The *.s.(t|f)* and *.d.(t|f)* forms rename the *.w.(t|f)* and *.l.(t|f)* forms for convenience.
4. In Vk, only the bits specified by the precision of the instruction are changed.

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

Purpose: To compare the contents of an address register with an immediate

Architecture: C100 Series, C200 Series

Format:



Operation:

```
if (Opcode_Test(Immediate,Ak) == TRUE) {
      C = 1;
} else {
      C = 0;
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
le.h #n,Ak	4C80	ST 0100110010	See note 1	Compare less than or equal halfword
le.h #N,Ak	1E00	ST 000111100	See note 1	Compare less than or equal halfword
le.w #n,Ak	4CC0	ST 0100110011	See note 1	Compare less than or equal word
le.w #N,Ak	1E80	ST 000111101	See note 1	Compare less than or equal word
lt.h #N,Ak	1F00	ST 000111110	See note 1	Compare less than halfword
lt.h #n,Ak	4E80	ST 0100111010	See note 1	Compare less than halfword
lt.w #n,Ak	4EC0	ST 0100111011	See note 1	Compare less than word
lt.w #N,Ak	1F80	ST 000111111	See note 1	Compare less than word
eq.h #n,Ak	4680	ST 0100011010	See note 1	Compare equal halfword
eq.h #N,Ak	1B00	ST 000110110	See note 1	Compare equal halfword
eq.w #n,Ak	46C0	ST 0100011011	See note 1	Compare equal word
eq.w #N,Ak	1B80	ST 000110111	See note 1	Compare equal word

Description: The truth value of the comparison between the contents of the address register Ak and the (sign-extended) immediate replaces the Carry (PSW<C>) bit. If the comparison is true, the PSW<C> bit is set to 1; otherwise, if the comparison is false, the PSW<C> bit is reset to 0. For example: *lt.w #1,a3* sets PSW<C> to 1 when 1 is less than the contents of address register A3.

Notes:

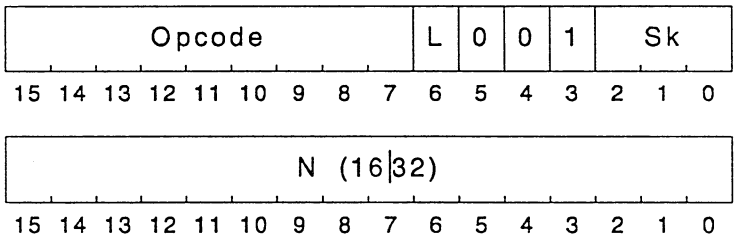
1. Address Carry (C) is affected as described by above operation pseudocode.
2. Sign extension does not occur for the 3 bits of the short immediate form.
3. Test for *not equal to*, *greater than*, and *greater than or equal to* by inverting the truth sense of the branch on carry instruction.

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

Purpose: To compare the contents of a scalar register and an immediate

Architecture: C100 Series, C200 Series

Format:



Operation:

```
if (Opcode_Test(Immediate,Sk) == TRUE) {
      SC = 1;
} else {
      SC = 0;
}
```

Exceptions:

(h|w): None

(s): Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
le.h #N,Sk	1E08	ST 000111100	See note 1	Compare less than or equal halfword
le.w #N,Sk	1E88	ST 000111101	See note 1	Compare less than or equal word
le.s #N,Sk	1A08	ST 000110100	RO,SC	Compare less than or equal single
lt.h #N,Sk	1F08	ST 000111110	See note 1	Compare less than halfword
lt.w #N,Sk	1F88	ST 000111111	See note 1	Compare less than word
lt.s #N,Sk	1A88	ST 000110101	RO,SC	Compare less than single
eq.h #N,Sk	1B08	ST 000110110	See note 1	Compare equal halfword
eq.w #N,Sk	1B88	ST 000110111	See note 1	Compare equal word

Description: The truth value of the comparison between the contents of the scalar register Sk and the (sign-extended) immediate replaces the Scalar Carry (SC) bit. If the comparison is true, the PSW<SC> bit is set to 1; otherwise, if the comparison is false, the PSW<SC> bit is reset to 0. Example: *lt.w #1,s3* sets PSW<SC> to 1 when 1 is less than the contents of scalar register S3.

Notes:

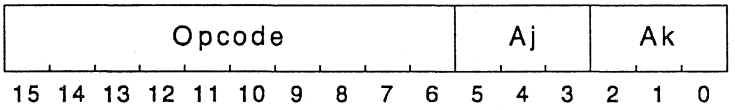
1. Scalar Carry (SC) is affected as described by the above operation pseudocode.
2. Test for *not equal to*, *greater than*, and *greater than or equal to* by inverting the truth sense of the branch-on-carry instruction.
3. Detect floating-point reserved operands by using a compare word immediate (for equality with the floating point reserved operand as its immediate value).
4. The *eq.s* instruction is synonomous to *eq.w*. However, when executed in IEEE mode, *eq.s # 0.0,Sk* does not treat negative zero as zero.

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

Purpose: To compare the contents of two address registers

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
if (Opcode_Test(Aj,Ak) == TRUE) {
      C = 1;
} else {
      C = 0;
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
le.h Aj,Ak	4C00	ST 0100110000	See note 1	Compare less than or equal signed halfword
le.w Aj,Ak	4C40	ST 0100110001	See note 1	Compare less than or equal signed word
lt.h Aj,Ak	4E00	ST 0100111000	See note 1	Compare less than signed halfword
lt.w Aj,Ak	4E40	ST 0100111001	See note 1	Compare less than signed word
eq.h Aj,Ak	4600	ST 0100011000	See note 1	Compare equal halfword
eq.w Aj,Ak	4640	ST 0100011001	See note 1	Compare equal word

Description: The truth value of the comparison between the contents of address register Ak and the contents of address register Aj replaces the Carry (PSW<C>) bit. If the comparison is true, the PSW<C> bit is set to 1; otherwise, if the comparison is false, the PSW<C> bit is reset to 0.

Example: *lt.w a1,a3* sets PSW<C> to 1 when the contents of address register A1 are less than the contents of address register A3.

Notes:

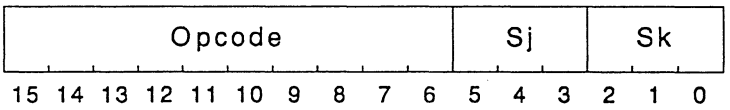
1. C is affected as described by above operation pseudocode
2. Test for *not equal to* by inverting the truth sense of the later branch-on-carry instruction. Test for *greater than* or *greater than or equal to* by inverting the truth sense of the branch on carry or by inverting the order of the operands of the compare.
3. Unsigned equal comparison is equivalent to signed equal comparison.

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

Purpose: To compare the contents of two scalar registers

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
if (Opcode_Test(Sj,Sk) == TRUE) {
      SC = 1;
} else {
      SC = 0;
}
```

Exceptions: (b|h|w|l): None

(s|d): Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
le.b Sj,Sk	4D00	ST 0100110100	See note 1	Compare less than or equal byte
le.h Sj,Sk	4D40	ST 0100110101	See note 1	Compare less than or equal halfword
le.w Sj,Sk	4D80	ST 0100110110	See note 1	Compare less than or equal word
le.l Sj,Sk	4DC0	ST 0100110111	See note 1	Compare less than or equal longword
le.s Sj,Sk	5400	ST 0101010000	RO,SC	Compare less than or equal single float
le.d Sj,Sk	5440	ST 0101010001	RO,SC	Compare less than or equal double float
lt.b Sj,Sk	4F00	ST 0100111100	See note 1	Compare less than byte
lt.h Sj,Sk	4F40	ST 0100111101	See note 1	Compare less than halfword
lt.w Sj,Sk	4F80	ST 0100111110	See note 1	Compare less than word
lt.l Sj,Sk	4FC0	ST 0100111111	See note 1	Compare less than longword
lt.s Sj,Sk	5480	ST 0101010010	RO,SC	Compare less than single float
lt.d Sj,Sk	54C0	ST 0101010011	RO,SC	Compare less than double float
eq.b Sj,Sk	4700	ST 0100011100	See note 1	Compare equal byte
eq.h Sj,Sk	4740	ST 0100011101	See note 1	Compare equal halfword
eq.w Sj,Sk	4780	ST 0100011110	See note 1	Compare equal word
eq.l Sj,Sk	47C0	ST 0100011111	See note 1	Compare equal longword
eq.s Sj,Sk	5600	ST 0101011000	RO,SC	Compare equal single float
eq.d Sj,Sk	5640	ST 0101011001	RO,SC	Compare equal double float

Description: The truth value of the comparison between the contents of scalar register Sk and the contents of scalar register Sj replaces the Scalar Carry (SC) bit. If the comparison is true, the PSW<SC> bit is set to 1; otherwise, if the comparison is false, the PSW<SC> bit is reset to 0. Example: *lt.w S2,S4* sets PSW<SC> to 1 when the contents of scalar register S2 are less than the contents of S4.

Notes:

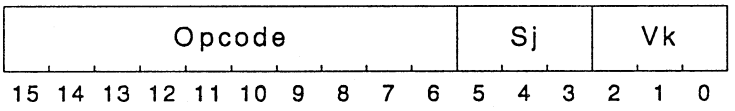
1. Scalar Carry (SC) is affected as described by the preceding operation pseudocode.
2. Test for *not equal to* by inverting the truth sense of the later branch-on-carry instruction. Test for *greater than* or *greater than or equal to* by inverting the truth sense of the branch-on-carry instruction or by inverting the order of the operands of the compare.
3. Unsigned equal comparison is equivalent to signed equal comparison.
4. The *eq.s Sj,Sk* instruction traps on reserved operands; *eq.w Sj,Sk* does not.

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

Purpose: To compare vector elements and the contents of a scalar register

Architecture: C100 Series, C200 Series

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      if (Opcode_Test(Sj,Vk[a]) == TRUE) {
          VM<a> = 1;
      } else {
          VM<a> = 0;
      }
}
if (VL < 128) {
      for (a = VL; a < 128; a++) {
          VM<a> = 0;
      }
}
```

Exceptions:

(b|h|w|l): None

(s|d): Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
le.b Sj,Vk	6B00	ST 0110101100	None	Compare less than or equal byte
le.h Sj,Vk	6B40	ST 0110101101	None	Compare less than or equal halfword
le.w Sj,Vk	6B80	ST 0110101110	None	Compare less than or equal word
le.l Sj,Vk	6BC0	ST 0110101111	None	Compare less than or equal longword
le.s Sj,Vk	6700	ST 0110011100	RO	Compare less than or equal single
le.d Sj,Vk	6740	ST 0110011101	RO	Compare less than or equal double float
lt.b Sj,Vk	6D00	ST 0110110100	None	Compare less than byte
lt.h Sj,Vk	6D40	ST 0110110101	None	Compare less than halfword
lt.w Sj,Vk	6D80	ST 0110110110	None	Compare less than word
lt.l Sj,Vk	6DC0	ST 0110110111	None	Compare less than longword
lt.s Sj,Vk	6780	ST 0110011110	RO	Compare less than single
lt.d Sj,Vk	67C0	ST 0110011111	RO	Compare less than double float
eq.b Sj,Vk	6900	ST 0110100100	None	Compare equal byte
eq.h Sj,Vk	6940	ST 0110100101	None	Compare equal halfword
eq.w Sj,Vk	6980	ST 0110100110	None	Compare equal word
eq.l Sj,Vk	69C0	ST 0110100111	None	Compare equal longword
eq.s Sj,Vk	6500	ST 0110010100	RO	Compare equal single
eq.d Sj,Vk	6540	ST 0110010101	RO	Compare equal double precision

Description: The truth value of the comparison between the contents of scalar register Sj and the respective contents of each of the first VL elements of vector register Vj replaces the respective bits of the VM register. If the comparison is true, the bit of VM is set to 1; otherwise, the bit of VM is cleared to 0. When VL is less than 128, remaining bits of VM are cleared to 0. Example: *lt.h* S4,V2 sets VM bits to 1 if the contents of scalar register S4 are less than the various contents of vector register V2[i].

Notes:

1. Test for *not equal to*, *greater than*, or *greater than or equal to* by inverting the truth sense of the branch on carry or by inverting the truth sense of the VM register.

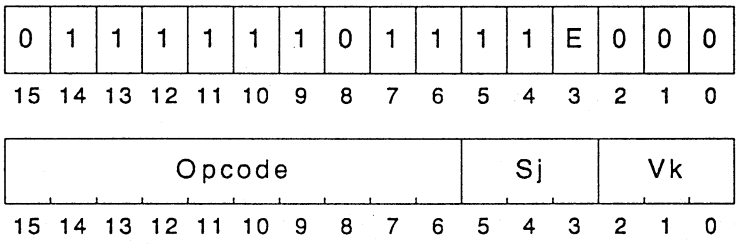
2. There are no unsigned vector compares.
3. If Sj or any of the unmasked elements of Vk are a reserved operand, a reserved operand exception is taken and the results in VM are ***undefined***.
4. The *plc* instruction calculates the number of successful compares.

(le|lt|eq).(b|h|w|l|s|d).(t|f) Sj,Vk — COMPARE VECTOR/SCALAR MASKED

Purpose: To compare a vector and a scalar and load the Vector Merge (VM) under control of the VM register at the instruction entry

Architecture: C200 Series only

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
      case TRUE:   /*  t */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 1) {        /* if VM<a> is TRUE */
                  if (Opcode_Test(Sj,Vk[a] == TRUE) {
                      VM<a> = 1;
                  } else {
                      VM<a> = 0;
                  }
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
      case FALSE:  /*  f */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 0) {        /* if VM<a> is FALSE */
                  if (Opcode_Test(Sj,Vk[a] == TRUE) {
                      VM<a> = 1;
                  } else {
                      VM<a> = 0;
                  }
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
}             /* end of switch */
if (VL < 128) {
      for (a = VL; a < 128; a++) {
          VM<a> = 0;
      }
}
```

Exceptions:

(b|h|w|l): None

(s|d): Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
le.b.t Sj,Vk	6B00	E1 0110101100	None	Compare less than or equal byte (VM)
le.b.f Sj,Vk	6B00	E0 0110101100	None	Compare less than or equal byte (!VM)
le.h.t Sj,Vk	6B40	E1 0110101101	None	Compare less than or equal half (VM)
le.h.f Sj,Vk	6B40	E0 0110101101	None	Compare less than or equal half (!VM)
le.w.t Sj,Vk	6B80	E1 0110101110	None	Compare less than or equal word (VM)
le.w.f Sj,Vk	6B80	E0 0110101110	None	Compare less than or equal word (!VM)
le.l.t Sj,Vk	6BC0	E1 0110101111	None	Compare less than or equal long (VM)
le.l.f Sj,Vk	6BC0	E0 0110101111	None	Compare less than or equal long (!VM)
le.s.t Sj,Vk	6700	E1 0110011100	RO	Compare less than or equal single (VM)
le.s.f Sj,Vk	6700	E0 0110011100	RO	Compare less than or equal single (!VM)
le.d.t Sj,Vk	6740	E1 0110011101	RO	Compare less than or equal double (VM)

le.d.f Sj,Vk	6740	E0 0110011101	RO	Compare less than or equal double (!VM)
lt.b.t Sj,Vk	6D00	E1 0110110100	None	Compare less than byte (VM)
lt.b.f Sj,Vk	6D00	E0 0110110100	None	Compare less than byte (!VM)
lt.h.t Sj,Vk	6D40	E1 0110110101	None	Compare less than halfword (VM)
lt.h.f Sj,Vk	6D40	E0 0110110101	None	Compare less than halfword (!VM)
lt.w.t Sj,Vk	6D80	E1 0110110110	None	Compare less than word (VM)
lt.w.f Sj,Vk	6D80	E0 0110110110	None	Compare less than word (!VM)
lt.l.t Sj,Vk	6DC0	E1 0110110111	None	Compare less than long (VM)
lt.l.f Sj,Vk	6DC0	E0 0110110111	None	Compare less than long (!VM)
lt.s.t Sj,Vk	6780	E1 0110011110	RO	Compare less than single (VM)
lt.s.f Sj,Vk	6780	E0 0110011110	RO	Compare less than single (!VM)
lt.d.t Sj,Vk	67C0	E1 0110011111	RO	Compare less than double (VM)
lt.d.f Sj,Vk	67C0	E0 0110011111	RO	Compare less than double (!VM)
eq.b.t Sj,Vk	6900	E1 0110100100	None	Compare equal byte (VM)
eq.b.f Sj,Vk	6900	E0 0110100100	None	Compare equal byte (!VM)
eq.h.t Sj,Vk	6940	E1 0110100101	None	Compare equal halfword (VM)
eq.h.f Sj,Vk	6940	E0 0110100101	None	Compare equal halfword (!VM)
eq.w.t Sj,Vk	6980	E1 0110100110	None	Compare equal word (VM)
eq.w.f Sj,Vk	6980	E0 0110100110	None	Compare equal word (!VM)
eq.l.t Sj,Vk	69C0	E1 0110100111	None	Compare equal long (VM)
eq.l.f Sj,Vk	69C0	E0 0110100111	None	Compare equal long (!VM)
eq.s.t Sj,Vk	6500	E1 0110010100	RO	Compare equal single (VM)
eq.s.f Sj,Vk	6500	E0 0110010100	RO	Compare equal single (!VM)
eq.d.t Sj,Vk	6540	E1 0110010101	RO	Compare equal double (VM)
eq.d.f Sj,Vk	6540	E0 0110010101	RO	Compare equal double (!VM)

Description: The truth value of the comparison of the contents of scalar register Sj compared to the respective contents of each of the first VL elements of vector register Vj replaces the respective bits of the VM register, if the corresponding initial (at beginning of the instruction) VM bit is set (clear for *.f*). If the comparison is true, the bit of VM is set to 1; otherwise it is reset to 0. When VL is less than 128, remaining bits of VM are reset to 0. Example: *lt.hS4,V2* sets VM bits to 1 if the contents of scalar register S4 are less than the various contents of vector register V2[i].

Notes:

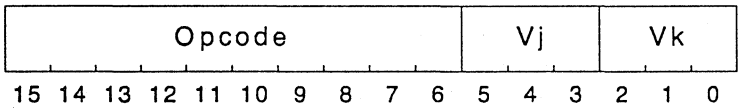
1. Test for *not equal to*, *greater than*, or *greater than or equal to* by inverting the truth sense of the VM register.
2. There are no unsigned vector compares.
3. If Sj or any of the unmasked elements of Vk are a reserved operand, a reserved operand exception is taken and the results in VM are ***undefined***.
4. The *plc* instruction calculates the number of successful compares.

(le|lt|eq).(b|h|w|l|s|d) Vj,Vk — COMPARE VECTOR/VECTOR

Purpose: To compare elements of two vector registers

Architecture: C100 Series, C200 Series

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      if (Opcode_Test(Vj[a],Vk[a]) == TRUE) {
          VM<a> = 1;
      } else {
          VM<a> = 0;
      }
}
if (VL < 128) {
      for (a = VL; a < 128; a++) {
          VM<a> = 0;
      }
}
```

Exceptions:

(b|h|w|l): None

(s|d): Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
le.b Vj,Vk	6A00	ST 0110101000	None	Compare less than or equal byte
le.h Vj,Vk	6A40	ST 0110101001	None	Compare less than or equal halfword
le.w Vj,Vk	6A80	ST 0110101010	None	Compare less than or equal word
le.l Vj,Vk	6AC0	ST 0110101011	None	Compare less than or equal longword
le.s Vj,Vk	6600	ST 0110011000	RO	Compare less than or equal single
le.d Vj,Vk	6640	ST 0110011001	RO	Compare less than or equal double float
lt.b Vj,Vk	6C00	ST 0110110000	None	Compare less than byte
lt.h Vj,Vk	6C40	ST 0110110001	None	Compare less than halfword
lt.w Vj,Vk	6C80	ST 0110110010	None	Compare less than word
lt.l Vj,Vk	6CC0	ST 0110110011	None	Compare less than longword
lt.s Vj,Vk	6680	ST 0110011010	RO	Compare less than single
lt.d Vj,Vk	66C0	ST 0110011011	RO	Compare less than double float
eq.b Vj,Vk	6800	ST 0110100000	None	Compare equal byte
eq.h Vj,Vk	6840	ST 0110100001	None	Compare equal halfword
eq.w Vj,Vk	6880	ST 0110100010	None	Compare equal word
eq.l Vj,Vk	68C0	ST 0110100011	None	Compare equal longword
eq.s Vj,Vk	6400	ST 0110010000	RO	Compare equal single
eq.d Vj,Vk	6440	ST 0110010001	RO	Compare equal double precision

Description: The truth value of the comparison of the contents of the first VL elements of vector register Vj to the respective contents of elements of vector register Vj replaces the respective bits of the VM register. If the comparison is true, the bit of VM is set to 1; otherwise it is reset to 0. When VL is less than 128, remaining bits of VM are reset to 0.

Example: *lt.h* V5,V1 sets VM<i> to 1 if the contents of vector register V5(i) are less than the contents of vector register V1[i].

Notes:

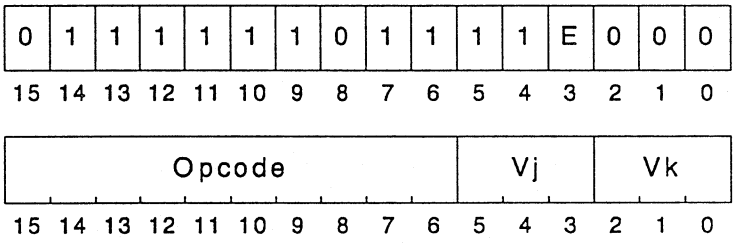
1. Test for *not equal to*, *greater than*, or *greater than or equal to* by inverting the truth sense of the VM register.
2. There are no unsigned vector compares.
3. The *plc.(t|f) VM,Sk* instruction calculates the number of successful compares.
4. Moving VM to a scalar register and performing a leading 1's position yields the index of the first successful compare.
5. If Sj or any of the unmasked elements of Vk are a reserved operand, a reserved operand exception is taken and the results in VM are ***undefined***.

(le|lt|eq).(b|h|w|l|s|d).(t|f) Vj,Vk — COMPARE VECTOR/VECTOR MASKED

Purpose: To compare two vectors and load Vector Merge (VM) under control of the VM register at the instruction entry

Architecture: **C200 Series only**

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
      case TRUE:   /* .t */
          for (a = 0; a < VL; a++) {
              if (VM<a>> == 1) {       /* if VM<a> is TRUE */
                  if (Opcode_Test(Vj[a],Vk[a]) == TRUE) {
                      VM<a> = 1;
                  } else {
                      VM<a> = 0;
                  }
              }
          }   /* end of for loop */
          break;   /* go to end of switch */
      case FALSE:  /* .f */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 0) {        /* if VM<a> is FALSE */
                  if (Opcode_Test(Vj[a],Vk[a]) == TRUE) {
                      VM<a> = 1;
                  } else {
                      VM<a> = 0;
                  }
              }
          }   /* end of for loop */
          break;   /* go to end of switch */
}             /* end of switch */
if (VL < 128) {
      for (a = VL; a < 128; a++) {
          VM<a> = 0;
      }
}
```

Exceptions:

(b|h|w|l): None

(s|d): Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
le.b.t Vj,Vk	6A00	E1 0110101000	None	Compare less than or equal byte (VM)
le.b.f Vj,Vk	6A00	E0 0110101000	None	Compare less than or equal byte (!VM)
le.h.t Vj,Vk	6A40	E1 0110101001	None	Compare less than or equal half (VM)
le.h.f Vj,Vk	6A40	E0 0110101001	None	Compare less than or equal half (!VM)
le.w.t Vj,Vk	6A80	E1 0110101010	None	Compare less than or equal word (VM)
le.w.f Vj,Vk	6A80	E0 0110101010	None	Compare less than or equal word (!VM)
le.l.t Vj,Vk	6AC0	E1 0110101011	None	Compare less than or equal long (VM)
le.l.f Vj,Vk	6AC0	E0 0110101011	None	Compare less than or equal long (!VM)
le.s.t Vj,Vk	6600	E1 0110011000	RO	Compare less than or equal single (VM)
le.s.f Vj,Vk	6600	E0 0110011000	RO	Compare less than or equal single (!VM)
le.d.t Vj,Vk	6640	E1 0110011001	RO	Compare less than or equal double (VM)

le.d.f Vj,Vk	6640	E0 0110011001	RO	Compare less than or equal double (!VM)
lt.b.t Vj,Vk	6C00	E1 0110110000	None	Compare less than byte (VM)
lt.b.f Vj,Vk	6C00	E0 0110110000	None	Compare less than byte (!VM)
lt.h.t Vj,Vk	6C40	E1 0110110001	None	Compare less than halfword (VM)
lt.h.f Vj,Vk	6C40	E0 0110110001	None	Compare less than halfword (!VM)
lt.w.t Vj,Vk	6C80	E1 0110110010	None	Compare less than word (VM)
lt.w.f Vj,Vk	6C80	E0 0110110010	None	Compare less than word (!VM)
lt.l.t Vj,Vk	6CC0	E1 0110110011	None	Compare less than long (VM)
lt.l.f Vj,Vk	6CC0	E0 0110110011	None	Compare less than long (!VM)
lt.s.t Vj,Vk	6680	E1 0110011010	RO	Compare less than single (VM)
lt.s.f Vj,Vk	6680	E0 0110011010	RO	Compare less than single (!VM)
lt.d.t Vj,Vk	66C0	E1 0110011011	RO	Compare less than double (VM)
lt.d.f Vj,Vk	66C0	E0 0110011011	RO	Compare less than double (!VM)
eq.b.t Vj,Vk	6800	E1 0110100000	None	Compare equal byte (VM)
eq.b.f Vj,Vk	6800	E0 0110100000	None	Compare equal byte (!VM)
eq.h.t Vj,Vk	6840	E1 0110100001	None	Compare equal halfword (VM)
eq.h.f Vj,Vk	6840	E0 0110100001	None	Compare equal halfword (!VM)
eq.w.t Vj,Vk	6880	E1 0110100010	None	Compare equal word (VM)
eq.w.f Vj,Vk	6880	E0 0110100010	None	Compare equal word (!VM)
eq.l.t Vj,Vk	68C0	E1 0110100011	None	Compare equal long (VM)
eq.l.f Vj,Vk	68C0	E0 0110100011	None	Compare equal long (!VM)
eq.s.t Vj,Vk	6400	E1 0110010000	RO	Compare equal single (VM)
eq.s.f Vj,Vk	6400	E0 0110010000	RO	Compare equal single (!VM)
eq.d.t Vj,Vk	6440	E1 0110010001	RO	Compare equal double (VM)
eq.d.f Vj,Vk	6440	E0 0110010001	RO	Compare equal double (!VM)

Description: The truth value of the comparison of the contents of the first VL respective elements, of vector register Vj to the respective contents of elements of vector register Vj replaces the respective bits of the VM register, if the corresponding initial (at beginning of the instruction) VM bit is set (clear for *.f*). If the comparison is true, the bit of VM is set to 1; otherwise, the bit of VM is cleared to 0. When VL is less than 128, remaining bits of the VM register are cleared to 0.

Example: *lt.h v5,v1* sets VM<i> to 1 if the contents of vector register V5[i] are less than the contents of vector register V1[i].

Notes:

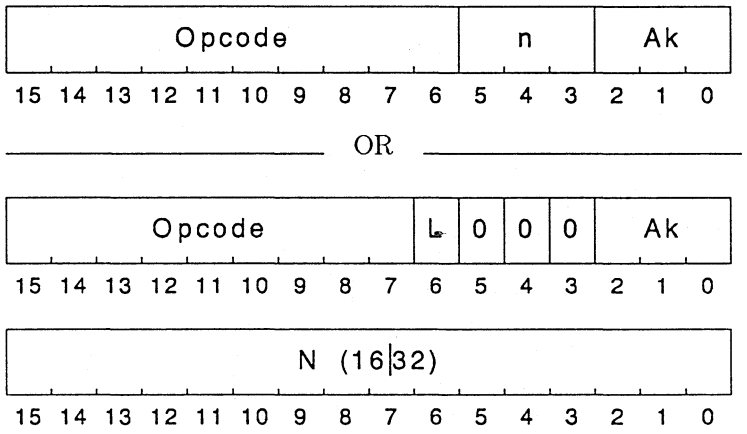
1. Test for *not equal to* by inverting the truth sense of the branch-on-carry instruction. Test for *greater than* or *greater than or equal to* by inverting the truth sense of the branch-on-carry instruction or by inverting the order of the operands of the compare.
2. There are no unsigned vector compares.
3. The *plc.(t|f) VM,Sk* instruction calculates the number of successful compares.
4. Moving VM to a scalar register and performing a leading zero count yields the index of the first successful compare.
5. If any of the elements of Vj or Vk are a reserved operand, a reserved operand exception is taken and the results in the VM register are ***undefined***.

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

Purpose: To compare the unsigned contents of an address register with an immediate

Architecture: C100 Series, C200 Series

Format:



Operation:

```
if (Opcode_Test(Immediate,Ak) == TRUE) {
     C = 1;
} else {
     C = 0;
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
leu.h #n,Ak	4880	ST 0100100010	See note 1	Compare unsigned less than or equal halfword
leu.h #N,Ak	1C00	ST 000111000	See note 1	Compare unsigned less than halfword
leu.w #n,Ak	48C0	ST 0100100011	See note 1	Compare unsigned less than or equal word
leu.w #N,Ak	1C80	ST 000111001	See note 1	Compare unsigned less than word
ltu.h #n,Ak	4A80	ST 0100101010	See note 1	Compare unsigned less than halfword
ltu.h #N,Ak	1D00	ST 000111010	See note 1	Compare unsigned less than halfword
ltu.w #n,Ak	4AC0	ST 0100101011	See note 1	Compare unsigned less than word
ltu.w #N,Ak	1D80	ST 000111011	See note 1	Compare unsigned less than word

Description: The truth value of the unsigned comparison between the contents of the address register Ak and the immediate replaces the Carry (PSW<C>) bit. If the comparison is true, the PSW<C> bit is set to 1; otherwise, if the comparison is false, the PSW<C> bit is cleared to 0. Example: *ltu.w #1,a3* sets C to 1 when 1 is less than the contents of address register A3.

Notes:

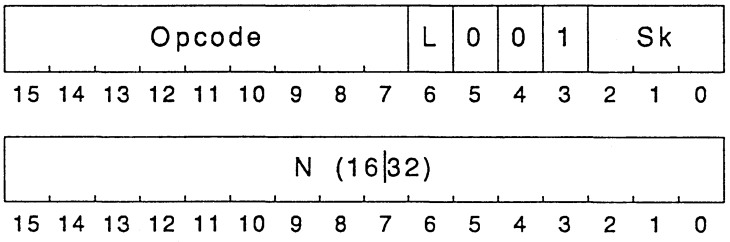
1. C is affected as described by above operation pseudocode
2. Unsigned equal comparison is equivalent to signed equal comparison.
3. Sign extension does not occur for the 3 bits of the short immediate form.
4. Test for *not equal to*, *greater than*, or *greater than or equal to* by inverting the truth sense of the branch-on-carry instruction.

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

Purpose: To compare the contents of a scalar register and an immediate

Architecture: C100 Series, C200 Series

Format:



Operation:

```
if (Opcode_Test(Immediate,Sk) == TRUE) {
      SC = 1;
} else {
      SC = 0;
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
leu.h #N,Sk	1C08	ST 000111000	See note 1	Compare unsigned less than or equal to halfword
leu.w #N,Sk	1C88	ST 000111001	See note 1	Compare unsigned less than or equal to word
ltu.h #N,Sk	1D08	ST 000111010	See note 1	Compare unsigned less than halfword
ltu.w #N,Sk	1D88	ST 000111011	See note 1	Compare unsigned less than word

Description: The truth value of the unsigned comparison between the contents of the scalar register Sk and the (sign-extended) immediate replaces the Scalar Carry (PSW<SC>) bit. If the comparison is true, the PSW<SC> bit is set to 1; otherwise, if the comparison is false, the PSW<SC> bit is cleared to 0. Example: *ltu.w #1,S5* sets SC to 1 when the 1 is less than the contents of scalar register S5.

Notes:

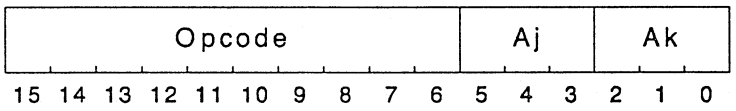
1. SC is affected as described by above operation pseudocode
2. Use the *eq.x* instruction for unsigned compares of equality.
3. Unsigned equal comparison is equivalent to signed equal comparison.
4. Test for *not equal to*, *greater than*, or *greater than or equal to* by inverting the truth sense of the branch-on-carry instruction.

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

Purpose: To compare the unsigned contents of two address registers

Architecture: C100 Series, C200 Series

Format:



Operation:

```
if (Opcode_Test(Aj,Ak) ==TRUE) {
     C = 1;
} else {
     C = 0;
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
leu.h Aj,Ak	4800	ST 0100100000	See note 1	Compare unsigned less than or equal to halfword
leu.w Aj,Ak	4840	ST 0100100001	See note 1	Compare unsigned less than or equal to word
ltu.h Aj,Ak	4A00	ST 0100101000	See note 1	Compare unsigned less than halfword
ltu.w Aj,Ak	4A40	ST 0100101001	See note 1	Compare unsigned less than word

Description: The truth value of the comparison between the contents of address register Ak and the contents of address register Aj replaces the Carry (PSW<C>) bit. If the comparison is true, the PSW<C> bit is set to 1; otherwise, if the comparison is false, the PSW<C> bit is cleared to 0. Example: *lt.h a2,a1* sets C to 1 if the contents of address register A2 are less than the contents of address register A1.

Notes:

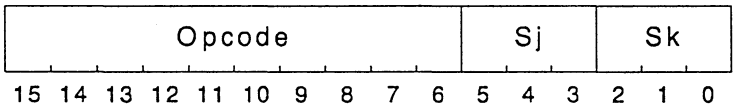
1. Carry (C) is affected as described by above operation pseudocode
2. Use the *eq.x* instruction for unsigned compares of equality.
3. Unsigned equal comparison is equivalent to signed equal comparison.
4. Test for *not equal to* by inverting the truth sense of the branch-on-carry instruction. Test for *greater than* or *greater than or equal to* by inverting the truth sense of the branch-on-carry instruction or by inverting the order of the operands.

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

Purpose: To compare the unsigned contents of two scalar registers

Architecture: C100 Series, C200 Series

Format:



Operation:

```
if (Opcode_Test(Sj,Sk) == TRUE) {
     SC = 1;
} else {
     SC = 0;
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
leu.b Sj,Sk	4900	ST 0100100100	See note 1	Compare less than or equal to byte
leu.h Sj,Sk	4940	ST 0100100101	See note 1	Compare less than or equal to halfword
leu.w Sj,Sk	4980	ST 0100100110	See note 1	Compare less than or equal to word
leu.l Sj,Sk	49C0	ST 0100100111	See note 1	Compare less than or equal to longword
ltu.b Sj,Sk	4B00	ST 0100101100	See note 1	Compare less than byte
ltu.h Sj,Sk	4B40	ST 0100101101	See note 1	Compare less than halfword
ltu.w Sj,Sk	4B80	ST 0100101110	See note 1	Compare less than word
ltu.l Sj,Sk	4BC0	ST 0100101111	See note 1	Compare less than longword

Description: The truth value of the comparison between the contents of scalar register Sk and the contents of scalar register Sj replaces the Scalar Carry (PSW<SC>) bit. If the comparison is true, the PSW<SC> bit is set to 1; otherwise, if the comparison is false, the PSW<SC> bit is cleared to 0. Example: *lt.h s3,s1* sets SC to 1 if the contents of scalar register S3 are less than the contents of S1.

Notes:

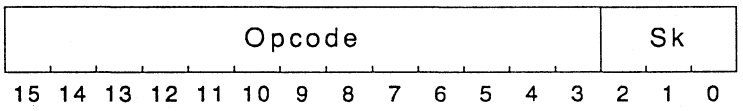
1. Scalar Carry (SC) is affected as described by above operation pseudocode
2. Use the *eq.x* instruction for unsigned compares of equality.
3. Unsigned equal comparison is equivalent to signed equal comparison.
4. Test for *not equal to* by inverting the truth sense of the branch-on-carry instruction. Test for *greater than* or *greater than or equal to* by inverting the truth sense of the branch-on-carry instruction or by inverting the order of the operands.

ln.(s|d) Sk — NATURAL LOGARITHM

Purpose: To compute the natural logarithm of the contents of a scalar register

Architecture: **C200 Series only**

Format:



Operation: `Sk = ln(Sk)`

Exceptions: (s|d): Floating Instrinsic Error

Opcode:

Mnemonic	Hex	Binary	PSW	Description
ln.s Sk	7C10	ST 0111110000010	FIN,IEC	Natural logarithm of a single precision number
ln.d Sk	7C18	ST 0111110000011	FIN,IEC	Natural logarithm of a double precision number

Description: The natural logarithm of the contents of Sk replaces the contents of Sk.

Notes:

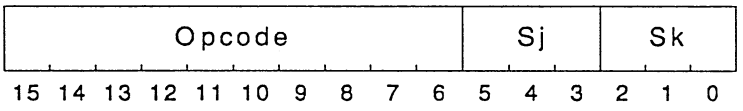
1. Intrinsic traps go through the same trap handler as other arithmetic traps (RO, FDZ, UN, etc.). If PSW <FUE> and/or PSW <FE> are set and and intrinsic traps are not (INE clear), these bits must be examined to determine the type of the current trap.
2. When PSW<FIN> is set, the PSW<IEC> bits contain a code that specifies the type of error encountered by the intrinsic instruction. Refer to the *CONVEX Architecture Reference*, "Register Set" chapter, "Program Status Word — C200 Series" section, for more information on the PSW<IEC> error codes and arithmetic trap conditions.

LEADING ONE'S POSITION SCALAR — lop Sj,Sk

Purpose: To calculate the leading one's position in a scalar register

Architecture: C100 Series, C200 Series

Format:



Operation:

```
for (a = 63; a >= 0; a--) {
     if (Sj<a> == 1) {
          break;  /* found leftmost 1, so break loop */
     }
}
Sk<7..0> = a;
Sk<63..8> = 0;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
lop Sj,Sk	61C0	ST 0110000111	None	Leading one's position in Sj

Description: The bit position of the leftmost 1 bit contained in Sj replaces the contents of Sk by scanning from bit <63> through bit <0>. If the most significant bit in Sj is set, 63 replaces the contents of Sk. If Sj is all zeros, then a ***byte*** value of -1 (0000 0000 0000 00FF) is loaded into Sk (as described above in the operation psuedocode).

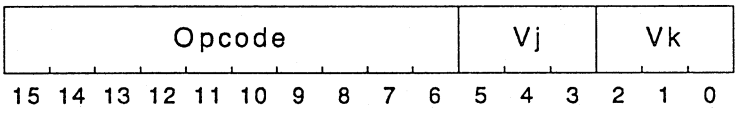
Notes: None

lop Vj,Vk — LEADING ONE'S POSITION VECTOR

Purpose: To calculate the leading one's position in each element of a vector register

Architecture: **C200 Series only**

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      for (b = 63; b >= 0; b--) {
          if (Vj[a]<b> == 1) {
              break;  /* found leftmost 1, so break loop */
          }
      }
      Vk[a]<7..0> = b;
      Vk[a]<63..8> = 0;
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
lop Vj,Vk	6240	ST 0110001001	None	Leading ones position vector

Description: The index number of the leftmost 1 bit of an element of Vj replaces replaces the first VL elements of Vk by scanning from bit <63> through bit <0>. If the most significant bit in an elment of Vj is set, 63 replaces the contents of VL elements of Vk. If an element of Vj is all zeros, then a ***byte*** value of -1 (0000 0000 0000 00FF) is loaded into VL elements of Vk (as described above in the operation psuedocode).

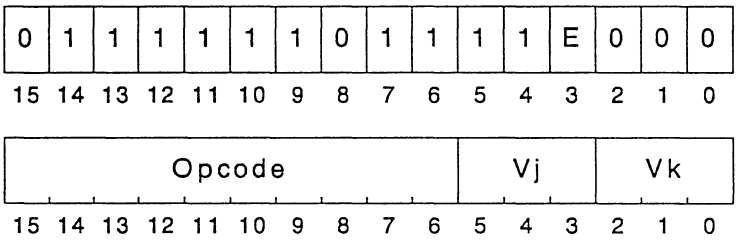
Notes: None

LEADING ONE'S POSITION VECTOR MASKED — lop.(t|f) Vj,Vk

Purpose: To calulate the leading one's position in each element of a vector register under control of the Vector Merge (VM) register

Architecture: **C200 Series only**

Format:



Operation:

```
switch (E) { /* prefix bit<3> */
     case TRUE:  /* .t */
         for (a = 0; a < VL; a++) {
             if (VM<a> == 1) {        /* if VM<a> is TRUE */
                 for (b = 63; b >= 0; b--) {
                     if (Vj[a]<b> == 1) {
                         break;  /* found leftmost 1 */
                     }
                 }
                 Vk[a]<7..0> = b;
                 Vk[a]<63..8> = 0;
             }   /* end if TRUE */
         }   /* end of for loop */
         break;  /* go to end of switch */
     case FALSE: /* .f */
         for (a = 0; a < VL; a++) {
             if (VM<a> == 0) {        /* if VM<a> is FALSE */
                 for (b = 63; b >= 0; b--) {
                     if (Vj[a]<b> == 1) {
                         break;  /* found leftmost 1 */
                     }
                 }
                 Vk[a]<7..0> = b;
                 Vk[a]<63..8> = 0;
             }   /* end if FALSE */
         }   /* end of for loop */
         break;  /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
lop.t Vj,Vk	6240	E1 0110001001	None	Leading ones position vector (VM)
lop.f Vj,Vk	6240	E0 0110001001	None	Leading ones position vector (!VM)

Description: The index number of the leftmost bit 1 bit of an element of Vj replaces replaces the first VL elements of Vk if the corresponding VM bit is set (clear for *.f*) by scanning from bit <63> through bit <0>. If the most significant bit in an elment of Vj is set, 63 replaces the contents of VL elements of Vk if the corresponding VM bit is set (clear for *.f*). If an element of Vj is all zeros, then a ***byte*** value of –1 (0000 0000 0000 00FF) is loaded into VL elements of Vk (as described above in the operation psuedocode).

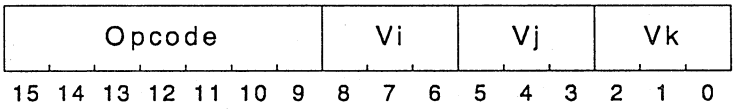
Notes: None

mask.t Vi,Vj,Vk

Purpose: To mask the elements of two vectors to the elements of a third vector

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
for (a = 0; a < VL); a++) {
      if (VM<a> == 1) {    /* if VM<a> is TRUE */
          Vk[a]= Vj[a];
      } else {
          Vk[a]= Vi[a];
      }
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mask.t Vi,Vj,Vk	8600	ST 1000011	None	Mask vector/vector

Description: Depending on the bits of the VM register, each of the first VL elements of vector register Vk is replaced by the corresponding element of Vi (if VM<a> is 1) or Vj (if VM<a> is 0).

Notes: Interchanging Vi and Vj is equivalent to using a complemented VM.

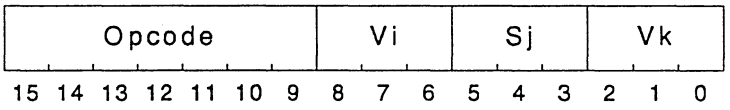
MASK VECTOR/SCALAR

mask.(t|f) Vi,Sj,Vk

Purpose: To mask a scalar to a vector

Architecture: C100 Series, C200 Series

Format:



Operation:

```
switch (opcode<10>) { /* opcode bit<10> */
     case TRUE:  /* .t */
         for (a = 0; a < VL; a++) {
             if (VM<a> == 1) {        /* if VM<a> is TRUE */
                 Vk[a]= Sj;
             } else {
                 Vk[a]= Vi[a];
             }
         }   /* end of for loop */
         break;  /* go to end of switch */
     case FALSE: /* .f */
         for (a = 0; a < VL; a++) {
             if (VM<a> == 0) {        /* if VM<a> is FALSE */
                 Vk[a]= Sj;
             } else {
                 Vk[a]= Vi[a];
             }
         }   /* end of for loop */
         break;  /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mask.t Vi,Sj,Vk	8E00	ST 1000111	None	Mask vector/scalar (VM)
mask.f Vi,Sj,Vk	8A00	ST 1000101	None	Mask vector/scalar (!VM)

Description: Each of the first VL elements of vector register Vk is replaced either by the 64 bits of scalar register Sj or the 64 bits of the corresponding element in Vj, depending on the value of the VM register bit corresponding to the vector elements. The *mask.t* and *mask.f* instructions differ only in their sense of interpreting the vector merge (VM) bit.

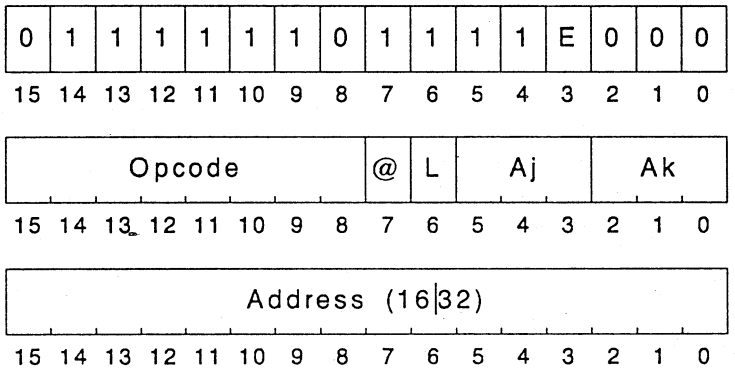
Notes: Load a scalar into all elements of a vector register by setting VM to all 1's and using *mask.t* (or use *mask.f* with a VM of all 0's).

mat.w Ak,<Ceffa> MATCH ADDRESS/COMMUNICATION

Purpose: Compare the address register contents with the communication register

Architecture: C200 Series only

Format:



Operation:

```
if ( c(Ceffa) == Ak ) {
      C = 1;
} else {
      C = 0;
}
```

Exceptions: Ring Violation (Invalid Communication Register Address)
Deadlock Exception

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mat.w Ak,<Ceffa>	2900	E0 0010100100	-C,CAT	Match address/communication

Description: Bits <31..0> of *c(Ceffa)* are compared to Ak; C is set if the two values are equal, otherwise, C is cleared if the two bits are not equal. Bits <63..32> of *c(Ceffa)* are not used. *c(Ceffa)* and *L(Ceffa)* are not modified.

Notes:

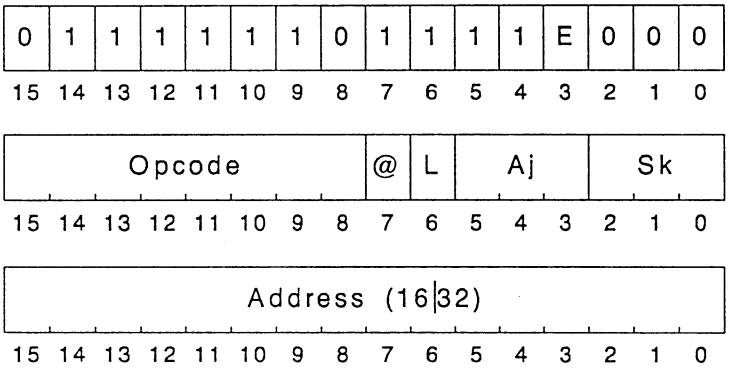
1. Address Carry (C) is affected as described by the preceding operation pseudocode.
2. This instruction provides the same functionality as an *equal* test followed by a branch on carry except that *mat.w* provides deadlock detection capability.

MATCH SCALAR/COMMUNICATION — mat.l Sk,<Ceffa>

Purpose: Compare the scalar register contents with the communication register

Architecture: C200 Series only

Format:



Operation:

```
if ( c(Ceffa) == Sk ) {
     SC = 1;
} else {
     SC = 0;
}
```

Exceptions: Ring Violation (Invalid Communication Register Address)
Deadlock Exception

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mat.l Sk,<Ceffa>	3100	E0 0011000100	SC,CAT	Match scalar/communication

Description: *c(Ceffa)* is compared to Sk; SC is set if the two values are equal, otherwise, SC is cleared if the two values are not equal. *c(Ceffa)* and *L(Ceffa)* are not modified.

Notes:

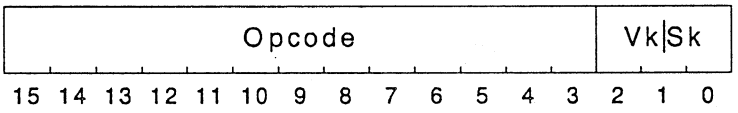
1. Scalar Carry (SC) is affected as described by the preceding operation pseudocode.
2. This instruction provides the same functionality as an *equal* test followed by a branch on carry except that *mat.l* provides deadlock detection capability.

max.(b|h|w|l|s|d) (Vk|Sk) — MAX VECTOR

Purpose: To find the maximum element of a vector register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      if (Vk[a] > Sk) {
          Sk = Vk[a];
      }
}
```

Exceptions: (b|h|w|l): None

(s|d): Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
max.b Vk	7E40	ST 0111111001000	None	Maximum of a vector of bytes
max.h Vk	7E48	ST 0111111001001	None	Maximum of a vector of halfwords
max.w Vk	7E50	ST 0111111001010	None	Maximum of a vector of words
max.l Vk	7E58	ST 0111111001011	None	Maximum of a vector of longwords
max.s Vk	7EA0	ST 0111111010100	RO	Maximum of a vector of single float
max.d Vk	7EA8	ST 0111111010101	RO	Maximum of a vector of double float

Description: The maximum of scalar register Sk and the first VL elements of vector register Vk replace Sk.

Notes:

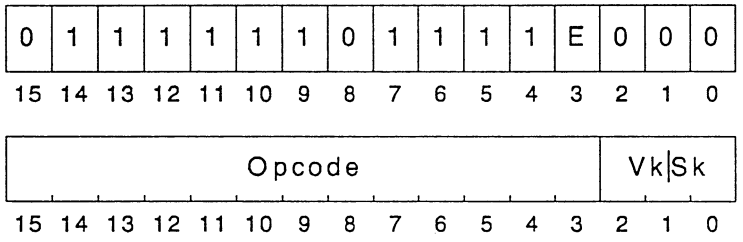
1. Initialize the scalar register Sk to the minimum value for the first use of the *max* instruction.
2. Either Vk or Sk may be used as a valid argument to this instruction. This instruction operates in distinct matched vector and scalar register pairs: (V0,S0), (V1,S1), (V2,S2), (V3,S3), (V4,S4), (V5,S5), (V6,S6), (V7,S7)

MAX VECTOR MASKED — max.(b|h|w|l|s|d).(t|f) (Vk|Sk)

Purpose: To find the maximum element of a subset of a vector register under control of the Vector Merge (VM) register

Architecture: C200 Series only

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
     case TRUE:   /*  .t */
          for (a = 0; a < VL; a++) {
               if (VM<a> == 1) {        /* if VM<a> is TRUE */
                    if (Vk[a] > Sk) {
                         Sk = Vk[a];
                    }
               }
          }   /* end of for loop */
          break;  /* go to end of switch */
     case FALSE:  /*  .f */
          for (a = 0; a < VL; a++) {
               if (VM<a> == 0) {        /* if VM<a> is FALSE */
                    if (Vk[a] > Sk) {
                         Sk = Vk[a];
                    }
               }
          }   /* end of for loop */
          break;  /* go to end of switch */
} /* end of switch */
```

Exceptions:

(b|h|w|l): None

(s|d): Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
max.b.t Vk	7E40	E1 0111111001000	None	Maximum of vector of bytes (VM)
max.b.f Vk	7E40	E0 0111111001000	None	Maximum of vector of bytes (!VM)
max.h.t Vk	7E48	E1 0111111001001	None	Maximum of vector of halfwords (VM)
max.h.f Vk	7E48	E0 0111111001001	None	Maximum of vector of halfwords (!VM)
max.w.t Vk	7E50	E1 0111111001010	None	Maximum of vector of words (VM)
max.w.f Vk	7E50	E0 0111111001010	None	Maximum of vector of words (!VM)
max.l.t Vk	7E58	E1 0111111001011	None	Maximum of vector of longwords (VM)
max.l.f Vk	7E58	E0 0111111001011	None	Maximum of vector of longwords (!VM)
max.s.t Vk	7EA0	E1 0111111010100	RO	Maximum of vector of singles (VM)
max.s.f Vk	7EA0	E0 0111111010100	RO	Maximum of vector of singles (!VM)
max.d.t Vk	7EA8	E1 0111111010101	RO	Maximum of vector of doubles (VM)
max.d.f Vk	7EA8	E0 0111111010101	RO	Maximum of vector of doubles (!VM)

Description: The maximum of scalar register Sk and the first VL elements of vector register Vk replace Sk. Only elements of Vk with corresponding VM bit set (clear for *.f*) participate in the search.

Notes:

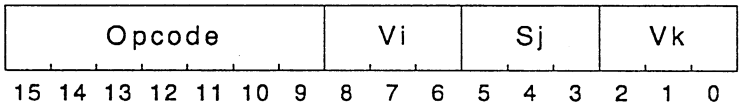
1. Initialize the scalar register Sk to the minimum value for the first use of the *max* instruction.
2. Either Vk or Sk may be used as a valid argument to this instruction. This instruction operates in distinct matched vector and scalar register pairs: (V0,S0), (V1,S1), (V2,S2), (V3,S3), (V4,S4), (V5,S5), (V6,S6), (V7,S7)

MERGE VECTOR/SCALAR — merg.(t|f) Vi,Sj,Vk

Purpose: To merge the contents of a scalar register and a vector register into the contents of a vector register

Architecture: C100 Series, C200 Series

Format:



Operation:

```
b = 0;
switch (opcode<10>) { /* opcode bit<10> */
     case TRUE:  /* .t */
          for (a = 0; a < VL; a++) {
               if (VM<a> == 1) {        /* if VM<a> is TRUE */
                    Vk[a] = Sj;
               } else {
                    Vk[a] = Vi[b];
                    b = b + 1;
               }
          }   /* end of for loop */
          break;  /* go to end of switch */
     case FALSE: /* .f */
          for (a = 0; a < VL; a++) {
               if (VM<a> == 0) {        /* if VM<a> is FALSE */
                    Vk[a] = Sj;
               } else {
                    Vk[a] = Vi[b];
                    b = b + 1;
               }
          }   /* end of for loop */
          break;  /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
merg.t Vi,Sj,Vk	8C00	ST 1000110	None	Merge vector/scalar
merg.f Vi,Sj,Vk	8800	ST 1000100	None	Merge vector/scalar (!VM)

Description: Depending on the value of the VM register, the first VL elements of vector register Vk are replaced either by Sj (when VM<a> is 1) or by the next uncopied element of Vi (if VM<a> is 0).

Notes:

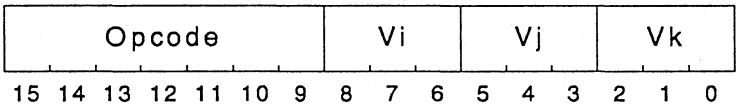
1. If Vk is the ***same*** register as Vi, then the behavior of this instruction is ***undefined.***
2. Typically, this operation uncompresses a vector compressed with one of the *cprs* instructions.

merg.t Vi,Vj,Vk — MERGE VECTOR/VECTOR

Purpose: To merge the contents of two vector registers into a third vector register

Architecture: C100 Series, C200 Series

Format:



Operation:

```
a = 0;
b = 0;
for (n = 0; n < VL; n++) {
     if (VM<n> == 1) {          /* if VM<n> is TRUE */
         Vk[n]; = Vj[b];
         b = b + 1;
     } else {
         Vk[n] = Vi[a];
         a = a + 1;
     }
}    /* end of for loop */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
merg.t Vi,Vj,Vk	8400	ST 1000010	None	Merge vector/vector

Description: Depending on the value of the VM register, the first VL elements of vector register Vk are replaced either by the next uncopied element of Vj (when VM<a> is 1) or the next uncopied element of Vi (when VM<a> is 0).

Notes:

1. The merge provides a convenient means to reassemble operands from two vectors into one vector. Typically, the operands were initially scrambled using a compress operation.
2. Merge using a complemented VM is equivalent to merge with Vi and Vj interchanged.
3. If Vk is the ***same*** vector register as Vi or Vj, then the behavior of this instruction is ***undefined***.

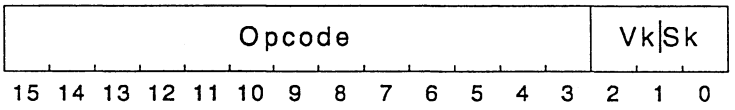
MIN VECTOR

min.(b|h|w|l|s|d) (Vk|Sk)

Purpose: To find the minimum element of a vector

Architecture: C100 Series, C200 Series

Format:



Operation:

```
for (a = 0; a < VL; a++) {
    if (Vk[a] < Sk) {
        Sk = Vk[a];
    }
}
```

Exceptions:

(b|h|w|l): None

(s|d): Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
min.b Vk	7E60	ST 0111111001100	None	Minimum of a vector of bytes
min.h Vk	7E68	ST 0111111001101	None	Minimum of a vector of halfwords
min.w Vk	7E70	ST 0111111001110	None	Minimum of a vector of words
min.l Vk	7E78	ST 0111111001111	None	Minimum of a vector of longwords
min.s Vk	7EB0	ST 0111111010110	RO	Minimum of a vector of single float
min.d Vk	7EB8	ST 0111111010111	RO	Minimum of a vector of double float

Description: The minimum of scalar register Sk and the first VL elements of vector register Vk replace Sk.

Notes:

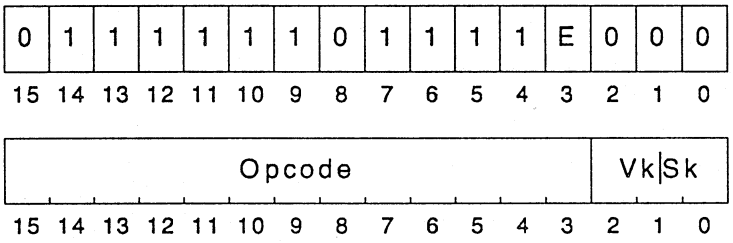
1. Initialize the scalar register Sk to the maximum value for the first use of the *min* instruction.
2. Either Vk or Sk may be used as a valid argument to this instruction. This instruction operates in distinct matched vector and scalar register pairs: (V0,S0), (V1,S1), (V2,S2), (V3,S3), (V4,S4), (V5,S5), (V6,S6), (V7,S7)

min.(b|h|w|l|s|d).(t|f) (Vk|Sk) — MIN VECTOR MASKED

Purpose: To find the minimum element of a subset of a vector register under control of the Vector Merge (VM) register

Architecture: **C200 Series only**

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
      case TRUE:   /*  .t */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 1) {        /* if VM<a> is TRUE */
                  if (Vk[a] < Sk) {
                      Sk = Vk[a];
                  }
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
      case FALSE:  /*  .f */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 0) {        /* if VM<a> is FALSE */
                  if (Vk[a] < Sk) {
                      Sk = Vk[a];
                  }
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
} /* end of switch */
```

Exceptions: (b|h|w|l): None

(s|d): Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
min.b.t Vk	7E60	E1 0111111001100	None	Minimum of vector of bytes (VM)
min.b.f Vk	7E60	E0 0111111001100	None	Minimum of vector of bytes (!VM)
min.h.t Vk	7E68	E1 0111111001101	None	Minimum of vector of halfwords (VM)
min.h.f Vk	7E68	E0 0111111001101	None	Minimum of vector of halfwords (!VM)
min.w.t Vk	7E70	E1 0111111001110	None	Minimum of vector of words (VM)
min.w.f Vk	7E70	E0 0111111001110	None	Minimum of vector of words (!VM)
min.l.t Vk	7E78	E1 0111111001111	None	Minimum of vector of longwords (VM)
min.l.f Vk	7E78	E0 0111111001111	None	Minimum of vector of longwords (!VM)
min.s.t Vk	7EB0	E1 0111111010110	RO	Minimum of vector of singles (VM)
min.s.f Vk	7EB0	E0 0111111010110	RO	Minimum of vector of singles (!VM)
min.d.t Vk	7EB8	E1 0111111010111	RO	Minimum of vector of doubles (VM)
min.d.f Vk	7EB8	E0 0111111010111	RO	Minimum of vector of doubles (!VM)

Description: The minimum of scalar register Sk and the first VL elements of vector register Vk replace Sk. Only elements of Vk with corresponding VM bit set (clear for *.f*) participate in the search.

Notes:

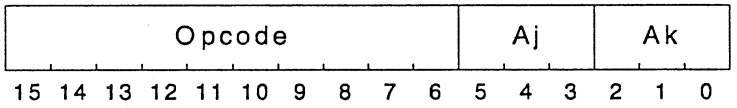
1. Initialize the scalar register Sk to the maximum value for the first use of the *min* instruction.
2. Either Vk or Sk may be used as a valid argument to this instruction. This instruction operates in distinct matched vector and scalar register pairs: (V0,S0), (V1,S1), (V2,S2), (V3,S3), (V4,S4), (V5,S5), (V6,S6), (V7,S7)

mov Aj,Ak — MOVE ADDRESS/ADDRESS

Purpose: To move (copy) the contents of one address register to another address register

Architecture: C100 Series, C200 Series

Format:



Operation:

```
Ak = Aj;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov Aj,Ak	5080	ST 0101000010	None	Move address register

Description: The 32 bits of address register Aj replace the contents of Ak.

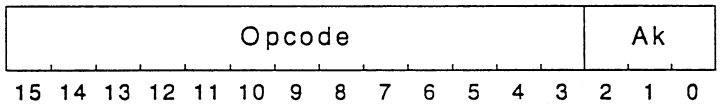
Notes: None

MOVE ADDRESS/PSW — mov Ak,PSW

Purpose: To move (copy) an address register to the Program Status Word (PSW)

Architecture: C100 Series, C200 Series

Format:



Operation:

```
PSW = Ak;
```

Exceptions: Any exceptions as dictated by the new contents of the PSW

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov Ak,PSW	7C48	ST 0111110001001	All bits	Load an address register into the PSW

Description: The contents of address register Ak replace the contents of the PSW.

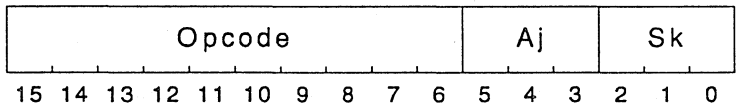
Notes: Before Ak is copied to PSW, all existing concurrent processing is completed. All exception flags and trap enables that are generated during concurrent processing are copied to the PSW before this instruction will execute. This ensures that the sequential state of the processor is accurately reflected and the appropriate action is taken for exception handling and trap processing.

mov Aj,Sk — MOVE ADDRESS/SCALAR

Purpose: To move (copy) the contents of an address register into a scalar register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Sk<31..0> = Aj;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov Aj,Sk	51C0	ST 0101000111	None	Move an address to a scalar

Description: The 32 bits of address register Aj replace the least significant 32 bits of scalar register Sk. The most significant 32 bits of Sk remain unchanged.

Notes: None

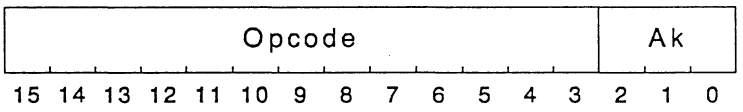
MOVE ADDRESS/VL

mov Ak,VL

Purpose: To move (copy) the contents of an address register to the Vector Length (VL) register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
if (Ak > 128) {
      VL = 128;
} else {
      if (Ak < 0) {
          VL = 0;
      } else {
          VL = Ak;
      }
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov Ak,VL	7D98	ST 0111110110011	None	Move Ak to VL

Description: The *mov Ak,VL* instruction replaces the contents of the VL register with the contents of address register Ak bracketed to the range 0..128.

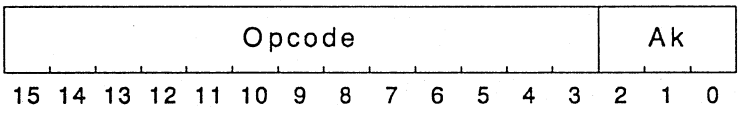
Notes: None

mov Ak,VS

Purpose: To move (copy) the contents of an address register to the Vector Stride (VS) register

Architecture: **C100 Series, C200 Series**

Format:



Operation: `VS = Ak;`

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov Ak,VS	7D88	ST 0111110110001	None	Move Ak to VS

Description: The contents of address register Ak replace the contents of VS.

Notes: None

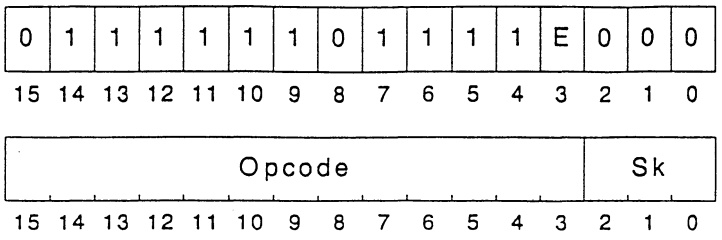
MOVE CIR/SCALAR

mov CIR,Sk

Purpose: To move Communication Index Register (CIR) to a scalar register

Architecture: C200 Series only

Format:



Operation: `Sk = CIR;`

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov CIR,Sk	7C08	E0 0111110000	None	Move CIR a scalar

Description: This instruction copies the zero-extended CIR for this CPU to Sk.

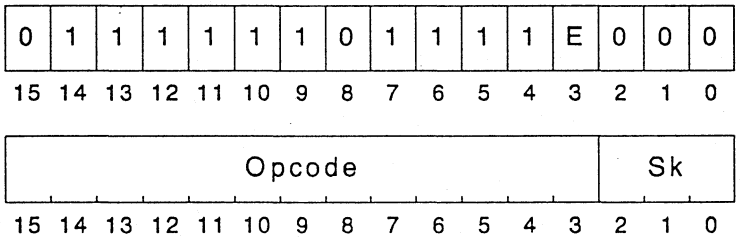
Notes: None

mov CPUID,Sk

Purpose: To move the read only CPUID register to a scalar register

Architecture: C200 Series only

Format:



Operation:

```
Sk = CPUID;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov CPUID,Sk	7C18	E0 0111110000	None	Move CPU identification to scalar

Description: This instruction copies the CPUID for this CPU into Sk. The value moved to Sk is a zero-extended longword between 0 and the maximum configurable CPU number.

Notes: None

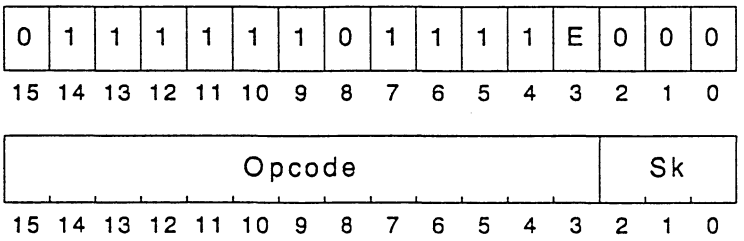
MOVE ICR/SCALAR

mov ICR,Sk

Purpose: To move the Interrupt Control Register (ICR) to a scalar register

Architecture: C200 Series only

Format:



Operation:

```
Sk = ICR.
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov ICR,Sk	7C68	E0 0111110001	None	Move ICR to scalar

Description: This instruction is used to read the ICR, which contains global interrupt control information for all CPUs.

Notes: The format of the ICR is described in the *CONVEX Architecture Reference*, "Exceptions and Interrupts" chapter.

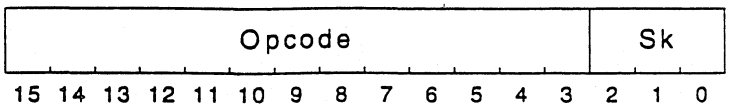
mov ITR,Sk

MOVE ITR/SCALAR

Purpose: To move (copy) the contents of the interval timer registers (ITC, NITC, and ITSR) to a scalar register

Architecture: **C100 Series**

Format:



Operation:

```
Sk<63..60> = ITSR;
Sk<59..40> = NITC;
Sk<27..8> = ITC;
```

Exceptions: Ring Violation (Privileged Instruction)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov ITR,Sk	7C60	ST 0111110001100	None	Move the ITC, ITSR, NITC into Sk

Description: The contents of the interval timer status register replace the most significant four bits of scalar register Sk. The contents of the next interval timer counter replace bits <59..40> of Sk. The contents of the interval timer counter replace bits <27..8> of Sk.

Notes: None

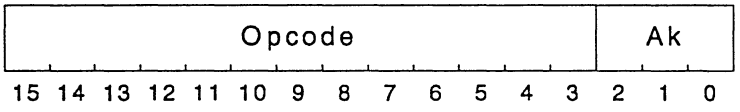
MOVE PC/ADDRESS

mov PC,Ak

Purpose: To move (copy) the address of the next instruction into an address register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Ak = Current_Address + 2;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov PC,Ak	7C50	ST 0111110001010	None	Load the next PC address

Description: The address of the instruction following the mov PC,Ak instruction replaces the contents of address register Ak.

Notes: None

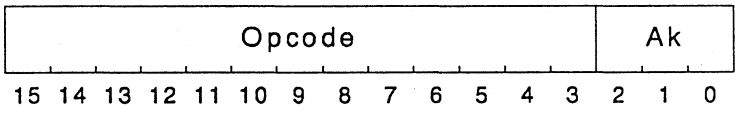
mov PSW,Ak

MOVE PSW/ADDRESS

Purpose: To move (copy) the Processor Status Word (PSW) to an address register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Ak = PSW;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov PSW,Ak	7C40	ST 0111110001000	None	Store the PSW into an address register

Description: The contents of the PSW replace the contents of address register Ak.

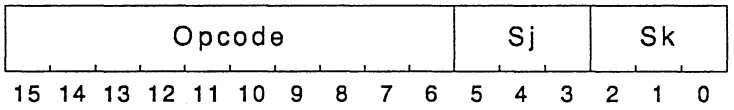
Notes: Before the PSW is moved to Ak, all existing concurrent processing is completed. This ensures that all exception condition flags accurately reflect the state of the processor.

MOVE SCALAR/SCALAR — mov.(w|l|s|d) Sj,Sk

Purpose: To move (copy) the contents of one scalar register to another

Architecture: C100 Series, C200 Series

Format:



Operation:

```
Sk<63..0> = Sj<63..0>;     /* mov.l, mov.d */
Sk<31..0> = Sj<31..0>;     /* mov.s, mov.w */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov.l Sj,Sk	5180	ST 0101000110	None	Move scalar register longword
mov.w Sj,Sk	5100	ST 0101000100	None	Move scalar register word
mov.d Sj,Sk	5180	ST 0101000110	None	Move scalar register single float
mov.s Sj,Sk	5100	ST 0101000100	None	Move scalar register double float

Description: The specified portion of scalar register Sj replaces the corresponding portion of Sk. The rest of the bits remain unchanged.

Notes: The *.s* and *.d* forms rename the *.w* and *.l* forms, respectively, for convenience.

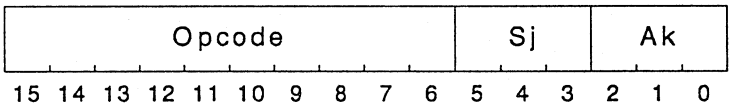
mov Sj,Ak

MOVE SCALAR/ADDRESS

Purpose: To move (copy) the contents of a scalar register into an address register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Ak = Sj<31..0>;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov Sj,Ak	50C0	ST 0101000011	None	Move 32 bits of Sj into Ak

Description: The least significant 32 bits of scalar register Sj replace the contents of address register Ak.

Notes: None

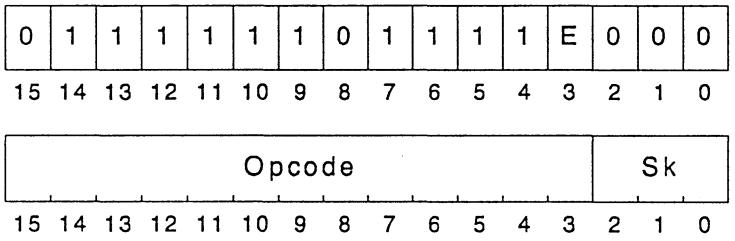
MOVE SCALAR/CIR

mov Sk,CIR

Purpose: To move scalar register to the Communication Index Register (CIR)

Architecture: C200 Series only

Format:



Operation: `CIR = Sk;`

Exceptions: Ring Violation (Privileged Instruction)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov Sk,CIR	7C00	E0 0111110000	None	Move scalar to CIR

Description: This instruction is used to associate a specific communication register set with the current CPU. Sk contains an integer index to one of the communication register sets.

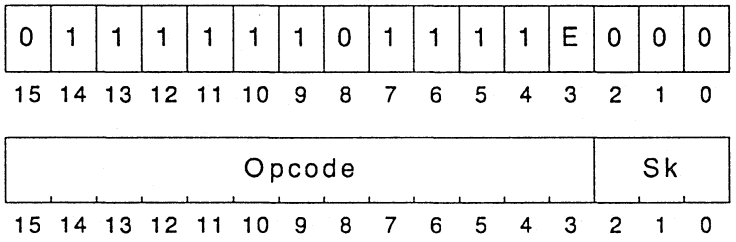
Notes: The thread count for the previously loaded CIR is ***not decremented.*** The thread count in the new CIR is ***not incremented.*** This can result in inconsistent thread allocation if care is not taken. The *mov Sk,TID* instruction must be executed to initialize the thread in the new CIR.

mov Sk,ICR

Purpose: To move scalar register to the Interrupt Control Register (ICR)

Architecture: **C200 Series only**

Format:



Operation: `ICR = Sk;`

Exceptions: Ring Violation (Privileged Instruction)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov Sk,ICR	7C60	E0 0111110001	None	Move Scalar to ICR

Description: This instruction is used to set the ICR, which contains global interrupt control information for all CPUs.

Notes:

1. The format of the ICR is detailed in the CONVEX Architecture Reference, "Exceptions and Interrupts" chapter.
2. Interrupts must be successfully disabled using *dsi* before this instruction is executed. Otherwise, unpredictable results can occur.

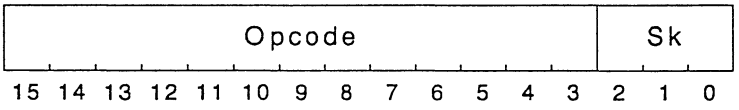
MOVE SCALAR/ITR

mov Sk,ITR

Purpose: To move the contents of a scalar register to the interval timer registers (NITC, ITSR, and ITC)

Architecture: C100 Series only

Format:



Operation:

```
ITSR = Sk<63..60>;
NITC = Sk<59..40>;
ITC = Sk<27..8>;
```

Exceptions: Ring Violation (Privileged Instruction)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov Sk,ITR	7C68	ST 0111110001101	None	Load NITC, ITC, ITSR from Sk

Description: Scalar register Sk bits <63..60> replace the contents of the interval timer status register. Bits Sk <59..40> replace the contents of the next iteration counter. Bits Sk <27..8> replace the contents of the interval timer counter.

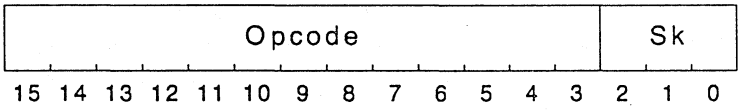
Notes: None

mov Sk,ITSR

Purpose: To move (copy) the four most significant bits of a scalar register to the ITSR register

Architecture: **C100 Series only**

Format:



Operation:

```
ITSR = Sk<63..60>;
```

Exceptions: Ring Violation (Privileged Instruction)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov Sk,ITSR	7C78	ST 0111110001111	None	Load ITSR with a scalar

Description: The most significant four bits of scalar register Sk replace the the interval timer status register.

Notes: None

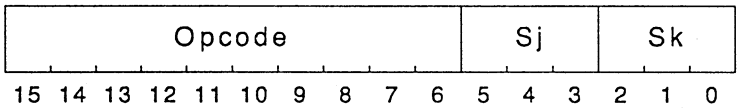
MOVE SCALAR/VM

mov Sj,Sk,VM

Purpose: To move (copy) the contents from a scalar register to the Vector Merge (VM) register.

Architecture: C100 Series, C200 Series

Format:



Operation:

```
if (Sj<0> == 0) {
      VM<63..0> = Sk;
} else {
      VM<127..64> = Sk;
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov Sj,Sk,VM	6100	ST 0110000100	None	Load VM(Sj) from Sk

Description: If Sj<0> is zero, then replace the contents of the Vector Merge register (VM<63..0>) with scalar register Sk. If Sj<0> is one, then replace the contents of VM<127..64> with the contents of scalar register Sk.

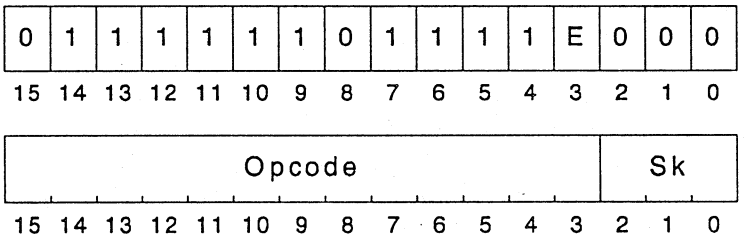
Notes: None

mov Sk,TCPU

Purpose: To move scalar register to the interrupt Target CPU (TCPU) register

Architecture: **C200 Series only**

Format:



Operation: `TCPU = Sk;`

Exceptions: Ring Violation (Privileged Instruction)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov Sk,TCPU	7C70	E0 0111110001	None	Move Scalar to TCPU register

Description: This instruction is used to set the interrupt TCPU register. The TCPU register is set to all ones to indicate that any CPU may be selected for interrupt delivery, or to a valid CPUID to vector all interrupts to the CPU associated with the specified CPUID.

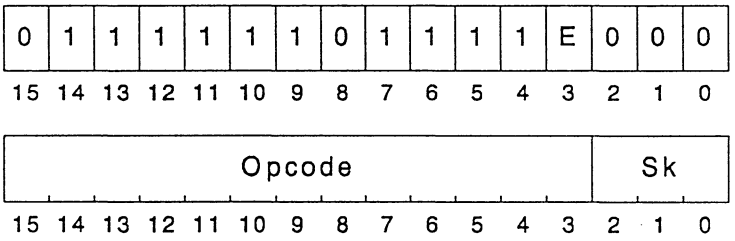
Notes: Interrupts must be successfully disabled using *dsi* before this instruction is executed. Otherwise, unpredictable results can occur.

MOVE SCALAR/THREAD ID — mov Sk,TID

Purpose: To move the contents of a scalar to the Thread ID (TID) register

Architecture: C200 Series only

Format:



Operation:

```
THREAD_ID = Sk;
```

Exceptions: Ring Violation (Privileged Instruction)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov Sk,TID	7CB0	E0 0111110010	None	Load TID from scalar

Description: This instruction sets the current CPU TID register to the value in Sk.

Notes: The TID being loaded should not be allocatable in the Thread Allocation Mask.

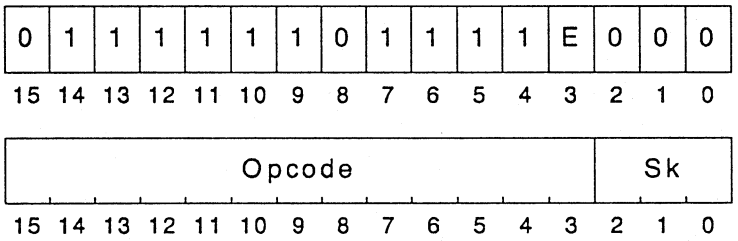
mov Sk,TTR

MOVE SCALAR/THREAD TIMER

Purpose: To move scalar register to Thread Timer (TTR) register

Architecture: **C200 Series only**

Format:



Operation:

```
THREAD_TIMER = Sk;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov Sk,TTR	7C20	E0 0111110000	None	Move scalar to TTR

Description: This instruction is used to initialize the current TTR register from a scalar register.

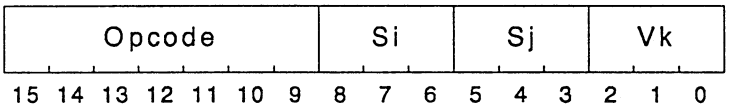
Notes: None

MOVE SCALAR/VECTOR ELEMENT — mov Si,Sj,Vk

Purpose: To move (copy) the contents of a scalar register to a vector register element

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Vk[Sj<6..0>] = Si;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov Si,Sj,Vk	8200	ST 1000001	None	Move a scalar to a vector element

Description: The contents of scalar register Si replace the contents of element Vk[Sj<6..0>]. Bits Sj<63..7> are ignored.

Notes: None

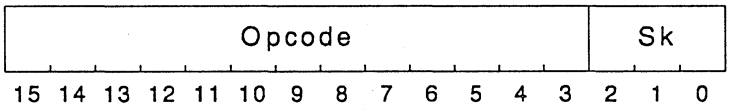
mov.w Sk,VL

MOVE SCALAR/VL

Purpose: To move (copy) the contents of a scalar register (Sk) to the Vector Length (VL) register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
if (Sk >= 128) {
     VL = 128;
} else {
     if (Sk < 0) {
          VL = 0;
     } else {
          VL = Sk;
     }
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov.w Sk,VL	7DB8	ST 0111110110111	None	Move Sk to VL

Description: The *mov Sk,VL* instruction replaces the contents of the VL register with the contents of scalar register Sk, bracketed to the range [0...128].

Notes: None

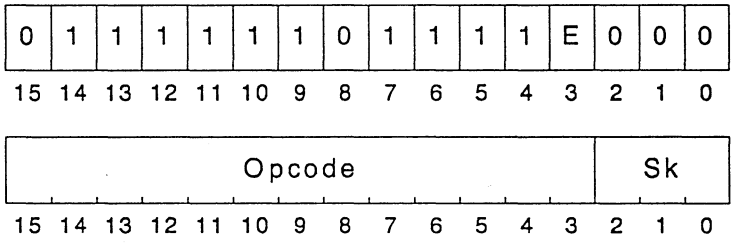
MOVE SCALAR/VM LOWER

mov Sk,VML

Purpose: To move a scalar register to the lower longword of the Vector Merge (VM) register

Architecture: C200 Series only

Format:



Operation:

```
VM<63..0> = Sk;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov Sk,VML	7C50	E0 0111110001010	None	Load VM<63..0> from Sk

Description: The *mov Sk,VML* instruction replaces The contents of the lower longword of VM with the contents of Sk.

Notes: None

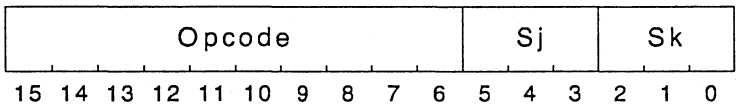
mov Sj,VM,Sk

MOVE VM/SCALAR

Purpose: To move (copy) half of the contents from the Vector Merge (VM) register to a scalar register.

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
if (Sj<0> == 0) {
      Sk = VM<63..0>;
} else {
      Sk = VM<127..64>;
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov Sj,VM,Sk	6140	ST 0110000101	None	Load Sk from VM(Sj)

Description: If Sj<0> is zero, then replace the contents of scalar register Sk with the contents of VM<63..0>. If Sj<0> is one, then replace the contents of scalar register Sk with the contents of Vector Merge (VM)<127..64>.

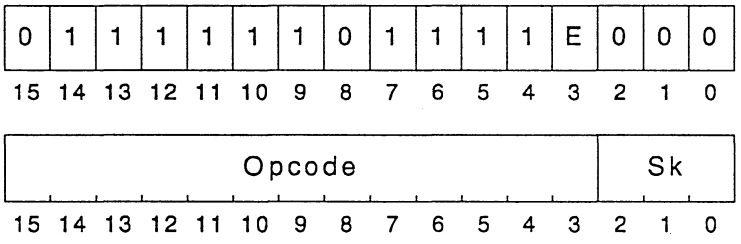
Notes: None

MOVE SCALAR/VM UPPER — mov Sk,VMU

Purpose: To move a scalar register to the upper longword of the Vector Merge (VM) register

Architecture: C200 Series only

Format:



Operation:

```
VM<127..64> = Sk;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov Sk,VMU	7C40	E0 0111110001000	None	Load VM<127..64> from Sk

Description: The *mov Sk, VMU* instruction replaces the contents of the upper longword of VM with the contents of Sk.

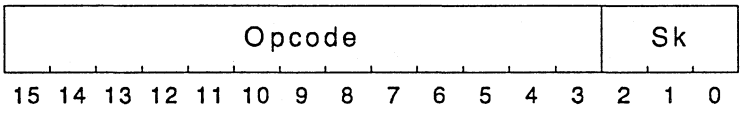
Notes: None

mov.w Sk,VS

Purpose: To move (copy) the contents of a scalar register Sk to the Vector Stride (VS) register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
VS = Sk<31..0>;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov.w Sk,VS	7DA8	ST 0111110110101	None	Move Sk to VS

Description: The least significant 32 bits of scalar register Sk replace the contents of VS.

Notes: None

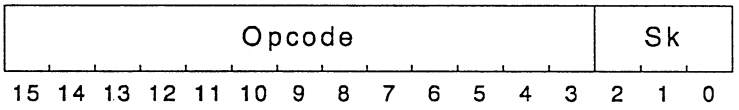
MOVE SCALAR/VV

mov Sk,VV

Purpose: To move (copy) the least significant bit of a scalar register to the Vector Valid (VV) flag

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
VV = Sk<0>;
```

Exceptions: Ring Violation (Privileged Instruction)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov Sk,VV	7D70	ST 0111110101110	None	Move scalar to vector valid flag

Description: The least significant bit of scalar register Sk replaces the VV flag.

Notes:

C100 Series only

1. The current ring of execution must be 0 or a privileged instruction exception will occur.

C200 Series only

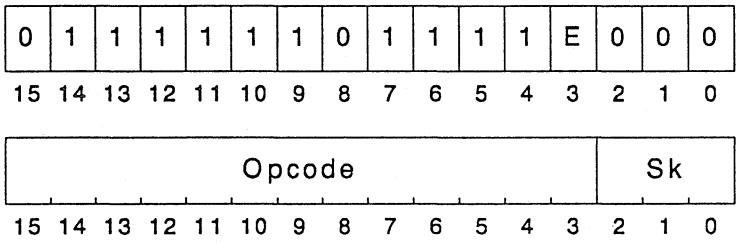
2. The *mov Sk, VV* instruction is NOT privileged, in order to allow Ring 4 (user) software to schedule the vector accumulators. This is incompatible with previous implementations of the CONVEX architecture.

mov TCPU,Sk

Purpose: To move the interrupt Target CPU (TCPU) register to a scalar register

Architecture: **C200 Series only**

Format:



Operation: `Sk = TCPU;`

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov TCPU,Sk	7C78	E0 0111110001	None	Move the TCPU Register to Scalar

Description: This instruction is used to read the interrupt TCPU register, which contains either a valid Target CPUID or all ones.

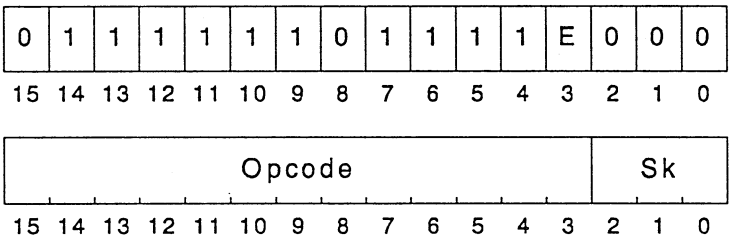
Notes: None

MOVE THREAD ID/SCALAR — mov TID,Sk

Purpose: To move the Thread ID (TID) register to a scalar register

Architecture: **C200 Series only**

Format:



Operation:

```
Sk = THREAD_ID;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov TID,Sk	7CB8	E0 0111110010	None	Load scalar with TID

Description: This instruction copies the TID for this CPU into Sk. The value moved to Sk is a zero-extended longword between 0 and the maximum number of threads.

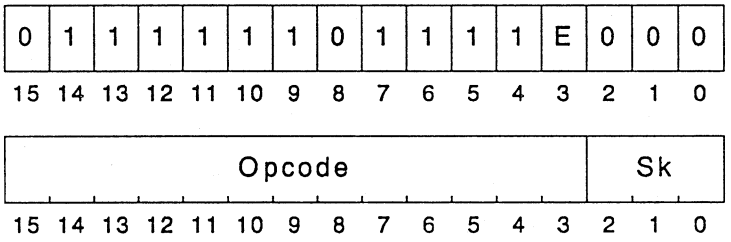
Notes: None

mov TOC,Sk

Purpose: To move the Time Of Century (TOC) counter to a scalar register

Architecture: **C130, C200 Series only**

Format:



Operation:

```
Sk = TOC;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov TOC,Sk	7C10	E0 0111110000	None	Move TOC to a scalar

Description: This instruction reads a pointer to the TOC clock from Ring 0 Page 0, and then combines the four 16-bit I/O locations that define the clock into scalar register Sk.

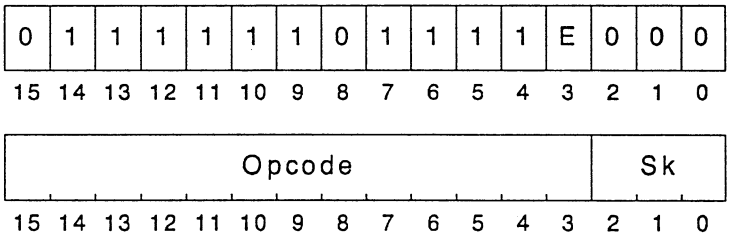
Notes: Refer to the CONVEX Architecture Reference, "Implementation-Specific Features" chapter for more information on the TOC.

MOVE THREAD TIMER/SCALAR — mov TTR,Sk

Purpose: To move the Thread Timer Register (TTR) to a scalar register

Architecture: C200 Series only

Format:



Operation:

```
Sk = THREAD_TIMER;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov TTR,Sk	7C28	E0 0111110000	None	Move TTR /scalar

Description: This instruction copies the value in the TTR to a scalar register.

Notes: This instruction has no effect on the timer register.

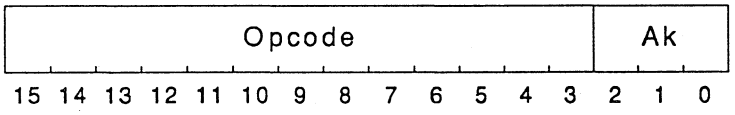
mov VL,Ak

MOVE VL/ADDRESS

Purpose: To move (copy) the contents of the Vector Length (VL) register to an address register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Ak = VL;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov VL,Ak	7D90	ST 0111110110010	None	Move VL to Ak

Description: The *mov VL,Ak* instruction replaces the contents of Ak with the contents of VL.

Notes: None

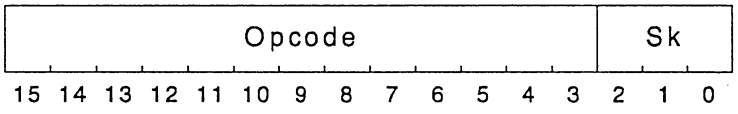
MOVE VL/SCALAR

mov.w VL,Sk

Purpose: To move (copy) the contents of the Vector Length (VL) register to a scalar register

Architecture: **C200 Series only**

Format:



Operation:

```
Sk = VL;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov.w VL,Sk	7DB0	ST 0111110110110	None	Move VL to Sk

Description: The *mov VL,Sk* instruction replaces the contents of scalar register Sk with the contents of VL.

Notes: None

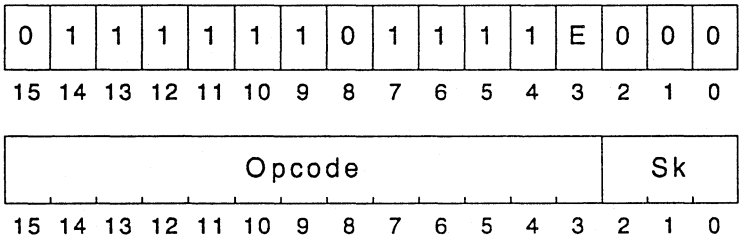
mov VML,Sk

MOVE VM LOWER/SCALAR

Purpose: To move the lower longwords of the Vector Merge (VM) register to a scalar register

Architecture: C200 Series only

Format:



Operation:

```
Sk = VM<63..0>;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov VML,Sk	7C58	E0 0111110001011	None	Load Sk from VM<63..0>

Description: The contents of Sk are replaced by the lower longword of the VM register.

Notes: None

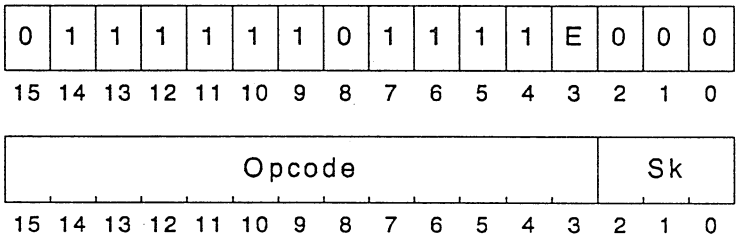
MOVE VM UPPER/SCALAR

mov VMU,Sk

Purpose: To move the upper longword of the Vector Merge (VM) register to a scalar register

Architecture: C200 Series only

Format:



Operation:

```
Sk = VM<127..64>;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov VMU,Sk	7C48	E0 0111110001001	None	Load Sk from VM<127..64>

Description: The contents of Sk are replaced by the upper longword of the VM register.

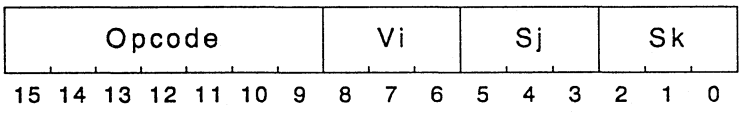
Notes: None

mov Vi,Sj,Sk — MOVE VECTOR ELEMENT/SCALAR

Purpose: To move (copy) the contents of a vector element into a scalar register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Sk = Vi[Sj<6..0>];
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov Vi,Sj,Sk	8000	ST 1000000	None	Move a vector element to a scalar

Description: The element of vector register Vi specified by the least significant seven bits of scalar register Sj replaces the contents of Sk.

Notes: None

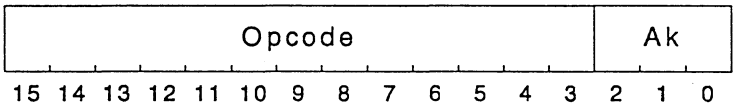
MOVE VS/ADDRESS

mov VS,Ak

Purpose: To move (copy) the contents of the Vector Stride (VS) register to an address register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Ak = VS;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov VS,Ak	7D80	ST 0111110110000	None	Move VS to Ak

Description: The contents of the VS register replace the contents of address register Ak.

Notes: None

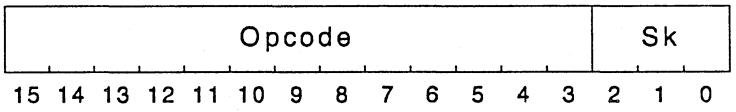
mov.w VS,Sk

MOVE VS/SCALAR

Purpose: To move (copy) the contents of the Vector Stride (VS) register to a scalar register Sk

Architecture: **C200 Series only**

Format:



Operation:

```
Sk<31..0> = VS;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mov.w VS,Sk	7DA8	ST 0111110110100	None	Move VS to Sk

Description: The contents of VS replace the least significant 32 bits of scalar register Sk.

Notes: None

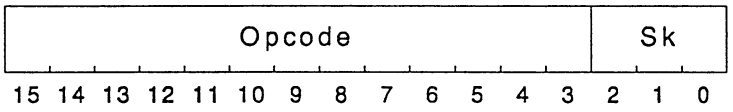
MASK INTERRUPT

mski Sk

Purpose: To mask the virtual channels

Architecture: **C100 Series only**

Format:



Operation: `Set the interrupt mask to Sk<7..0>`

Exceptions: Ring Violation (Privileged Instruction)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mski Sk	7D60	ST 0111110101100	None	Mask out interrupt

Description: The least significant eight bits of Sk replace the interrupt mask register. Bit <i> masks out virtual channel i; a zero inhibits the interrupt from a channel. When concurrent interrupts occur, the lowest numbered interrupts are serviced first.

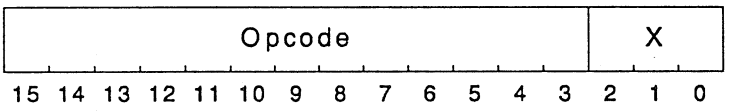
Notes: The operating system must explicitly perform *mski* during interrupt service. This may require the saving of any previous mask values.

msync — SYNCHRONIZE MEMORY

Purpose: To wait for all previous memory stores to complete

Architecture: C200 Series only

Format:



Operation:

```
while (memory_pipeline != EMPTY) {
        wait();
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
msync	7D58	ST 0111110110	None	Synchronize stores to memory

Description: This instruction will not complete if the CPU has store data in the memory pipeline. This allows a producer process to know that data has reached memory prior to informing a consumer process of its presence.

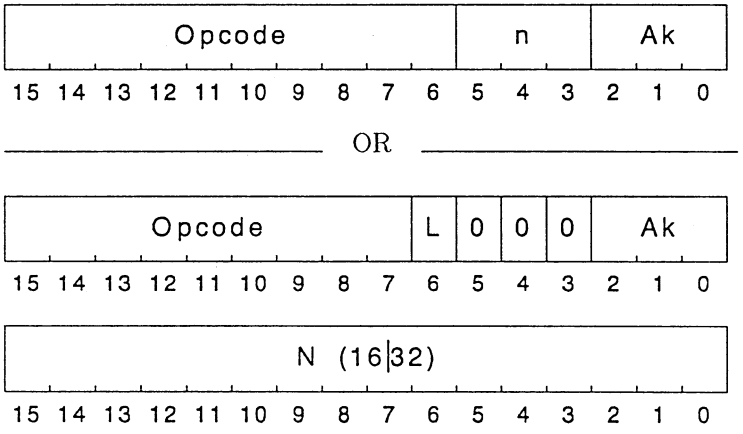
Notes: The X field is unused.

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

Purpose: To multiply the contents of an address register by an immediate field

Architecture: C100 Series, C200 Series

Format:



Operation:

```
Ak = Ak * Immediate;
```

Exceptions: (h|w): Integer Overflow

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mul.h #n,Ak	5C80	ST 0101110010	AIV	Multiply short immediate address halfword
mul.w #n,Ak	5CC0	ST 0101110011	AIV	Multiply short immediate address word
mul.h #N,Ak	1600	ST 000101100	AIV	Multiply immediate address halfword
mul.w #N,Ak	1680	ST 000101101	AIV	Multiply immediate address word

Description: The product of the contents of address register Ak and the (sign-extended) immediate replace the contents of Ak.

Notes:

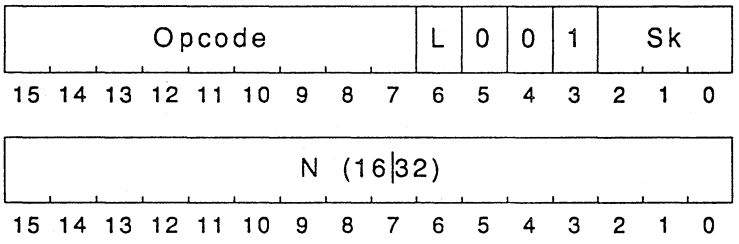
1. Sign extension does not occur for the three bits of the short immediate form.
2. The precision of the result is equal to the precision of the specified register and immediate.

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

Purpose: To multiply the contents of a scalar register by an immediate

Architecture: C100 Series, C200 Series

Format:



Operation:

```
Sk = Sk * Immediate;
```

Exceptions:

(b|h|w|l): Integer Overflow

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mul.h #N,Sk	1608	ST 000101100	SIV	Multiply scalar/immediate integer halfword
mul.w #N,Sk	1688	ST 000101101	SIV	Multiply scalar/immediate integer word
mul.s #N,Sk	1908	ST 000110010	RO,OV,UN	Multiply scalar/immediate single float

Description: The product of the contents of the (sign-extended) immediate field and the contents of scalar register Sk replaces the contents of Sk.

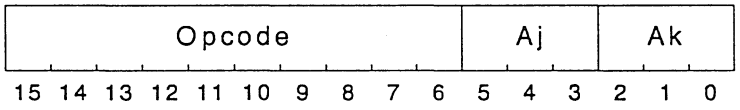
Notes: The precision of the result is equal to the precision of the specified register and immediate.

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

Purpose: To multiply the contents of two address registers

Architecture: C100 Series, C200 Series

Format:



Operation:

```
Ak = Ak * Aj;
```

Exceptions: (h|w): Integer Overflow

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mul.h Aj,Ak	5C00	ST 0101110000	AIV	Multiply address register halfword
mul.w Aj,Ak	5C40	ST 0101110001	AIV	Multiply address register word

Description: The product of the contents of address registers Aj and Ak replaces the contents of Ak.

Notes: The precision of the result is equal to the precision of the two specified registers.

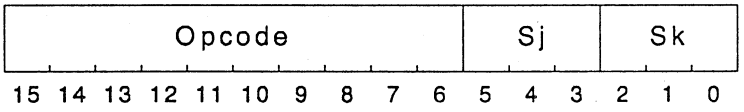
mul.(b|h|w|l|s|d) Sj,Sk

MULTIPLY SCALAR/SCALAR

Purpose: To multiply the contents two scalar registers

Architecture: **C100 Series, C200 Series**

Format:



Operation: `Sk = Sk * Sj;`

Exceptions:

(b|h|w|l): Integer Overflow

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mul.b Sj,Sk	5D00	ST 0101110100	SIV	Multiply scalar/scalar integer byte
mul.h Sj,Sk	5D40	ST 0101110101	SIV	Multiply scalar/scalar integer halfword
mul.w Sj,Sk	5D80	ST 0101110110	SIV	Multiply scalar/scalar integer word
mul.l Sj,Sk	5DC0	ST 0101110111	SIV	Multiply scalar/scalar integer longword
mul.s Sj,Sk	5700	ST 0101011100	OV,UN,RO	Multiply scalar/scalar single float
mul.d Sj,Sk	5740	ST 0101011101	OV,UN,RO	Multiply scalar/scalar double float

Description: The product of the contents of scalar registers Sj and Sk replaces the contents of Sk.

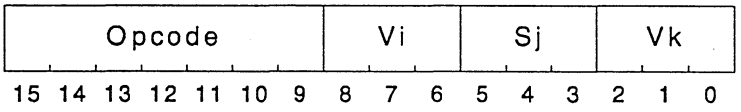
Notes: The precision of the result is equal to the precision of the two specified registers.

MULTIPLY VECTOR/SCALAR — mul.(b|h|w|l|s|d) Vi,Sj,Vk

Purpose: To multiply the contents of a vector register by the contents of a scalar register

Architecture: C100 Series, C200 Series

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      Vk[a] = Vi[a] * Sj;
}
```

Exceptions:

(b|h|w|l): Integer Overflow

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mul.b Vi,Sj,Vk	E800	ST 1110100	SIV	Multiply vector/scalar integer byte
mul.h Vi,Sj,Vk	EA00	ST 1110101	SIV	Multiply vector/scalar integer halfword
mul.w Vi,Sj,Vk	EC00	ST 1110110	SIV	Multiply vector/scalar integer word
mul.l Vi,Sj,Vk	EE00	ST 1110111	SIV	Multiply vector/scalar integer longword
mul.s Vi,Sj,Vk	9800	ST 1001100	OV,UN,RO	Multiply vector/scalar single float
mul.d Vi,Sj,Vk	9A00	ST 1001101	OV,UN,RO	Multiply vector/scalar double float

Description: Each of the contents of the first VL elements of vector register Vk is replaced by the product of the contents of the corresponding element of Vi and the contents of scalar register Sj.

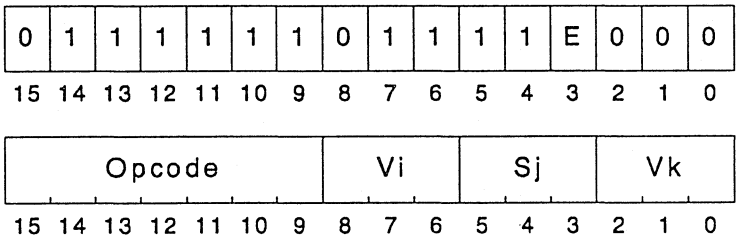
Notes: The precision of the result is equal to the precision of the specified registers.

mul.(b|h|w|l|s|d).(t|f) Vi,Sj,Vk — MULTIPLY VECTOR/SCALAR MASKED

Purpose: To multiply a vector by a scalar under control of the Vector Merge (VM) register

Architecture: **C200 Series only**

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
      case TRUE:   /* .t */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 1) {        /* if VM<a> is TRUE */
                  Vk[a] = Vi[a] * Sj;
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
      case FALSE:  /* .f */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 0) {        /* if VM<a> is FALSE */
                  Vk[a] = Vi[a] * Sj;
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
} /* end of switch */
```

Exceptions:

(b|h|w|l): Integer Overflow

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mul.b.t Vi,Sj,Vk	E800	E1 1110100	SIV	Multiply vector/scalar byte (VM)
mul.b.f Vi,Sj,Vk	E800	E0 1110100	SIV	Multiply vector/scalar byte (!VM)
mul.h.t Vi,Sj,Vk	EA00	E1 1110101	SIV	Multiply vector/scalar halfword (VM)
mul.h.f Vi,Sj,Vk	EA00	E0 1110101	SIV	Multiply vector/scalar halfword (!VM)
mul.w.t Vi,Sj,Vk	EC00	E1 1110110	SIV	Multiply vector/scalar word (VM)
mul.w.f Vi,Sj,Vk	EC00	E0 1110110	SIV	Multiply vector/scalar word (!VM)
mul.l.t Vi,Sj,Vk	EE00	E1 1110111	SIV	Multiply vector/scalar longword (VM)
mul.l.f Vi,Sj,Vk	EE00	E0 1110111	SIV	Multiply vector/scalar longword (!VM)
mul.s.t Vi,Sj,Vk	9800	E1 1001100	OV,UN,RO	Multiply vector/scalar single (VM)
mul.s.f Vi,Sj,Vk	9800	E0 1001100	OV,UN,RO	Multiply vector/scalar single (!VM)
mul.d.t Vi,Sj,Vk	9A00	E1 1001101	OV,UN,RO	Multiply vector/scalar double (VM)
mul.d.f Vi,Sj,Vk	9A00	E0 1001101	OV,UN,RO	Multiply vector/scalar double (!VM)

Description: Each of the contents of the first VL elements of vector register Vk is replaced by the product of the contents of the corresponding element of Vi and the contents of scalar register Sj if the corresponding VM bit is set (clear for *.f*).

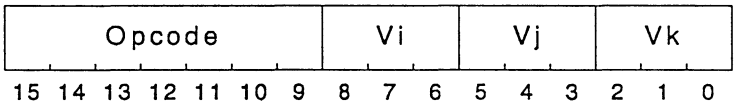
Notes: The precision of the result is equal to the precision of the specified registers.

MULTIPLY VECTOR/VECTOR — mul.(b|h|w|l|s|d) Vi,Vj,Vk

Purpose: To multiply the corresponding elements of two vector registers

Architecture: C100 Series, C200 Series

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      Vk[a] = Vi[a] * Vj[a];
}
```

Exceptions:

(b|h|w|l): Integer Overflow

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mul.b Vi,Vj,Vk	E000	ST 1110000	SIV	Multiply vector/vector integer byte
mul.h Vi,Vj,Vk	E200	ST 1110001	SIV	Multiply vector/vector integer halfword
mul.w Vi,Vj,Vk	E400	ST 1110010	SIV	Multiply vector/vector integer word
mul.l Vi,Vj,Vk	E600	ST 1110011	SIV	Multiply vector/vector integer longword
mul.s Vi,Vj,Vk	9000	ST 1001000	OV,UN,RO	Multiply vector/vector single float
mul.d Vi,Vj,Vk	9200	ST 1001001	OV,UN,RO	Multiply vector/vector double float

Description: The product of the contents of the corresponding elements of Vi and Vj replaces the contents of the first VL elements of vector register Vk.

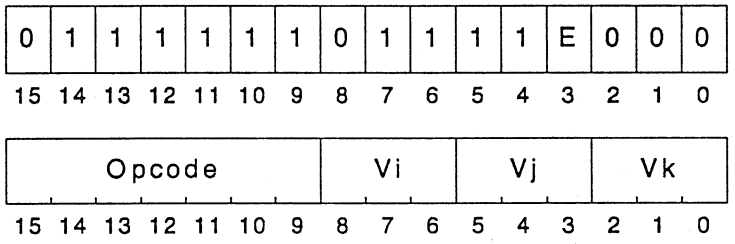
Notes: The precision of the result is equal to the precision of the specified registers.

mul.(b|h|w|l|s|d).(t|f) Vi,Vj,Vk — MULTIPLY VECTOR/VECTOR MASKED

Purpose: To multiply two vectors under control of the Vector Merge (VM) register

Architecture: **C200 Series only**

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
      case TRUE:   /*  .t */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 1) {        /* if VM<a> is TRUE */
                  Vk[a] = Vi[a] * Vj[a];
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
      case FALSE:  /*  .f */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 0) {        /* if VM<a> is FALSE */
                  Vk[a] = Vi[a] * Vj[a];
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
} /* end of switch */
```

Exceptions:

(b|h|w|l): Integer Overflow

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
mul.b.t Vi,Vj,Vk	E000	E1 1110000	SIV	Multiply byte vectors (VM)
mul.b.f Vi,Vj,Vk	E000	E0 1110000	SIV	Multiply byte vectors (!VM)
mul.h.t Vi,Vj,Vk	E200	E1 1110001	SIV	Multiply halfword vectors (VM)
mul.h.f Vi,Vj,Vk	E200	E0 1110001	SIV	Multiply halfword vectors (!VM)
mul.w.t Vi,Vj,Vk	E400	E1 1110010	SIV	Multiply word vectors (VM)
mul.w.f Vi,Vj,Vk	E400	E0 1110010	SIV	Multiply word vectors (!VM)
mul.l.t Vi,Vj,Vk	E600	E1 1110011	SIV	Multiply longword vectors (VM)
mul.l.f Vi,Vj,Vk	E600	E0 1110011	SIV	Multiply longword vectors (!VM)
mul.s.t Vi,Vj,Vk	9000	E1 1001000	OV,UN,RO	Multiply single vectors (VM)
mul.s.f Vi,Vj,Vk	9000	E0 1001000	OV,UN,RO	Multiply single vectors (!VM)
mul.d.t Vi,Vj,Vk	9200	E1 1001001	OV,UN,RO	Multiply double vectors (VM)
mul.d.f Vi,Vj,Vk	9200	E0 1001001	OV,UN,RO	Multiply double vectors (!VM)

Description: Each of the contents of first VL elements of vector register Vk is replaced by the product of the contents of the corresponding elements of Vi and Vj if the corresponding VM bit is set (clear for *.f*).

Notes: The precision of the result is equal to the precision of the specified registers.

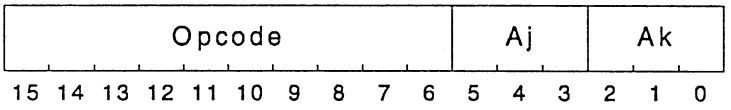
NEGATE ADDRESS

neg.(h|w) Aj,Ak

Purpose: To negate arithmetically the contents of an address register

Architecture: C100 Series, C200 Series

Format:



Operation:

```
Ak = 0 - Aj;
```

Exceptions: (h|w): Integer Overflow

Opcode:

Mnemonic	Hex	Binary	PSW	Description
neg.h Aj,Ak	5680	ST 0101011010	C,AIV	Negate address register halfword
neg.w Aj,Ak	56C0	ST 0101011011	C,AIV	Negate address register word

Description: The 2's complement of address register Aj replaces the contents of Ak.

Notes: Overflow can occur for the negation of the most negative integer.

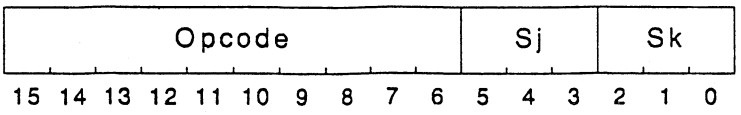
neg.(b|h|w|l|s|d) Sj,Sk

NEGATE SCALAR

Purpose: To negate arithmetically the contents of a scalar register

Architecture: C100 Series, C200 Series

Format:



Operation:

```
Sk = 0 - Sj;
```

Exceptions: (b|h|w|l): Integer Overflow

(s|d): Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
neg.b Sj,Sk	6F00	ST 0110111100	SIV	Negate scalar/scalar integer byte
neg.h Sj,Sk	6F40	ST 0110111101	SIV	Negate scalar/scalar integer halfword
neg.w Sj,Sk	6F80	ST 0110111110	SIV	Negate scalar/scalar integer word
neg.l Sj,Sk	6FC0	ST 0110111111	SIV	Negate scalar/scalar integer longword
neg.s Sj,Sk	6580	ST 0110010110	RO	Negate scalar/scalar single float
neg.d Sj,Sk	65C0	ST 0110010111	RO	Negate scalar/scalar double float

Description: The arithmetic negation of scalar register Sj replaces Sk. The result is identical to subtracting Sj from 0.

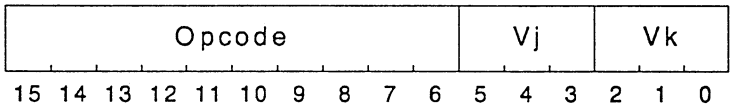
Notes: Overflow can occur for the negation of the most negative integer.

NEGATE VECTOR — neg.(b|h|w|l|s|d) Vj,Vk

Purpose: To negate arithmetically the contents of a vector register

Architecture: C100 Series, C200 Series

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      Vk[a] = 0 - Vj[a];
}
```

Exceptions: (b|h|w|l): Integer Overflow

(s|d): Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
neg.b Vj,Vk	6E00	ST 0110111000	SIV	Negate vector/vector integer byte
neg.h Vj,Vk	6E40	ST 0110111001	SIV	Negate vector/vector integer halfword
neg.w Vj,Vk	6E80	ST 0110111010	SIV	Negate vector/vector integer word
neg.l Vj,Vk	6EC0	ST 0110111011	SIV	Negate vector/vector integer longword
neg.s Vj,Vk	6480	ST 0110010010	RO	Negate vector/vector single float
neg.d Vj,Vk	64C0	ST 0110010011	RO	Negate vector/vector double float

Description: Each of the first VL elements of vector register Vk is replaced by zero minus the contents of the corresponding element of Vj.

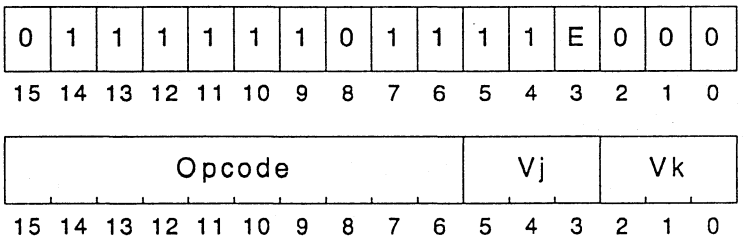
Notes: Overflow can occur for the negation of the most negative integer.

neg.(b|h|w|l|s|d).(t|f) Vj,Vk — NEGATE VECTOR MASKED

Purpose: To negate a vector under control of the Vector Merge (VM) register

Architecture: C200 Series only

Format:



Operation:

```
switch (E) { /* prefix bit<3> */
     case TRUE:  /* .t */
         for (a = 0; a < VL; a++) {
             if (VM<a> == 1) {        /* if VM<a> is TRUE */
                 Vk[a] = 0 - Vj[a];
             }
         }   /* end of for loop */
         break;  /* go to end of switch */
     case FALSE: /* .f */
         for (a = 0; a < VL; a++) {
             if (VM<a> == 0) {        /* if VM<a> is FALSE */
                 Vk[a] = 0 - Vj[a];
             }
         }   /* end of for loop */
         break;  /* go to end of switch */
} /* end of switch */
```

Exceptions:

(b|h|w|l): Integer Overflow

(s|d): Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
neg.b.t Vj,Vk	6E00	E1 0110111000	SIV	Negate byte vector (VM)
neg.b.f Vj,Vk	6E00	E0 0110111000	SIV	Negate byte vector (!VM)
neg.h.t Vj,Vk	6E40	E1 0110111001	SIV	Negate halfword (VM)
neg.h.f Vj,Vk	6E40	E0 0110111001	SIV	Negate halfword (!VM)
neg.w.t Vj,Vk	6E80	E1 0110111010	SIV	Negate word (VM)
neg.w.f Vj,Vk	6E80	E0 0110111010	SIV	Negate word (!VM)
neg.l.t Vj,Vk	6EC0	E1 0110111011	SIV	Negate longword (VM)
neg.l.f Vj,Vk	6EC0	E0 0110111011	SIV	Negate longword (!VM)
neg.s.t Vj,Vk	6480	E1 0110010010	RO	Negate single (VM)
neg.s.f Vj,Vk	6480	E0 0110010010	RO	Negate single (!VM)
neg.d.t Vj,Vk	64C0	E1 0110010011	RO	Negate double (VM)
neg.d.f Vj,Vk	64C0	E0 0110010011	RO	Negate double (!VM)

Description: Each of the first VL elements of vector register Vk is replaced by zero minus the contents of the corresponding element of Vj if the corresponding VM bit is set (clear for *.f*).

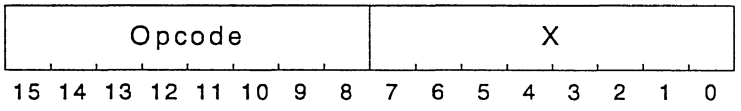
Notes: Overflow can occur for the negation of the most negative integer.

NO OPERATION — nop

Purpose: To perform no operation

Architecture: C100 Series, C200 Series

Format:



Operation: `(Perform no operation);`

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
nop	7000	ST 01110000	None	No operation (branch never)

Description: This instruction will perform no operation.

Notes: The X field is unused.

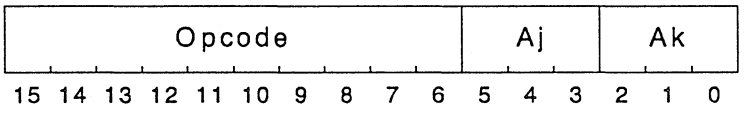
not Aj,Ak

COMPLEMENT ADDRESS

Purpose: To complement the contents of an address register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Ak = ~Aj;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
not Aj,Ak	52C0	ST 0101001011	None	Complement address register

Description: The 1's complement of the contents of address register Aj replaces the contents of Ak.

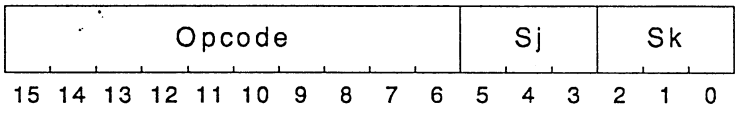
Notes: None

COMPLEMENT SCALAR — not Sj,Sk

Purpose: To complement the contents of a scalar register

Architecture: C100 Series, C200 Series

Format:



Operation:

```
Sk = ~Sj;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
not Sj,Sk	53C0	ST 0101001111	None	Complement scalar/scalar

Description: The 1's complement of scalar register Sj replaces the contents of Sk.

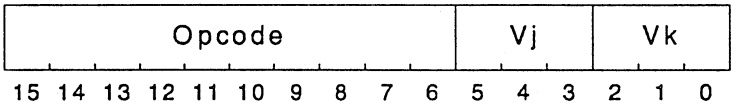
Notes: None

not Vj,Vk — COMPLEMENT VECTOR

Purpose: To complement the elements of a vector

Architecture: C100 Series, C200 Series

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      Vk[a] = ~Vj[a];
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
not Vj,Vk	62C0	ST 0110001011	None	Complement a vector

Description: The 1's complement of the contents of each of the first VL elements of vector register Vj replaces the corresponding elements of Vk.

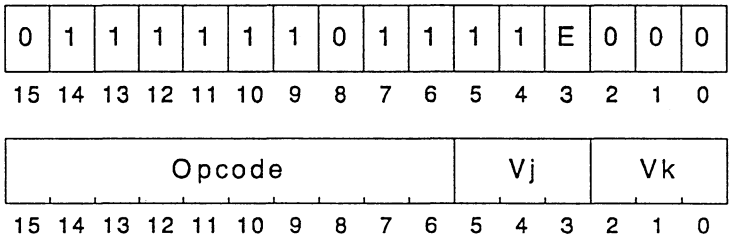
Notes: None

COMPLEMENT VECTOR MASKED — not.(t|f) Vj,Vk

Purpose: To complement the contents of a vector under control of the Vector Merge (VM) register

Architecture: C200 Series only

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
      case TRUE:   /*  .t */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 1) {        /* if VM<a> is TRUE */
                  Vk[a] = ~Vj[a];
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
      case FALSE:  /*  .f */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 0) {        /* if VM<a> is FALSE */
                  Vk[a] = ~Vj[a];
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
not.t Vj,Vk	62C0	E1 0110001011	None	Complement a vector (VM)
not.f Vj,Vk	62C0	E0 0110001011	None	Complement a vector (!VM)

Description: The 1's complement of the contents of each of the first VL elements of vector register Vj replaces the corresponding elements of Vk if the corresponding VM bit is set (clear for *.f*).

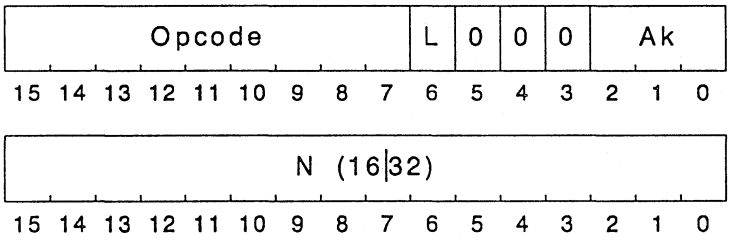
Notes: None

or #N,Ak — OR ADDRESS/IMMEDIATE

Purpose: To OR the contents of an address register and an immediate

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Ak = Ak | Immediate;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
or #N,Ak	1280	ST 000100101	None	OR immediate to address register

Description: The logical OR of the (sign-extended) immediate field and the contents of address register Ak replace the contents of Ak.

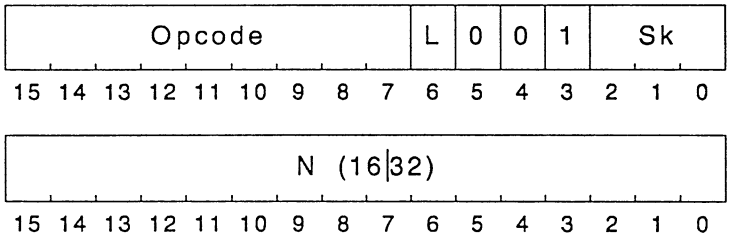
Notes: None

OR SCALAR/IMMEDIATE — or #N,Sk

Purpose: To OR the contents of a scalar register and an immediate

Architecture: C100 Series, C200 Series

Format:



Operation:

```
Sk<31..0> = Sk<31..0> | Immediate;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
or #N,Sk	1288	ST 000100101	None	OR scalar/immediate

Description: The logical OR of the (sign-extended) immediate field and the contents of the least significant 32 bits of scalar register Sk replace the least significant 32 bits of Sk. The most significant 32 bits of Sk are not affected.

Notes: None

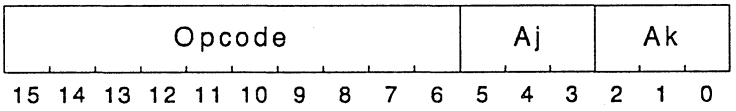
or Aj,Ak

OR ADDRESS/ADDRESS

Purpose: To OR the contents of two address registers

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Ak = Ak | Aj;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
or Aj,Ak	5240	ST 0101001001	None	OR address register

Description: The logical OR of the contents of address registers Aj and Ak replaces the contents of Ak.

Notes: None

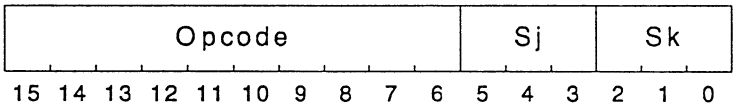
OR SCALAR/SCALAR

or Sj,Sk

Purpose: To OR the contents of two scalar registers

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Sk = Sk | Sj;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
or Sj,Sk	5340	ST 0101001101	None	OR scalar/scalar

Description: The logical OR of the contents of scalar registers Sj and Sk replaces the contents of Sk.

Notes: None

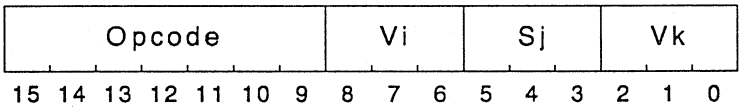
or Vi,Sj,Vk

OR VECTOR/SCALAR

Purpose: To OR the elements of a vector register and the contents of a scalar register

Architecture: C100 Series, C200 Series

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      Vk[a] = Vi[a] | Sj;
}
```

Exceptions: None

Opcode:

Mnemonic	Hex Binary	PSW	Description
or Vi,Sj,Vk	AA00 ST 1010101	None	OR vector/scalar

Description: The logical OR of the contents of the corresponding element of Vi and the contents of scalar register Sj replaces each of the first VL elements of vector register Vk.

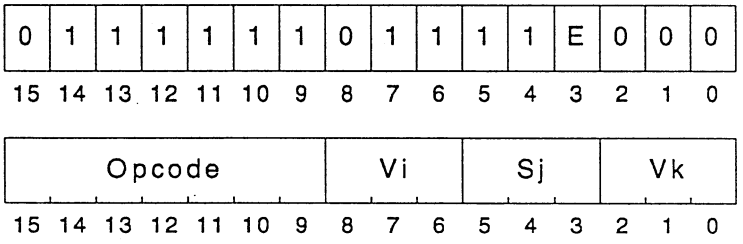
Notes: None

OR VECTOR/SCALAR MASKED — or.(t|f) Vi,Sj,Vk

Purpose: To OR the contents of a vector and a scalar under control of the Vector Merge (VM) register

Architecture: C200 Series only

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
      case TRUE:   /*  .t */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 1) {        /* if VM<a> is TRUE */
                  Vk[a] = Vi[a] | Sj;
              }
          }   /* end of for loop */
          break;   /* go to end of switch */
      case FALSE:  /*  .f */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 0) {        /* if VM<a> is FALSE */
                  Vk[a] = Vi[a] | Sj;
              }
          }   /* end of for loop */
          break;   /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
or.t Vi,Sj,Vk	AA00	E1 1010101	None	OR vector/scalar (VM)
or.f Vi,Sj,Vk	AA00	E0 1010101	None	OR vector/scalar (!VM)

Description: Each of the first VL elements of vector register Vk is replaced by the logical OR of the contents of the corresponding element of Vi and the contents of scalar register Sj if the corresponding VM bit is set (clear for *.f*).

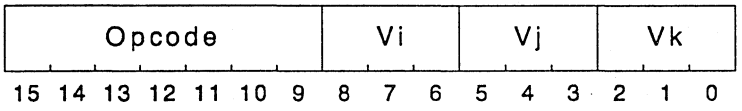
Notes: None

or Vi,Vj,Vk — OR VECTOR/VECTOR

Purpose: To OR the elements of two vector registers

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      Vk[a] = Vi[a] | Vj[a];
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
or Vi,Vj,Vk	A200	ST 1010001	None	OR two vectors

Description: The logical OR of the contents of the corresponding elements of Vi and Vj replaces each of the first VL elements of vector register Vk.

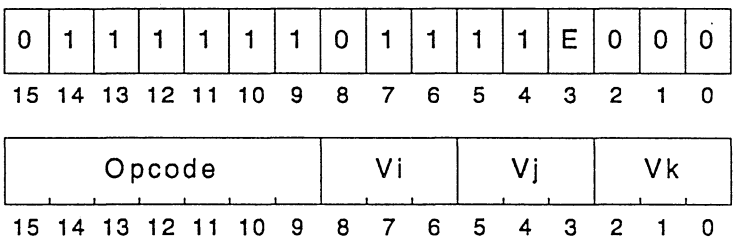
Notes: None

OR VECTOR/VECTOR MASKED — or.(t|f) Vi,Vj,Vk

Purpose: To OR the contents of two vectors under control of the Vector Merge (VM) register

Architecture: C200 Series only

Format:



Operation:

```
switch (E) { /* prefix bit<3> */
     case TRUE:  /* .t */
         for (a = 0; a < VL; a++) {
             if (VM<a> == 1) {        /* if VM<a> is TRUE */
                 Vk[a] = Vi[a] | Vj[a];
             }
         }   /* end of for loop */
         break;  /* go to end of switch */
     case FALSE: /* .f */
         for (a = 0; a < VL; a++) {
             if (VM<a> == 0) {        /* if VM<a> is FALSE */
                 Vk[a] = Vi[a] | Vj[a];
             }
         }   /* end of for loop */
         break;  /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
or.t Vi,Vj,Vk	A200	E1 1010001	None	OR two vectors (VM)
or.f Vi,Vj,Vk	A200	E0 1010001	None	OR two vectors (!VM)

Description: Each of the first VL elements of vector register Vk is replaced by the logical OR of the contents of the corresponding elements of Vi and Vj if the corresponding VM bit is set (clear for *.f*).

Notes: None

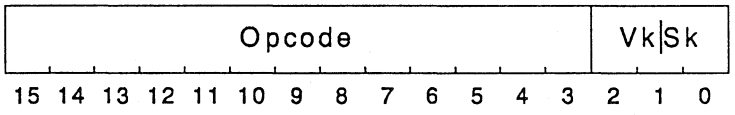
parity (Vk|Sk)

EXCLUSIVE OR REDUCE VECTOR

Purpose: To exclusive OR reduce all the elements of a vector register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      Vk = Sk ^ Vk[a];
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
parity Vk	7E30	ST 0111111000110	None	Exclusive OR reduce a vector

Description: The logical OR of all 64 bits of the contents of scalar register Sk and the first VL elements of vector register Vk replace Sk.

Notes:

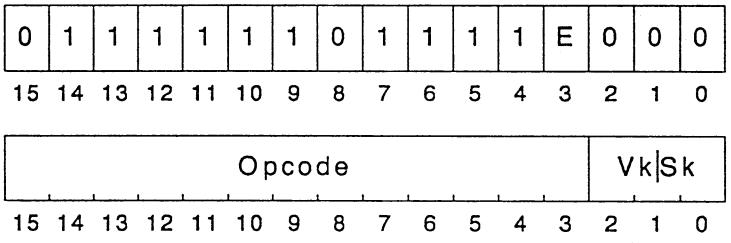
1. Initialize the scalar register properly for the first use of this instruction (probably to 0).
2. Either Vk or Sk may be used as a valid argument to this instruction. This instruction operates in distinct matched vector and scalar register pairs: (V0,S0), (V1,S1), (V2,S2), (V3,S3), (V4,S4), (V5,S5), (V6,S6), (V7,S7)

EXCLUSIVE OR REDUCE VECTOR MASKED — parity.(t|f) (Vk|Sk)

Purpose: To exclusive OR reduce the elements of a subset of a vector register under control of the Vector Merge (VM) register

Architecture: C200 Series only

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
      case TRUE:   /* .t */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 1) {        /* if VM<a> is TRUE */
                  Vk[a] = Sk ^ Vk[a];
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
      case FALSE:  /* .f */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 0) {        /* if VM<a> is FALSE */
                  Vk[a] = Sk ^ Vk[a];
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
parity.t Vk	7E30	E1 0111111000110	None	Exclusive OR reduce vector (VM)
parity.f Vk	7E30	E0 0111111000110	None	Exclusive OR reduce vector (!VM)

Description: The logical OR of all 64 bits of the contents of scalar register Sk and the first VL elements of vector register Vk replace Sk. Only elements of Vk with corresponding VM bit set (clear for *.f*) participate in the reduction.

Notes:

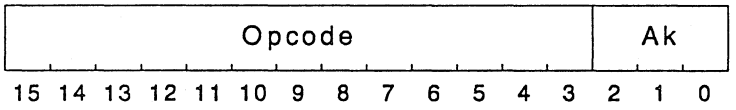
1. Initialize the scalar register properly for the first use of this instruction (probably to 0).
2. Either Vk or Sk may be used as a valid argument to this instruction. This instruction operates in distinct matched vector and scalar register pairs: (V0,S0), (V1,S1), (V2,S2), (V3,S3), (V4,S4), (V5,S5), (V6,S6), (V7,S7)

pate Ak — PURGE ATU ENTRY

Purpose: To purge an Address Translation Unit (ATU) entry

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
if (Architecture == C200) {    /* C200 Series */
     ATU_valid_bit(Ak) = 0;    /* purge All CPU(s) */
} else {
     ATU_valid_bit(Ak) = 0;    /* C100 Series */
     (Purge L_cache);
     (Purge I_cache);
}
```

Exceptions: Ring Violation (Privileged Instruction)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
pate Ak	7C28	ST 0111110000101	None	Purge ATU entry

Description: If there is an ATU entry associated with the address contained in address register Ak, then that ATU entry is purged, i.e., marked invalid. All other ATU entries are left unchanged.

Notes:

1. The *pate Ak* instruction is typically used when a page frame is added to the working set of a process after an Address Translation Fault (ATF). The ATU information concerning that page is invalid after the fault has been resolved by the operating system; hence, the ATU entry must be purged.

C200 Series only

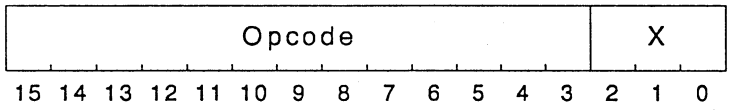
2. These processors *do not* purge the Instruction Cache (Icache).
3. The *pate Ak* instruction purges (i.e., marks invalid) a single ATU entry associated with the address contained in address register Ak from each CPU in the Complex. All other ATU entries are left unchanged.

PURGE ATU — patu

Purpose: To purge all Address Translation Unit (ATU) entries

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
if (Architecture == C200) {   /* C200 Series only */
      All_CPUs_ATU_valid_bits = 0;
} else {
      ATU_valid_bits = 0;  /* C100 Series only */
      (Purge L_cache);
      (Purge I_cache);
}
```

Exceptions: Ring Violation (Privileged Instruction)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
patu	7C20	ST 0111110000100	None	Purge the entire ATU

Description: All entries in the ATU are purged, including the logical and instruction caches.

Notes:

1. The X field is ignored.

C200 Series only

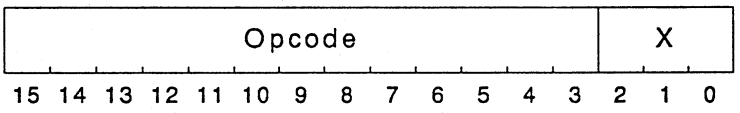
2. All the entries in each CPUs ATU are purged.
3. CPU Complexes implementing the C200 Series architecture ***do not*** purge the Logical Cache (Lcache) or the Instruction Cache (Icache).

pbkpt — PROCESS BREAKPOINT

Purpose: To force all threads associated with the current process to stop execution

Architecture: C200 Series only

Format:



Operation:

```
(Set the process breakpoint bit in all trap instruction registers greater
than or equal to the ring of execution);
(Allocate a ring 0 stack if crossing rings);
psw[FRL] = 01;          /* extended frame */
push(thread_timer);
push(S0); push(S1); push(S2); push(S3); push(S4); push(S5); push(S6); push(S7);
push(A0); push(A1); push(A2); push(A3); push(A4); push(A5); push(A6); push(A7);
push(PSW);
push(next_instruction_address);
psw[FRL] = 0; psw[C]  = 0; psw[SC] =  0; psw[AIV] = 0; psw[ADZ] = 0;
psw[UN]  = 0; psw[OV] = 0; psw[FDZ] = 0; psw[RO]  = 0; psw[SIV] = 0;
psw[SDZ] = 0; psw[FIN] = 0;
SP = SP - 112;          /* Extended return block */
FP = SP;
A5 = 0x00001400;
S0 = (Trap Instruction Register that trapped);
(Enter the ring 0 system exception handler);
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
pbkpt	7DC8	ST 0111110111001	See note 1	Force process breakpoint exception

Description: A Ring 0 system exception will occur for all threads in rings greater than or equal to the ring of execution. The process breakpoint bit in all trap instruction registers is set to indicate a process breakpoint. The Ring 0 exception handler is executed with a code of 14 (hex) and 0 qualifier.

Notes:

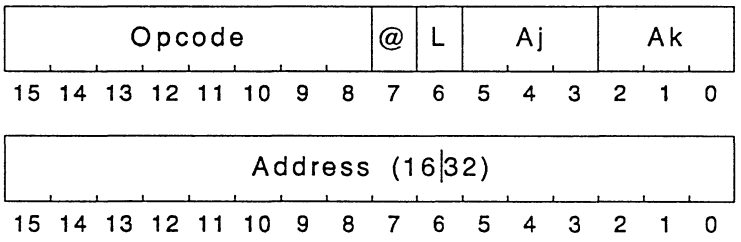
1. The PSW is cleared to all zero after the PSW is pushed on the stack.
2. The X field is unused.
3. The trap condition remains outstanding until all bits in the hardware communication trap instruction register are cleared. If a thread attempts to return to an outer ring that has any bits set in the ring's trap instruction register, it will immediately enter the Ring 0 exception handler.
4. Refer to the *CONVEX Architecture Reference*, "Exceptions and Interrupts" chapter, for a more detailed explanation of the *pbkpt* instruction.

POST A FORK — pfork <effa>,Ak

Purpose: To post the need for a CPU to join in a process computation

Architecture: C200 Series only

Format:



Operation:

```
Set the Hardware Communication Fork Event Registers in the current CIR,
posting the need for another CPU to join the process:

if ( C = snd.l(forklck, FP::AP) ) {  /* C = 1 if snd() succeeds */
      fork.PC        = <effa>;
      fork.PSW       = PSW;
      fork.source_PC = PC;
      snd.l(forkposted, PFORKED::Ak);
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
pfork <effa>,Ak	2300	ST 0010001100	See note 1	Post a fork

Description: This instruction posts a fork which can be taken by any CPU to create a concurrent thread of execution at the specified effective address. If a fork is already posted, it must be taken prior to posting another. Address Carry (C) is set to 0 if a fork is already posted, and is set to 1 if the posting of the fork was successful.

Note that the *snd* operation to *forklck* loads FP into *fork.FP* and AP into *fork.AP*. The communication registers are loaded only if *forklck* is zero, i.e., there is no currently posted fork. Similarly, the snd to forkposted loads the constant "PFORKED" into *fork.type* and Ak into *fork.SP*.

Notes:

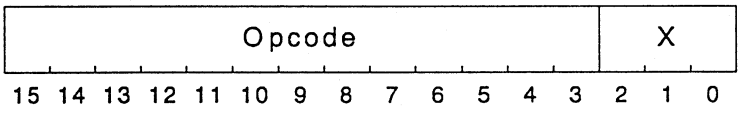
1. Address Carry (C) of the PSW is set to 1 if the fork is successfully posted, otherwise it is reset to 0.
2. At most, *pfork* will add one CPU to the process. Use *spawn* to attempt to add multiple CPUs to a process.
3. Once a fork is posted, the fork must either be taken by a CPU or cleared with *cfork* before another fork can be successfully posted.
4. Refer to the *CONVEX Architecture Reference*, "Multiprocessor Management," chapter for more information on *pfork* and forking operations.

pich — PURGE INSTRUCTION CACHE

Purpose: To purge the entire Instruction Cache (Icache)

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Icache_valid_bits = 0;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
pich	7C30	ST 0111110000110	None	Purge the Icache

Description: All the entries in the Instruction Cache (Icache) are purged.

Notes:

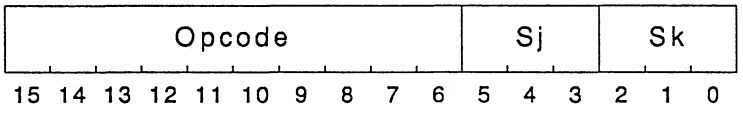
1. The *pich* instruction affects only the performance of the currently executing program, not its results. Language debuggers use the *pich* instruction to purge the Icache after a modification of instruction space is performed.
2. This is not a privileged instruction.
3. The X field is ignored.

POPULATION COUNT SCALAR — plc.t Sj,Sk

Purpose: To determine the number of 1's in a scalar register

Architecture: C100 Series, C200 Series

Format:



Operation:

```
temp = 0;
for (a = 0; a <= 63; a++) {
      if (Sj<a> == 1) {
          temp = temp + 1;
      }
}
Sk = temp;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
plc.t Sj,Sk	4580	ST 0100010110	None	Count the number of 1's in Sj

Description: The number of 1 bits in the 64 bits of scalar register Sj replaces the contents of Sk.

Notes: None

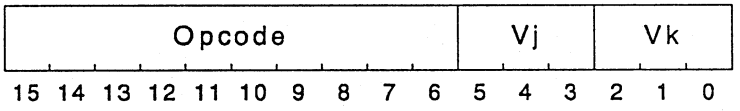
plc.t Vj,Vk

POPULATION COUNT VECTOR

Purpose: To count the number of 1's in each vector element

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      temp = 0;
      for(b = 0; b <= 63; b++) {
          if (Vj[a]<b> == 1) {
              temp = temp + 1;
          }
      }
      Vk[a] = temp;
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
plc.t Vj,Vk	6340	ST 0110001101	None	Population count of a vector

Description: The number of 1's contained in the 64 bits of the corresponding element of Vj replaces each of the first VL elements of Vk.

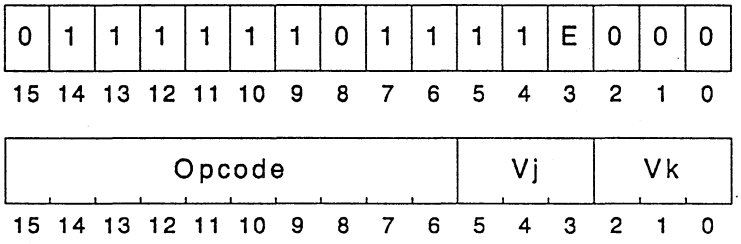
Notes: None

POPULATION COUNT VECTOR MASKED — plc.t.(t|f) Vj,Vk

Purpose: To count the number of 1's in each vector element under control of the Vector Merge (VM) register

Architecture: C200 Series only

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
     case TRUE:   /* .t */
          for (a = 0; a < VL; a++) {
               if (VM<a> == 1) {        /* if VM<a> is TRUE */
                    temp = 0;
                    for (b = 0; b <= 63; j++) {
                         if (Vj<b> == 1) {
                              temp = temp + 1;
                         }
                    }
                    Vk[a] = temp;
               }
          }   /* end of for loop */
          break;   /* go to end of switch */
     case FALSE:  /* .f */
          for (a = 0; a < VL; a++) {
               if (VM<a> == 0) {        /* if VM<a> is FALSE */
                    temp = 0;
                    for (b = 0; b <= 63; j++) {
                         if (Vj<b> == 1) {
                              temp = temp + 1;
                         }
                    }
                    Vk[a] = temp;
               }
          }   /* end of for loop */
          break;   /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
plc.t.t Vj,Vk	6340	E1 0110001101	None	Population count of vector (VM)
plc.t.f Vj,Vk	6340	E0 0110001101	None	Population count of vector (!VM)

Description: Each of the first VL elements of Vk is replaced by the number of 1's contained in the 64 bits of the corresponding element of Vj if the corresponding VM bit is set (clear for *.f*).

Notes: None

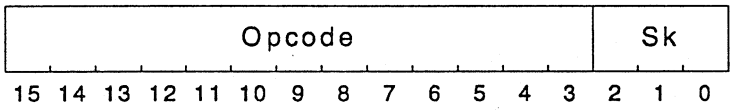
plc.(t|f) VM,Sk

POPULATION COUNT VM

Purpose: To load the number of 1's or 0's in Vector Merge (VM) into a scalar register

Architecture: C100 Series, C200 Series

Format:



Operation:

```
temp = 0;
switch (E) {  /* opcode bit<3> */
      case TRUE:   /* .t */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 1) {        /* if VM<a> is TRUE */
                  temp = temp + 1;
              }
          }   /* end of for loop */
          Sk = temp;
          break;  /* go to end of switch */
      case FALSE:  /* .f */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 0) {        /* if VM<a> is FALSE */
                  temp = temp + 1;
              }
          }   /* end of for loop */
          Sk = temp;
          break;  /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
plc.f VM,Sk	7EE0	ST 0111111011100	None	Load the number of 0's in VM into Sk
plc.t VM,Sk	7EE8	ST 0111111011101	None	Load the number of 1's in VM into Sk

Description: The number of 1's (for *plc.t*; 0's for *plc.f*) in the VM register replaces the entire contents of scalar register Sk.

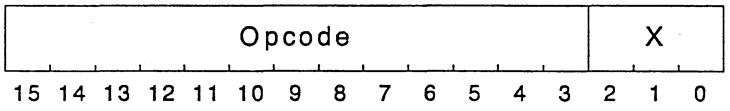
Notes: None

PURGE LOGICAL CACHE — plch

Purpose: To purge the Logical Cache (Lcache)

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Lcache_valid_bits = 0;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
plch	7C38	ST 0111110000111	None	Purge the Lcache

Description: All the entries in the logical cache (Lcache) are purged.

Notes:

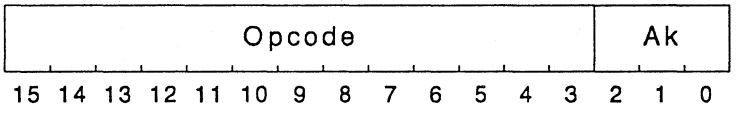
1. Execution of this instruction on a C201, C202, C210, C220, C230, or C240 CPU Complex has no effect.
2. The *plch* instruction enables the program to look anew at a location that may already be accelerated into the Lcache. If an I/O device (or I/O processor) changes a location whose contents have found their way into the the Lcache, the only guaranteed way to observe that change is to purge the Lcache.
3. This instruction is not privileged.
4. The X field is ignored.

pop.w Ak — POP ADDRESS REGISTER

Purpose: To pop a word from the stack into an address register

Architecture: C100 Series, C200 Series

Format:



Operation:

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

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
pop.w Ak	7D10	ST 0111110100010	None	Pop word into address register

Description: The word found at the location referenced by address register A0 (the stack pointer) replaces the contents of Ak. A0 is then increased by four.

Notes: None

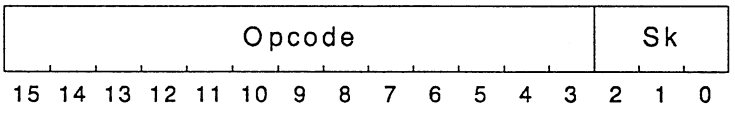
POP SCALAR REGISTER

pop.(w|l) Sk

Purpose: To pop one element from the stack into a scalar register

Architecture: C100 Series, C200 Series

Format:



Operation:

```
Sk<31..0> = c(A0);      /* pop.w */
A0 = A0 + 4;

Sk = c(A0);             /* pop.l */
A0 = A0 + 8;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
pop.w Sk	7D30	ST 0111110100110	None	Pop Sk <31..0> from the stack
pop.l Sk	7D38	ST 0111110100111	None	Pop Sk <63..0> from the stack

Description: *pop.w*: The word found at the location referenced by address register A0 (the stack pointer) replaces the bottom 32 bits of scalar register Sk. A0 is then incremented by four.

pop.l: The longword found at the location referenced by address register A0 (the stack pointer) replaces scalar register Sk. A0 is then incremented by eight.

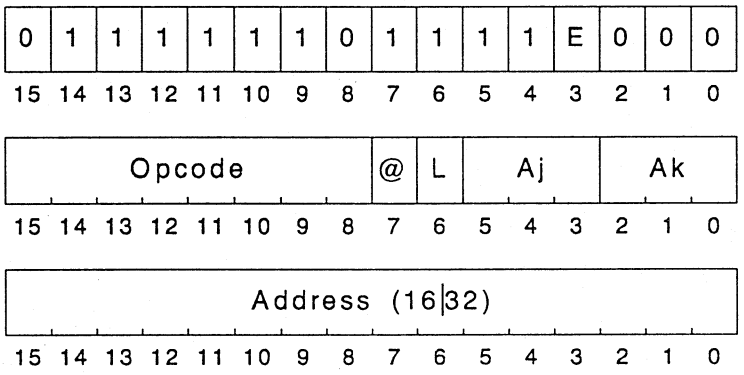
Notes: None

popr <effa>,Ak — POP RESOURCE/ADDRESS

Purpose: To pop a word from a resource structure into an address register

Architecture: **C200 Series only**

Format:



Operation:

```
if ( tas(effa.lock) ) {
     C = 1;
     if (c(effa.data) == 0) {
          SC = 0;
     } else {
          Ak = c(effa.data + c(effa.data));
          c(effa.data) = c(effa.data) - 4;
          SC = 1;
     }
     tac(effa.lock);
} else {
     C = 0;
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
popr Ak,<effa>	0800	E0 0000100000	See note 1	Pop resource/address register

Description: The *popr* instruction atomically pops a word from the resource structure checking for underflow. If C is set to 0, then *popr* was unable to lock the resource structure, the pop operation failed, and SC is unmodified. If C is set to 1, the lock was successful and the Scalar Carry (SC) is set with success or failure status of the operation. SC = 0 for a resource structure underflow; SC = 1 for a successful pop operation.

Notes:

1 Address Carry (C) and Scalar Carry (SC) are affected as described by the preceding operation pseudocode.

2 This instruction is atomic.

3 When an underflow occurs (i.e., the resource structure is empty), the contents of Ak are undefined.

4 The "valid" byte of the resource structure is ignored.

5 Refer to the *CONVEX Architecture Reference*, "Memory Management" chapter, for more information on the *popr* instruction and resource structures.

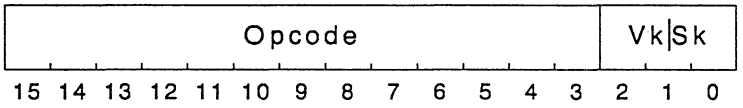
PRODUCT VECTOR

prod.(b|h|w|l|s|d) (Vk|Sk)

Purpose: To obtain the product of all elements of a vector register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
for (a = 0; a < VL; a++) {
    Sk = Sk * Vk[a];     /* See following notes */
}
```

Exceptions:

(b|h|w|l): Integer Overflow

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
prod.b Vk	7EC0	ST 0111111011000	SIV	Multiply reduce a vector of bytes
prod.h Vk	7EC8	ST 0111111011001	SIV	Multiply reduce a vector of halfwords
prod.w Vk	7ED0	ST 0111111011010	SIV	Multiply reduce a vector of words
prod.l Vk	7ED8	ST 0111111011011	SIV	Multiply reduce a vector of longwords
prod.s Vk	7E90	ST 0111111010010	OV,UN,RO	Multiply reduce a vector of single float
prod.d Vk	7E98	ST 0111111010011	OV,UN,RO	Multiply reduce a vector of double float

Description: The product of the contents of scalar register Sk and the first VL elements of vector register Vk replaces Sk.

Notes:

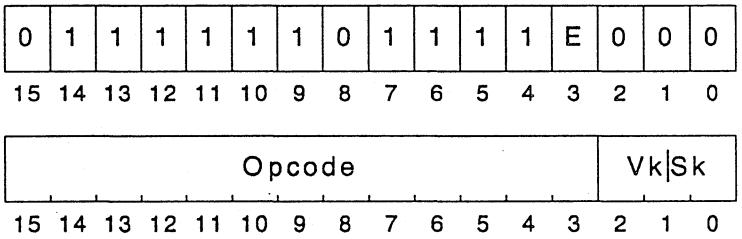
1. Initialize the scalar register properly for the first use of the *prod* reduce instruction (probably to 1).
2. The sequence of the products performed by the hardware is *not* identical to the psuedocode sequence as noted above, i.e., it is implementation-specific. The C100 Series and C200 Series architectures each execute a different order of reduction and may get different results. For more information, refer to the discussion of vector operations in the *CONVEX Architecture Reference*, "Instruction Set" chapter.
3. Either Vk or Sk may be used as a valid argument to this instruction. This instruction operates in distinct matched vector and scalar register pairs: (V0,S0), (V1,S1), (V2,S2), (V3,S3), (V4,S4), (V5,S5), (V6,S6), (V7,S7).

prod.(b|h|w|l|s|d).(t|f) (Vk|Sk) — PRODUCT VECTOR MASKED

Purpose: To obtain the product of a subset of the elements of a vector register under control of the Vector Merge (VM) register

Architecture: C200 Series only

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
     case TRUE:   /*  .t */
          for (a = 0; a < VL; a++) {
               if (VM<a> == 1) {        /* if VM<a> is TRUE */
                    Sk = Sk * Vk[a]; /* See notes below */
               }
          }   /* end of for loop */
          break;  /* go to end of switch */
     case FALSE:  /*  .f */
          for (a = 0; a < VL; a++) {
               if (VM<a> == 0) {        /* if VM<a> is FALSE */
                    Sk = Sk * Vk[a]; /* See notes below */
               }
          }   /* end of for loop */
          break;  /* go to end of switch */
} /* end of switch */
```

Exceptions:

(b|h|w|l): Integer Overflow

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
prod.b.t Vk	7EC0	E1 0111111011000	SIV	Multiply reduce byte vector (VM)
prod.b.f Vk	7EC0	E0 0111111011000	SIV	Multiply reduce byte vector (!VM)
prod.h.t Vk	7EC8	E1 0111111011001	SIV	Multiply reduce halfword vector (VM)
prod.h.f Vk	7EC8	E0 0111111011001	SIV	Multiply reduce halfword vector (!VM)
prod.w.t Vk	7ED0	E1 0111111011010	SIV	Multiply reduce word vector (VM)
prod.w.f Vk	7ED0	E0 0111111011010	SIV	Multiply reduce word vector (!VM)
prod.l.t Vk	7ED8	E1 0111111011011	SIV	Multiply reduce longword vector (VM)
prod.l.f Vk	7ED8	E0 0111111011011	SIV	Multiply reduce longword vector (!VM)
prod.s.t Vk	7E90	E1 0111111010010	OV,UN,RO	Multiply reduce single vector (VM)
prod.s.f Vk	7E90	E0 0111111010010	OV,UN,RO	Multiply reduce single vector (!VM)
prod.d.t Vk	7E98	E1 0111111010011	OV,UN,RO	Multiply reduce double vector (VM)
prod.d.f Vk	7E98	E0 0111111010011	OV,UN,RO	Multiply reduce double vector (!VM)

Description: The product of the contents of scalar register Sk and the first VL elements of vector register Vk replaces Sk. Only elements of Vk with corresponding VM bit set (clear for *.f*) participate in the product.

Notes:

1. Initialize the scalar register properly for the first use of the *prod* reduce instruction (probably to 1).

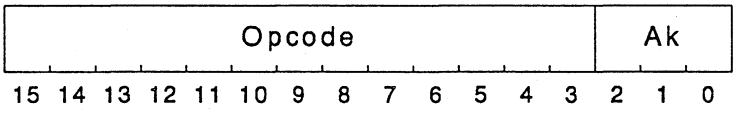
2. The sequence of the products performed by the hardware is *not* identical to the psuedocode sequence as noted above, i.e., it is implementation-specific. The C100 Series and C200 Series architectures each execute the reduction in a different order and may get different results. For more information, refer to the discussion of vector operations in the *CONVEX Architecture Reference*, "Instruction Set" chapter.
3. Either Vk or Sk may be used as a valid argument to this instruction. This instruction operates in distinct matched vector and scalar register pairs: (V0,S0), (V1,S1), (V2,S2), (V3,S3), (V4,S4), (V5,S5), (V6,S6), (V7,S7).

psh.w Ak — PUSH ADDRESS REGISTER

Purpose: To push the contents of an address register onto the stack

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
A0 = A0 - 4;
c(A0) = Ak;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
psh.w Ak	7D00	ST 0111110100000	None	Push an address register

Description: After A0 is decremented by four, the contents of address register Ak replace the contents of the memory addressed by A0.

Notes: None

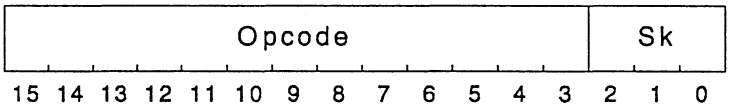
PUSH SCALAR REGISTER

psh.(w|l) Sk

Purpose: To push the contents of a scalar register onto the stack

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
A0 = A0 - 4;  /* psh.w */
c(A0) = Sk<31..0>;

A0 = A0 - 8;  /* psh.l */
c(A0) = Sk<63..0>;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
psh.w Sk	7D20	ST 0111110100100	None	Push Sk<31..0> onto the stack
psh.l Sk	7D28	ST 0111110100101	None	Push Sk<63..0> onto the stack

Description: *psh.w*: After A0 is decremented by four, the least significant 32 bits of scalar register Sk replace the contents of the word of memory addressed by A0.

psh.l: After A0 is decremented by eight, the contents of scalar register Sk replace the contents of the longword of memory addressed by A0.

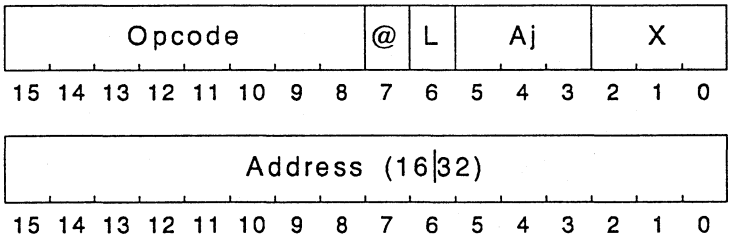
Notes: None

pshea <effa> — PUSH EFFECTIVE ADDRESS

Purpose: To push an effective address onto the stack

Architecture: C100 Series, C200 Series

Format:



Operation:

```
c(A0-4) = Effective_Address;
A0 = A0 - 4;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
pshea <effa>	0D00	ST 00001101	None	Push effective address

Description: The effective address is determined by evaluating the @, L, Aj, and address fields. Address register A0 is then decremented by four after the 32-bit effective address is evaluated. The effective address then replaces the word of memory specified by the contents of A0.

Notes:

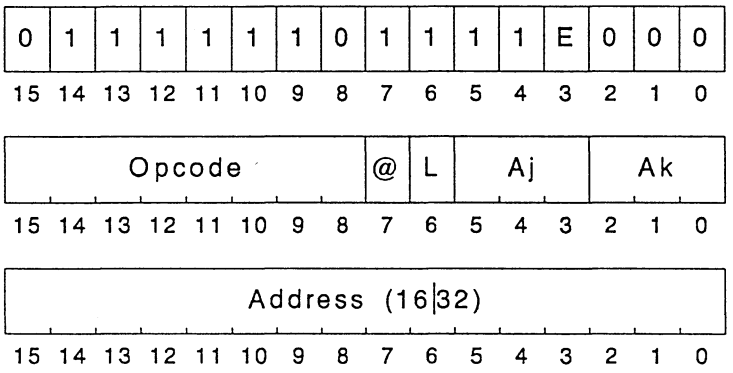
1. This instruction is useful for pushing arbitrary constants onto the stack.
2. No ring violation occurs if the developed effective address references an inner ring.
3. During address calculation, references to A0 use the original value of A0 (before the decrement).
4. The X field is ignored.

PUSH ADDRESS/RESOURCE — pshr Ak,<effa>

Purpose: To push the address register onto a resource structure

Architecture: C200 Series only

Format:



Operation:

```
if ( tas(effa.lock) ) {
     c(effa.data) = c(effa.data) + 4;
     c(effa.data + c(effa.data)) = Ak;
     Ak = c(effa.data);
     C = 1;
     tac(effa.lock);
} else {
     C = 0;
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
pshr Ak,<effa>	0900	E0 0000100100	See note 1	Push address register/resource

Description: The *pshr* instruction atomically pushes a word onto a resource structure and returns the new depth of the resource structure in address register Ak. The success or failure status of the push operation is returned in the Address Carry (C).

Notes:

1. Address Carry (C) of the PSW is affected as described by the preceding operation pseudocode.
2. This instruction is atomic.
3. The "valid" byte in the resource structure is ignored.
4. Refer to the *CONVEX Architecture Reference*, "Memory Management" chapter, for more information on the *pshr* instruction and resource structures.

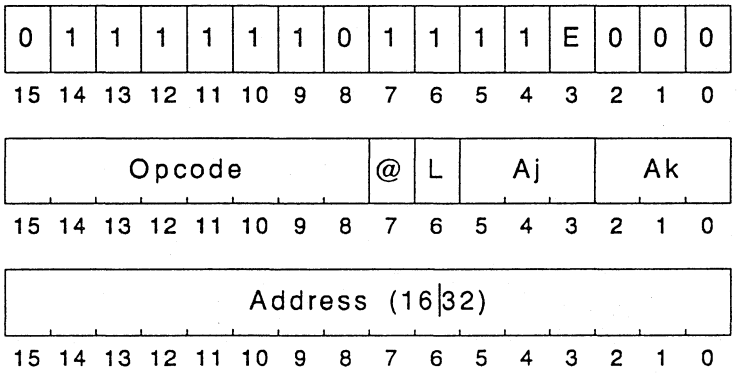
put.w Ak,<Ceffa>

PUT ADDRESS/COMMUNICATION

Purpose: To copy the contents of the address register into a communication register

Architecture: C200 Series only

Format:



Operation:

```
c(Ceffa)<31..0> = Ak;
```

Exceptions: Ring Violation (Invalid Communication Register Address)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
put.w Ak,<Ceffa>	2E00	E0 0010111000	CAT	Put address/communication

Description: A word of data is moved from Ak to c(Ceffa) bits <31..0>. Bits <63..32> of the addressed communication register are not modified. The lock bit, L(Ceffa), is not modified.

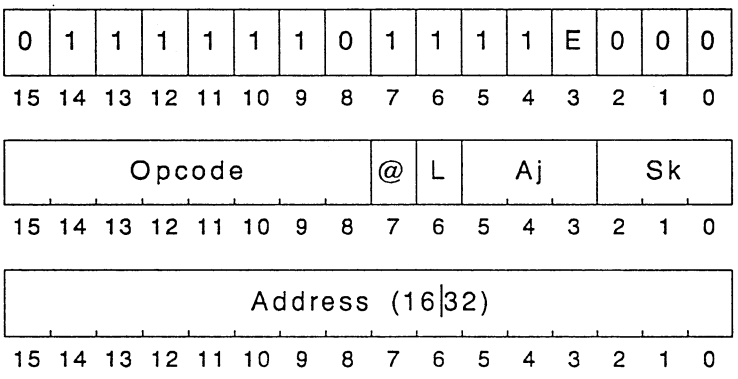
Notes: None

PUT SCALAR/COMMUNICATION — put.l Sk,<Ceffa>

Purpose: To copy the contents of a scalar register into a communication register

Architecture: C200 Series only

Format:



Operation: `c(Ceffa) = Sk;`

Exceptions: Ring Violation (Invalid Communication Register Address)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
put.l Sk,<Ceffa>	3600	E0 0011011000	CAT	Put scalar/communication

Description: A longword of data is moved from Sk to c(Ceffa). The lock bit, L(Ceffa), is not modified.

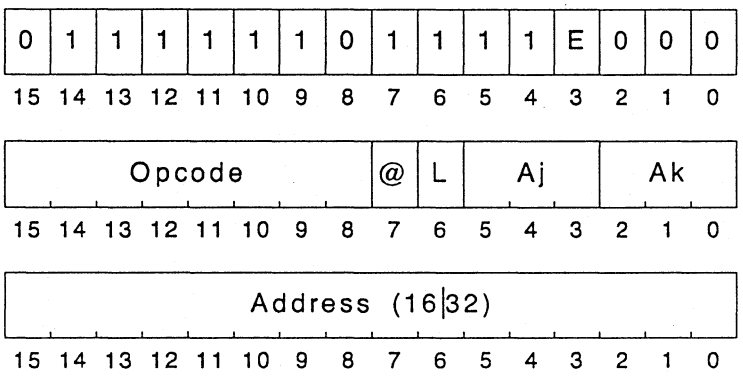
Notes: None

rcv.w <Ceffa>,Ak — RECEIVE COMMUNICATION/ADDRESS

Purpose: To receive the contents of a communication register into an address register

Architecture: C200 Series only

Format:



Operation:

```
Ak = c(Ceffa);
if (L(Ceffa) == 1) {
     L(Ceffa) = 0;
     C = 1;
}
} else {
     C = 0;
}
```

Exceptions: Ring Violation (Invalid Communication Register Address)
Deadlock Exception

Opcode:

Mnemonic	Hex	Binary	PSW	Description
rcv.w <Ceffa>,Ak	2b00	E0 0010101100	C,CAT	Receive communication/address

Description: If the lock bit for the addressed communication register is clear, the lock bit is not modified and "fail" status (C=0) is returned. If the lock bit for the addressed communication register is set, bits <31..0> are moved from c(Ceffa) to Ak, the lock bit is cleared, and "success" status (C=1) is returned. The contents of Ak are ***undefined*** if the communication register contains no valid data, i.e., if the lock bit was clear when the operation started.

Notes:

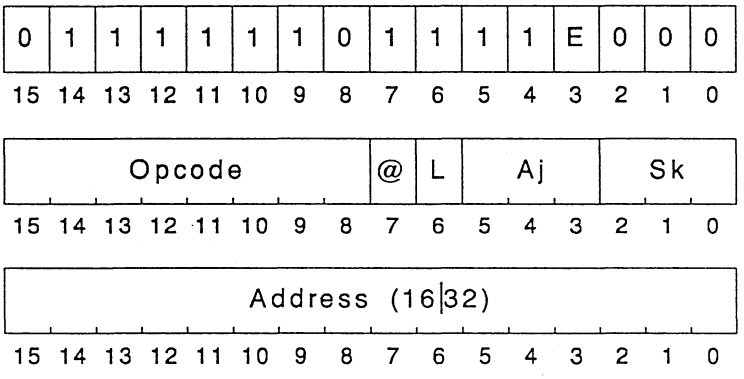
1. Address Carry (C) of the PSW is affected as described by the preceding operation pseudocode.
2. This is an atomic instruction.
3. The memory dual of this instruction is *rcvr.w <effa>,Ak*.

RECEIVE COMMUNICATION/SCALAR — rcv.l <Ceffa>,Sk

Purpose: To receive the contents of a communication register into a scalar register

Architecture: **C200 Series only**

Format:



Operation:

```
Sk = c(Ceffa);
if (L(Ceffa) == 1) {
     L(Ceffa) = 0;
     SC = 1;
} else {
     SC = 0;
}
```

Exceptions: Ring Violation (Invalid Communication Register Address)
Deadlock Exception

Opcode:

Mnemonic	Hex	Binary	PSW	Description
rcv.l <Ceffa>,Sk	3300	E0 0011001100	SC,CAT	Receive communication/scalar

Description: If the lock bit for the addressed communication register is clear, the lock bit is not modified, and "fail" status (SC=0) is returned. If the lock bit for the addressed communication register is set, a word of data is moved from c(Ceffa) to Sk, the lock bit is cleared, and "success" status (SC=1) is returned. The contents of Sk are undefined if the communication register contains no valid data, i.e., if the lock bit was clear when the operation started.

Notes:

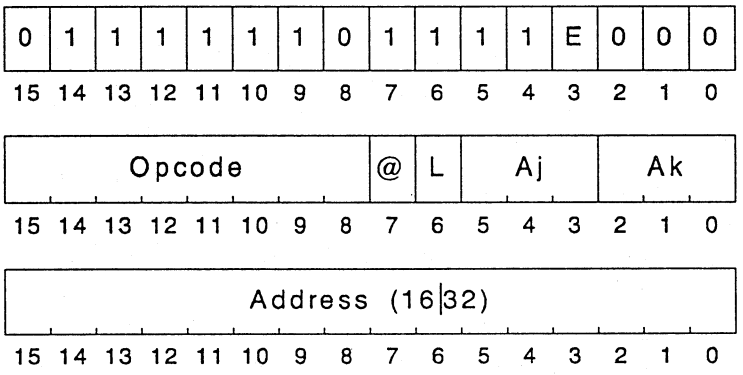
1. Scalar Carry (SC) of the PSW is affected as described by the preceding operation pseudocode
2. This is an atomic instruction.
3. The memory dual of this instruction is *rcvr.l <effa>,Sk*.

rcvr.w <effa>,Ak — RECEIVE RESOURCE/ADDRESS

Purpose: To receive the contents of a synchronized resource structure in memory into an address register

Architecture: C200 Series only

Format:



Operation:

```
msync;
if (tas(effa.lock)) {
     Ak = c(effa.data);
     if (effa.valid == 0xff) {
         c(effa.valid) = 0x00;
         C = 1;
     } else {
         C = 0;        /* fail - no valid data there */
     }
     msync;
     tac(effa.lock);
} else {
     C = 0;            /* fail - structure in transition */
}
```

Exceptions: Deadlock Exception

Opcode:

Mnemonic	Hex	Binary	PSW	Description
rcvr.w <effa>,Ak	0A00	E0 0000101000	C	Receive address register/resource

Description: The *rcvr.w* instruction atomically receives a word (*C(effa.data)*) from a resource structure into an address register Ak. If C is returned as 0, *rcvr.w* was unable to lock the resource structure (the resource structure was in transition) or the structure was "invalid." If C is returned as 1, the operation was successful, the received data is valid, and the resource structure is unlocked and invalid. The received data may only be considered valid if C is returned as 1. The contents of Ak are ***undefined*** if the communication register contains no valid data, i.e., if the lock bit was clear when the operation started.

Notes:

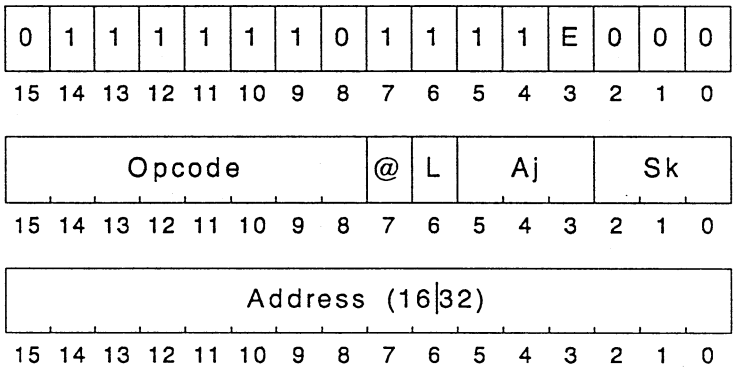
1. Address Carry (C) of the PSW is affected as described by the preceding operation pseudocode.
2. This instruction is atomic.

RECEIVE RESOURCE/SCALAR — rcvr.l <effa>,Sk

Purpose: To receive the contents of a synchronized resource structure in memory into a scalar register

Architecture: C200 Series only

Format:



Operation:

```
msync;
if (tas(effa.lock)) {
     Sk = c(effa.data);
     if (effa.valid == 0xff) {
         c(effa.valid) = 0x00;
         SC = 1;
     } else {
         SC = 0;          /* fail - no valid data there */
     }
     msync;
     tac(effa.lock);
} else {
     SC = 0;              /* fail - structure in transition */
}
```

Exceptions: Deadlock Exception

Opcode:

Mnemonic	Hex	Binary	PSW	Description
rcvr.l <effa>,Sk	0E00	E0 0000111000	SC	Receive scalar register/resource

Description: The *rcvr.l* instruction atomically receives a word from a resource structure into a scalar register. If SC is returned as 0, *rcvr.l* was unable to lock the resource structure (the resource structure was in transition) or the resource structure was "invalid." If SC is returned as 1, the operation was successful, the received data is valid, and the resource structure is unlocked and invalid. The contents of Ak are ***undefined*** if the communication register contains no valid data, i.e., if the lock bit was clear when the operation started.

Notes:

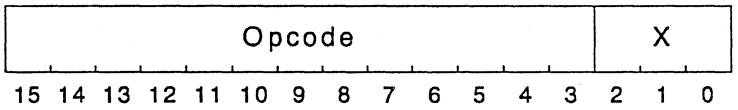
1. Scalar Carry (SC) of the PSW is affected as described by the preceding operation pseudocode.
2. This instruction is atomic.

rtn — RETURN FROM SUBROUTINE

Purpose: To return from a subroutine

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
SP = FP;  /* remove local save area */
Unwind stack to previous frame (restore stack built by call, calls,
sysc, or exception condition handler).
```

Exceptions: Ring Violation (Inward Return)
Ring Violation (Invalid Frame Length)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
rtn	7C90	ST 0111110010010	All bits	Return from subroutine call

Description: A subroutine exit is performed. The contents of the FP replace the SP to re-establish the top of the stack. This instruction assumes that the subroutine was entered using a *call*, *calls*, *sysc*, or an exception condition that pushed an extended return block.

There are four types of return blocks: short, long, extended, and context. Subroutine calls create short and long return blocks. Return from a short call causes address registers A6 and A7, the Processor Status Word (PSW), and the appropriate Program Counter (PC) to be returned. Return from a long call additionally restores scalar registers S1 through S7 besides address registers A1 through A5.

System calls and traps create extended blocks; faults create context blocks. 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:

- A check for outward ring crossing is made.
- If the ring crossing is inward, a system exception condition occurs.

  C100 Series The stack pointer after the pop is stored in bytes 72..75 of page 0 of the ring containing the *rtn* instruction.

  C200 Series The stack pointer after the pop is pushed on the system resource structure.

  When Frame Length is zero (FRL=00), indicating a context block, *rtnc* (return from context) must be executed. If *rtn* is executed instead of *rtnc*, then a frame length trap is generated.

Notes:

1. The PSW is loaded from the current stack frame.
2. 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.
3. The *rtn* instruction is restricted to be within the current ring. That is, if the current ring is 4, the most significant bit of the effective address is ignored. Otherwise, the most significant 3 bits of the effective address are ignored.
4. Before the PSW is pushed on the stack, all existing concurrent processing is completed. This ensures that all exception condition flags accurately reflect the state of the CPU.

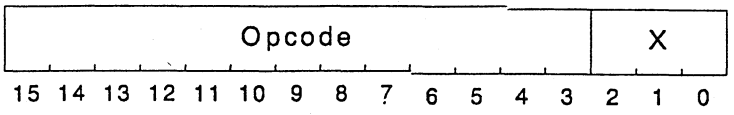
5. Refer to the *CONVEX Architecture Reference*, "Memory Management" chapter for more information on process return blocks, stack management, and subrouting calls and returns.
6. The X field is ignored.

rtnc — RETURN FROM CONTEXT BLOCK

Purpose: To return from a context block

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Processor_State = Context_Block;  /* FRL = 00 */
if (Architecture == C200 ) {
     (Push A0 on system resource structure);  /* C200 Series */
} else {
     (Ring 0, bytes <36..39)> = A0;       /* C100 Series */
}
A0 = (stack pointer from context block);
```

Exceptions: Ring Violation (Privileged Instruction)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
rtnc	7CA8	ST 0111110010101	None	Return from a context block

Description: The context block on the Ring 0 stack is popped.

C100 Series The new value of the stack pointer (A0) after the pop then replaces the context stack pointer in the word at byte address 0x36 of Ring 0.

C200 Series The new value of the stack pointer (A0) of the inner ring is pushed on the system resource structure pointed to by the system resource structure pointer located at byte address 0000 0048 of page 0 of Ring 0.

Finally, the stack pointer value contained within the context block just popped replaces A0.

Notes:

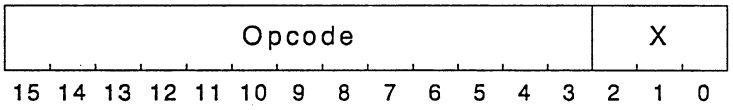
1. The entire processor state was stored in the context block at the time that the condition that initiated the exception occurred. The cause of the exception was loaded into A5 after the context block was stored. This permits the operating system to recover from the exception condition, if possible. The faulted instruction can then resume execution.
2. The Frame Length (FRL) bits in the PSW are checked. If the FRL bits are not clear (00), then a hard error is generated. Refer to the *CONVEX Architecture Reference*, "Exceptions and Interrupts" chapter, for more information on invalid frame lengths.
3. Before the PSW is pushed on the stack, all existing concurrent processing is completed. This ensures that all exception condition flags accurately reflect the state of the CPU.
4. Refer to the *CONVEX Architecture Reference*, "Memory Management" chapter, for more information on process return blocks, stack management, and subroutining calls and returns.
5. The X field is ignored.

POP PC and JUMP — rtnq

Purpose: To pop the top element of the stack into the Program Counter (PC)

Architecture: C100 Series, C200 Series

Format:



Operation:

```
temp = c(A0);
A0 = A0 + 4;
if (PC<31> == 1) {
      PC<30..0> = temp<30..0>;
} else {
      PC<28..0> = temp<28..0>;
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
rtnq	7C80	ST 0111110010000	None	Pop the PC and jump

Description: The top of the stack contains a previously pushed PC value. This value replaces the present PC (though adjusted to stay within the current ring); the stack is adjusted; and execution continues at the instruction referenced by the popped PC.

Notes:

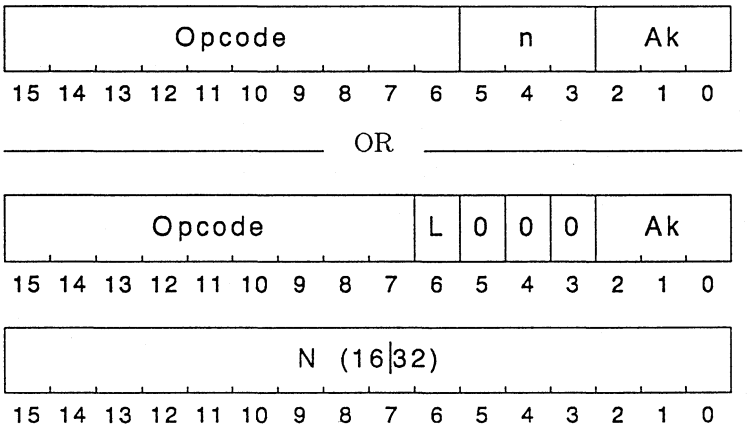
1. The current ring of execution does not change.
2. The *rtn* instruction is restricted to be within the current ring. That is, if the current ring is 4, the most significant bit of the effective address is ignored. Otherwise, the most significant 3 bits of the effective address are ignored.
3. The X field is ignored.

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

Purpose: To shift logically the contents of an address register by an immediate field

Architecture: C100 Series, C200 Series

Format:



Operation:

```
if (Immediate >= 0) {
     Ak = Ak << Immediate<7..0>;  /* shift left */
} else {
     Ak = Ak >> -Immediate<7..0>; /* shift right */
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
shf #n,Ak	4440	ST 0100010001	None	Logical shift short immediate
shf #N,Ak	1380	ST 00010011	None	Logical shift immediate

Description: Address register Ak logically shifts left by the number of bits specified by the (sign-extended) immediate (bits <7..0>) and replaces the old contents of Ak. Logical right shifts occur when the (sign-extended) immediate is negative. Logical shifts always zero-fill.

Notes:

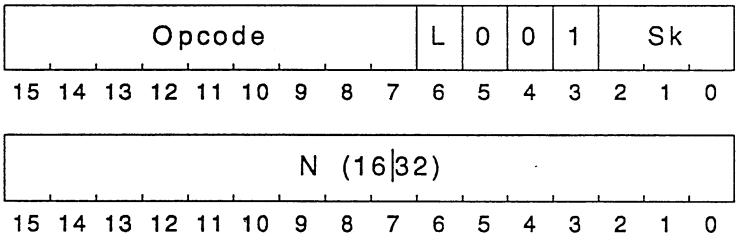
1. Sign extension does not occur for the 3 bits of the short immediate form.
2. Use multiplies or divides to implement arithmetic shifts.

LOGICAL SHIFT SCALAR/IMMEDIATE — shf #N,Sk

Purpose: To shift logically the contents of a scalar register by an immediate

Architecture: C100 Series, C200 Series

Format:



Operation:

```
if (Immediate >= 0) {
     Sk = Sk << Immediate<7..0>; /* shift left */
} else {
     Sk = Sk >> -Immediate<7..0>; /* shift right */
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
shf #N,Sk	1388	ST 000100111	None	Shift scalar/immediate

Description: The contents of scalar register Sk logically shift left by the number of bits specified by (sign-extended) bits <7..0> of the immediate and replace the old contents of Sk. Logical right shifts occur when the (sign-extended) immediate bits <7..0> represent a negative number. Logical shifts always zero-fill.

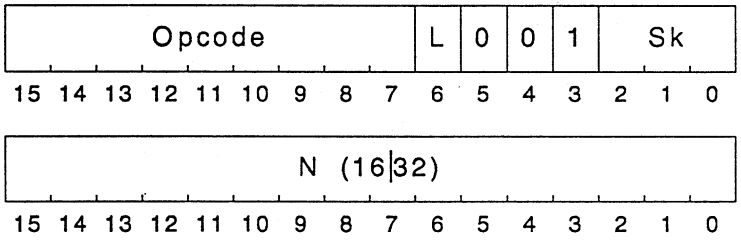
Notes: Use multiplies and divides to implement arithmetic shifts.

shf.w #N,Sk — LOGICAL SHIFT SCALAR/IMMEDIATE

Purpose: To shift logically the contents of the low order word of a scalar register by an immediate

Architecture: **C200 Series only**

Format:



Operation:

```
if (Immediate > 0) {
     Sk<31..0> = Sk<31..0> << Immediate<7..0>;     /* shift left */
} else {
     Sk<31..0> = Sk<31..0> >> -Immediate<7..0>;    /* shift right */
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
shf.w #N,Sk	1980	ST 000110011	None	Shift scalar word/immediate

Description: The contents of scalar register Sk logically shift left by the number of bits specified by (sign-extended) bits <7..0> of the immediate and replace the old contents of Sk. Logical right shifts occur when the (sign-extended) immediate bits <7..0> represent a negative number. Logical shifts always zero-fill.

Notes:

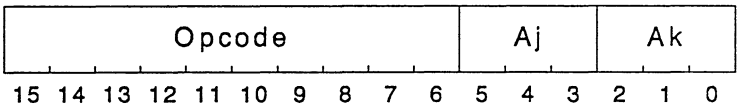
1. Use multiplies and divides to implement arithmetic shifts.
2. The *shf.w #N, Sk* instruction uses only the least significant 32 bits contained in a scalar register Sk in the shift operation.
3. Execution of this opcode in a C1 or C120 processor will result in an unimplemented instruction trap.

LOGICAL SHIFT ADDRESS/ADDRESS — shf Aj,Ak

Purpose: To shift logically the contents of an address register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
if (Aj<7..0> > 0) {
     Ak = Ak << Aj<7..0>; /* shift left */
} else {
     Ak = Ak >> -Aj<7..0>;    /* shift right */
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
shf Aj,Ak	5040	ST 0101000001	None	Shift an address

Description: Address register Ak logically shifts by the number of bits specified by the (sign-extended) bottom 8 bits of Aj and replaces the old contents of Ak. Logical left shifts occur when the sign-extension is positive. Logical right shifts occur when the sign-extension is negative. Logical shifts always zero-fill.

Notes: Use multiplies or divides to implement arithmetic shifts.

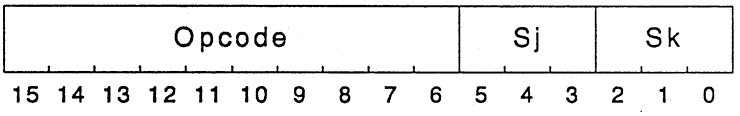
shf Sj,Sk

LOGICAL SHIFT SCALAR/SCALAR

Purpose: To shift logically the contents of a scalar register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
if (Sj<7..0> > 0) {
     Sk = Sk<63..0> << Sj<7..0>;  /* shift left */
} else {
     Sk = Sk<63..0> >> -Sj<7..0>; /* shift right */
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
shf Sj,Sk	5140	ST 0101000101	None	Shift a scalar

Description: Scalar register Sk logically shifts by the number of bits specified by the (sign-extended) bottom 8 bits of Sj and replaces the old contents of Sk. Logical left shifts occur when the sign-extension is positive. Logical right shifts occur when the (sign-extended) contents of Sj are negative. Logical shifts always zero-fill.

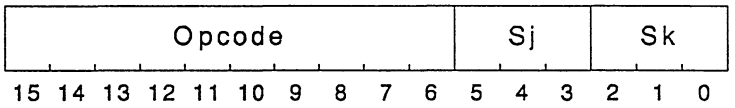
Notes: Use multiplies or divides to implement arithmetic shifts.

LOGICAL SHIFT SCALAR WORD/SCALAR — shf.w Sj,Sk

Purpose: To shift logically the contents of the lower order word of a scalar register

Architecture: **C200 Series only**

Format:



Operation:

```
if (Sj<7..0> >= 0) {
     Sk<31..0> = Sk<31..0> << Sj<7..0>;   /* shift left */
} else {
     Sk<31..0> = Sk<31..0> >> -Sj<7..0>;  /* shift right */
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
shf.w Sj,Sk	4440	E0 0100010001	None	Shift a scalar word

Description: Scalar register Sk logically shifts by the number of bits specified by the (sign-extended) bottom 8 bits of Sj and replaces the old contents of Sk. Logical left shifts occur when the sign-extension is positive. Logical right shifts occur when the (sign-extended) contents of Sj are negative. Logical shifts always zero-fill.

Notes:

1. Use multiplies or divides to implement arithmetic shifts.
2. The *shf.w Sj, Sk* instruction only uses the least significant 32 bits of Sk in the shift operation.
3. Execution of the *shf.w Sj, Sk* instruction on a C1 or C120 processor will result in an unimplemented instruction trap.

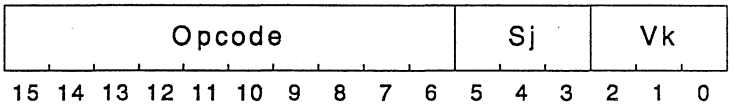
shf Sj,Vk

LOGICAL SHIFT VECTOR/SCALAR

Purpose: To shift logically the elements of a vector register by the contents of a scalar register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
for (a = 0; a < VL; a++) {
     if (Sj<7..0> >= 0) {
         Vk[a] = Vk[a] << Sj<7..0>;   /* shift left */
     } else {
         Vk[a] = Vk[a] >> -Sj<7..0>;  /* shift right */
     }
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
shf Sj,Vk	6300	ST 0110001100	None	Shift a vector accumulator

Description: Each of the first VL elements of vector register Vk, logically shifted by the number of bits specified by the (sign-extended) bottom eight bits of Sj, replace the old contents of Vk[i]. Logical left shifts occur when the sign-extension is positive. Logical right shifts occur when the (sign-extended) immediate is negative. Logical shifts always zero-fill.

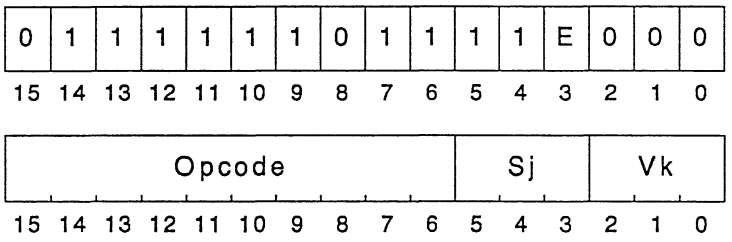
Notes: Use multiplies or divides to implement arithmetic shifts.

LOGICAL SHIFT VECTOR/SCALAR MASKED — shf.(t|f) Sj,Vk

Purpose: To logically shift the contents of a vector register by a scalar register under control of the Vector Merge (VM) register

Architecture: C200 Series only

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
      case TRUE:   /* .t */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 1) {        /* if VM<a> is TRUE */
                  if (Sj<7..0> >= 0) {
                      Vk[a] = Vk[a] << Sj<7..0>; /* left */
                  } else {
                      Vk[a] = Vk[a] >>- Sj<7..0>; /* right */
                  }
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
      case FALSE:  /* .f */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 0) {        /* if VM<a> is FALSE */
                  if (Sj<7..0> >= 0) {
                      Vk[a] = Vk[a] <<- Sj<7..0>; /* left */
                  } else {
                      Vk[a] = Vk[a] >> Sj<7..0>; /* right */
                  }
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
shf.t Sj,Vk	6300	E1 0110001100	None	Shift vector/scalar (VM)
shf.f Sj,Vk	6300	E0 0110001100	None	Shift vector/scalar (!VM)

Description: Each of the first VL elements of vector register Vk, logically shifted by the number of bits specified by the (sign-extended) bottom 8 bits of Sj, replace the old contents of Vk[i], if the corresponding VM bit is set (clear for *.f*). Logical left shifts occur when the sign-extension is positive. Logical right shifts occur when the (sign-extended) contents of Sj are negative. Logical shifts always zero-fill.

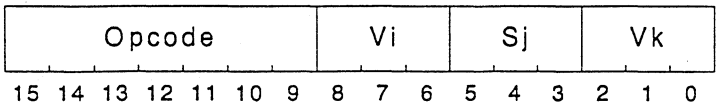
Notes: Use multiplies or divides to implement arithmetic shifts.

shf Vi,Sj,Vk — LOGICAL SHIFT VECTOR/SCALAR

Purpose: To logically shift the contents of a vector register by a scalar register and store the result in another vector register

Architecture: C200 Series only

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      if (Sj<7..0> >= 0) {
          Vk[a] = Vi[a] << Sj<7..0>;   /* shift left */
      } else {
          Vk[a] = Vi[a] >> -Sj<7..0>;  /* shift right */
      }
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
shf Vi,Sj,Vk	AE00	ST 1010111000	None	Shift a vector accumulator

Description: Each of the first VL elements of vector register Vi, logically shifted by the number of bits specified by the (sign-extended) bottom 8 bits of Sj, replace the old contents of Vk[i]. Logical left shifts occur when the sign-extension is positive. Logical right shifts occur when the (sign-extended) contents of Sj are negative. Logical shifts always zero-fill.

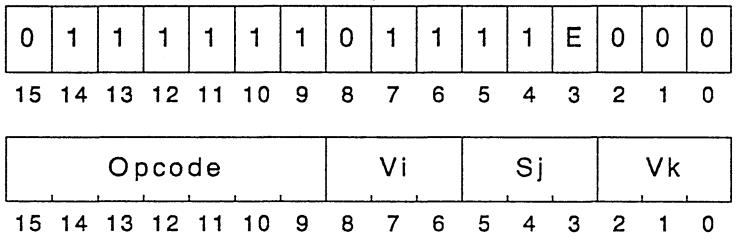
Notes: Use multiplies or divides to implement arithmetic shifts.

LOGICAL SHIFT VECTOR/SCALAR MASKED — shf.(t|f) Vi,Sj,Vk

Purpose: To logically shift the contents of a vector register by a scalar register under control of the Vector Merge (VM) register, and store the contents in another vector register

Architecture: C200 Series only

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
      case TRUE:   /* .t */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 1) {        /* if VM<a> is TRUE */
                  if (Sj<7..0> >= 0) {
                      Vk[a] = Vi[a] << Sj<7..0>; /* left */
                  } else {
                      Vk[a] = Vi[a] >> -Sj<7..0>; /* right */
                  }
              }
          }   /* end of for loop */
          break;   /* go to end of switch */
      case FALSE:  /* .f */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 0) {        /* if VM<a> is FALSE */
                  if (Sj<7..0> >= 0) {
                      Vk[a] = Vi[a] << Sj<7..0>; /* left */
                  } else {
                      Vk[a] = Vi[a] >> -Sj<7..0>; /* right */
                  }
              }
          }   /* end of for loop */
          break;   /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
shf.t Vi,Sj,Vk	AE00	E1 1010111000	None	Shift vector/scalar (VM)
shf.f Vi,Sj,Vk	AE00	E0 1010111000	None	Shift vector/scalar (!VM)

Description: Each of the first VL elements of vector register Vi, logically shifted by the number of bits specified by the (sign-extended) bottom eight bits of Sj, replaces the old contents of Vk[i], if the corresponding VM bit is set (clear for *.f*). Logical left shifts occur when the sign-extension is positive. Logical right shifts occur when the (sign-extended) contents of Sj are negative. Logical shifts always zero-fill.

Notes: Use multiplies or divides to implement arithmetic shifts.

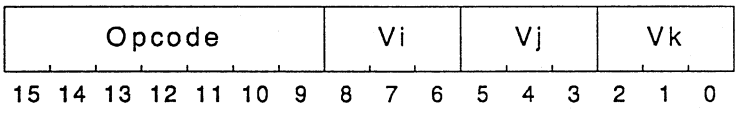
shf Vi,Vj,Vk

LOGICAL SHIFT VECTOR/VECTOR

Purpose: To logically shift the contents of one vector register by another vector register

Architecture: **C200 Series only**

Format:



Operation:

```
for (a = 0 a < VL; a++) {
      if (Vj[a]<7..0> >= 0) {
          Vk[a] = Vi[a] << Vj[a]<7..0>;     /* shift left */
      } else {
          Vk[a] = Vi[a] >> -Vj[a]<7..0>;    /* shift right */
      }
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
shf Vi,Vj,Vk	A600	ST 1010011	None	Shift vector/vector

Description: Each of the first VL elements of vector register Vi, logically shifted by the number of bits specified by the (sign-extended) bottom eight bits of the corresponding elements of Vj, replace the old contents of the corresponding elements of Vk. Logical left shifts occur when the sign-extension is positive. Logical right shifts occur when the (sign-extended) elements of Vj are negative. Logical shifts always zero-fill.

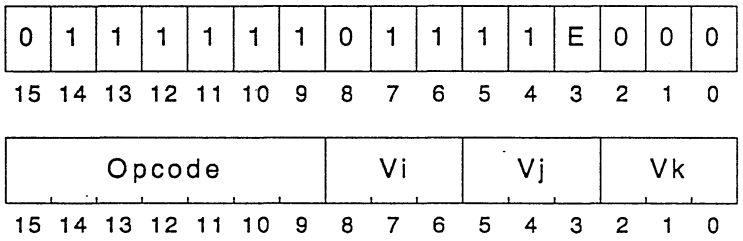
Notes: Use multiplies or divides to implement arithmetic shifts.

LOGICAL SHIFT VECTOR/VECTOR MASKED — shf.(t|f) Vi,Vj,Vk

Purpose: To logically shift the contents of one vector register by another vector register under control of the Vector Merge (VM) register

Architecture: **C200 Series only**

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
      case TRUE:   /* .t */
          for (a = 0 a < VL; a++) {
              if (VM<a> == 1) {        /* if VM<a> is TRUE */
                  if (Vj[a]<7..0> >= 0) {
                      Vk[a] = Vi[a] << Vj[a]<7..0>;
                  } else {
                      Vk[a] = Vi[a] >> -Vj[a]<7..0>;
                  }
              }
          }   /* end of for loop */
          break;   /* go to end of switch */
      case FALSE:  /* .f */
          for (a = 0 a < VL; a++) {
              if (VM<a> == 0) {        /* if VM<a> is FALSE */
                  if (Vj[a]<7..0> >= 0) {
                      Vk[a] = Vi[a] << Vj[a]<7..0>;
                  } else {
                      Vk[a] = Vi[a] >> -Vj[a]<7..0>;
                  }
              }
          }   /* end of for loop */
          break;   /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
shf.t Vi,Vj,Vk	A600	E1 1010011	None	Shift vector/vector (VM)
shf.f Vi,Vj,Vk	A600	E0 1010011	None	Shift vector/vector (!VM)

Description: Each of the first VL elements of vector register Vi, logically shifted by the number of bits specified by the (sign-extended) bottom eight bits of the corresponding elements of Vj, replace the old contents of the corresponding elements of Vk if the corresponding VM bit is set (clear for *.f*). Logical left shifts occur when the sign-extension is positive. Logical right shifts occur when the (sign-extended) elements of Vj are negative. Logical shifts always zero-fill.

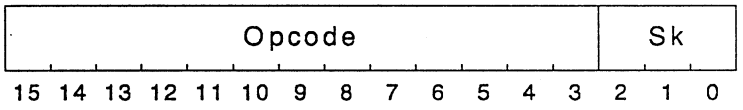
Notes: Use multiplies or divides to implement arithmetic shifts.

sin.(s|d) Sk

Purpose: To compute take the trigonometric sine of the contents of a scalar register

Architecture: **C200 Series only**

Format:



Operation: `Sk = sin(Sk)`

Exceptions: (s|d): Reserved Operand
Floating Instrinsic Error

Opcode:

Mnemonic	Hex	Binary	PSW	Description
sin.s Sk	7CC0	ST 0111110011000	RO,FIN,IEC	Sine of a single precision number
sin.d Sk	7CC8	ST 0111110011001	RO,FIN,IEC	Sine of a double precision number

Description: The sine of the contents of Sk replaces the contents of Sk.

Notes:

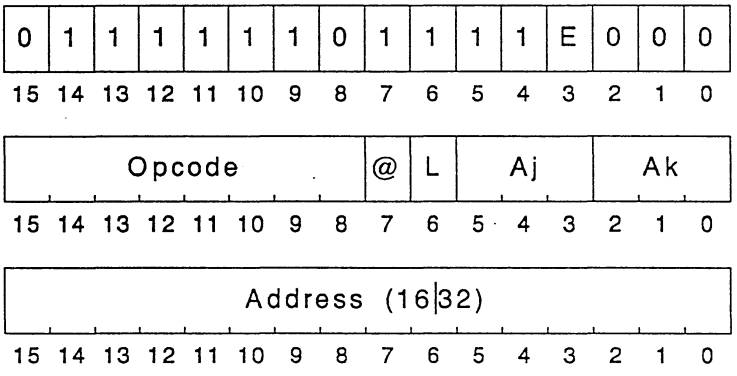
1. The input operand is interpreted as an angle in radians.
2. Intrinsic traps go through the same trap handler as other arithmetic traps (RO, FDZ, UN, etc.). If PSW <FUE> and/or PSW <FE> are set and and intrinsic traps are not (INE clear), these bits must be examined to determine the type of the current trap.
3. When PSW<FIN> is set, the PSW<IEC> bits contain a code that specifies the type of error encountered by the intrinsic instruction. Refer to the *CONVEX Architecture Reference*, "Register Set" chapter, "Program Status Word — C200 Series" section, for more information on the PSW<IEC> error codes and arithmetic trap conditions.

SEND ADDRESS/COMMUNICATION — snd.w Ak,<Ceffa>

Purpose: To send the contents of an address register to a communication register

Architecture: **C200 Series only**

Format:



Operation:

```
if (L(Ceffa) == 0) {
      c(Ceffa) = Ak;
      L(Ceffa) = 1;
      C = 1;
} else {
      C = 0;
}
```

Exceptions: Ring Violation (Invalid Communication Register Address)
Deadlock Exception

Opcode:

Mnemonic	Hex	Binary	PSW	Description
snd.w Ak,<Ceffa>	2F00	E0 0010111100	C,CAT	Send address/communication

Description: If the lock bit for the addressed communication register is set, the communication registers are not modified and "fail" status (C=0) is returned. If the lock bit for the addressed communication register is clear, a word of data is moved from Ak to c(Ceffa) bits <31..0>, the lock bit is set, and "success" status (C=1) is returned. Bits <63..32> of the addressed communication register are not modified.

Notes:

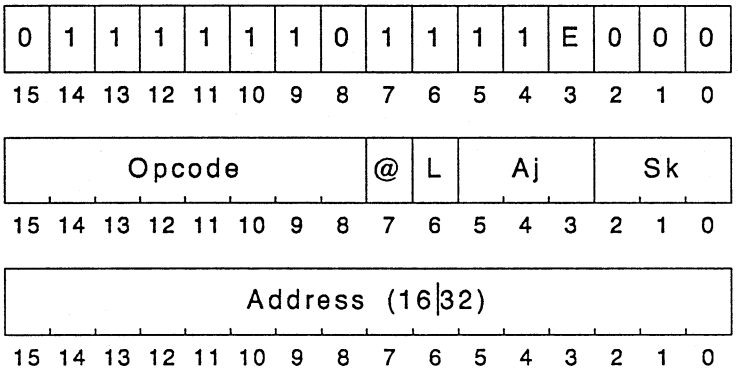
1. Address Carry (C) of the PSW is affected as described by the preceding operation pseudocode.
2. This is an atomic instruction.
3. The memory dual of this instruction is *sndr.w <effa>,Ak*.

snd.l Sk,<Ceffa> SEND COMMUNICATION/SCALAR

Purpose: To send the contents of a scalar register to a communication register

Architecture: **C200 Series only**

Format:



Operation:

```
if (L(Ceffa) == 0) {
     c(Ceffa) = Sk;
     L(Ceffa) = 1;
     SC = 1;
} else {
     SC = 0;
}
```

Exceptions: Ring Violation (Invalid Communication Register Address)
Deadlock Exception

Opcode:

Mnemonic	Hex	Binary	PSW	Description
snd.l Sk,<Ceffa>	3700	E0 0011011100	SC,CAT	Send scalar/communication

Description: If the lock bit for the addressed communication register is set, the communication registers are not modified and "fail" status (SC=0) is returned. If the lock bit for the addressed communication register is clear, a word of data is moved from Sk to c(Ceffa), the lock bit is set, and "success" status (SC=1) is returned.

Notes:

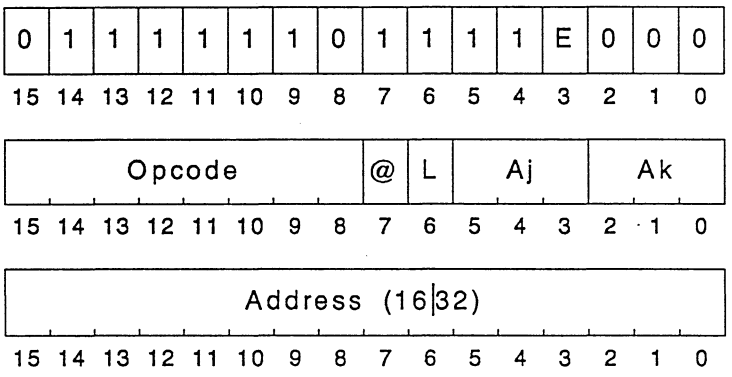
1. Scalar Carry (SC) of the PSW is affected as described by the preceding operation pseudocode.
2. This is an atomic instruction.
3. The memory dual of this instruction is *sndr.l <effa>,Sk*.

SEND ADDRESS/RESOURCE — sndr.w Ak,<effa>

Purpose: To send the contents of an address register to a synchronized resource structure in memory

Architecture: **C200 Series only**

Format:



Operation:

```
msync;
if (tas(effa.lock)) {
     if (effa.valid == 0x00) {
         c(effa.data) = Ak;
         c(effa.valid) = 0xFF;
         C = 1;
     } else {
         C = 0;          /* fail - already valid data there */
     }
     msync;
     tac(effa.lock);
} else {
     C = 0;              /* fail - structure in transition */
}
```

Exceptions: Deadlock Exception

Opcode:

Mnemonic	Hex	Binary	PSW	Description
sndr.w Ak,<effa>	0C00	E0 0000110000	C	Send address register/resource

Description: The *sndr.w* instruction atomically sends a word from an address register to a resource structure if it previously contained no valid data. If C is returned as 0, *sndr.w* was unable to lock the resource structure or there was valid data already present (the structure was "valid"). If C is returned as 1, the operation was successful and the resource structure is unlocked with valid data in it.

Notes:

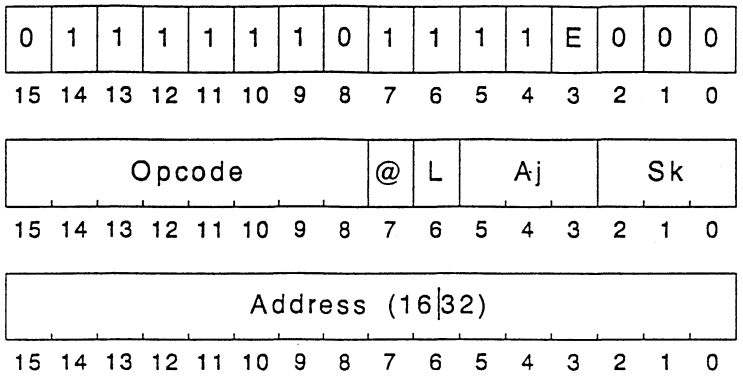
1. Address Carry (C) of the PSW is affected as described by the preceding operation pseudocode.
2. This instruction is atomic.
3. The communication register dual of this instruction is *snd.w <Ceffa>,Ak*.

sndr.l Sk,<effa> — SEND RESOURCE/SCALAR

Purpose: To send the contents of a scalar register to a synchronized long resource structure in memory

Architecture: C200 Series only

Format:



Operation:

```
msync;
if (tas(effa.lock)) {
     if (effa.valid == 0x00) {
         c(effa.data) = Sk;
         c(effa.valid) = 0xFF;
         SC = 1;
     } else {
         SC = 0;          /* fail - already valid data there */
     }
     msync;
     tac(effa.lock);
} else {
     SC = 0;              /* fail - structure in transition */
}
```

Exceptions: Deadlock Exception

Opcode:

Mnemonic	Hex	Binary	PSW	Description
sndr.l Sk,<effa>	0D00	E0 0000110100	SC	Send scalar register/resource

Description: The *sndr.l* instruction atomically sends a longword from a scalar register to a resource structure if the resource structure previously contained no valid data. If SC is returned as 0, *sndr.l* was unable to lock the resource structure or there was valid data already present (the structure was "valid"). If SC is returned as 1, the operation was successful and the resource structure is unlocked with valid data in it.

Notes:

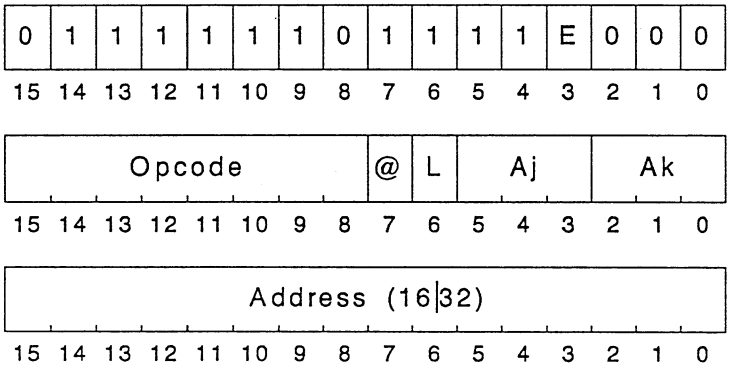
1. Scalar Carry (SC) of the PSW is affected as described by the preceding operation pseudocode.
2. This instruction is atomic.
3. The communication register dual of this instruction is *snd.l <Ceffa>,Sk*.

SPAWN A FORK — spawn <effa>,Ak

Purpose: To post the need for as many CPUs as possible to join in a process computation

Architecture: C200 Series only

Format:



Operation:

```
if (C = snd.l(forklck, FP::AP)) {      /* C = 1 if snd.l() succeeds */
      fork.PC    = <effa>;
      fork.PSW   = PSW;
      fork.source_PC = PC;
      snd.l(forkposted, SPAWNED::Ak);
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
spawn <effa>,Ak	0500	E0 0000010100	C	Spawn a fork

Description: This instruction posts a fork that can be taken by any CPU to create a concurrent thread of execution at the specified effective address. The fork remains posted, continuing to be taken by any idle CPUs, until one thread reaches a *join* instruction, which sets *fork.type* to "STOPPED," inhibiting other CPUs from taking the fork. If a fork is already posted, it must be taken prior to posting another.

Note that the *snd* to *forklck* loads FP into *fork.FP* and AP into *fork*.AP. The communication registers are loaded only if *forklck* is zero, i.e., there is no currently posted fork. The *snd* to *forkposted* loads the constant "SPAWNED" into *fork.type* and Ak into *fork.SP*.

Notes:

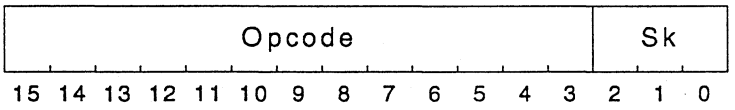
1. Address Carry (C) of the PSW is set to 1 if the fork is successfully posted, otherwise C is cleared to 0.
2. A *spawn* will potentially add multiple processors to the process. Use *pfork* to add a single processor.
3. Once a fork is posted, it must either be taken by a CPU or cleared with *cfork* before another fork can be successfully posted.
4. Refer to the *CONVEX Architecture Reference*, "Multiprocessor Management" chapter, for more information on *pfork* and forking operations.

sqrt.(s|d) Sk — SQUARE ROOT SCALAR

Purpose: To compute the square root of the contents of a scalar register

Architecture: **C200 Series only**

Format:



Operation:

```
Sk = sqrt(Sk);
```

Exceptions: (s|d): Reserved Operand
Floating Instrinsic Error

Opcode:

Mnemonic	Hex	Binary	PSW	Description
sqrt.s Sk	7DD0	ST 0111110111010	RO,FIN,IEC	Square root of a single float
sqrt.d Sk	7DD8	ST 0111110111011	RO,FIN,IEC	Square root of a double float

Description: The square root of the contents of Sk replaces the contents of Sk.

Notes:

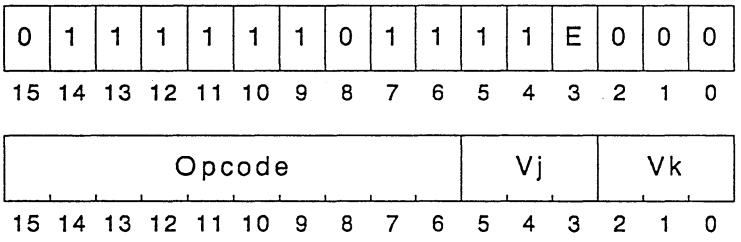
1. Intrinsic traps go through the same trap handler as other arithmetic traps (RO, FDZ, UN, etc.). If PSW <FUE> and/or PSW <FE> are set and and intrinsic traps are not (INE clear), these bits must be examined to determine the type of the current trap.
2. When PSW<FIN> is set, the PSW<IEC> bits contain a code that specifies the type of error encountered by the intrinsic instruction. Refer to the *CONVEX Architecture Reference*, "Register Set" chapter, "Program Status Word — C200 Series" section, for more information on the PSW<IEC> error codes and arithmetic trap conditions.

SQUARE ROOT VECTOR — sqrt.(s|d) Vj,Vk

Purpose: To compute the square root of the contents a vector register

Architecture: **C200 Series only**

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      Vk[a] = sqrt(Vj[a]);
}
```

Exceptions: (s|d): Reserved Operand
Floating Instrinsic Error

Opcode:

Mnemonic	Hex	Binary	PSW	Description
sqrt.s Vj,Vk	5D00	E0 0101110100	RO,FIN,IEC	Square root single vector/vector
sqrt.d Vj,Vk	5D40	E0 0101110101	RO,FIN,IEC	Square root double vector/vector

Description: Each of the contents of the first VL elements of vector register Vk is replaced by the square root of the corresponding element of vector register Vj. If any of the first VL elements of Vj is negative, FIN is set in the PSW, the appropriate error code is loaded into PSW <IEC>, and the corresponding element of Vk is loaded with the square root of the absolute value of Vj.

Notes:

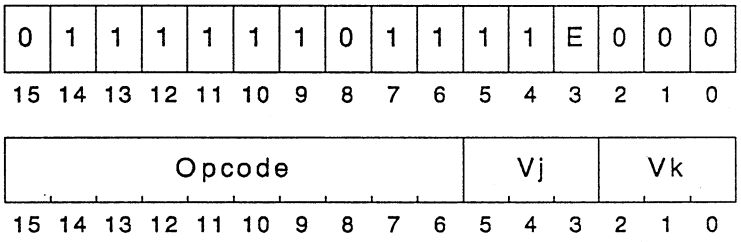
1. Intrinsic traps go through the same trap handler as other arithmetic traps (RO, FDZ, UN, etc.). If PSW <FUE> and/or PSW <FE> are set and and intrinsic traps are not (INE clear), these bits must be examined to determine the type of the current trap.
2. When PSW<FIN> is set, the PSW<IEC> bits contain a code that specifies the type of error encountered by the intrinsic instruction. Refer to the *CONVEX Architecture Reference*, "Register Set" chapter, "Program Status Word — C200 Series" section, for more information on the PSW<IEC> error codes and arithmetic trap conditions.

sqrt.(s|d).(t|f) Vj,Vk — SQUARE ROOT VECTOR MASKED

Purpose: To compute the square root of the contents of a vector register under control of the Vector Merge (VM) register

Architecture: **C200 Series only**

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
      case TRUE:   /* .t */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 1) {        /* if VM<a> is TRUE */
                  Vk[a] = sqrt(Vj[a]);
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
      case FALSE:  /* .f */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 0) {        /* if VM<a> is FALSE */
                  Vk[a] = sqrt(Vj[a]);
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
} /* end of switch */
```

Exceptions: (s|d): Reserved Operand
Floating Instrinsic Error

Opcode:

Mnemonic	Hex	Binary	PSW	Description
sqrt.s.t Vj,Vk	5F00	E1 0101111100	RO,FIN,IEC	Square root single (VM)
sqrt.s.f Vj,Vk	5F00	E0 0101111100	RO,FIN,IEC	Square root single (!VM)
sqrt.d.t Vj,Vk	5F40	E1 0101111101	RO,FIN,IEC	Square root double (VM)
sqrt.d.f Vj,Vk	5F40	E0 0101111101	RO,FIN,IEC	Square root double (!VM)

Description: Each of the contents of the first VL elements of vector register Vk is replaced by the square root of the corresponding element of vector register Vj, if the corresponding VM bit is set (clear for *.f*). If any of the first VL elements of Vj that are enabled for the operation (VM bit is 1 for *.t*, 0 for *.f*) is negative, FIN is set in the PSW, the appropriate error code is loaded into PSW <IEC>, and the corresponding element of Vk is loaded with the square root of the absolute value of Vj.

Notes:

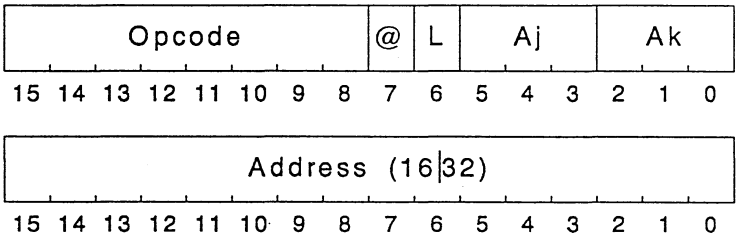
1. Intrinsic traps go through the same trap handler as other arithmetic traps (RO, FDZ, UN, etc.). If PSW <FUE> and/or PSW <FE> are set and and intrinsic traps are not (INE clear), these bits must be examined to determine the type of the current trap.
2. When PSW<FIN> is set, the PSW<IEC> bits contain a code that specifies the type of error encountered by the intrinsic instruction. Refer to the *CONVEX Architecture Reference*, "Register Set" chapter, "Program Status Word — C200 Series" section, for more information on the PSW<IEC> error codes and arithmetic trap conditions.

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

Purpose: To store the contents of an address register into memory

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
c(Effective_Address) = Ak;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
st.b Ak,<effa>	2C00	ST 00101100	None	Store address register byte
st.h Ak,<effa>	2D00	ST 00101100	None	Store address register halfword
st.w Ak,<effa>	2E00	ST 00101110	None	Store address register word

Description: The contents of the address register Ak replace the contents the memory location referenced by the effective address.

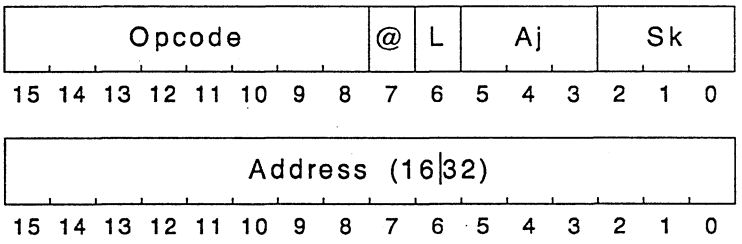
Notes: Higher order bits of the Ak register are ignored for byte and halfword stores.

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

Purpose: To store the contents of a scalar register into memory

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
c(Effective_Address) = Sk;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
st.b Sk,<effa>	3400	ST 00110100	None	Store scalar byte
st.h Sk,<effa>	3500	ST 00110100	None	Store scalar halfword
st.w Sk,<effa>	3600	ST 00110100	None	Store scalar word
st.l Sk,<effa>	3700	ST 00110111	None	Store scalar longword
st.s Sk,<effa>	3600	ST 00110110	None	Store scalar single float
st.d Sk,<effa>	3700	ST 00110111	None	Store scalar double float

Description: The contents of scalar register Sk replace the contents of memory specified by the effective address.

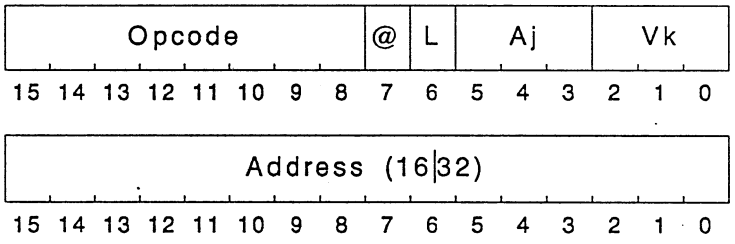
Notes: Single precision floating point data and 32-bit integer data occupy the same bit positions within a scalar register. The *.s* and *.d* forms rename the *.w* and *.l* forms, respectively, for convenience.

STORE VECTOR REGISTER — st.(b|h|w|l|s|d) Vk,<effa>

Purpose: To store the elements of a vector register into memory

Architecture: C100 Series, C200 Series

Format:



Operation:

```
temp = Effective_Address
for (a = 0; a < VL; a++) {
      c(temp) = Vk[a];
      temp = temp + VS;
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
st.b Vk,<effa>	3C00	ST 001111000	None	Store vector byte
st.h Vk,<effa>	3D00	ST 001111010	None	Store vector halfword
st.w Vk,<effa>	3E00	ST 001111100	None	Store vector word
st.l Vk,<effa>	3F00	ST 001111110	None	Store vector longword
st.s Vk,<effa>	3E00	ST 001111100	None	Store vector single float
st.d Vk,<effa>	3F00	ST 001111110	None	Store vector double float

Description: VL elements of vector Vj replace VL memory elements. The address produced by evaluating the @, L, Aj, and address fields specifies the first memory element. Successive elements come from addresses formed by incrementing the effective address by the contents of the Vector Stride (VS) register. The signed value of VS is the distance in bytes.

Notes:

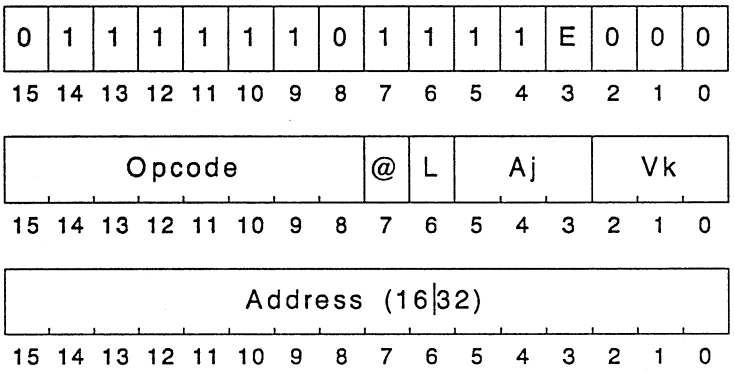
1. The *.s* and *.d* forms rename the *.w* and *.l* forms, respectively, for convenience.
2. If the absolute value of VS is less than the size of the elements being stored, then results are unpredictable.

st.(b|h|w|l|s|d).(t|f) Vk,<effa> STORE VECTOR REGISTER MASKED

Purpose: To store a vector from a vector register under control of the Vector Merge (VM) register

Architecture: C200 Series only

Format:



Operation:

```
temp = Effective_Address;
switch (E) { /* prefix bit<3> */
     case TRUE:  /* .t */
          for (a = 0; a < VL; a++) {
               if (VM<a> == 1) {        /* if VM<a> is TRUE */
                    c(temp) = Vk[a];
               }
               temp = temp + VS;
          }   /* end of for loop */
          break;  /* go to end of switch */
     case FALSE: /* .f */
          for (a = 0; a < VL; a++) {
               if (VM<a> == 0) {        /* if VM<a> is FALSE */
                    c(temp) = Vk[a];
               }
               temp = temp + VS;
          }   /* end of for loop */
          break;  /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
st.b.t Vk,<effa>	3C00	E1 001111000	None	Store vector byte (VM)
st.b.f Vk,<effa>	3C00	E0 001111000	None	Store vector byte (!VM)
st.h.t Vk,<effa>	3D00	E1 001111010	None	Store vector halfword (VM)
st.h.f Vk,<effa>	3D00	E0 001111010	None	Store vector halfword (!VM)
st.w.t Vk,<effa>	3E00	E1 001111100	None	Store vector word (VM)
st.w.f Vk,<effa>	3E00	E0 001111100	None	Store vector word (!VM)
st.l.t Vk,<effa>	3F00	E1 001111110	None	Store vector longword (VM)
st.l.f Vk,<effa>	3F00	E0 001111110	None	Store vector longword (!VM)
st.s.t Vk,<effa>	3E00	E1 001111100	None	Store vector single float (VM)
st.s.f Vk,<effa>	3E00	E0 001111100	None	Store vector single float (!VM)
st.d.t Vk,<effa>	3F00	E1 001111110	None	Store vector double float (VM)
st.d.f Vk,<effa>	3F00	E0 001111110	None	Store vector double float (!VM)

Description: VL elements of vector Vj replace VL memory elements if the corresponding VM bit is set (clear for *.f*). The address produced by evaluating the L, @, Aj, and address fields specifies the first memory element. Successive elements come from addresses formed by incrementing the effective address by the contents of the VS register. The

signed value of VS is the distance in bytes.

Notes:

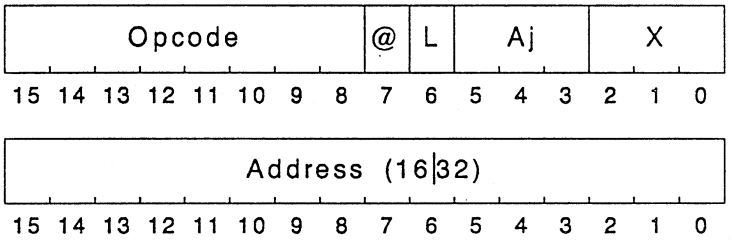
1. The *.s.(t|f)* and *.d.(t|f)* forms rename the *.w.(t|f)* and *.l.(t|f)* forms for convenience.
2. If the absolute value of VS is less than the size of the elements being stored, then results are unpredictable.

st.l VLS, <effa> — STORE VS AND VL

Purpose: To store the contents of the Vector Stride (VS) register and the Vector Length (VL) register into memory

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
c(Effective_Address)<63..32> = VS;
c(Effective_Address)<31..0> = VL;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
st.l VLS,<effa>	0E00	ST 000011100	None	Store VS and VL to memory

Description: The contents of the 32-bit VS register replace the word specified by the effective address. The contents of the 32-bit VL register replace the word specified by four more than the effective address.

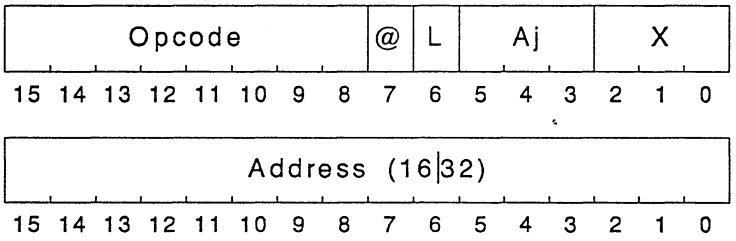
Notes:

1. The X field is ignored.

Purpose: To store the contents of the Vector Merge (VM) register into memory

Architecture: C100 Series, C200 Series

Format:



Operation: `c(Effective_Address)<127..0> = VM`

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
st.x VM,<effa>	0F00	ST 000011110	None	Store VM into memory

Description: The contents of the VM register replace the 16 bytes beginning with the byte referenced by the effective address.

Notes:

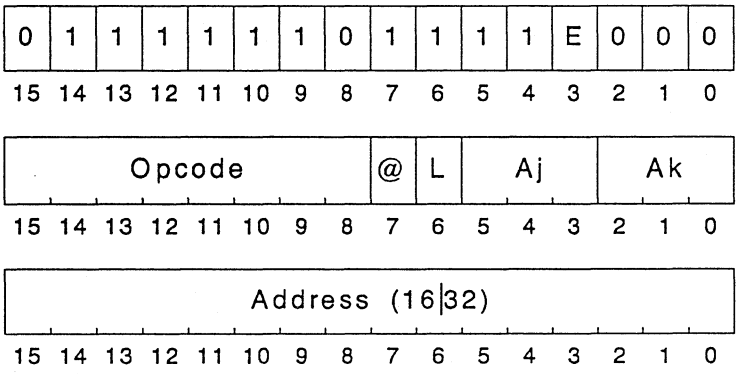
1. VM <127..120> are stored in the byte referenced by the effective address.
2. VM <7..0> are stored in the byte referenced by the effective address plus 15.
3. The X field is ignored.

stcmr Ak,<effa> — STORE COMMUNICATION REGISTERS

Purpose: Store a specified set of communication registers in memory

Architecture: **C200 Series only**

Format:



Operation:

```
/* For the CIR specified in Ak: */

/* assume current CIR = Ak */
(update CPU timer register for current ring);

if ((ring 0 comm register modified bit) == 1 ) {
      (Store the hardware communication registers);
      (Store the ring 0 communication registers);
      (Store the hardware lock bits);
      (Store the ring 0 lock bits);
}
if ( (ring 4 comm register modified bit) == 1 ) {
      (Store the ring 4 communication registers);
      (Store the ring 4 lock bits);
}
(ring 0 comm register valid bit) = (ring 0 comm register modified bit);
(ring 4 comm register valid bit) = (ring 4 comm register modified bit);
```

Exceptions: Ring Violation (Privileged Instruction)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
stcmr Ak,<effa>	0700	E0 0000011100	None	Store communication registers

Description: The communication register set index contained in Ak specifies a communication register set that is stored in memory.

The lock bit longwords are accumulated as the communication registers are stored. The memory map that defines the exact format of the stored data is implementation-specific. Refer to the *CONVEX Architecture Reference*, "Implementation-Specific" chapter for more information concerning *stcmr/ldcmr* memory map. If the communication registers have not been modified since last loaded via the *ldcmr* instruction, then the communication registers are not stored into memory.

Notes:

1. Prior to the communication registers in memory are the register set modified bit longword, and enough longword locations to hold the lock bits. A full longword contains 64 lock bits.
2. The lock bits in the physical communication registers are cleared by this instruction.

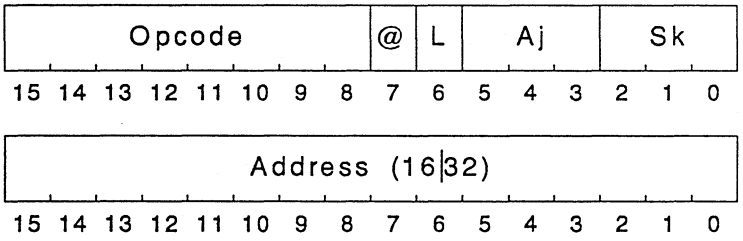
3. The communication register modified bits are unchanged by this instruction.

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

Purpose: To store the contents of a scalar register repetitively into memory

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
temp = Effective_Address;
for (a = 0; a < VL; a++) {
      c<temp> = Sk;
      temp = temp + VS;
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
ste.b Sk,<effa>	2400	ST 001001000	None	Store an extended scalar byte
ste.h Sk,<effa>	2500	ST 001001010	None	Store an extended scalar halfword
ste.w Sk,<effa>	2600	ST 001001100	None	Store an extended scalar word
ste.l Sk,<effa>	2700	ST 001001110	None	Store an extended scalar longword
ste.s Sk,<effa>	2600	ST 001001100	None	Store an extended scalar single float
ste.d Sk,<effa>	2700	ST 001001110	None	Store an extended scalar double float

Description: The contents of scalar register Sk replace VL elements of memory. The effective address is produced by evaluating the @, L, Aj, and address fields and specifies the first memory address to be replaced. Addresses of successive elements are formed by incrementing the effective address by the contents of the Vector Stride (VS) register. The signed value of VS is the distance in bytes.

Notes:

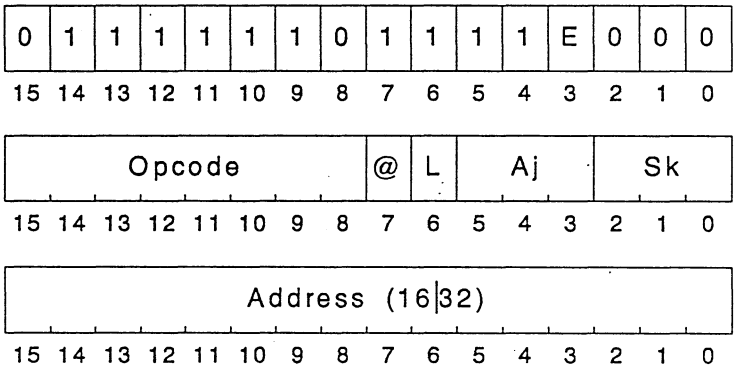
1. The *.s* and *.d* forms rename the *.w* and *.l* forms, respectively, for convenience.
2. If the absolute value of VS is less than the size of the elements being stored, results are unpredictable.

STORE SCALAR EXTENDED MASKED — ste.(b|h|w|l|s|d).(t|f) Sk,<effa>

Purpose: To store a scalar register repetitively into memory under control of the Vector Merge (VM) register

Architecture: **C200 Series only**

Format:



Operation:

```
temp = Effective_Address;
switch (E) { /* prefix bit<3> */
     case TRUE:  /* .t */
          for (a = 0; a < VL; a++) {
               if (VM<a> == 1) {        /* if VM<a> is TRUE */
                    c<temp> = Sk;
               }
               temp = temp + VS;
          }   /* end of for loop */
          break;  /* go to end of switch */
     case FALSE: /* .f */
          for (a = 0; a < VL; a++) {
               if (VM<a> == 0) {        /* if VM<a> is FALSE */
                    c<temp> = Sk;
               }
               temp = temp + VS;
          }   /* end of for loop */
          break;  /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
ste.b.t Sk,<effa>	2400	E1 001001000	None	Store extended scalar byte (VM)
ste.b.f Sk,<effa>	2400	E0 001001000	None	Store extended scalar byte (!VM)
ste.h.t Sk,<effa>	2500	E1 001001010	None	Store extended scalar halfword (VM)
ste.h.f Sk,<effa>	2500	E0 001001010	None	Store extended scalar halfword (!VM)
ste.w.t Sk,<effa>	2600	E1 001001100	None	Store extended scalar word (VM)
ste.w.f Sk,<effa>	2600	E0 001001100	None	Store extended scalar word (!VM)
ste.l.t Sk,<effa>	2700	E1 001001110	None	Store extended scalar longword (VM)
ste.l.f Sk,<effa>	2700	E0 001001110	None	Store extended scalar longword (!VM)
ste.s.t Sk,<effa>	2600	E1 001001100	None	Store extended scalar single (VM)
ste.s.f Sk,<effa>	2600	E0 001001100	None	Store extended scalar single (!VM)
ste.d.t Sk,<effa>	2700	E1 001001110	None	Store extended scalar double (VM)
ste.d.f Sk,<effa>	2700	E0 001001110	None	Store extended scalar double (!VM)

Description: The contents of scalar register Sk replace VL elements of memory. The effective address produced by evaluating the L, @, Aj, and address fields specifies the first memory address to be replaced. Successive elements' addresses are formed by incrementing the effective address by the contents of the Vector Stride (VS) register.

These stores are performed if the corresponding VM bit is set (clear for *.f*). The signed value of VS is the distance in bytes.

Notes:

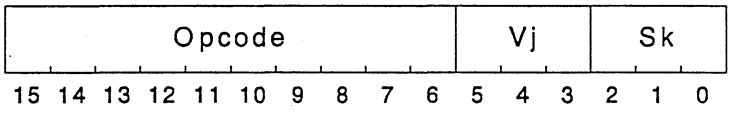
1. The *.s.(t|f)* and *.d.(t|f)* forms rename the *.w.(t|f)* and *.l.(t|f)* forms for convenience.
2. If the absolute value of VS is less than the size of the elements being stored, then results are unpredictable.

STORE SCALAR/VECTOR INDEX — stvi.(b|h|w|l|s|d) Sk,Vj

Purpose: To store the contents of a scalar register using the contents of a vector register as a set of indices

Architecture: C100 Series, C200 Series

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      c(Vj[a]<31..0> + A5) = Sk;
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
stvi.b Sk,Vj	7B00	ST 0111101100	None	Scalar index store vector byte
stvi.h Sk,Vj	7B40	ST 0111101101	None	Scalar index store vector halfword
stvi.w Sk,Vj	7B80	ST 0111101110	None	Scalar index store vector word
stvi.l Sk,Vj	7BC0	ST 0111101111	None	Scalar index store vector longword
stvi.s Sk,Vj	7B80	ST 0111101110	None	Scalar index store vector single float
stvi.d Sk,Vj	7BC0	ST 0111101111	None	Scalar index store vector double float

Description: The bottom 32 bits of each of the first VL elements of vector register Vj specify a set of address offsets. Each address offset is summed with the contents of address register A5 to yield a set of addresses that specify the destinations of operands whose contents are replaced by the contents of scalar register Sk.

Notes:

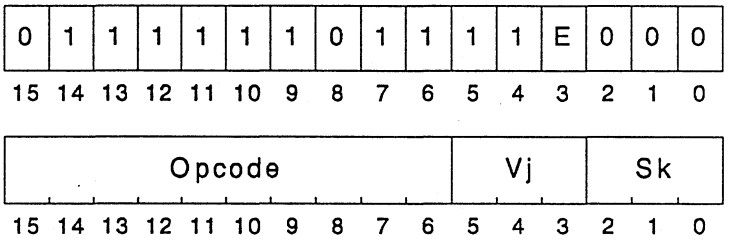
1. An *ldea* instruction typically loads A5 before execution of this instruction.
2. If the distance between successive elements is less than the precision of an element, unpredictable actions occur.
3. The *.s* and *.d* forms rename the *.w* and *.l* forms, respectively, for convenience.

stvi.(b|h|w|l|s|d).(t|f) Sk,Vj — STORE SCALAR/VECTOR INDEX MASKED

Purpose: To store a vector from a scalar register using the contents of a vector register as set of indices, under control of the Vector Merge (VM) register

Architecture: **C200 Series only**

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
      case TRUE:   /*  .t */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 1) {        /* if VM<a> is TRUE */
                  c(Vj[a]<31..0> + A5) = Sk;
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
      case FALSE:  /*  .f */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 0) {        /* if VM<a> is FALSE */
                  c(Vj[a]<31..0> + A5) = Sk;
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
stvi.b.t Sk,Vj	7B00	E1 0111101100	None	Scalar index store vector byte (VM)
stvi.b.f Sk,Vj	7B00	E0 0111101100	None	Scalar index store vector byte (!VM)
stvi.h.t Sk,Vj	7B40	E1 0111101101	None	Scalar index store vector half (VM)
stvi.h.f Sk,Vj	7B40	E0 0111101101	None	Scalar index store vector half (!VM)
stvi.w.t Sk,Vj	7B80	E1 0111101110	None	Scalar index store vector word (VM)
stvi.w.f Sk,Vj	7B80	E0 0111101110	None	Scalar index store vector word (!VM)
stvi.l.t Sk,Vj	7BC0	E1 0111101111	None	Scalar index store vector long (VM)
stvi.l.f Sk,Vj	7BC0	E0 0111101111	None	Scalar index store vector long (!VM)
stvi.s.t Sk,Vj	7B80	E1 0111101110	None	Scalar index store vector single (VM)
stvi.s.f Sk,Vj	7B80	E0 0111101110	None	Scalar index store vector single (!VM)
stvi.d.t Sk,Vj	7BC0	E1 0111101111	None	Scalar index store vector double (VM)
stvi.d.f Sk,Vj	7BC0	E0 0111101111	None	Scalar index store vector double (!VM)

Description: The bottom 32 bits of each of the first VL elements of vector register Vj specify a set of address offsets. These address offsets are each summed with the contents of address register A5 to yield a set of addresses that specify the destinations of operands whose contents are replaced by the contents of scalar register Sk, if the corresponding VM bit is set (clear for *.f*).

Notes:

1. An *ldea* instruction typically loads A5 before execution of this instruction.
2. If the distance between successive elements is less than the precision of an element, unpredictable actions occur.

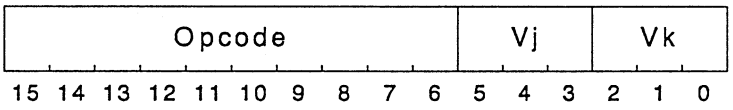
3. The *.s.(t|f)* and *.d.(t|f)* forms rename the *.w.(t|f)* and *.l.(t|f)* forms for convenience.

stvi.(b|h|w|l|s|d) Vk,Vj — STORE VECTOR/VECTOR INDEX

Purpose: To store the elements of a vector register using the contents of a vector register as set of indices ("vector scatter")

Architecture: C100 Series, C200 Series

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      c(Vj[a]<31..0> + A5) = Vk[a];
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
stvi.b Vk,Vj	7A00	ST 0111101000	None	Index store vector byte
stvi.h Vk,Vj	7A40	ST 0111101001	None	Index store vector halfword
stvi.w Vk,Vj	7A80	ST 0111101010	None	Index store vector word
stvi.l Vk,Vj	7AC0	ST 0111101011	None	Index store vector longword
stvi.s Vk,Vj	7A80	ST 0111101010	None	Index store vector single
stvi.d Vk,Vj	7AC0	ST 0111101011	None	Index store vector double

Description: The bottom 32 bits of each of the first VL elements of vector register Vj specify a set of address offsets. Each address offset is summed with the contents of address register A5 to yield a set of addresses that specify the destinations of operands whose contents are replaced by successive elements of Vk.

Notes:

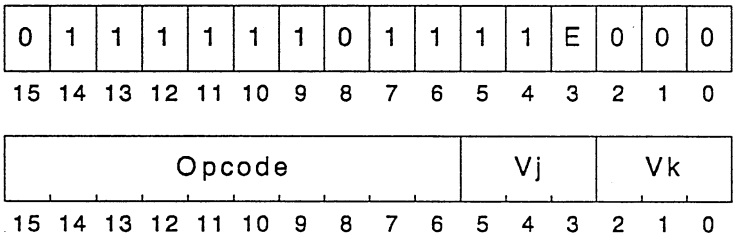
1. An *ldea* instruction typically loads A5 before execution of this instruction.
2. If the distance between successive elements is less than the precision of an element, unpredictable actions occur.
3. The *.s* and *.d* forms rename the *.w* and *.l* forms, respectively, for convenience.

STORE VECTOR/VECTOR INDEX MASKED — stvi.(b|h|w|l|s|d).(t|f) Vk,Vj

Purpose: To store the elements of a vector register using the contents of a vector register as set of indices, under control of the Vector Merge (VM) register ("vector scatter")

Architecture: C200 Series only

Format:



Operation:

```
switch (E) { /* prefix bit<3> */
     case TRUE:  /* .t */
          for (a = 0; a < VL; a++) {
               if (VM<a> == 1) {        /* if VM<a> is TRUE */
                    c(Vj[a]<31..0> + A5) = Vk[a];
               }
          }   /* end of for loop */
          break;  /* go to end of switch */
     case FALSE: /* .f */
          for (a = 0; a < VL; a++) {
               if (VM<a> == 0) {        /* if VM<a> is FALSE */
                    c(Vj[a]<31..0> + A5) = Vk[a];
               }
          }   /* end of for loop */
          break;  /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
stvi.b.t Vk,Vj	7A00	E1 0111101000	None	Index store vector byte (VM)
stvi.b.f Vk,Vj	7A00	E0 0111101000	None	Index store vector byte (!VM)
stvi.h.t Vk,Vj	7A40	E1 0111101001	None	Index store vector halfword (VM)
stvi.h.f Vk,Vj	7A40	E0 0111101001	None	Index store vector halfword (!VM)
stvi.w.t Vk,Vj	7A80	E1 0111101010	None	Index store vector word (VM)
stvi.w.f Vk,Vj	7A80	E0 0111101010	None	Index store vector word (!VM)
stvi.l.t Vk,Vj	7AC0	E1 0111101011	None	Index store vector longword (VM)
stvi.l.f Vk,Vj	7AC0	E0 0111101011	None	Index store vector longword (!VM)
stvi.s.t Vk,Vj	7A80	E1 0111101010	None	Index store vector single (VM)
stvi.s.f Vk,Vj	7A80	E0 0111101010	None	Index store vector single (!VM)
stvi.d.t Vk,Vj	7AC0	E1 0111101011	None	Index store vector double (VM)
stvi.d.f Vk,Vj	7AC0	E0 0111101011	None	Index store vector double (!VM)

Description: The bottom 32 bits of each of the first VL elements of vector register Vj specify a set of address offsets. These address offsets are each summed with the contents of address register A5 to yield a set of addresses that specify the destinations of operands whose contents are replaced by successive elements of Vk, if the corresponding VM bit is set (clear for *.f*).

Notes:

1. An *ldea* instruction typically loads A5 before execution of this instruction.
2. If the distance between successive elements is less than the precision of an element, unpredictable actions occur.

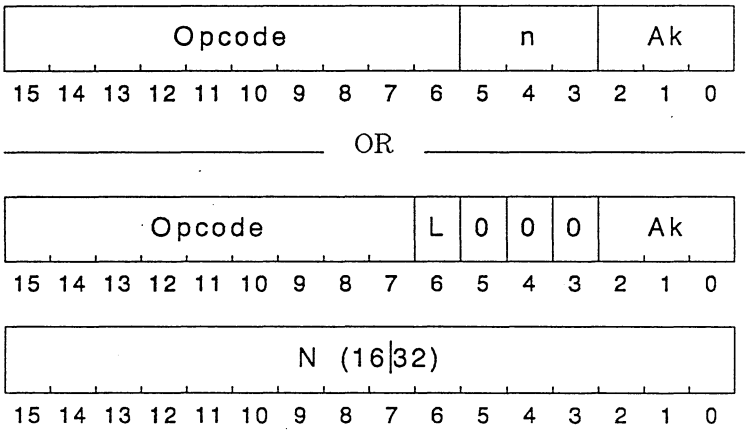
3. The *.s.(t|f)* and *.d.(t|f)* forms rename the *.w.(t|f)* and *.l.(t|f)* forms for convenience.

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

Purpose: To subtract an immediate from the contents of an address register

Architecture: C100 Series, C200 Series

Format:



Operation:

```
Ak = Ak - Immediate;
```

Exceptions: (h|w): Integer Overflow

Opcode:

Mnemonic	Hex	Binary	PSW	Description
sub.h #n,Ak	5A80	ST 0101101010	C,AIV	Subtract short immediate address halfword
sub.w #n,Ak	5AC0	ST 0101101011	C,AIV	Subtract short immediate address word
sub.h #N,Ak	1500	ST 000101010	C,AIV	Subtract immediate address halfword
sub.w #N,Ak	1580	ST 000101011	C,AIV	Subtract immediate address word

Description: The contents of address register Ak minus the (sign-extended) immediate field replaces the contents of Ak.

Notes: Sign extension does not occur for the three bits of the short immediate form.

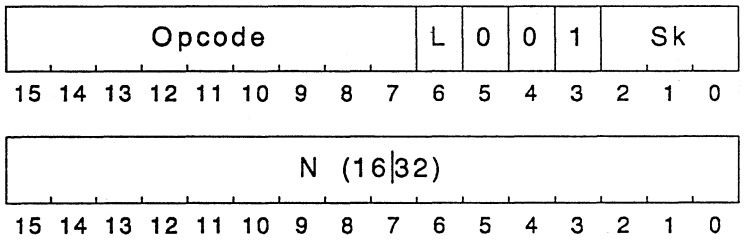
sub.(h|w|s) #N,Sk

SUBTRACT SCALAR/IMMEDIATE

Purpose: To subtract an immediate from the contents of a scalar register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Sk = Sk - Immediate;
```

Exceptions:

(h|w): Integer Overflow

(s): Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
sub.h #N,Sk	1508	ST 000101010	SC,SIV	Subtract scalar/immediate integer halfword
sub.w #N,Sk	1588	ST 000101011	SC,SIV	Subtract scalar/immediate integer word
sub.s #N,Sk	1888	ST 000110001	OV,UN,RO	Subtract scalar/immediate single float

Description: The contents of scalar register Sk minus the (sign-extended) immediate field replaces the contents of Sk.

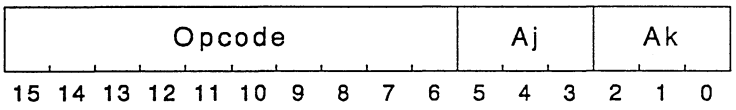
Notes: The carry out of the subtraction is returned in the PSW<SC> bit.

SUBTRACT ADDRESS/ADDRESS — sub.(h|w) Aj,Ak

Purpose: To subtract the contents of one address register from the contents of another address register

Architecture: C100 Series, C200 Series

Format:



Operation:

```
Ak = Ak - Aj;
```

Exceptions: (h|w): Integer Overflow

Opcode:

Mnemonic	Hex	Binary	PSW	Description
sub.h Aj,Ak	5A00	ST 0101101000	C,AIV	Subtract address register halfword
sub.w Aj,Ak	5A40	ST 0101101001	C,AIV	Subtract address register word

Description: The contents of address register Ak minus the contents of Aj replaces the contents of Ak.

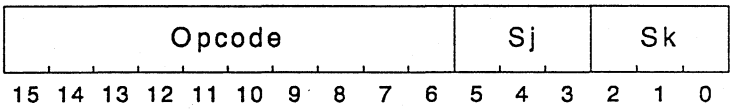
Notes: None

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

Purpose: To subtract the contents of one scalar register from the contents of another scalar register

Architecture: **C100 Series, C200 Series**

Format:



Operation: `Sk = Sk - Sj;`

Exceptions:

(b|h|w|l): Integer Overflow

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
sub.b Sj,Sk	5B00	ST 0101101100	SC,SIV	Subtract scalar/scalar integer byte
sub.h Sj,Sk	5B40	ST 0101101101	SC,SIV	Subtract scalar/scalar integer halfword
sub.w Sj,Sk	5B80	ST 0101101110	SC,SIV	Subtract scalar/scalar integer word
sub.l Sj,Sk	5BC0	ST 0101101111	SC,SIV	Subtract scalar/scalar integer longword
sub.s Sj,Sk	5580	ST 0101010110	OV,UN,RO	Subtract scalar/scalar single float
sub.d Sj,Sk	55C0	ST 0101010111	OV,UN,RO	Subtract scalar/scalar double float

Description: The contents of scalar register Sk minus the contents of Sj replaces the contents of Sk.

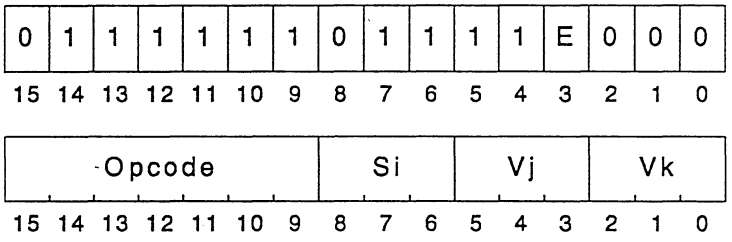
Notes: None

REVERSE SUBTRACT SCALAR/VECTOR — sub.(s|d) Si,Vj,Vk

Purpose: To subtract each element of a vector register from the contents of a scalar register

Architecture: C200 Series only

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      Vk[a] = Si - Vj[a];
}
```

Exceptions: (s|d): Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
sub.s Si,Vj,Vk	8000	E0 1000000	OV,UN,RO	Subtract scalar/vector single float
sub.d Si,Vj,Vk	8200	E0 1000001	OV,UN,RO	Subtract scalar/vector double float

Description: The contents of each of the first VL elements of vector register Vk is replaced by the evaluation of the contents of scalar register Si minus the contents of the corresponding element of Vj.

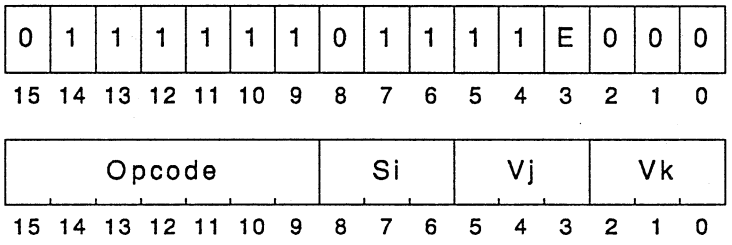
Notes: None

sub.(s|d).(t|f) Si,Vj,Vk REVERSE SUBTRACT SCALAR/VECTOR MASKED

Purpose: To subtract each element of a vector register from the contents of a scalar register under control of the Vector Merge (VM) register

Architecture: C200 Series only

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
      case TRUE:   /* .t */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 1) {        /* if VM<a> is TRUE */
                  Vk[a] = Si - Vj[a];
              }
          }   /* end of for loop */
          break;   /* go to end of switch */
      case FALSE:  /* .f */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 0) {        /* if VM<a> is FALSE */
                  Vk[a] = Si - Vj[a];
              }
          }   /* end of for loop */
          break;   /* go to end of switch */
} /* end of switch */
```

Exceptions: (s|d): Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
sub.s.t Si,Vj,Vk	8800	E1 1000100	OV,UN,RO	Subtract scalar/vector single (VM)
sub.s.f Si,Vj,Vk	8800	E0 1000100	OV,UN,RO	Subtract scalar/vector single (!VM)
sub.d.t Si,Vj,Vk	8A00	E1 1000101	OV,UN,RO	Subtract scalar/vector double (VM)
sub.d.f Si,Vj,Vk	8A00	E0 1000101	OV,UN,RO	Subtract scalar/vector double (!VM)

Description: The contents of each of the first VL elements of vector register Vk is replaced by the evaluation of the contents of scalar register Si minus the contents of the corresponding element of Vj if the corresponding VM bit is set (clear for *.f*).

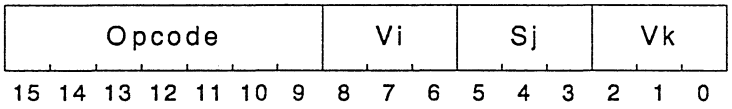
Notes: None

SUBTRACT VECTOR/SCALAR — sub.(b|h|w|l|s|d) Vi,Sj,Vk

Purpose: To subtract the contents of a scalar register from the elements of a vector register

Architecture: C100 Series, C200 Series

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      Vk[a] = Si - Vj[a];
}
```

Exceptions:

(b|h|w|l): Integer Overflow

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
sub.b Vi,Sj,Vk	D800	ST 1101100	SIV	Subtract vector/scalar integer byte
sub.h Vi,Sj,Vk	DA00	ST 1101101	SIV	Subtract vector/scalar integer halfword
sub.w Vi,Sj,Vk	DC00	ST 1101110	SIV	Subtract vector/scalar integer word
sub.l Vi,Sj,Vk	DE00	ST 1101111	SIV	Subtract vector/scalar integer longword
sub.s Vi,Sj,Vk	BC00	ST 1011110	OV,UN,RO	Subtract vector/scalar single float
sub.d Vi,Sj,Vk	BE00	ST 1011111	OV,UN,RO	Subtract vector/scalar double float

Description: The evaluation of the contents of the corresponding element of Vi minus the contents of scalar register Sj replaces the contents of each of the first VL elements of vector register Vk.

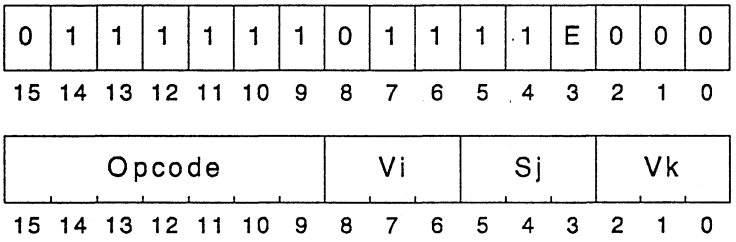
Notes: None

sub.(b|h|w|l|s|d).(t|f) Vi,Sj,Vk — SUBTRACT VECTOR/SCALAR MASKED

Purpose: To subtract the contents of a scalar register from the elements of a vector register under control of the Vector Merge (VM) register

Architecture: C200 Series only

Format:



Operation:

```
switch (E) { /* prefix bit<3> */
      case TRUE:  /* .t */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 1) {        /* if VM<a> is TRUE */
                  Vk[a] = Vi[a] - Sj;
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
      case FALSE: /* .f */
          for (a = 0; a < VL; a++) {
              if (VM<a> == 0) {        /* if VM<a> is FALSE */
                  Vk[a] = Vi[a] - Sj;
              }
          }   /* end of for loop */
          break;  /* go to end of switch */
} /* end of switch */
```

Exceptions:

(b|h|w|l): Integer Overflow

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
sub.b.t Vi,Sj,Vk	D800	E1 1101100	SIV	Subtract vector/scalar byte (VM)
sub.b.f Vi,Sj,Vk	D800	E0 1101100	SIV	Subtract vector/scalar byte (!VM)
sub.h.t Vi,Sj,Vk	DA00	E1 1101101	SIV	Subtract vector/scalar halfword (VM)
sub.h.f Vi,Sj,Vk	DA00	E0 1101101	SIV	Subtract vector/scalar halfword (!VM)
sub.w.t Vi,Sj,Vk	DC00	E1 1101110	SIV	Subtract vector/scalar word (VM)
sub.w.f Vi,Sj,Vk	DC00	E0 1101110	SIV	Subtract vector/scalar word (!VM)
sub.l.t Vi,Sj,Vk	DE00	E1 1101111	SIV	Subtract vector/scalar longword (VM)
sub.l.f Vi,Sj,Vk	DE00	E0 1101111	SIV	Subtract vector/scalar longword (!VM)
sub.s.t Vi,Sj,Vk	BC00	E1 1011110	OV,UN,RO	Subtract vector/scalar single (VM)
sub.s.f Vi,Sj,Vk	BC00	E0 1011110	OV,UN,RO	Subtract vector/scalar single (!VM)
sub.d.t Vi,Sj,Vk	BE00	E1 1011111	OV,UN,RO	Subtract vector/scalar double (VM)
sub.d.f Vi,Sj,Vk	BE00	E0 1011111	OV,UN,RO	Subtract vector/scalar double (!VM)

Description: The contents of each of the first VL elements of vector register Vk is replaced by the evaluation of the contents of corresponding element of Vi minus the contents of scalar register Sj if the corresponding VM bit is set (clear for *.f*).

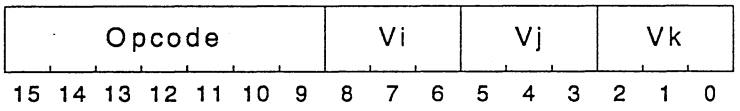
Notes: None

SUBTRACT VECTOR/VECTOR — sub.(b|h|w|l|s|d) Vi,Vj,Vk

Purpose: To subtract the elements of two vector registers

Architecture: C100 Series, C200 Series

Format:



Operation:

```
for (a = 0; a < VL; a++) {
     Vk[a] = Vi[a] - Vj[a];
}
```

Exceptions:

(b|h|w|l): Integer Overflow

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
sub.b Vi,Vj,Vk	D000	ST 1101000	SIV	Subtract vector/vector integer byte
sub.h Vi,Vj,Vk	D200	ST 1101001	SIV	Subtract vector/vector integer halfword
sub.w Vi,Vj,Vk	D400	ST 1101010	SIV	Subtract vector/vector integer word
sub.l Vi,Vj,Vk	D600	ST 1101011	SIV	Subtract vector/vector integer longword
sub.s Vi,Vj,Vk	B400	ST 1011010	OV,UN,RO	Subtract vector/vector single float
sub.d Vi,Vj,Vk	B600	ST 1011011	OV,UN,RO	Subtract vector/vector double float

Description: The evaluation of the contents of the corresponding element of Vi minus the contents of the corresponding element of Vj replaces the contents of each of the first VL elements of vector register Vk.

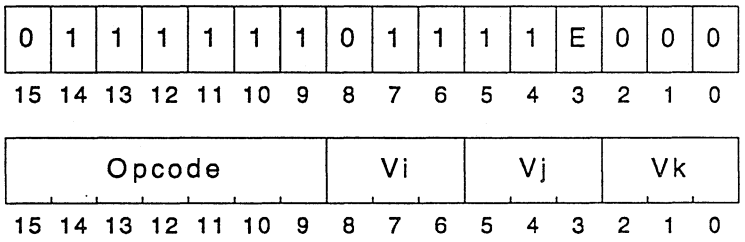
Notes: None

sub.(b|h|w|l|s|d).(t|f) Vi,Vj,Vk — SUBTRACT VECTOR/VECTOR MASKED

Purpose: To subtract the elements of two vector registers under control of the Vector Merge (VM) register

Architecture: **C200 Series only**

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
     case TRUE:   /* .t */
          for (a = 0; a < VL; a++) {
               if (VM<a> == 1) {        /* if VM<a> is TRUE */
                    Vk[a] = Vi[a] - Vj[a];
               }
          }   /* end of for loop */
          break;   /* go to end of switch */
     case FALSE:  /* .f */
          for (a = 0; a < VL; a++) {
               if (VM<a> == 0) {        /* if VM<a> is FALSE */
                    Vk[a] = Vi[a] - Vj[a];
               }
          }   /* end of for loop */
          break;   /* go to end of switch */
} /* end of switch */
```

Exceptions:

(b|h|w|l): Integer Overflow

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
sub.b.t Vi,Vj,Vk	D000	E1 1101000	SIV	Subtract byte vectors (VM)
sub.b.f Vi,Vj,Vk	D000	E0 1101000	SIV	Subtract byte vectors (!VM)
sub.h.t Vi,Vj,Vk	D200	E1 1101001	SIV	Subtract halfword vectors (VM)
sub.h.f Vi,Vj,Vk	D200	E0 1101001	SIV	Subtract halfword vectors (!VM)
sub.w.t Vi,Vj,Vk	D400	E1 1101010	SIV	Subtract word vectors (VM)
sub.w.f Vi,Vj,Vk	D400	E0 1101010	SIV	Subtract word vectors (!VM)
sub.l.t Vi,Vj,Vk	D600	E1 1101011	SIV	Subtract longword vectors (VM)
sub.l.f Vi,Vj,Vk	D600	E0 1101011	SIV	Subtract longword vectors (!VM)
sub.s.t Vi,Vj,Vk	B400	E1 1011010	OV,UN,RO	Subtract single vectors (VM)
sub.s.f Vi,Vj,Vk	B400	E0 1011010	OV,UN,RO	Subtract single vectors (!VM)
sub.d.t Vi,Vj,Vk	B600	E1 1011011	OV,UN,RO	Subtract double vectors (VM)
sub.d.f Vi,Vj,Vk	B600	E0 1011011	OV,UN,RO	Subtract double vectors (!VM)

Description: The contents of each of the first VL elements of vector register Vk are replaced by the evaluation of the contents of corresponding element of Vi minus the contents of the corresponding element of Vj if the corresponding VM bit is set (clear for *.f*).

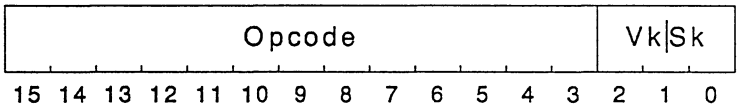
Notes: None

SUM VECTOR — sum.(b|h|w|l|s|d) (Vk|Sk)

Purpose: To sum all the elements of a vector register

Architecture: C100 Series, C200 Series

Format:



Operation:

```
for (a = 0; a < VL; a++) {
      Sk = Sk + Vk[a]; /* see following notes */
}
```

Exceptions:

(b|h|w|l): Integer Overflow

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
sum.b Vk	7E00	ST 0111111000000	SIV	Sum a vector of bytes
sum.h Vk	7E08	ST 0111111000001	SIV	Sum a vector of halfwords
sum.w Vk	7E10	ST 0111111000010	SIV	Sum a vector of words
sum.l Vk	7E18	ST 0111111000011	SIV	Sum a vector of longwords
sum.s Vk	7E80	ST 0111111010000	OV,UN,RO	Sum a vector of single float
sum.d Vk	7E88	ST 0111111010001	OV,UN,RO	Sum a vector of double float

Description: The sum of the contents of scalar register Sk and the contents of the first VL elements of vector register Vk replace Sk.

Notes:

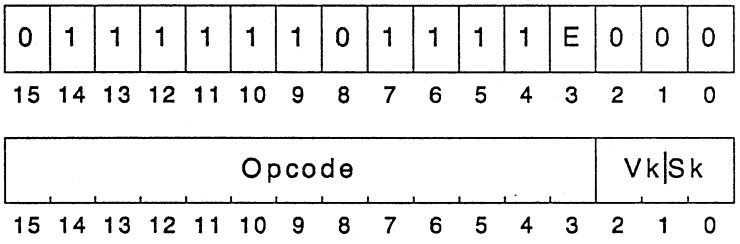
1. Initialize the scalar register properly for the first use of the *sum* reduce instruction (probably to 0).
2. The sequence of the sum executed by the hardware is *not* identical to the preceding operation psuedocode sequence, i.e., the execution sequence is implementation-specfic. For more information, refer to the discussion of vector operations in the *CONVEX Architecture Reference*, "Instruction Set" chapter.
3. Either Vk or Sk may be used as a valid argument to this instruction. This instruction operates in distinct matched vector and scalar register pairs: (V0,S0), (V1,S1), (V2,S2), (V3,S3), (V4,S4), (V5,S5), (V6,S6), (V7,S7).

sum.(b|h|w|l|s|d).(t|f) (Vk|Sk) — SUM VECTOR MASKED

Purpose: To sum a subset of the elements of a vector under control of the Vector Merge (VM) register

Architecture: **C200 Series only**

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
     case TRUE:   /*  .t */
         for (a = 0; a < VL; a++) {
             if (VM<a> == 1) {        /* if VM<a> is TRUE */
                 Sk = Sk + Vk[a];
             }
         }   /* end of for loop */
         break;  /* go to end of switch */
     case FALSE:  /*  .f */
         for (a = 0; a < VL; a++) {
             if (VM<a> == 0) {        /* if VM<a> is FALSE */
                 Sk = Sk + Vk[a];
             }
         }   /* end of for loop */
         break;  /* go to end of switch */
} /* end of switch */
```

Exceptions:

(b|h|w|l): Integer Overflow

(s|d): Exponent Overflow
Exponent Underflow
Reserved Operand

Opcode:

Mnemonic	Hex	Binary	PSW	Description
sum.b.t Vk	7E00	E1 0111111000000	SIV	Sum a vector of bytes (VM)
sum.b.f Vk	7E00	E0 0111111000000	SIV	Sum a vector of bytes (!VM)
sum.h.t Vk	7E08	E1 0111111000001	SIV	Sum a vector of halfwords (VM)
sum.h.f Vk	7E08	E0 0111111000001	SIV	Sum a vector of halfwords (!VM)
sum.w.t Vk	7E10	E1 0111111000010	SIV	Sum a vector of words (VM)
sum.w.f Vk	7E10	E0 0111111000010	SIV	Sum a vector of words (!VM)
sum.l.t Vk	7E18	E1 0111111000011	SIV	Sum a vector of longwords (VM)
sum.l.f Vk	7E18	E0 0111111000011	SIV	Sum a vector of longwords (!VM)
sum.s.t Vk	7E80	E1 0111111010000	OV,UN,RO	Sum a vector of single (VM)
sum.s.f Vk	7E80	E0 0111111010000	OV,UN,RO	Sum a vector of single (!VM)
sum.d.t Vk	7E88	E1 0111111010001	OV,UN,RO	Sum a vector of double (VM)
sum.d.f Vk	7E88	E0 0111111010001	OV,UN,RO	Sum a vector of double (!VM)

Description: The sum of the contents of scalar register Sk and the contents of the first VL elements of vector register Vk replace Sk. Only elements of Vk with corresponding VM bit set (clear for *.f*) participate in the sum.

Notes:

1. Initialize the scalar register properly for the first use of the *sum* reduce instruction (probably to 0).

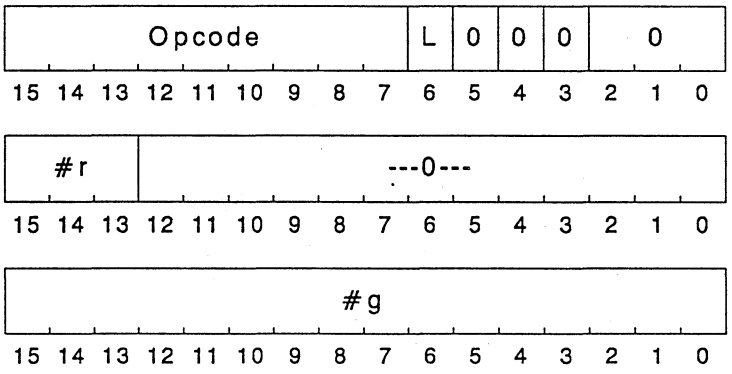
2. The sequence of the sum executed by the hardware is *not* identical to the preceding operation psuedocode sequence, i.e., the execution sequence is implementation-specfic. For more information, refer to the discussion of vector operations in the *CONVEX Architecture Reference*, "Instruction Set" chapter.
3. Either Vk or Sk may be used as a valid argument to this instruction. This instruction operates in distinct matched vector and scalar register pairs: (V0,S0), (V1,S1), (V2,S2), (V3,S3), (V4,S4), (V5,S5), (V6,S6), (V7,S7).

sysc #r,#g — SYSTEM CALL

Purpose: To perform a system call to ring specified in #r

Architecture: C100 Series, C200 Series

Format:



Operation:

```
psw[FRL] = 01;          /* extended frame */
if (Architecture == C200) {
     push(thread_timer);
}
push(S0); push(S1); push(S2); push(S3);
push(S4); push(S5); push(S6); push(S7);
push(A0); push(A1); push(A2); push(A3);
push(A4); push(A5); push(A6); push(A7);
push(PSW);
push(next_instruction_address);
psw[FRL] = 0; psw[C]  = 0; psw[SC] =  0; psw[AIV] = 0; psw[ADZ] = 0;
psw[UN]  = 0; psw[OV] = 0; psw[FDZ] = 0; psw[RO]  = 0; psw[SIV] = 0;
psw[SDZ] = 0; psw[FIN] = 0;
(Execute a call to ring #r, GATE(#g) );
(Get new stack pointer);
if (Architecture == C200) {
     SP = SP - 112;   /* Extended return block */
} else {
     SP = SP - 104;   /* Extended return block */
}
FP = SP;
PC<31..29> = #r;
PC<28..1> = GATE_ARRAY(#g);
```

Exceptions: Ring Violation (outward call)
Ring Violation (invalid gate)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
sysc #r,#g	1080	ST 000100001	All bits	Perform a system call

Description: An inward or current ring crossing with protection checks is performed. Temp = PC + instruction length, where PC is the address of this 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.

C100 Series The SP contained in page 0 of the inner ring is used to define the base of the inner ring stack.

C200 Series The stack pointer contained in the extended return block is popped off the system resource structure.

The *# g* field references an entry in a gate and is used to obtain the address of the called instruction for both intra- or inward-ring crossing. The *# r* field is the number of the called ring. The base of the gate is referenced by bytes <76..79> of page 0 of the ring named by #r.

After the new ring stack is established, an extended return block is pushed. All A and S (including S0 and A0) registers are saved, as well as the PSW and the address of the instruction following this call instruction.

Notes:

1. All PSW bits are cleared to zero after the extended frame is pushed.
2. Refer to the *CONVEX Architecture Reference*, "Memory Management" 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.

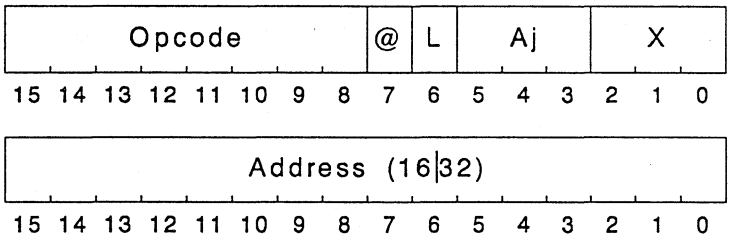
tac <effa>

TEST AND CLEAR BYTE

Purpose: To indivisibly test and clear a byte in memory

Architecture: **C200 Series only**

Format:



Operation:

```
msync;
if ( c(Effective_Address) == 0xFF ) {
     C = 1;
} else {
     C = 0;
}
c(Effective_Address) = 0;
```

Exceptions: Deadlock Exception

Opcode:

Mnemonic	Hex	Binary	PSW	Description
tac <effa>	0800	ST 00001000000	C	Test and clear a byte in memory

Description: This instruction is the "signal" corresponding to the test and set instruction (*tas*). It performs an implicit *msync*, i.e., waits for all previous stores to reach memory before proceeding. Next, it clears the "lock" byte, returning PSW<C> = 1 if the lock byte was set.

Notes:

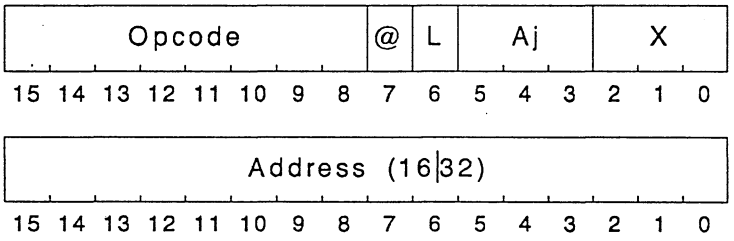
1. Address Carry (C) of the PSW is affected as described by the preceding operation pseudocode.
2. The test and clear byte is used to clear a byte in memory indivisibly. No I/O operation is permitted between the read and write of the referenced byte.
3. The X field is unused.

TEST AND SET BYTE — tas <effa>

Purpose: To indivisibly test and set a byte in memory

Architecture: C100 Series, C200 Series

Format:



Operation:

```
msync;
if ( c(Effective_Address) == 0 ) {
     C = 1;
} else {
     C = 0;
}
c(Effective_Address) = 0xFF;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
tas <effa>	0C00	ST 000011000	C	Test and set a memory byte

Description: This instruction is the "signal" corresponding to the test and clear byte instruction (*tac*). It performs an implicit *msync*, i.e., waits for all previous stores to reach memory before proceeding. It sets the "lock" byte, returning PSW<C> = 1 if the lock byte was clear.

Notes:

1. Address Carry (C) of the PSW is affected as described by the preceding operation pseudocode.
2. The test and set byte is used to test a byte in memory indivisibly, i.e., this instruction is atomic. No I/O operation is permitted between the read and write of the referenced byte.
3. The X field is ignored.

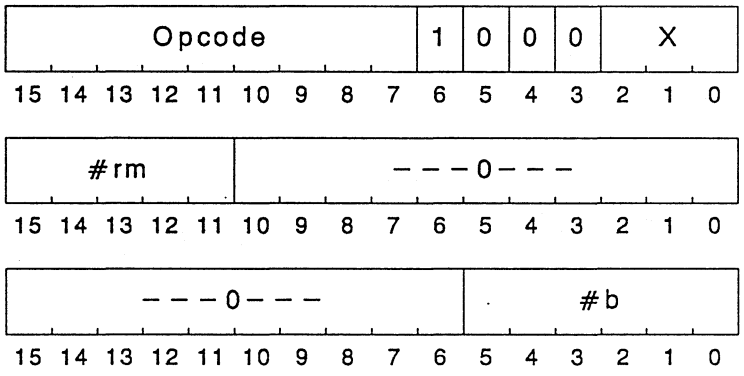
trap #rm,#b

TRAP PROCESS

Purpose: To selectively force all threads sharing the same communication register set to enter the exception handler

Architecture: **C200 Series only**

Format:



Operation:

```
if (crossing_rings) {
     (Allocate a ring 0 stack from the system resource structure);
}
psw[FRL] = 01;          /* extended frame */
push(thread_timer);
push(S0); push(S1); push(S2); push(S3);
push(S4); push(S5); push(S6); push(S7);
push(A0); push(A1); push(A2); push(A3);
push(A4); push(A5); push(A6); push(A7);
push(PSW);
push(next_instruction_address);
psw = 0;
SP = SP - 112;          /* extended return block */
FP = SP;
A5 = 0x00001400;
S0 = (Trap Instruction Register that trapped);
(Enter the ring 0 system exception handler);
```

Exceptions: Invalid Trap Instruction

Opcode:

Mnemonic	Hex	Binary	PSW	Description
trap #rm,#b	1A00	ST 0001101000	All bits	Force a trap system exception

Description: Validate ring *#rm* and bit number *#b*. The most significant bit of *#rm* specifies Ring 4, down to the least significant bit, which specifies Ring 0. If they are valid, set bit *#b* (a number from 0–63) in trap instruction registers specified by the ring mask *#rm*. If *#rm* specifies the current ring, then take a trap on this CPU in the manner described in the preceding psuedocode.

A Ring 0 system exception occurs for all threads within the specified rings that share the same communication register set. The specified bit is ORed with the specified ring's trap instruction register in the hardware communication registers. If the ring mask specifies the current ring, the Ring 0 exception handler is executed with a code of 14 (hex) and 0 qualifier.

Notes:

1. The PSW is cleared to all zero.
2. The trap condition remains outstanding until all bits in the hardware communication trap instruction register are cleared. If a thread attempts to enter a ring that has any bits set in the ring's trap instruction register, it will immediately enter the Ring 0 exception handler.

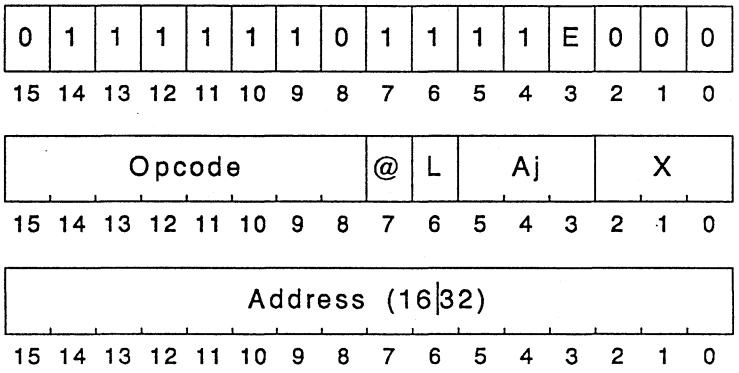
3. Refer to the *CONVEX Architecture Reference*, "Instruction Set" chapter, for a detailed explanation of the *trap* instruction.
4. The X field is ignored.

tst <Ceffa> TEST COMMUNICATION REGISTER LOCK

Purpose: To test the value of the lock bit associated with the communication register

Architecture: **C200 Series only**

Format:



Operation: `C = L(Ceffa);`

Exceptions: Ring Violation (Invalid Communication Register Address)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
tst <Ceffa>	0100	E0 0000000100	C,CAT	Test communication register lock bit

Description: C is loaded with the current value of the lock bit for the addressed communication register. Neither L(Ceffa) or C<Ceffa> are modified.

Notes:

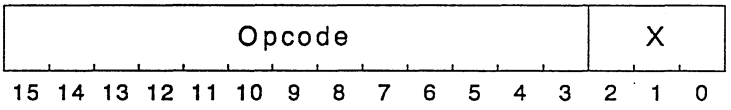
1. Address Carry (C) of the PSW is affected by the preceding operation pseudocode
2. The X field is unused.

TEST VECTOR VALID — tstvv

Purpose: To test the value of the Vector Valid (VV) flag

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
if (VV == 1) {
     SC = 1;
} else {
     SC = 0;
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
tstvv	7D78	ST 0111110101111	SC	Test value of vector valid flag

Description: The value of the VV flag replaces the SC bit.

Notes:

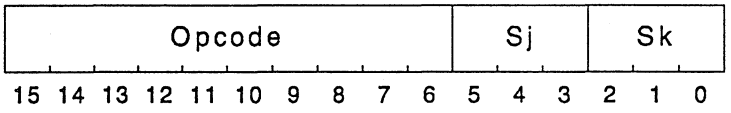
1. Scalar Carry (SC) of the PSW is affected as described by the preceding operation pseudocode.
2. The X field is ignored.
3. This instruction is not privileged.

tzc Sj,Sk — TRAILING ZERO COUNT SCALAR

Purpose: To determine the number of trailing zero bits in a scalar register

Architecture: C100 Series, C200 Series

Format:



Operation:

```
for (a = 0; a <= 63; a++) {
     if (Sj<a> != 0) {
         break;  /* found rightmost 1, so break loop */
     }
}
Sk = a;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
tzc Sj,Sk	45C0	ST 0100010111	None	Count of trailing zeros in Sj

Description: The number of trailing zero bits in scalar register Sj replaces Sk. If Sj contains all zeros, Sk becomes 64.

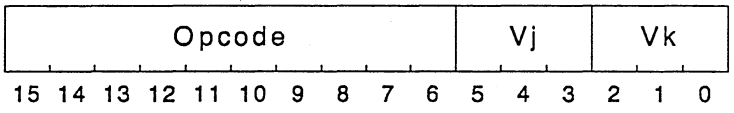
Notes: None

TRAILING ZERO COUNT VECTOR — tzc Vj,Vk

Purpose: To count the number of trailing zero bits in each element of a vector register

Architecture: C200 Series only

Format:



Operation:

```
for (a = 0; a < VL; a++) {
     for (b = 0; b <= 63; b++) {
          if (Vj[a]<b>e != 0) {
               break;  /* found rightmost 1, so break loop */
          }
     }
     Vk[a] = b;
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
tzc Vj,Vk	6200	ST 0110001000	None	Trailing zero count vector

Description: Each of the first VL elements of Vk is replaced by the number of trailing zeros contained in the 64 bits of the corresponding element of Vj. If an element of Vj contains all zeros, the corresponding element of Vk becomes 64.

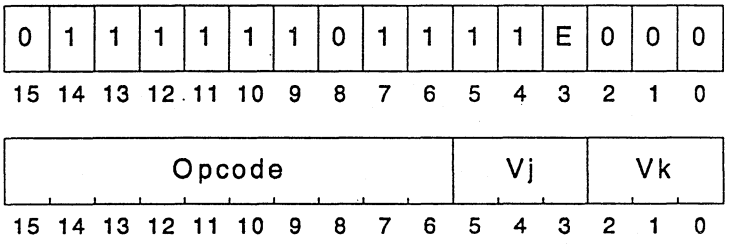
Notes: None

tzc.(t|f) Vj,Vk — TRAILING ZERO COUNT VECTOR MASKED

Purpose: To count the number of trailing zero bits in each element of a vector register under control of the Vector Merge (VM) register

Architecture: **C200 Series only**

Format:



Operation:

```
switch (E) {  /* prefix bit<3> */
     case TRUE:   /* .t */
          for (a = 0; a < VL; a++) {
               if (VM<a> == 1) {        /* if VM<a> is TRUE */
                    for (b = 0; b <= 63; b++) {
                         if (Vj[a]<b> != 0) {
                              break;  /* found rightmost 1,
                                   /* so break loop */
                         }
                    }
                    Vk[a] = b;
               }   /* end if TRUE */
          }   /* end of for loop */
          break;  /* go to end of switch */
     case FALSE:  /* .f */
          for (a = 0; a < VL; a++) {
               if (VM<a> == 0) {        /* if VM<a> is FALSE */
                    for (b = 0; b <= 63; b++) {
                         if (Vj[a]<b> != 0) {
                              break;  /* found rightmost 1,
                                   /* so break loop */
                         }
                    }
                    Vk[a] = temp;
               }   /* end if FALSE */
          }   /* end of for loop */
          break;  /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
tzc.t Vj,Vk	6200	E1 0110001000	None	Trailing zero count vector (VM)
tzc.f Vj,Vk	6200	E0 0110001000	None	Trailing zero count vector (!VM)

Description: Each of the first VL elements of Vk is replaced by the number of trailing zeros contained in the 64 bits of the corresponding element of Vj if the corresponding VM bit is set (clear for *.f*).

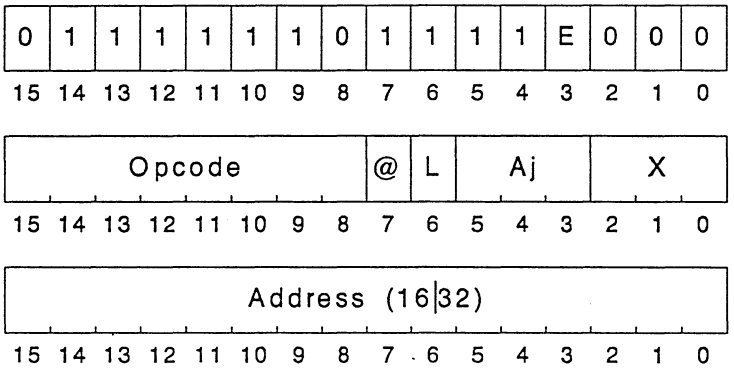
Notes: None

UNLOCK COMMUNICATION REGISTER — ulk <Ceffa>

Purpose: To unlock a communication register

Architecture: C200 Series only

Format:



Operation:

```
if (L(Ceffa) == 1) {
     L(Ceffa) = 0;
     C = 1;
} else {
     C = 0;
}
```

Exceptions: Ring Violation (Invalid Communication Register Address)
Deadlock Exception

Opcode:

Mnemonic	Hex	Binary	PSW	Description
ulk <Ceffa>	0300	E0 0000001100	C,CAT	Unlock communication register

Description: If the lock bit for the addressed communication register is cleared, the communication registers are not modified and "fail" status (C = 0) is returned. If the lock bit for the addressed communication register is set, the lock bit is cleared, and "success" status (C = 1) is returned.

Notes:

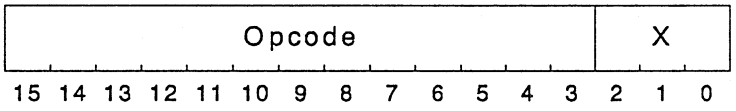
1. Address Carry (C) of the PSW is affected as described by the preceding operation psuedocode.
2. This is an atomic instruction.
3. The X field is unused.

wfork — WAIT FOR A FORK

Purpose: Terminate a thread, idle the current CPU and look for any posted forks

Architecture: **C200 Series only**

Format:



Operation:

```
if (!rcv(threadcount))          /* loop until rcv() succeeds,    */
     (restart instruction);     /* ...accepting interrupts */
if (threadcount == 1) {
     if (rcv(forkposted)) {
          if (fork.type == PFORKED) {  /* take fork in this CIR */
               ulk(forklck);          /* clear fork */
          } else {
               rcv(forklck);          /* clear fork */
               snd(threadcount);
               (Deadlock);            /* mixed wfork and join */
          }
     } else {
          snd(threadcount);
          (Deadlock);                 /* last thread termination */
     }
} else {
     (idle the CPU);
}
```

Exceptions: Deadlock Exception (Last Thread Termination)
Deadlock Exception (Mixed *wfork* and *join*)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
wfork	7C98	ST 0111110010	None	Wait for a fork

Description: The *wfork* instruction returns the CPU to the idle state and disassociates the CPU from any process context. It then attempts to find other posted forks. The current CIR is examined first; if a posted fork is found there, then the fork is taken directly and the CPU need not be idled. If there is no fork in the current CIR, the CPU is idled and forks are accepted from other CIRs. The specific actions of an idle CPU are described in the *CONVEX Architecture Reference*, "Multiprocess Management" chapter.

Notes:

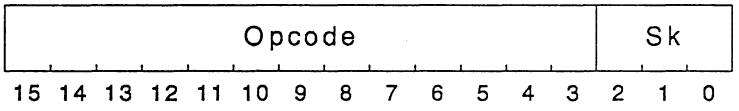
1. Programs that mix the *wfork* and *join* instructions after a *spawn* instruction must properly synchronize execution of these instructions to insure that last thread termination deadlocks do not occur.
2. This instruction may be traced, i.e., if the either the PSW<TTR> bit or PSW<TIT> bit is set and the CPU attempts to go idle, an instruction trace trap will occur.
3. The X field is ignored.

TRANSMIT INTERRUPT — xmti Sk

Purpose: To interrupt a channel

Architecture: C100 Series, C200 Series

Format:



Operation:

```
(Assert the channel interrupt line of virtual channel c(Sk)<7..0>);
```

Exceptions: Ring Violation (Privileged Instruction)

Opcode:

Mnemonic	Hex	Binary	PSW	Description
xmti Sk	7D68	ST 0111110101101	C	Transmit interrupt

Description: The *xmti* instruction asserts an interrupt to one of 256 virtual interrupt channels specified by Sk <7..0>.

Notes:

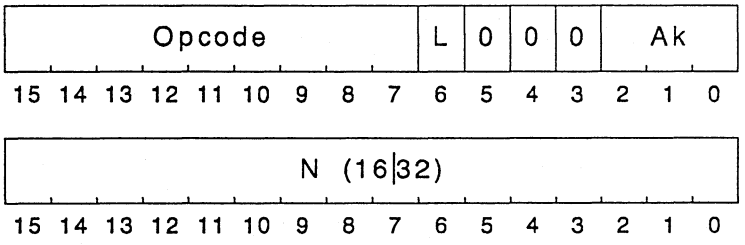
1. Channels <0..7> are associated with the CPU and are maskable (refer to the *mski* instruction).
2. Address Carry (C) of the PSW is set to 1 if the operation times out, and C is cleared to zero if the interrupt is successfully transmitted.

xor #N,Ak — EXCLUSIVE OR ADDRESS/IMMEDIATE

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

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Ak = Ak ^ Immediate;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
xor #N,Ak	1300	ST 000100110	None	Exclusive OR immediate to address register

Description: The exclusive OR of the (sign-extended) immediate field and the contents of address register Ak replace the contents of Ak.

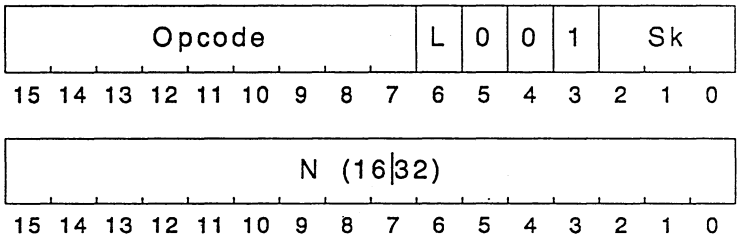
Notes: None

EXCLUSIVE OR SCALAR/IMMEDIATE — xor #N,Sk

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

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Sk<31..0>  = Sk<31..0> ^ Immediate;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
xor #N,Sk	1308	ST 000100110	None	Exclusive OR scalar/immediate

Description: The exclusive OR of the (sign-extended) immediate field and the least significant 32 bits of scalar register Sk replace the least significant 32 bits of Sk.

Notes:

1. The most significant bits of Sk are not affected.

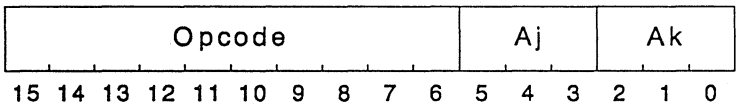
xor Aj,Ak

EXCLUSIVE OR ADDRESS/ADDRESS

Purpose: To exclusive OR the contents of two address registers

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Ak = Ak ^ Aj;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
xor Aj,Ak	5280	ST 0101001010	None	Exclusive OR address register

Description: The exclusive OR of the contents of address registers Aj and Ak replaces the contents of Ak.

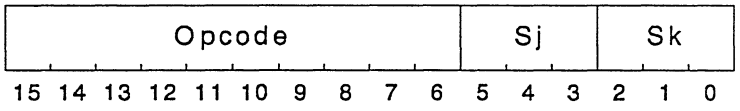
Notes: None

EXCLUSIVE OR SCALAR/SCALAR — xor Sj,Sk

Purpose: To exclusive OR the contents of two scalar registers

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
Sk = Sk ^ Sj;
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
xor Sj,Sk	5380	ST 0101001110	None	Exclusive OR scalar/scalar

Description: The exclusive OR of the contents of scalar registers Sj and Sk replaces the contents of Sk.

Notes: None

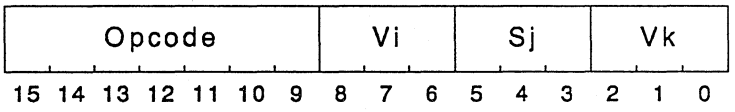
xor Vi,Sj,Vk

EXCLUSIVE OR VECTOR/SCALAR

Purpose: To exclusive OR the elements of a vector register and the contents of a scalar register

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
for (a = 0; a < VL; a++) {
     Vk[a] = Vi[a] ^ Sj;
}
```

Exceptions: None

Opcode:

Mnemonic	Hex Binary	PSW	Description
xor Vi,Sj,Vk	AC00 ST 1010110	None	Exclusive OR vector/scalar

Description: The exclusive OR of the contents of the corresponding element of Vi and the contents of scalar register Sj replaces the contents of each of the first VL elements of vector register Vk.

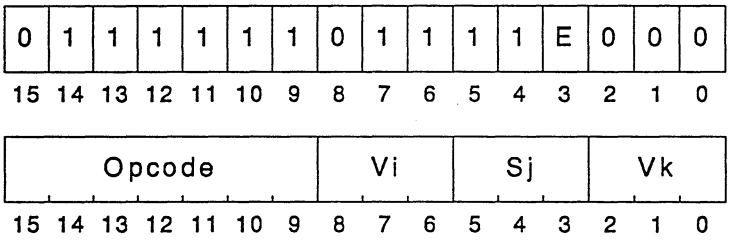
Notes: None

EXCLUSIVE OR VECTOR/SCALAR MASKED — xor.(t|f) Vi,Sj,Vk

Purpose: To exclusive OR the contents of a vector and a scalar under control of the Vector Merge (VM) register

Architecture: C200 Series only

Format:



Operation:

```
switch (E) { /* prefix bit<3> */
     case TRUE:  /* .t */
         for (a = 0; a < VL; a++) {
             if (VM<a> == 1) {        /* if VM<a> is TRUE */
                 Vk[a] = Vi[a] ^ Sj;
             }
         }  /* end of for loop */
         break;  /* go to end of switch */
     case FALSE: /* .f */
         for (a = 0; a < VL; a++) {
             if (VM<a> == 0) {        /* if VM<a> is FALSE */
                 Vk[a] = Vi[a] ^ Sj;
             }
         }  /* end of for loop */
         break;  /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
xor.t Vi,Sj,Vk	AC00	E1 1010110	None	Exclusive OR vector/scalar (VM)
xor.f Vi,Sj,Vk	AC00	E0 1010110	None	Exclusive OR vector/scalar (!VM)

Description: The contents each of the first VL elements of vector register Vk are replaced by the exclusive OR of the contents of the corresponding element of Vi and the contents of scalar register Sj, if the corresponding VM bit is set (clear for *.f*).

Notes: None

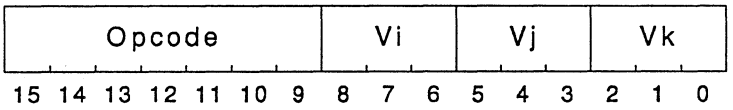
xor Vi,Vj,Vk

EXCLUSIVE OR VECTOR/VECTOR

Purpose: To exclusive OR the elements of two vector registers

Architecture: **C100 Series, C200 Series**

Format:



Operation:

```
for (a = 0; a < VL; a++) {
     Vk[a] = Vi[a] ^ Vj[a];
}
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
xor Vi,Vj,Vk	A400	ST 1010010	None	Exclusive OR two vectors

Description: The exclusive OR of the contents of the corresponding elements of Vi and Vj replaces the contents of each of the first VL elements of vector register Vk.

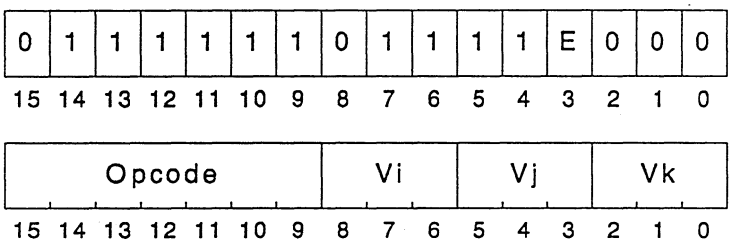
Notes: None

EXCLUSIVE OR VECTOR/VECTOR MASKED — xor.(t|f) Vi,Vj,Vk

Purpose: To exclusive OR the contents of two vector registers under control of the Vector Merge (VM) register

Architecture: C200 Series only

Format:



Operation:

```
switch (E) { /* prefix bit<3> */
     case TRUE:  /* .t */
         for (a = 0; a < VL; a++) {
             if (VM<a> == 1) {        /* if VM<a> is TRUE */
                 Vk[a] = Vi[a] ^ Vj[a];
             }
         }   /* end of for loop */
         break;  /* go to end of switch */
     case FALSE: /* .f */
         for (a = 0; a < VL; a++) {
             if (VM<a> == 0) {        /* if VM<a> is FALSE */
                 Vk[a] = Vi[a] ^ Vj[a];
             }
         }   /* end of for loop */
         break;  /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
xor.t Vi,Vj,Vk	A400	E1 1010010	None	Exclusive OR two vectors (VM)
xor.f Vi,Vj,Vk	A400	E0 1010010	None	Exclusive OR two vectors (!VM)

Description: The contents of each of the first VL elements of vector register Vk are replaced by the exclusive OR of the contents of the corresponding elements of Vi and Vj, if the corresponding VM bit is set (clear for *.f*).

Notes: None

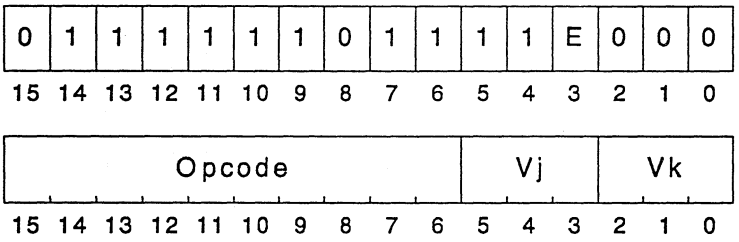
xpnd.(t|f) Vj,Vk

EXPAND VECTOR

Purpose: To expand a vector using the Vector Merge (VM) register

Architecture: **C200 Series only**

Format:



Operation:

```
a = 0;
switch (E) {  /* prefix bit<3> */
      case TRUE:   /* .t */
          for (b = 0; b < VL; b++) {
              if (VM<b> == 1) {    /* if VM<b> is TRUE */
                  Vk[b] = Vj[a];
                  a = a + 1;
              }
          }   /* end of for loop */
          break;   /* go to end of switch */
      case FALSE:  /* .f */
          for (b = 0; b < VL; b++) {
              if (VM<b> == 0) {    /* if VM<b> is FALSE */
                  Vk[b] = Vj[a];
                  a = a + 1;
              }
          }   /* end of for loop */
          break;   /* go to end of switch */
} /* end of switch */
```

Exceptions: None

Opcode:

Mnemonic	Hex	Binary	PSW	Description
xpnd.t Vj,Vk	6280	E1 0110001010	None	Expand a vector (VM)
xpnd.f Vj,Vk	6280	E0 0110001010	None	Expand a vector (!VM)

Description: The *xpnd* instruction copies all 64 bits of elements from the source vector register Vj to positions in the destination vector register Vk that are selected (or not selected for *cprs.f*) by the VM register. The number of elements copied to Vk is equal to the number of 1's (or 0's for *cprs.f*) in the VM register.

Notes: The *plc VM* instruction calculates the number of elements copied from Vj.

B

Opcodes Sorted by Name

ST 5900	A-6	add.b Sj,SkAdd scalar/scalar integer byte
ST C800	A-8	add.b Vi,Sj,VkAdd vector/scalar integer byte
ST C000	A-10	add.b Vi,Vj,VkAdd vector/vector integer byte
E0 C800	A-9	add.b.f Vi,Sj,VkAdd vector/scalar byte (!VM)
E0 C000	A-11	add.b.f Vi,Vj,VkAdd vector/vector byte (!VM)
E1 C800	A-9	add.b.t Vi,Sj,VkAdd vector/scalar byte (VM)
E1 C000	A-11	add.b.t Vi,Vj,VkAdd vector/vector byte (VM)
ST 5540	A-6	add.d Sj,SkAdd scalar/scalar double float
ST BA00	A-8	add.d Vi,Sj,VkAdd vector/scalar double float
ST B200	A-10	add.d Vi,Vj,VkAdd vector/vector double float
E0 BA00	A-9	add.d.f Vi,Sj,VkAdd vector/scalar double (!VM)
E0 B200	A-11	add.d.f Vi,Vj,VkAdd vector/vector double (!VM)
E1 BA00	A-9	add.d.t Vi,Sj,VkAdd vector/scalar double (VM)
E1 B200	A-11	add.d.t Vi,Vj,VkAdd vector/vector double (VM)
ST 1400	A-3	add.h #N,AkAdd immediate address halfword
ST 5880	A-3	add.h #n,AkAdd short immediate address halfword
ST 1408	A-4	add.h #N,SkAdd scalar/immediate integer halfword
ST 5800	A-5	add.h Aj,AkAdd address register halfword
ST 5940	A-6	add.h Sj,SkAdd scalar/scalar integer halfword
ST CA00	A-8	add.h Vi Sj,VkAdd vector/scalar integer halfword
ST C200	A-10	add.h Vi,Vj,VkAdd vector/vector integer halfword
E0 CA00	A-9	add.h.f Vi Sj,VkAdd vector/scalar halfword (!VM)
E0 C200	A-11	add.h.f Vi,Vj,VkAdd vector/vector halfword (!VM)
E1 CA00	A-9	add.h.t Vi Sj,VkAdd vector/scalar halfword (VM)
E1 C200	A-11	add.h.t Vi,Vj,VkAdd vector/vector halfword (VM)
ST 59C0	A-6	add.l Sj,SkAdd scalar/scalar integer longword
ST CE00	A-8	add.l Vi,Sj,VkAdd vector/scalar integer longword
ST C600	A-10	add.l Vi,Vj,VkAdd vector/vector integer longword
E0 CE00	A-9	add.l.f Vi,Sj,VkAdd vector/scalar longword (!VM)
E0 C600	A-11	add.l.f Vi,Vj,VkAdd vector/vector longword (!VM)
E1 CE00	A-9	add.l.t Vi,Sj,VkAdd vector/scalar longword (VM)
E1 C600	A-11	add.l.t Vi,Vj,VkAdd vector/vector longword (VM)
ST 1808	A-4	add.s #N,SkAdd scalar/immediate single float
ST 5500	A-6	add.s Sj,SkAdd scalar/scalar single float
ST B800	A-8	add.s Vi,Sj,VkAdd vector/scalar single float
ST B000	A-10	add.s Vi,Vj,VkAdd vector/vector single float
E0 B800	A-9	add.s.f Vi,Sj,VkAdd vector/scalar single (!VM)
E0 B000	A-11	add.s.f Vi,Vj,VkAdd vector/vector single (!VM)
E1 B800	A-9	add.s.t Vi,Sj,VkAdd vector/scalar single (VM)
E1 B000	A-11	add.s.t Vi,Vj,VkAdd vector/vector single (VM)
ST 1480	A-3	add.w #N,AkAdd immediate address word
ST 58C0	A-3	add.w #n,AkAdd short immediate address word
ST 1488	A-4	add.w #N,SkAdd scalar/immediate integer word
ST 5840	A-5	add.w Aj,AkAdd address register word
ST 5000	A-7	add.w Sj,AkAdd scalar to address word
ST 5980	A-6	add.w Sj,SkAdd scalar/scalar integer word
ST CC00	A-8	add.w Vi,Sj,VkAdd vector/scalar integer word
ST C400	A-10	add.w Vi,Vj,VkAdd vector/vector integer word

E0 CC00	A-9	add.w.f Vi,Sj,VkAdd vector/scalar word (!VM)
E0 C400	A-11	add.w.f Vi,Vj,VkAdd vector/vector word (!VM)
E1 CC00	A-9	add.w.t Vi,Sj,VkAdd vector/scalar word (VM)
E1 C400	A-11	add.w.t Vi,Vj,VkAdd vector/vector word (VM)
ST 7E20	A-12	all SkAND reduce a vector
ST 7E20	A-12	all VkAND reduce a vector
E0 7E20	A-13	all.f Sk AND reduce a vector (!VM)
E0 7E20	A-13	all.f Vk AND reduce a vector (!VM)
E1 7E20	A-13	all.t Sk AND reduce a vector (VM)
E1 7E20	A-13	all.t Vk AND reduce a vector (VM)
ST 1200	A-14	and #N,AkAND immediate to address register
ST 1208	A-15	and #N,SkAND scalar/immediate
ST 5200	A-16	and Aj,AkAND address register
ST 5300	A-17	and Sj,SkAND scalar/scalar
ST A800	A-18	and Vi,Sj,VkAND vector/scalar
ST A000	A-20	and Vi,Vj,VkAND two vectors
E0 A800	A-19	and.f Vi,Sj,VkAND vector/scalar (!VM)
E0 A000	A-21	and.f Vi,Vj,VkAND two vectors (!VM)
E1 A800	A-19	and.t Vi,Sj,VkAND vector/scalar (VM)
E1 A000	A-21	and.t Vi,Vj,VkAND two vectors (VM)
ST 7E28	A-22	any Sk OR reduce a vector
ST 7E28	A-22	any Vk OR reduce a vector
E0 7E28	A-23	any.f SkOR reduce a vector (!VM)
E0 7E28	A-23	any.f VkOR reduce a vector (!VM)
E1 7E28	A-23	any.t SkOR reduce a vector (VM)
E1 7E28	A-23	any.t VkOR reduce a vector (VM)
ST 7DF8	A-24	atan.d SkArc-tangent of a double float
ST 7DF0	A-24	atan.s SkArc-tangent of a single float
ST 7D50	A-25	bkptBreakpoint
ST 7100	A-26	brBranch always
ST 7400	A-26	bra.fBranch on address carry false
ST 7500	A-26	bra.tBranch on address carry true
ST 7200	A-26	bri.fBranch on ION false[5]
ST 7300	A-26	bri.tBranch on ION true[5]
ST 7600	A-26	brs.fBranch on scalar carry false
ST 7700	A-26	brs.tBranch on scalar carry true
ST 2000	A-27	call <effa>Call a subroutine, long frame
ST 2200	A-29	callq <effa>Push the PC and jump
ST 2100	A-30	calls <effa>Call a subroutine, short frame
ST 7C88	A-32	cforkClear a fork
ST 7CE8	A-33	cos.d SkCosine of a double-precision number
ST 7CE0	A-33	cos.s SkCosine of a single-precision number
ST 6380	A-34	cprs.f Vj,VkCompress a vector (!VM)
ST 63C0	A-34	cprs.t Vj,VkCompress a vector (VM)
E0 7C38	A-35	ctrsgMove scalar to CPU timer
ST 4080	A-36	cvtb.w Aj,AkConvert byte to word
ST 4180	A-37	cvtb.w Sj,SkConvert byte to word
E0 4080	A-39	cvtb.w Vj,VkConvert byte to word
E0 4180	A-41	cvtb.w.f Vj,VkConvert byte to word (!VM)
E1 4180	A-41	cvtb.w.t Vj,VkConvert byte to word (VM)
ST 4340	A-37	cvtd.l Sj,SkConvert double float to longword
ST 60C0	A-39	cvtd.l Vj,VkConvert double float to longword
E0 60C0	A-42	cvtd.l.f Vj,VkConvert double to longword (!VM)
E1 60C0	A-41	cvtd.l.t Vj,VkConvert double to longword (VM)
ST 4280	A-37	cvtd.s Sj,SkConvert double float to single float
ST 6000	A-39	cvtd.s Vj,VkConvert double float to single float
E0 6000	A-41	cvtd.s.f Vj,VkConvert double to single (!VM)
E1 6000	A-41	cvtd.s.t Vj,VkConvert double to single (VM)
E0 4540	A-37	cvtd.w Sj,SkConvert double float to word

E0 4240	A-39	cvtd.w Vj,VkConvert double to word float
E0 4340	A-42	cvtd.w.f Vj,VkConvert double to word (!VM)
E1 4340	A-42	cvtd.w.t Vj,VkConvert double to word (VM)
ST 40C0	A-36	cvth.w Aj,AkConvert halfword to word
ST 41C0	A-37	cvth.w Sj,SkConvert halfword to word
E0 40C0	A-39	cvth.w Vj,VkConvert halfword to word
E0 41C0	A-41	cvth.w.f Vj,VkConvert halfword to word (!VM)
E1 41C0	A-41	cvth.w.t Vj,VkConvert halfword to word (VM)
ST 43C0	A-37	cvtl.d Sj,SkConvert longword to double float
ST 6080	A-39	cvtl.d Vj,VkConvert longword to double float
E0 6080	A-42	cvtl.d.f Vj,VkConvert longword to double (!VM)
E1 6080	A-42	cvtl.d.t Vj,VkConvert longword to double (VM)
ST 4380	A-37	cvtl.s Sj,SkConvert longword to single float
E0 4280	A-39	cvtl.s Vj,VkConvert longword to single float
E0 4380	A-42	cvtl.s.f Vj,VkConvert longword to single (!VM)
E1 4380	A-42	cvtl.s.t Vj,VkConvert longword to single (VM)
ST 4500	A-37	cvtl.w Sj,SkConvert longword to word
ST 79C0	A-39	cvtl.w Vj,VkConvert longword to word
E0 79C0	A-42	cvtl.w.f Vj,VkConvert longword to word (!VM)
E1 79C0	A-42	cvtl.w.t Vj,VkConvert longword to word (VM)
ST 42C0	A-37	cvts.d Sj,SkConvert single float to double float
ST 6040	A-39	cvts.d Vj,VkConvert single float to double float
E0 6040	A-41	cvts.d.f Vj,VkConvert single to double (!VM)
E1 6040	A-41	cvts.d.t Vj,VkConvert single to double (VM)
ST 4300	A-37	cvts.l Sj,SkConvert single float to longword
E0 4200	A-39	cvts.l Vj,VkConvert single float to longword
E0 4300	A-41	cvts.l.f Vj,VkConvert single to longword (!VM)
E1 4300	A-41	cvts.l.t Vj,VkConvert single to longword (VM)
ST 4240	A-37	cvts.w Sj,SkConvert single float to word
ST 7940	A-39	cvts.w Vj,VkConvert single float to word
E0 7940	A-41	cvts.w.f Vj,VkConvert single to word (!VM)
E1 7940	A-41	cvts.w.t Vj,VkConvert single to word (VM)
ST 4000	A-36	cvtw.b Aj,AkConvert word to byte
ST 4100	A-37	cvtw.b Sj,SkConvert word to byte
E0 4000	A-39	cvtw.b Vj,VkConvert word to byte
E0 4100	A-41	cvtw.b.f Vj,VkConvert word to byte (!VM)
E1 4100	A-41	cvtw.b.t Vj,VkConvert word to byte (VM)
E0 4500	A-37	cvtw.d Sj,SkConvert word to double float
E0 42C0	A-39	cvtw.d Vj,VkConvert word to double float
E0 43C0	A-42	cvtw.d.f Vj,VkConvert word to double (!VM)
E1 43C0	A-42	cvtw.d.t Vj,VkConvert word to double (VM)
ST 4040	A-36	cvtw.h Aj,AkConvert word to halfword
ST 4140	A-37	cvtw.h Sj,SkConvert word to halfword
E0 4040	A-39	cvtw.h Vj,VkConvert word to halfword
E0 4140	A-41	cvtw.h.f Vj,VkConvert word to halfword (!VM)
E1 4140	A-41	cvtw.h.t Vj,VkConvert word to halfword (VM)
ST 4540	A-37	cvtw.l Sj,SkConvert word to longword
ST 7980	A-39	cvtw.l Vj,VkConvert word to longword
E0 7980	A-42	cvtw.l.f Vj,VkConvert word to longword (!VM)
E1 7980	A-42	cvtw.l.t Vj,VkConvert word to longword (VM)
ST 4200	A-37	cvtw.s Sj,SkConvert word to single float
ST 7900	A-39	cvtw.s Vj,VkConvert word to single float
E0 7900	A-41	cvtw.s.f Vj,VkConvert word to single (!VM)
E1 7900	A-41	cvtw.s.t Vj,VkConvert word to single (VM)
ST 7DC0	A-43	diag AkExecute nonstandard microcode sequence
ST 5F00	A-48	div.b Sj,SkDivide scalar/scalar integer byte
ST F800	A-51	div.b Vi,Sj,VkDivide vector/scalar integer byte
ST F000	A-54	div.b Vi,Vj,VkDivide vector/vector integer byte
E0 F800	A-52	div.b.f Vi,Sj,VkDivide vector/scalar byte (!VM)

E0 F000	A-55	div.b.f Vi,Vj,VkDivide byte vectors (!VM)
E1 F800	A-52	div.b.t Vi,Sj,VkDivide vector/scalar byte (VM)
E1 F000	A-55	div.b.t Vi,Vj,VkDivide byte vectors (VM)
E0 8600	A-49	div.d Si,Vj,VkDivide scalar/vector double float
ST 57C0	A-48	div.d Sj,SkDivide scalar/scalar double float
ST 9E00	A-51	div.d Vi,Sj,VkDivide vector/scalar double float
ST 9600	A-54	div.d Vi,Vj,VkDivide vector/vector double float
E0 8E00	A-50	div.d.f Si,Vj,VkDivide scalar/vector double (!VM)
E1 9E00	A-52	div.d.f Vi,Sj,VkDivide vector/scalar double (!VM)
E0 9600	A-55	div.d.f Vi,Vj,VkDivide double vectors (!VM)
E1 8E00	A-50	div.d.t Si,Vj,VkDivide scalar/vector double (VM)
E0 9E00	A-52	div.d.t Vi,Sj,VkDivide vector/scalar double (VM)
E1 9600	A-55	div.d.t Vi,Vj,VkDivide double vectors (VM)
ST 1700	A-45	div.h #N,AkDivide immediate address halfword
ST 5E80	A-45	div.h #n,AkDivide short immediate address halfword
ST 1708	A-46	div.h #N,SkDivide scalar/scalar integer halfword
ST 5E00	A-47	div.h Aj,AkDivide address register halfword
ST 5F40	A-48	div.h Sj,SkDivide scalar/scalar integer halfword
ST FA00	A-51	div.h Vi,Sj,VkDivide vector/scalar integer halfword
ST F200	A-54	div.h Vi,Vj,VkDivide vector/vector integer halfword
E0 FA00	A-52	div.h.f Vi,Sj,VkDivide vector/scalar halfword (!VM)
E0 F200	A-55	div.h.f Vi,Vj,VkDivide halfword vectors (!VM)
E1 FA00	A-52	div.h.t Vi,Sj,VkDivide vector/scalar halfword (VM)
E1 F200	A-55	div.h.t Vi,Vj,VkDivide halfword vectors (VM)
ST 5FC0	A-48	div.l Sj,SkDivide scalar/scalar integer longword
ST FE00	A-51	div.l Vi,Sj,VkDivide vector/scalar integer longword
ST F600	A-54	div.l Vi,Vj,VkDivide vector/vector integer longword
E0 FE00	A-52	div.l.f Vi,Sj,VkDivide vector/scalar longword (!VM)
E0 F600	A-55	div.l.f Vi,Vj,VkDivide longword vectors (!VM)
E1 FE00	A-52	div.l.t Vi,Sj,VkDivide vector/scalar longword (VM)
E1 F600	A-55	div.l.t Vi,Vj,VkDivide longword vectors (VM)
ST 1988	A-46	div.s #N,SkDivide scalar/scalar single float
E0 8400	A-49	div.s Si,Vj,VkDivide scalar/vector single float
ST 5780	A-48	div.s Sj,SkDivide scalar/scalar single float
ST 9C00	A-51	div.s Vi,Sj,VkDivide vector/scalar single float
ST 9400	A-54	div.s Vi,Vj,VkDivide vector/vector single float
E0 8C00	A-50	div.s.f Si,Vj,VkDivide scalar/vector single (!VM)
E1 9C00	A-52	div.s.f Vi,Sj,VkDivide vector/scalar single (!VM)
E0 9400	A-55	div.s.f Vi,Vj,VkDivide single vectors (!VM)
E1 8C00	A-50	div.s.t Si,Vj,VkDivide scalar/vector single (VM)
E1 9C00	A-52	div.s.t Vi,Sj,VkDivide vector/scalar single (VM)
E1 9400	A-55	div.s.t Vi,Vj,VkDivide single vectors (VM)
ST 1780	A-45	div.w #N,AkDivide immediate address word
ST 5EC0	A-45	div.w #n,AkDivide short immediate address word
ST 1788	A-46	div.w #N,SkDivide scalar/scalar integer word
ST 5E40	A-47	div.w Aj,AkDivide address register word
ST 5F80	A-48	div.w Sj,SkDivide scalar/scalar integer word
ST FC00	A-51	div.w Vi,Sj,VkDivide vector/scalar integer word
ST F400	A-54	div.w Vi,Vj,VkDivide vector/vector integer word
E0 FC00	A-52	div.w.f Vi,Sj,VkDivide vector/scalar word (!VM)
E0 F400	A-55	div.w.f Vi,Vj,VkDivide word vectors (!VM)
E1 FC00	A-52	div.w.t Vi,Sj,VkDivide vector/scalar word (VM)
E1 F400	A-55	div.w.t Vi,Vj,VkDivide word vectors (VM)
ST 7D48	A-57	dsiDisable interrupts; reset ION to 0
E0 4480	A-58	enag Sj,SkEnable all global CPU interrupts
E0 4400	A-59	enal Sj,SkEnable local CPU interrupt
ST 7D40	A-60	eniEnable interrupts; set ION to 1
ST 4700	A-103	eq.b Sj,SkCompare equal byte
ST 6900	A-104	eq.b Sj,VkCompare equal byte

ST 6800	A-108	eq.b Vj,VkCompare equal byte
E0 6900	A-107	eq.b.f Sj,VkCompare equal byte (!VM)
E0 6800	A-111	eq.b.f Vj,VkCompare equal byte (!VM)
E1 6900	A-107	eq.b.t Sj,VkCompare equal byte (VM)
E1 6800	A-111	eq.b.t Vj,VkCompare equal byte (VM)
ST 5640	A-103	eq.d Sj,SkCompare equal double float
ST 6540	A-104	eq.d Sj,VkCompare equal double precision
ST 6440	A-108	eq.d Vj,VkCompare equal double precision
E0 6540	A-107	eq.d.f Sj,VkCompare equal double (!VM)
E0 6440	A-111	eq.d.f Vj,VkCompare equal double (!VM)
E1 6540	A-107	eq.d.t Sj,VkCompare equal double (VM)
E1 6440	A-111	eq.d.t Vj,VkCompare equal double (VM)
ST 1B00	A-100	eq.h #N,AkCompare equal halfword
ST 4680	A-100	eq.h #n,AkCompare equal halfword
ST 1B08	A-101	eq.h #N,SkCompare equal halfword
ST 4600	A-102	eq.h Aj,AkCompare equal halfword
ST 4740	A-103	eq.h Sj,SkCompare equal halfword
ST 6940	A-104	eq.h Sj,VkCompare equal halfword
ST 6840	A-108	eq.h Vj,VkCompare equal halfword
E0 6940	A-107	eq.h.f Sj,VkCompare equal halfword (!VM)
E0 6840	A-111	eq.h.f Vj,VkCompare equal halfword (!VM)
E1 6940	A-107	eq.h.t Sj,VkCompare equal halfword (VM)
E1 6840	A-111	eq.h.t Vj,VkCompare equal halfword (VM)
ST 47C0	A-103	eq.l Sj,SkCompare equal longword
ST 69C0	A-104	eq.l Sj,VkCompare equal longword
ST 68C0	A-108	eq.l Vj,VkCompare equal longword
E0 69C0	A-107	eq.l.f Sj,VkCompare equal long (!VM)
E0 68C0	A-111	eq.l.f Vj,VkCompare equal long (!VM)
E1 69C0	A-107	eq.l.t Sj,VkCompare equal long (VM)
E1 68C0	A-111	eq.l.t Vj,VkCompare equal long (VM)
ST 5600	A-103	eq.s Sj,SkCompare equal single float
ST 6500	A-104	eq.s Sj,VkCompare equal single
ST 6400	A-108	eq.s Vj,VkCompare equal single
E0 6500	A-107	eq.s.f Sj,VkCompare equal single (!VM)
E0 6400	A-111	eq.s.f Vj,VkCompare equal single (!VM)
E1 6500	A-107	eq.s.t Sj,VkCompare equal single (VM)
E1 6400	A-111	eq.s.t Vj,VkCompare equal single (VM)
ST 1B80	A-100	eq.w #N,AkCompare equal word
ST 46C0	A-100	eq.w #n,AkCompare equal word
ST 1B88	A-101	eq.w #N,SkCompare equal word
ST 4640	A-102	eq.w Aj,AkCompare equal word
ST 4780	A-103	eq.w Sj,SkCompare equal word
ST 6980	A-104	eq.w Sj,VkCompare equal word
ST 6880	A-108	eq.w Vj,VkCompare equal word
E0 6980	A-107	eq.w.f Sj,VkCompare equal word (!VM)
E0 6880	A-111	eq.w.f Vj,VkCompare equal word (!VM)
E1 6980	A-107	eq.w.t Sj,VkCompare equal word (VM)
E1 6880	A-111	eq.w.t Vj,VkCompare equal word (VM)
ST 0000	A-61	exitError exit instruction
ST 7CB8	A-62	exp.d SkExponent of a double float
ST 7CB0	A-62	exp.s SkExponent of a single float
E0 4740	A-63	frint.d Sj,SkIntegerize float double scalar
E0 58C0	A-64	frint.d Vj,VkIntegerize float double vector
E0 59C0	A-65	frint.d.f Vj,VkIntegerize double vector (!VM)
E1 59C0	A-65	frint.d.t Vj,VkIntegerize double vector (VM)
E0 4700	A-63	frint.s Sj,SkIntegerize float single scalar
E0 5880	A-64	frint.s Vj,VkIntegerize float single vector
E0 5980	A-65	frint.s.f Vj,VkIntegerize single vector (!VM)
E1 5980	A-65	frint.s.t Vj,VkIntegerize single vector (VM)

E0 3200	A-68	get.l <Ceffa>,SkGet communication/scalar
E0 2A00	A-67	get.w <Ceffa>,AkGet communication/address
ST 1000	A-69	halt #N,AkHalt the CPU
ST 7C58	A-70	idle SkIdle the CPU
E0 2D00	A-72	inc.l <Ceffa>,SkIncrement communication/scalar
E0 2D00	A-71	inc.w <Ceffa>,AkIncrement communication/address
ST 2F00	A-74	incr.l <effa>,SkIncrement long resource structure
ST 2B00	A-73	incr.w <effa>,AkIncrement resource structure data
ST 0100	A-75	jmp <effa>Jump always
ST 0400	A-75	jmpa.f <effa>Jump on address carry false
ST 0500	A-75	jmpa.t <effa>Jump on address carry true
ST 0200	A-75	jmpi.f <effa>Jump on ION false
ST 0300	A-75	jmpi.t <effa>Jump on ION true
ST 0600	A-75	jmps.f <effa>Jump on scalar carry false
ST 0700	A-75	jmps.t <effa>Jump on scalar carry true
ST 7CA0	A-76	joinJoin all threads
E0 0200	A-77	lck <Ceffa>Lock communication register
ST 2800	A-82	ld.b <effa>,AkLoad address register byte
ST 3000	A-83	ld.b <effa>,SkLoad scalar byte
ST 3800	A-84	ld.b <effa>,VkLoad vector byte
E0 3800	A-85	ld.b.f <effa>,VkLoad vector byte (!VM)
E1 3800	A-85	ld.b.t <effa>,VkLoad vector byte (VM)
ST 1008	A-79	ld.d #N,SkLoad double float immediate upper 32 bits
ST 3300	A-83	ld.d <effa>,SkLoad scalar double float
ST 3B00	A-84	ld.d <effa>,VkLoad vector double float
E0 3B00	A-85	ld.d.f <effa>,VkLoad vector double float (!VM)
E1 3B00	A-85	ld.d.t <effa>,VkLoad vector double float (VM)
ST 1188	A-79	ld.dl #N,SkLoad 64-bit floating immediate, lower half
ST 1088	A-79	ld.du #N,SkLoad 64-bit floating immediate, upper half
ST 1100	A-78	ld.h #N,AkLoad halfword immediate into Ak
ST 4480	A-78	ld.h #n,AkLoad short immediate into Ak
ST 2900	A-82	ld.h <effa>,AkLoad address register halfword
ST 3100	A-83	ld.h <effa>,SkLoad scalar halfword
ST 3900	A-84	ld.h <effa>,VkLoad vector halfword
E0 3900	A-85	ld.h.f <effa>,VkLoad vector halfword (!VM)
E1 3900	A-85	ld.h.t <effa>,VkLoad vector halfword (VM)
ST 1108	A-79	ld.l #N,SkLoad 32-bit immediate sign-extended to 64 bits
ST 3300	A-83	ld.l <effa>,SkLoad scalar longword
ST 3B00	A-84	ld.l <effa>,VkLoad vector longword
ST 0A00	A-87	ld.l <effa>,VLSLoad VS and VL from memory
E0 3B00	A-85	ld.l.f <effa>,VkLoad vector longword (!VM)
E1 3B00	A-85	ld.l.t <effa>,VkLoad vector longword (VM)
ST 1188	A-79	ld.ll #N,SkLoad 64-bit integer immediate, lower half
ST 1088	A-79	ld.lu #N,SkLoad 64-bit integer immediate, upper half
ST 1188	A-79	ld.s #N,SkLoad a single float immediate
ST 3200	A-83	ld.s <effa>,SkLoad scalar single float
ST 3A00	A-84	ld.s <effa>,VkLoad vector single float
E0 3A00	A-85	ld.s.f <effa>,VkLoad vector single float (!VM)
E1 3A00	A-85	ld.s.t <effa>,VkLoad vector single float (VM)
ST 1088	A-79	ld.u #N,SkLoad immediate, upper half
ST 1180	A-78	ld.w #N,AkLoad immediate into Ak
ST 44C0	A-78	ld.w #n,AkLoad short immediate into Ak
ST 1188	A-79	ld.w #N,SkLoad a 32-bit immediate
ST 1800	A-80	ld.w #N,VLLoad VL with an immediate
ST 1880	A-81	ld.w #N,VSLoad VS from an immediate
ST 2A00	A-82	ld.w <effa>,AkLoad address register word
ST 3200	A-83	ld.w <effa>,SkLoad scalar word
ST 3A00	A-84	ld.w <effa>,VkLoad vector word
E0 3A00	A-85	ld.w.f <effa>,VkLoad vector word (!VM)

E1 3A00	A-85	ld.w.t <effa>,VkLoad vector word (VM)
ST 0B00	A-88	ld.x <effa>, VMLoad VM from memory
E0 0600	A-89	ldcmr <effa>,AkLoad communication registers
ST 0900	A-90	ldea <effa>,AkLoad effective address
E0 0400	A-91	ldea <effa>,SkLoad effective address/scalar
ST 7C08	A-92	ldkdr AkLoad all eight SDRs
ST 4400	A-94	ldpa Aj,AkLoad a physical byte address into Ak
ST 7C00	A-96	ldsdr AkLoad process SDRs
ST 7800	A-97	ldvi.b Vj,VkIndex load vector byte
E0 7800	A-98	ldvi.b.f Vj,VkIndex load vector byte (!VM)
E1 7800	A-98	ldvi.b.t Vj,VkIndex load vector byte (VM)
ST 78C0	A-97	ldvi.d Vj,VkIndex load vector double float
E0 78C0	A-98	ldvi.d.f Vj,VkIndex load vector double (!VM)
E1 78C0	A-98	ldvi.d.t Vj,VkIndex load vector double (VM)
ST 7840	A-97	ldvi.h Vj,VkIndex load vector halfword
E0 7840	A-98	ldvi.h.f Vj,VkIndex load vector halfword (!VM)
E1 7840	A-98	ldvi.h.t Vj,VkIndex load vector halfword (VM)
ST 78C0	A-97	ldvi.l Vj,VkIndex load vector longword
E0 78C0	A-98	ldvi.l.f Vj,VkIndex load vector longword (!VM)
E1 78C0	A-98	ldvi.l.t Vj,VkIndex load vector longword (VM)
ST 7880	A-97	ldvi.s Vj,VkIndex load vector single float
E0 7880	A-98	ldvi.s.f Vj,VkIndex load vector single (!VM)
E1 7880	A-98	ldvi.s.t Vj,VkIndex load vector single (VM)
ST 7880	A-97	ldvi.w Vj,VkIndex load vector word
E0 7880	A-98	ldvi.w.f Vj,VkIndex load vector word (!VM)
E1 7880	A-98	ldvi.w.t Vj,VkIndex load vector word (VM)
ST 4D00	A-103	le.b Sj,SkCompare less than or equal byte
ST 6B00	A-104	le.b Sj,VkCompare less than or equal byte
ST 6A00	A-108	le.b Vj,VkCompare less than or equal byte
E0 6B00	A-106	le.b.f Sj,VkCompare less than or equal byte (!VM)
E0 6A00	A-110	le.b.f Vj,VkCompare less than or equal byte (!VM)
E1 6B00	A-106	le.b.t Sj,VkCompare less than or equal byte (VM)
E1 6A00	A-110	le.b.t Vj,VkCompare less than or equal byte (VM)
ST 5440	A-103	le.d Sj,SkCompare less than or equal double float
ST 6740	A-104	le.d Sj,VkCompare less than or equal double float
ST 6640	A-108	le.d Vj,VkCompare less than or equal double float
E0 6740	A-107	le.d.f Sj,VkCompare less than or equal double (!VM)
E0 6640	A-111	le.d.f Vj,VkCompare less than or equal double (!VM)
E1 6740	A-106	le.d.t Sj,VkCompare less than or equal double (VM)
E1 6640	A-110	le.d.t Vj,VkCompare less than or equal double (VM)
ST 1E00	A-100	le.h #N,AkCompare less than or equal halfword
ST 4C80	A-100	le.h #n,AkCompare less than or equal halfword
ST 1E08	A-101	le.h #N,SkCompare less than or equal halfword
ST 4C00	A-102	le.h Aj,AkCompare less than or equal signed halfword
ST 4D40	A-103	le.h Sj,SkCompare less than or equal halfword
ST 6B40	A-104	le.h Sj,VkCompare less than or equal halfword
ST 6A40	A-108	le.h Vj,VkCompare less than or equal halfword
E0 6B40	A-106	le.h.f Sj,VkCompare less than or equal half (!VM)
E0 6A40	A-110	le.h.f Vj,VkCompare less than or equal half (!VM)
E1 6B40	A-106	le.h.t Sj,VkCompare less than or equal half (VM)
E1 6A40	A-110	le.h.t Vj,VkCompare less than or equal half (VM)
ST 4DC0	A-103	le.l Sj,SkCompare less than or equal longword
ST 6BC0	A-104	le.l Sj,VkCompare less than or equal longword
ST 6AC0	A-108	le.l Vj,VkCompare less than or equal longword
E0 6BC0	A-106	le.l.f Sj,VkCompare less than or equal long (!VM)
E0 6AC0	A-110	le.l.f Vj,VkCompare less than or equal long (!VM)
E1 6BC0	A-106	le.l.t Sj,VkCompare less than or equal long (VM)
E1 6AC0	A-110	le.l.t Vj,VkCompare less than or equal long (VM)
ST 1A08	A-101	le.s #N,SkCompare less than or equal single

ST 5400	A-103	le.s Sj,SkCompare less than or equal single float
ST 6700	A-104	le.s Sj,VkCompare less than or equal single
ST 6600	A-108	le.s Vj,VkCompare less than or equal single
E0 6700	A-106	le.s.f Sj,VkCompare less than or equal single (!VM)
E0 6600	A-110	le.s.f Vj,VkCompare less than or equal single (!VM)
E1 6700	A-106	le.s.t Sj,VkCompare less than or equal single (VM)
E1 6600	A-110	le.s.t Vj,VkCompare less than or equal single (VM)
ST 1E80	A-100	le.w #N,AkCompare less than or equal word
ST 4CC0	A-100	le.w #n,AkCompare less than or equal word
ST 1E88	A-101	le.w #N,SkCompare less than or equal word
ST 4C40	A-102	le.w Aj,AkCompare less than or equal signed word
ST 4D80	A-103	le.w Sj,SkCompare less than or equal word
ST 6B80	A-104	le.w Sj,VkCompare less than or equal word
ST 6A80	A-108	le.w Vj,VkCompare less than or equal word
E0 6B80	A-106	le.w.f Sj,VkCompare less than or equal word (!VM)
E0 6A80	A-110	le.w.f Vj,VkCompare less than or equal word (!VM)
E1 6B80	A-106	le.w.t Sj,VkCompare less than or equal word (VM)
E1 6A80	A-110	le.w.t Vj,VkCompare less than or equal word (VM)
ST 4900	A-115	leu.b Sj,SkCompare less than or equal to byte
ST 1C00	A-112	leu.h #N,AkCompare unsigned less than halfword
ST 4880	A-112	leu.h #n,AkCompare unsigned less than or equal halfword
ST 1C08	A-113	leu.h #N,SkCompare unsigned less than or equal to halfword
ST 4800	A-114	leu.h Aj,AkCompare unsigned less than or equal to halfword
ST 4940	A-115	leu.h Sj,SkCompare less than or equal to halfword
ST 49C0	A-115	leu.l Sj,SkCompare less than or equal to longword
ST 48C0	A-112	leu.w #n,AkCompare unsigned less than or equal word
ST 1C80	A-112	leu.w #N,AkCompare unsigned less than word
ST 1C88	A-113	leu.w #N,SkCompare unsigned less than or equal to word
ST 4840	A-114	leu.w Aj,AkCompare unsigned less than or equal to word
ST 4980	A-115	leu.w Sj,SkCompare less than or equal to word
ST 7C18	A-116	ln.d SkNatural logarithm of a double precision number
ST 7C10	A-116	ln.s SkNatural logarithm of a single precision number
ST 61C0	A-117	lop Sj,SkLeading one_s position in Sj
ST 6240	A-118	lop Vj,VkLeading ones position vector
E0 6240	A-119	lop.f Vj,VkLeading ones position vector (!VM)
E1 6240	A-119	lop.t Vj,VkLeading ones position vector (VM)
ST 4F00	A-103	lt.b Sj,SkCompare less than byte
ST 6D00	A-104	lt.b Sj,VkCompare less than byte
ST 6C00	A-108	lt.b Vj,VkCompare less than byte
E0 6D00	A-107	lt.b.f Sj,VkCompare less than byte (!VM)
E0 6C00	A-111	lt.b.f Vj,VkCompare less than byte (!VM)
E1 6D00	A-107	lt.b.t Sj,VkCompare less than byte (VM)
E1 6C00	A-111	lt.b.t Vj,VkCompare less than byte (VM)
ST 54C0	A-103	lt.d Sj,SkCompare less than double float
ST 67C0	A-104	lt.d Sj,VkCompare less than double float
ST 66C0	A-108	lt.d Vj,VkCompare less than double float
E0 67C0	A-107	lt.d.f Sj,VkCompare less than double (!VM)
E0 66C0	A-111	lt.d.f Vj,VkCompare less than double (!VM)
E1 67C0	A-107	lt.d.t Sj,VkCompare less than double (VM)
E1 66C0	A-111	lt.d.t Vj,VkCompare less than double (VM)
ST 1F00	A-100	lt.h #N,AkCompare less than halfword
ST 4E80	A-100	lt.h #n,AkCompare less than halfword
ST 1F08	A-101	lt.h #N,SkCompare less than halfword
ST 4E00	A-102	lt.h Aj,AkCompare less than signed halfword
ST 4F40	A-103	lt.h Sj,SkCompare less than halfword
ST 6D40	A-104	lt.h Sj,VkCompare less than halfword
ST 6C40	A-108	lt.h Vj,VkCompare less than halfword
E0 6D40	A-107	lt.h.f Sj,VkCompare less than halfword (!VM)
E0 6C40	A-111	lt.h.f Vj,VkCompare less than halfword (!VM)

E1 6D40	A-107	lt.h.t Sj,VkCompare less than halfword (VM)
E1 6C40	A-111	lt.h.t Vj,VkCompare less than halfword (VM)
ST 4FC0	A-103	lt.l Sj,SkCompare less than longword
ST 6DC0	A-104	lt.l Sj,VkCompare less than longword
ST 6CC0	A-108	lt.l Vj,VkCompare less than longword
E0 6DC0	A-107	lt.l.f Sj,VkCompare less than long (!VM)
E0 6CC0	A-111	lt.l.f Vj,VkCompare less than long (!VM)
E1 6DC0	A-107	lt.l.t Sj,VkCompare less than long (VM)
E1 6CC0	A-111	lt.l.t Vj,VkCompare less than long (VM)
ST 1A88	A-101	lt.s #N,SkCompare less than single
ST 5480	A-103	lt.s Sj,SkCompare less than single float
ST 6780	A-104	lt.s Sj,VkCompare less than single
ST 6680	A-108	lt.s Vj,VkCompare less than single
E0 6780	A-107	lt.s.f Sj,VkCompare less than single (!VM)
E0 6680	A-111	lt.s.f Vj,VkCompare less than single (!VM)
E1 6780	A-107	lt.s.t Sj,VkCompare less than single (VM)
E1 6680	A-111	lt.s.t Vj,VkCompare less than single (VM)
ST 1F80	A-100	lt.w #N,AkCompare less than word
ST 4EC0	A-100	lt.w #n,AkCompare less than word
ST 1F88	A-101	lt.w #N,SkCompare less than word
ST 4E40	A-102	lt.w Aj,AkCompare less than signed word
ST 4F80	A-103	lt.w Sj,SkCompare less than word
ST 6D80	A-104	lt.w Sj,VkCompare less than word
ST 6C80	A-108	lt.w Vj,VkCompare less than word
E0 6D80	A-107	lt.w.f Sj,VkCompare less than word (!VM)
E0 6C80	A-111	lt.w.f Vj,VkCompare less than word (!VM)
E1 6D80	A-107	lt.w.t Sj,VkCompare less than word (VM)
E1 6C80	A-111	lt.w.t Vj,VkCompare less than word (VM)
ST 4B00	A-115	ltu.b Sj,SkCompare less than byte
ST 1D00	A-112	ltu.h #N,AkCompare unsigned less than halfword
ST 4A80	A-112	ltu.h #n,AkCompare unsigned less than halfword
ST 1D08	A-113	ltu.h #N,SkCompare unsigned less than halfword
ST 4A00	A-114	ltu.h Aj,AkCompare unsigned less than halfword
ST 4B40	A-115	ltu.h Sj,SkCompare less than halfword
ST 4BC0	A-115	ltu.l Sj,SkCompare less than longword
ST 1D80	A-112	ltu.w #N,AkCompare unsigned less than word
ST 4AC0	A-112	ltu.w #n,AkCompare unsigned less than word
ST 1D88	A-113	ltu.w #N,SkCompare unsigned less than word
ST 4A40	A-114	ltu.w Aj,AkCompare unsigned less than word
ST 4B80	A-115	ltu.w Sj,SkCompare less than word
ST 8A00	A-121	mask.f Vi,Sj,VkMask vector/scalar (!VM)
ST 8E00	A-121	mask.t Vi,Sj,VkMask vector/scalar (VM)
ST 8600	A-120	mask.t Vi,Vj,VkMask vector/vector
E0 3100	A-123	mat.l Sk,<Ceffa>Match scalar/communication
E0 2900	A-122	mat.w Ak,<Ceffa>Match address/communication
ST 7E40	A-124	max.b VkMaximum of a vector of bytes
E0 7E40	A-125	max.b.f VkMaximum of vector of bytes (!VM)
E1 7E40	A-125	max.b.t VkMaximum of vector of bytes (VM)
ST 7EA8	A-124	max.d VkMaximum of a vector of double float
E0 7EA8	A-125	max.d.f VkMaximum of vector of doubles (!VM)
E1 7EA8	A-125	max.d.t VkMaximum of vector of doubles (VM)
ST 7E48	A-124	max.h VkMaximum of a vector of halfwords
E0 7E48	A-125	max.h.f VkMaximum of vector of halfwords (!VM)
E1 7E48	A-125	max.h.t VkMaximum of vector of halfwords (VM)
ST 7E58	A-124	max.l VkMaximum of a vector of longwords
E0 7E58	A-125	max.l.f VkMaximum of vector of longwords (!VM)
E1 7E58	A-125	max.l.t VkMaximum of vector of longwords (VM)
ST 7EA0	A-124	max.s VkMaximum of a vector of single float
E0 7EA0	A-125	max.s.f VkMaximum of vector of singles (!VM)

E1 7EA0	A-125	max.s.t VkMaximum of vector of singles (VM)
ST 7E50	A-124	max.w VkMaximum of a vector of words
E0 7E50	A-125	max.w.f VkMaximum of vector of words (!VM)
E1 7E50	A-125	max.w.t VkMaximum of vector of words (VM)
ST 8800	A-127	merg.f Vi,Sj,VkMerge vector/scalar (!VM)
ST 8C00	A-127	merg.t Vi,Sj,VkMerge vector/scalar
ST 8400	A-128	merg.t Vi,Vj,VkMerge vector/vector
ST 7E60	A-129	min.b VkMinimum of a vector of bytes
E0 7E60	A-130	min.b.f VkMinimum of vector of bytes (!VM)
E1 7E60	A-130	min.b.t VkMinimum of vector of bytes (VM)
ST 7EB8	A-129	min.d VkMinimum of a vector of double float
E0 7EB8	A-130	min.d.f VkMinimum of vector of doubles (!VM)
E1 7EB8	A-130	min.d.t VkMinimum of vector of doubles (VM)
ST 7E68	A-129	min.h VkMinimum of a vector of halfwords
E0 7E68	A-130	min.h.f VkMinimum of vector of halfwords (!VM)
E1 7E68	A-130	min.h.t VkMinimum of vector of halfwords (VM)
ST 7E78	A-129	min.l VkMinimum of a vector of longwords
E0 7E78	A-130	min.l.f VkMinimum of vector of longwords (!VM)
E1 7E78	A-130	min.l.t VkMinimum of vector of longwords (VM)
ST 7EB0	A-129	min.s VkMinimum of a vector of single float
E0 7EB0	A-130	min.s.f VkMinimum of vector of singles (!VM)
E1 7EB0	A-130	min.s.t VkMinimum of vector of singles (VM)
ST 7E70	A-129	min.w VkMinimum of a vector of words
E0 7E70	A-130	min.w.f VkMinimum of vector of words (!VM)
E1 7E70	A-130	min.w.t VkMinimum of vector of words (VM)
ST 5080	A-132	mov Aj,Ak Move address register
ST 51C0	A-134	mov Aj,SkMove an address to a scalar
ST 7C48	A-133	mov Ak,PSWLoad an address register into the PSW
ST 7D98	A-135	mov Ak,VL Move Ak to VL
ST 7D88	A-136	mov Ak,VSMove Ak to VS
E0 7C08	A-137	mov CIR,SkMove CIR a scalar
E0 7C18	A-138	mov CPUID,SkMove CPU identification to scalar
E0 7C68	A-139	mov ICR,SkMove ICR to scalar
ST 7C60	A-140	mov ITR,SkMove the ITC, ITSR, NITC into Sk
ST 7C50	A-141	mov PC,AkLoad the next PC address
ST 7C40	A-142	mov PSW,AkStore the PSW into an address register
ST 8200	A-153	mov Si,Sj,VkMove a scalar to a vector element
ST 50C0	A-144	mov Sj,AkMove 32 bits of Sj into Ak
ST 6100	A-149	mov Sj,Sk,VMLoad VM(Sj) from Sk
ST 6140	A-156	mov Sj,VM,SkLoad Sk from VM(Sj)
E0 7C00	A-145	mov Sk,CIRMove scalar to CIR
E0 7C60	A-146	mov Sk,ICRMove Scalar to ICR
ST 7C68	A-147	mov Sk,ITRLoad NITC, ITC, ITSR from Sk
ST 7C78	A-148	mov Sk,ITSRLoad ITSR with a scalar
E0 7C70	A-150	mov Sk,TCPUMove Scalar to TCPU register
E0 7CB0	A-151	mov Sk,TIDLoad TID from scalar
E0 7C20	A-152	mov Sk,TTRMove scalar to TTR
E0 7C50	A-155	mov Sk,VMLLoad VM<63..0> from Sk
E0 7C40	A-157	mov Sk,VMULoad VM<127..64> from Sk
ST 7D70	A-159	mov Sk,VVMove scalar to vector valid flag
E0 7C78	A-160	mov TCPU,SkMove the TCPU Register to Scalar
E0 7CB8	A-161	mov TID,SkLoad scalar with TID
E0 7C10	A-162	mov TOC,SkMove TOC to a scalar
E0 7C28	A-163	mov TTR,SkMove TTR /scalar
ST 8000	A-168	mov Vi,Sj,SkMove a vector element to a scalar
ST 7D90	A-164	mov VL,Ak Move VL to Ak
E0 7C58	A-166	mov VML,SkLoad Sk from VM<63..0>
E0 7C48	A-167	mov VMU,SkLoad Sk from VM<127..64>
ST 7D80	A-169	mov VS,AkMove VS to Ak

ST 5180	A-143	mov.d Sj,SkMove scalar register single float
ST 5180	A-143	mov.l Sj,SkMove scalar register longword
ST 5100	A-143	mov.s Sj,SkMove scalar register double float
ST 5100	A-143	mov.w Sj,SkMove scalar register word
ST 7DB8	A-154	mov.w Sk,VLMove Sk to VL
ST 7DA8	A-158	mov.w Sk,VSMove Sk to VS
ST 7DB0	A-165	mov.w VL,SkMove VL to Sk
ST 7DA8	A-170	mov.w VS,SkMove VS to Sk
ST 7D60	A-171	mski SkMask out interrupt
ST 7D58	A-172	msyncSynchronize stores to memory
ST 5D00	A-176	mul.b Sj,SkMultiply scalar/scalar integer byte
ST E800	A-177	mul.b Vi,Sj,VkMultiply vector/scalar integer byte
ST E000	A-179	mul.b Vi,Vj,VkMultiply vector/vector integer byte
E0 E800	A-178	mul.b.f Vi,Sj,VkMultiply vector/scalar byte (!VM)
E0 E000	A-180	mul.b.f Vi,Vj,VkMultiply byte vectors (!VM)
E1 E800	A-178	mul.b.t Vi,Sj,VkMultiply vector/scalar byte (VM)
E1 E000	A-180	mul.b.t Vi,Vj,VkMultiply byte vectors (VM)
ST 5740	A-176	mul.d Sj,SkMultiply scalar/scalar double float
ST 9A00	A-177	mul.d Vi,Sj,VkMultiply vector/scalar double float
ST 9200	A-179	mul.d Vi,Vj,VkMultiply vector/vector double float
E0 9A00	A-178	mul.d.f Vi,Sj,VkMultiply vector/scalar double (!VM)
E0 9200	A-180	mul.d.f Vi,Vj,VkMultiply double vectors (!VM)
E1 9A00	A-178	mul.d.t Vi,Sj,VkMultiply vector/scalar double (VM)
E1 9200	A-180	mul.d.t Vi,Vj,VkMultiply double vectors (VM)
ST 1600	A-173	mul.h #N,AkMultiply immediate address halfword
ST 5C80	A-173	mul.h #n,AkMultiply short immediate address halfword
ST 1608	A-174	mul.h #N,SkMultiply scalar/immediate integer halfword
ST 5C00	A-175	mul.h Aj,AkMultiply address register halfword
ST 5D40	A-176	mul.h Sj,SkMultiply scalar/scalar integer halfword
ST EA00	A-177	mul.h Vi,Sj,VkMultiply vector/scalar integer halfword
ST E200	A-179	mul.h Vi,Vj,VkMultiply vector/vector integer halfword
E0 EA00	A-178	mul.h.f Vi,Sj,VkMultiply vector/scalar halfword (!VM)
E0 E200	A-180	mul.h.f Vi,Vj,VkMultiply halfword vectors (!VM)
E1 EA00	A-178	mul.h.t Vi,Sj,VkMultiply vector/scalar halfword (VM)
E1 E200	A-180	mul.h.t Vi,Vj,VkMultiply halfword vectors (VM)
ST 5DC0	A-176	mul.l Sj,SkMultiply scalar/scalar integer longword
ST EE00	A-177	mul.l Vi,Sj,VkMultiply vector/scalar integer longword
ST E600	A-179	mul.l Vi,Vj,VkMultiply vector/vector integer longword
E0 EE00	A-178	mul.l.f Vi,Sj,VkMultiply vector/scalar longword (!VM)
E0 E600	A-180	mul.l.f Vi,Vj,VkMultiply longword vectors (!VM)
E1 EE00	A-178	mul.l.t Vi,Sj,VkMultiply vector/scalar longword (VM)
E1 E600	A-180	mul.l.t Vi,Vj,VkMultiply longword vectors (VM)
ST 1908	A-174	mul.s #N,SkMultiply scalar/immediate single float
ST 5700	A-176	mul.s Sj,SkMultiply scalar/scalar single float
ST 9800	A-177	mul.s Vi,Sj,VkMultiply vector/scalar single float
ST 9000	A-179	mul.s Vi,Vj,VkMultiply vector/vector single float
E0 9800	A-178	mul.s.f Vi,Sj,VkMultiply vector/scalar single (!VM)
E0 9000	A-180	mul.s.f Vi,Vj,VkMultiply single vectors (!VM)
E1 9800	A-178	mul.s.t Vi,Sj,VkMultiply vector/scalar single (VM)
E1 9000	A-180	mul.s.t Vi,Vj,VkMultiply single vectors (VM)
ST 1680	A-173	mul.w #N,AkMultiply immediate address word
ST 5CC0	A-173	mul.w #n,AkMultiply short immediate address word
ST 1688	A-174	mul.w #N,SkMultiply scalar/immediate integer word
ST 5C40	A-175	mul.w Aj,AkMultiply address register word
ST 5D80	A-176	mul.w Sj,SkMultiply scalar/scalar integer word
ST EC00	A-177	mul.w Vi,Sj,VkMultiply vector/scalar integer word
ST E400	A-179	mul.w Vi,Vj,VkMultiply vector/vector integer word
E0 EC00	A-178	mul.w.f Vi,Sj,VkMultiply vector/scalar word (!VM)
E0 E400	A-180	mul.w.f Vi,Vj,VkMultiply word vectors (!VM)

E1 EC00	A-178	mul.w.t Vi,Sj,VkMultiply vector/scalar word (VM)
E1 E400	A-180	mul.w.t Vi,Vj,VkMultiply word vectors (VM)
ST 6F00	A-182	neg.b Sj,SkNegate scalar/scalar integer byte
ST 6E00	A-183	neg.b Vj,VkNegate vector/vector integer byte
E0 6E00	A-184	neg.b.f Vj,VkNegate byte vector (!VM)
E1 6E00	A-184	neg.b.t Vj,VkNegate byte vector (VM)
ST 65C0	A-182	neg.d Sj,SkNegate scalar/scalar double float
ST 64C0	A-183	neg.d Vj,VkNegate vector/vector double float
E0 64C0	A-184	neg.d.f Vj,VkNegate double (!VM)
E1 64C0	A-184	neg.d.t Vj,VkNegate double (VM)
ST 5680	A-181	neg.h Aj,AkNegate address register halfword
ST 6F40	A-182	neg.h Sj,SkNegate scalar/scalar integer halfword
ST 6E40	A-183	neg.h Vj,VkNegate vector/vector integer halfword
E0 6E40	A-184	neg.h.f Vj,VkNegate halfword (!VM)
E1 6E40	A-184	neg.h.t Vj,VkNegate halfword (VM)
ST 6FC0	A-182	neg.l Sj,SkNegate scalar/scalar integer longword
ST 6EC0	A-183	neg.l Vj,VkNegate vector/vector integer longword
E0 6EC0	A-184	neg.l.f Vj,VkNegate longword (!VM)
E1 6EC0	A-184	neg.l.t Vj,VkNegate longword (VM)
ST 6580	A-182	neg.s Sj,SkNegate scalar/scalar single float
ST 6480	A-183	neg.s Vj,VkNegate vector/vector single float
E0 6480	A-184	neg.s.f Vj,VkNegate single (!VM)
E1 6480	A-184	neg.s.t Vj,VkNegate single (VM)
ST 56C0	A-181	neg.w Aj,AkNegate address register word
ST 6F80	A-182	neg.w Sj,SkNegate scalar/scalar integer word
ST 6E80	A-183	neg.w Vj,VkNegate vector/vector integer word
E0 6E80	A-184	neg.w.f Vj,VkNegate word (!VM)
E1 6E80	A-184	neg.w.t Vj,VkNegate word (VM)
ST 7000	A-185	nopNo operation (branch never)
ST 52C0	A-186	not Aj,AkComplement address register
ST 53C0	A-187	not Sj,SkComplement scalar/scalar
ST 62C0	A-188	not Vj,VkComplement a vector
E0 62C0	A-189	not.f Vj,VkComplement a vector (!VM)
E1 62C0	A-189	not.t Vj,VkComplement a vector (VM)
ST 1280	A-190	or #N,AkOR immediate to address register
ST 1288	A-191	or #N,SkOR scalar/immediate
ST 5240	A-192	or Aj,AkOR address register
ST 5340	A-193	or Sj,SkOR scalar/scalar
ST AA00	A-194	or Vi,Sj,VkOR vector/scalar
ST A200	A-196	or Vi,Vj,VkOR two vectors
E0 AA00	A-195	or.f Vi,Sj,VkOR vector/scalar (!VM)
E0 A200	A-197	or.f Vi,Vj,VkOR two vectors (!VM)
E1 AA00	A-195	or.t Vi,Sj,VkOR vector/scalar (VM)
E1 A200	A-197	or.t Vi,Vj,VkOR two vectors (VM)
ST 7E30	A-198	parity VkExclusive OR reduce a vector
E0 7E30	A-199	parity.f VkExclusive OR reduce vector (!VM)
E1 7E30	A-199	parity.t VkExclusive OR reduce vector (VM)
ST 7C28	A-200	pate Ak Purge ATU entry
ST 7C20	A-201	patuPurge the entire ATU
ST 7DC8	A-202	pbkptForce process breakpoint exception
ST 2300	A-203	pfork <effa>,AkPost a fork
ST 7C30	A-204	pichPurge the Icache
ST 7EE0	A-208	plc.f VM,SkLoad the number of 0's in VM into Sk
ST 4580	A-205	plc.t Sj,SkCount the number of 1's in Sj
ST 6340	A-206	plc.t Vj,VkPopulation count of a vector
ST 7EE8	A-208	plc.t VM,SkLoad the number of 1's in VM into Sk
E0 6340	A-207	plc.t.f Vj,VkPopulation count of vector (!VM)
E1 6340	A-207	plc.t.t Vj,VkPopulation count of vector (VM)
ST 7C38	A-209	plchPurge the Lcache

ST 7D38	A-211	pop.l SkPop Sk <63..0> from the stack
ST 7D10	A-210	pop.w AkPop word into address register
ST 7D30	A-211	pop.w SkPop Sk <31..0> from the stack
E0 0800	A-212	popr Ak,<effa>Pop resource/address register
ST 7EC0	A-213	prod.b VkMultiply reduce a vector of bytes
E0 7EC0	A-214	prod.b.f VkMultiply reduce byte vector (!VM)
E1 7EC0	A-214	prod.b.t VkMultiply reduce byte vector (VM)
ST 7E98	A-213	prod.d VkMultiply reduce a vector of double float
E0 7E98	A-214	prod.d.f VkMultiply reduce double vector (!VM)
E1 7E98	A-214	prod.d.t VkMultiply reduce double vector (VM)
ST 7EC8	A-213	prod.h VkMultiply reduce a vector of halfwords
E0 7EC8	A-214	prod.h.f VkMultiply reduce halfword vector (!VM)
E1 7EC8	A-214	prod.h.t VkMultiply reduce halfword vector (VM)
ST 7ED8	A-213	prod.l VkMultiply reduce a vector of longwords
E0 7ED8	A-214	prod.l.f VkMultiply reduce longword vector (!VM)
E1 7ED8	A-214	prod.l.t VkMultiply reduce longword vector (VM)
ST 7E90	A-213	prod.s VkMultiply reduce a vector of single float
E0 7E90	A-214	prod.s.f VkMultiply reduce single vector (!VM)
E1 7E90	A-214	prod.s.t VkMultiply reduce single vector (VM)
ST 7ED0	A-213	prod.w VkMultiply reduce a vector of words
E0 7ED0	A-214	prod.w.f VkMultiply reduce word vector (!VM)
E1 7ED0	A-214	prod.w.t VkMultiply reduce word vector (VM)
ST 7D28	A-217	psh.l SkPush Sk<63..0> onto the stack
ST 7D00	A-216	psh.w AkPush an address register
ST 7D20	A-217	psh.w SkPush Sk<31..0> onto the stack
ST 0D00	A-218	pshea <effa>Push effective address
E0 0900	A-219	pshr Ak,<effa>Push address register/resource
E0 3600	A-221	put.l Sk,<Ceffa>Put scalar/communication
E0 2E00	A-220	put.w Ak,<Ceffa>Put address/communication
E0 3300	A-223	rcv.l <Ceffa>,SkReceive communication/scalar
E0 2b00	A-222	rcv.w <Ceffa>,AkReceive communication/address
E0 0E00	A-225	rcvr.l <effa>,SkReceive scalar register/resource
E0 0A00	A-224	rcvr.w <effa>,AkReceive address register/resource
ST 7C90	A-226	rtnReturn from subroutine call
ST 7CA8	A-228	rtncReturn from a context block
ST 7C80	A-229	rtnqPop the PC and jump
ST 1380	A-230	shf #N,AkLogical shift immediate
ST 4440	A-230	shf #n,AkLogical shift short immediate
ST 1388	A-231	shf #N,SkShift scalar/immediate
ST 5040	A-233	shf Aj,AkShift an address
ST 5140	A-234	shf Sj,SkShift a scalar
ST 6300	A-236	shf Sj,VkShift a vector accumulator
ST AE00	A-238	shf Vi,Sj,VkShift a vector accumulator
ST A600	A-240	shf Vi,Vj,VkShift vector/vector
E0 6300	A-237	shf.f Sj,VkShift vector/scalar (!VM)
E0 AE00	A-239	shf.f Vi,Sj,VkShift vector/scalar (!VM)
E0 A600	A-241	shf.f Vi,Vj,VkShift vector/vector (!VM)
E1 6300	A-237	shf.t Sj,VkShift vector/scalar (VM)
E1 AE00	A-239	shf.t Vi,Sj,VkShift vector/scalar (VM)
E1 A600	A-241	shf.t Vi,Vj,VkShift vector/vector (VM)
ST 1980	A-232	shf.w #N,SkShift scalar word/immediate
E0 4440	A-235	shf.w Sj,SkShift a scalar word
ST 7CC8	A-242	sin.d SkSine of a double precision number
ST 7CC0	A-242	sin.s SkSine of a single precision number
E0 3700	A-244	snd.l Sk,<Ceffa>Send scalar/communication
E0 2F00	A-243	snd.w Ak,<Ceffa>Send address/communication
E0 0D00	A-246	sndr.l Sk,<effa>Send scalar register/resource
E0 0C00	A-245	sndr.w Ak,<effa>Send address register/resource
E0 0500	A-247	spawn <effa>,AkSpawn a fork

ST 7DD8	A-248	sqrt.d SkSquare root of a double float
E0 5D40	A-249	sqrt.d Vj,VkSquare root double vector/vector
E0 5F40	A-250	sqrt.d.f Vj,VkSquare root double (!VM)
E1 5F40	A-250	sqrt.d.t Vj,VkSquare root double (VM)
ST 7DD0	A-248	sqrt.s SkSquare root of a single float
E0 5D00	A-249	sqrt.s Vj,VkSquare root single vector/vector
E0 5F00	A-250	sqrt.s.f Vj,VkSquare root single (!VM)
E1 5F00	A-250	sqrt.s.t Vj,VkSquare root single (VM)
ST 2C00	A-251	st.b Ak,<effa>Store address register byte
ST 3400	A-252	st.b Sk,<effa>Store scalar byte
ST 3C00	A-253	st.b Vk,<effa>Store vector byte
E0 3C00	A-254	st.b.f Vk,<effa>Store vector byte (!VM)
E1 3C00	A-254	st.b.t Vk,<effa>Store vector byte (VM)
ST 3700	A-252	st.d Sk,<effa>Store scalar double float
ST 3F00	A-253	st.d Vk,<effa>Store vector double float
E0 3F00	A-254	st.d.f Vk,<effa>Store vector double float (!VM)
E1 3F00	A-254	st.d.t Vk,<effa>Store vector double float (VM)
ST 2D00	A-251	st.h Ak,<effa>Store address register halfword
ST 3500	A-252	st.h Sk,<effa>Store scalar halfword
ST 3D00	A-253	st.h Vk,<effa>Store vector halfword
E0 3D00	A-254	st.h.f Vk,<effa>Store vector halfword (!VM)
E1 3D00	A-254	st.h.t Vk,<effa>Store vector halfword (VM)
ST 3700	A-252	st.l Sk,<effa>Store scalar longword
ST 3F00	A-253	st.l Vk,<effa>Store vector longword
ST 0E00	A-256	st.l VLS,<effa>Store VS and VL to memory
E0 3F00	A-254	st.l.f Vk,<effa>Store vector longword (!VM)
E1 3F00	A-254	st.l.t Vk,<effa>Store vector longword (VM)
ST 3600	A-252	st.s Sk,<effa>Store scalar single float
ST 3E00	A-253	st.s Vk,<effa>Store vector single float
E0 3E00	A-254	st.s.f Vk,<effa>Store vector single float (!VM)
E1 3E00	A-254	st.s.t Vk,<effa>Store vector single float (VM)
ST 2E00	A-251	st.w Ak,<effa>Store address register word
ST 3600	A-252	st.w Sk,<effa>Store scalar word
ST 3E00	A-253	st.w Vk,<effa>Store vector word
E0 3E00	A-254	st.w.f Vk,<effa>Store vector word (!VM)
E1 3E00	A-254	st.w.t Vk,<effa>Store vector word (VM)
ST 0F00	A-257	st.x VM,<effa>Store VM into memory
E0 0700	A-258	stcmr Ak,<effa>Store communication registers
ST 2400	A-260	ste.b Sk,<effa>Store an extended scalar byte
E0 2400	A-261	ste.b.f Sk,<effa>Store extended scalar byte (!VM)
E1 2400	A-261	ste.b.t Sk,<effa>Store extended scalar byte (VM)
ST 2700	A-260	ste.d Sk,<effa>Store an extended scalar double float
E0 2700	A-261	ste.d.f Sk,<effa>Store extended scalar double (!VM)
E1 2700	A-261	ste.d.t Sk,<effa>Store extended scalar double (VM)
ST 2500	A-260	ste.h Sk,<effa>Store an extended scalar halfword
E0 2500	A-261	ste.h.f Sk,<effa>Store extended scalar halfword (!VM)
E1 2500	A-261	ste.h.t Sk,<effa>Store extended scalar halfword (VM)
ST 2700	A-260	ste.l Sk,<effa>Store an extended scalar longword
E0 2700	A-261	ste.l.f Sk,<effa>Store extended scalar longword (!VM)
E1 2700	A-261	ste.l.t Sk,<effa>Store extended scalar longword (VM)
ST 2600	A-260	ste.s Sk,<effa>Store an extended scalar single float
E0 2600	A-261	ste.s.f Sk,<effa>Store extended scalar single (!VM)
E1 2600	A-261	ste.s.t Sk,<effa>Store extended scalar single (VM)
ST 2600	A-260	ste.w Sk,<effa>Store an extended scalar word
E0 2600	A-261	ste.w.f Sk,<effa>Store extended scalar word (!VM)
E1 2600	A-261	ste.w.t Sk,<effa>Store extended scalar word (VM)
ST 7B00	A-263	stvi.b Sk,VjScalar index store vector byte
ST 7A00	A-266	stvi.b Vk,VjIndex store vector byte
E0 7B00	A-264	stvi.b.f Sk,VjScalar index store vector byte (!VM)

E0 7A00	A-267	stvi.b.f Vk,VjIndex store vector byte (!VM)
E1 7B00	A-264	stvi.b.t Sk,VjScalar index store vector byte (VM)
E1 7A00	A-267	stvi.b.t Vk,VjIndex store vector byte (VM)
ST 7BC0	A-263	stvi.d Sk,VjScalar index store vector double float
ST 7AC0	A-266	stvi.d Vk,VjIndex store vector double
E0 7BC0	A-264	stvi.d.f Sk,VjScalar index store vector double (!VM)
E0 7AC0	A-267	stvi.d.f Vk,VjIndex store vector double (!VM)
E1 7BC0	A-264	stvi.d.t Sk,VjScalar index store vector double (VM)
E1 7AC0	A-267	stvi.d.t Vk,VjIndex store vector double (VM)
ST 7B40	A-263	stvi.h Sk,VjScalar index store vector halfword
ST 7A40	A-266	stvi.h Vk,VjIndex store vector halfword
E0 7B40	A-264	stvi.h.f Sk,VjScalar index store vector half (!VM)
E0 7A40	A-267	stvi.h.f Vk,VjIndex store vector halfword (!VM)
E1 7B40	A-264	stvi.h.t Sk,VjScalar index store vector half (VM)
E1 7A40	A-267	stvi.h.t Vk,VjIndex store vector halfword (VM)
ST 7BC0	A-263	stvi.l Sk,VjScalar index store vector longword
ST 7AC0	A-266	stvi.l Vk,VjIndex store vector longword
E0 7BC0	A-264	stvi.l.f Sk,VjScalar index store vector long (!VM)
E0 7AC0	A-267	stvi.l.f Vk,VjIndex store vector longword (!VM)
E1 7BC0	A-264	stvi.l.t Sk,VjScalar index store vector long (VM)
E1 7AC0	A-267	stvi.l.t Vk,VjIndex store vector longword (VM)
ST 7B80	A-263	stvi.s Sk,VjScalar index store vector single float
ST 7A80	A-266	stvi.s Vk,VjIndex store vector single
E0 7B80	A-264	stvi.s.f Sk,VjScalar index store vector single (!VM)
E0 7A80	A-267	stvi.s.f Vk,VjIndex store vector single (!VM)
E1 7B80	A-264	stvi.s.t Sk,VjScalar index store vector single (VM)
E1 7A80	A-267	stvi.s.t Vk,VjIndex store vector single (VM)
ST 7B80	A-263	stvi.w Sk,VjScalar index store vector word
ST 7A80	A-266	stvi.w Vk,VjIndex store vector word
E0 7B80	A-264	stvi.w.f Sk,VjScalar index store vector word (!VM)
E0 7A80	A-267	stvi.w.f Vk,VjIndex store vector word (!VM)
E1 7B80	A-264	stvi.w.t Sk,VjScalar index store vector word (VM)
E1 7A80	A-267	stvi.w.t Vk,VjIndex store vector word (VM)
ST 5B00	A-272	sub.b Sj,SkSubtract scalar/scalar integer byte
ST D800	A-275	sub.b Vi,Sj,VkSubtract vector/scalar integer byte
ST D000	A-277	sub.b Vi,Vj,VkSubtract vector/vector integer byte
E0 D800	A-276	sub.b.f Vi,Sj,VkSubtract vector/scalar byte (!VM)
E0 D000	A-278	sub.b.f Vi,Vj,VkSubtract byte vectors (!VM)
E1 D800	A-276	sub.b.t Vi,Sj,VkSubtract vector/scalar byte (VM)
E1 D000	A-278	sub.b.t Vi,Vj,VkSubtract byte vectors (VM)
E0 8200	A-273	sub.d Si.Vj,VkSubtract scalar/vector double float
ST 55C0	A-272	sub.d Sj,SkSubtract scalar/scalar double float
ST BE00	A-275	sub.d Vi,Sj,VkSubtract vector/scalar double float
ST B600	A-277	sub.d Vi,Vj,VkSubtract vector/vector double float
E0 8A00	A-274	sub.d.f Si,Vj,VkSubtract scalar/vector double (!VM)
E0 BE00	A-276	sub.d.f Vi,Sj,VkSubtract vector/scalar double (!VM)
E0 B600	A-278	sub.d.f Vi,Vj,VkSubtract double vectors (!VM)
E1 8A00	A-274	sub.d.t Si,Vj,VkSubtract scalar/vector double (VM)
E1 BE00	A-276	sub.d.t Vi,Sj,VkSubtract vector/scalar double (VM)
E1 B600	A-278	sub.d.t Vi,Vj,VkSubtract double vectors (VM)
ST 1500	A-269	sub.h #N,AkSubtract immediate address halfword
ST 5A80	A-269	sub.h #n,AkSubtract short immediate address halfword
ST 1508	A-270	sub.h #N,SkSubtract scalar/immediate integer halfword
ST 5A00	A-271	sub.h Aj,AkSubtract address register halfword
ST 5B40	A-272	sub.h Sj,SkSubtract scalar/scalar integer halfword
ST DA00	A-275	sub.h Vi,Sj,VkSubtract vector/scalar integer halfword
ST D200	A-277	sub.h Vi,Vj,VkSubtract vector/vector integer halfword
E0 DA00	A-276	sub.h.f Vi,Sj,VkSubtract vector/scalar halfword (!VM)
E0 D200	A-278	sub.h.f Vi,Vj,VkSubtract halfword vectors (!VM)

E1 DA00	A-276	sub.h.t Vi,Sj,VkSubtract vector/scalar halfword (VM)
E1 D200	A-278	sub.h.t Vi,Vj,VkSubtract halfword vectors (VM)
ST 5BC0	A-272	sub.l Sj,SkSubtract scalar/scalar integer longword
ST DE00	A-275	sub.l Vi,Sj,VkSubtract vector/scalar integer longword
ST D600	A-277	sub.l Vi,Vj,VkSubtract vector/vector integer longword
E0 DE00	A-276	sub.l.f Vi,Sj,VkSubtract vector/scalar longword (!VM)
E0 D600	A-278	sub.l.f Vi,Vj,VkSubtract longword vectors (!VM)
E1 DE00	A-276	sub.l.t Vi,Sj,VkSubtract vector/scalar longword (VM)
E1 D600	A-278	sub.l.t Vi,Vj,VkSubtract longword vectors (VM)
ST 1888	A-270	sub.s #N,SkSubtract scalar/immediate single float
E0 8000	A-273	sub.s Si,Vj,VkSubtract scalar/vector single float
ST 5580	A-272	sub.s Sj,SkSubtract scalar/scalar single float
ST BC00	A-275	sub.s Vi,Sj,VkSubtract vector/scalar single float
ST B400	A-277	sub.s Vi,Vj,VkSubtract vector/vector single float
E0 8800	A-274	sub.s.f Si,Vj,VkSubtract scalar/vector single (!VM)
E0 BC00	A-276	sub.s.f Vi,Sj,VkSubtract vector/scalar single (!VM)
E0 B400	A-278	sub.s.f Vi,Vj,VkSubtract single vectors (!VM)
E1 8800	A-274	sub.s.t Si,Vj,VkSubtract scalar/vector single (VM)
E1 BC00	A-276	sub.s.t Vi,Sj,VkSubtract vector/scalar single (VM)
E1 B400	A-278	sub.s.t Vi,Vj,VkSubtract single vectors (VM)
ST 1580	A-269	sub.w #N,AkSubtract immediate address word
ST 5AC0	A-269	sub.w #n,AkSubtract short immediate address word
ST 1588	A-270	sub.w #N,SkSubtract scalar/immediate integer word
ST 5A40	A-271	sub.w Aj,AkSubtract address register word
ST 5B80	A-272	sub.w Sj,SkSubtract scalar/scalar integer word
ST DC00	A-275	sub.w Vi,Sj,VkSubtract vector/scalar integer word
ST D400	A-277	sub.w Vi,Vj,VkSubtract vector/vector integer word
E0 DC00	A-276	sub.w.f Vi,Sj,VkSubtract vector/scalar word (!VM)
E0 D400	A-278	sub.w.f Vi,Vj,VkSubtract word vectors (!VM)
E1 DC00	A-276	sub.w.t Vi,Sj,VkSubtract vector/scalar word (VM)
E1 D400	A-278	sub.w.t Vi,Vj,VkSubtract word vectors (VM)
ST 7E00	A-279	sum.b VkSum a vector of bytes
E0 7E00	A-280	sum.b.f VkSum a vector of bytes (!VM)
E1 7E00	A-280	sum.b.t VkSum a vector of bytes (VM)
ST 7E88	A-279	sum.d VkSum a vector of double float
E0 7E88	A-280	sum.d.f VkSum a vector of double (!VM)
E1 7E88	A-280	sum.d.t VkSum a vector of double (VM)
ST 7E08	A-279	sum.h VkSum a vector of halfwords
E0 7E08	A-280	sum.h.f VkSum a vector of halfwords (!VM)
E1 7E08	A-280	sum.h.t VkSum a vector of halfwords (VM)
ST 7E18	A-279	sum.l VkSum a vector of longwords
E0 7E18	A-280	sum.l.f VkSum a vector of longwords (!VM)
E1 7E18	A-280	sum.l.t VkSum a vector of longwords (VM)
ST 7E80	A-279	sum.s VkSum a vector of single float
E0 7E80	A-280	sum.s.f VkSum a vector of single (!VM)
E1 7E80	A-280	sum.s.t VkSum a vector of single (VM)
ST 7E10	A-279	sum.w VkSum a vector of words
E0 7E10	A-280	sum.w.f VkSum a vector of words (!VM)
E1 7E10	A-280	sum.w.t VkSum a vector of words (VM)
ST 1080	A-282	sysc #r,#gPerform a system call
ST 0800	A-284	tac <effa>Test and clear a byte in memory
ST 0C00	A-285	tas <effa>Test and set a memory byte
ST 1A00	A-286	trap #rm,#bForce a trap system exception
E0 0100	A-288	tst <Ceffa>Test communication register lock bit
ST 7D78	A-289	tstvvTest value of vector valid flag
ST 45C0	A-290	tzc Sj,SkCount of trailing zeros in Sj
ST 6200	A-291	tzc Vj,VkTrailing zero count vector
E0 6200	A-292	tzc.f Vj,VkTrailing zero count vector (!VM)
E1 6200	A-292	tzc.t Vj,VkTrailing zero count vector (VM)

E0 0300	A-293	ulk <Ceffa>Unlock communication register
ST 7C98	A-294	wforkWait for a fork
ST 7D68	A-295	xmti SkTransmit interrupt
ST 1300	A-296	xor #N,AkExclusive OR immediate to address register
ST 1308	A-297	xor #N,SkExclusive OR scalar/immediate
ST 5280	A-298	xor Aj,AkExclusive OR address register
ST 5380	A-299	xor Sj,SkExclusive OR scalar/scalar
ST AC00	A-300	xor Vi,Sj,VkExclusive OR vector/scalar
ST A400	A-302	xor Vi,Vj,VkExclusive OR two vectors
E0 AC00	A-301	xor.f Vi,Sj,VkExclusive OR vector/scalar (!VM)
E0 A400	A-303	xor.f Vi,Vj,VkExclusive OR two vectors (!VM)
E1 AC00	A-301	xor.t Vi,Sj,VkExclusive OR vector/scalar (VM)
E1 A400	A-303	xor.t Vi,Vj,VkExclusive OR two vectors (VM)
E0 6280	A-304	xpnd.f Vj,VkExpand a vector (!VM)
E1 6280	A-304	xpnd.t Vj,VkExpand a vector (VM)

C

Reporting Problems

C.1 Introduction

The *contact* utility is the recommended way to report software and documentation problems to the Technical Assistance Center (TAC). It is an interactive tool that prompts you for the information necessary to report a problem to the TAC.

You must have a UNIX-to-UNIX Communications Protocol (UUCP) connection to the TAC to use *contact*. A UUCP system allows communication between UNIX systems by either dial-up or hard-wired communication lines. See *uucp*(1) or the entry in *info*(1) (online information system) for more information.

You must know the name and version number of the product involved. If you do not know the version number of the program or utility you are having trouble with, use the *vers* command. The syntax for the command is

vers *filename*

where *filename* is the the full pathname of the program. If you don't know the full pathname of the program, type

which *program*

For more information on these commands, see *vers*(1) and *which*(1) in the *CONVEX UNIX Programmer's Manual*, Part I.

C.2 Information Required to Report a Problem

contact requires the following information:

1. Your name, title, phone number, and corporate name.

2. The name and version of the product involved. Use the *vers* command if you don't know the version number of the program or utility.

3. A short (1 line) summary of the problem.

4. A detailed description of the problem. Include source code and a stack backtrace whenever possible. (See *adb*(1) or *csd*(1) for information on obtaining stack backtraces.) The more information provided, the quicker your problem can be isolated and solved.

5. The priority of the problem. You are shown a list of six levels from which to select.

6. Instructions on how to reproduce the problem, including the command syntax used, any flags invoked, or anything else you attempted to make your program run.

7. Any other comments about the problem or files you wish to submit.

You will have a chance to review your report before you submit it. You can edit the report if you find an error in what you have typed. If you change your mind and don't want to submit the report, you can abort the *contact* session; the file is saved in your home directory in a file named *dead.report*.

The following figure is a sample *contact* session. User input is in bold lettering, and the system response is in constant-width lettering.

Figure C–1: Sample *contact* Session

```
%contact (RETURN)
Welcome to contact version 0.14 ()

Enter your name, title, phone number, and corporate name (^D to terminate)
> Margaret Atwood, systems programmer, 814-4444, University (RETURN)
> of Chicago (RETURN)
> (CTRL-D)

Enter the name of the product involved
> CONVEX UNIX Programmer's Manual, Part I (RETURN)

Enter the version number (in the form X.X or X.X.X.X) of the product
> Revision 4.0 (RETURN)

Enter a short (1 line) summary of the problem
> The finger command manual page lists nonexistent bug (RETURN)

Enter a detailed description of the problem (^D to terminate)
> The finger(1) man page says, under the BUGS section, that "Only the first
line of the .project file is printed."  Happily, this is not true! (RETURN)
> (CTRL-D)

Enter a problem priority, based on the following:
1) Critical    - work cannot proceed until the problem is resolved.
2) Serious     - work can proceed around the problem, with difficulty.
3) Necessary   - problem has to be fixed.
4) Annoying    - problem is bothersome.
5) Enhancement - requested enhancement.
6) Informative - for informational purposes only.
> 4 (RETURN)

Enter the instructions by which the problem may be reproduced (^D to terminate)
> a) put more than one line in .project (RETURN)
> b) read the man page for finger(1) (RETURN)
> (CTRL-D)

Enter any comments that are applicable (^D to terminate) (RETURN)
> (CTRL-D)

Do you have any suggestions or comments on the documentation that you
referenced when you were trying to resolve your problem (for example,
additions, corrections organization, accessibility)? (^D to terminate)
> The man page should be updated. (RETURN)
> (CTRL-D)

Are there any files that should be included in this report (yes | no)?
> no (RETURN)

Please select one of the following options:
1) Review the problem report.
2) Edit the problem report.
3) Submit the problem report.
4) Abort the problem report.
> 3 (RETURN)

Problem report submitted.
%
```

Index

Software Documentation

Index Enhancements

So that we can continue to provide better indexing in CONVEX documentation, please keep track of the words or phrases you look up in an index, but don't find. Then, list under which index entry you ultimately found the information you were seeking. You can mail one of these postage-paid forms to the CONVEX Software Documentation Department monthly, or you can submit the information to the Technical Assistance Center in the form of a bug report. You can get more forms by writing to CONVEX at the address below, or by calling us. You can also photocopy this form and mail it back in an envelope. Thank you for helping us to serve you better.

Name:_____ **Company:**_____

Phone:_____ **Date:**_____

Manual Title/Rev. No.	Looked Up This Word	Found Information Under This Word

CONVEX Computer Corporation — P.O. Box 833851, Richardson, TX 75083-3851 (214)952-0200

(Fold Here First)

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1046 RICHARDSON, TEXAS

POSTAGE WILL BE PAID BY ADDRESSEE

CUSTOMER SERVICE
CONVEX Computer Corp.
P.O. Box 833851
Richardson, TX 75083-3851

(Fold Here Second)

(Tape or Staple)

CONVEX Assembly-Language User's Guide
Document No. 740-000030-204, Sixth Edition

Reader's Forum

You are invited to submit your comments concerning the clarity and service of this manual. Constructive critical comments are most welcome and will help us continue in our efforts to generate quality customer documentation. (Please list the page number for questions and comments):

From:

Name_____ Title_____

Company_____ Date_____

Address and Phone No._____

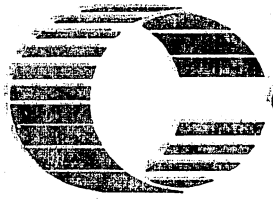
FOR ADDITIONAL INFORMATION/DOCUMENTATION:

Location	Phone Number
Within continental U.S.	1(800)952-0379
Outside continental U.S.	Contact local CONVEX office

Direct Mail Orders to:

CONVEX Computer Corporation
Customer Service
P.O. Box 833851
Richardson, Texas 75083-3851 USA

(Fold Here First)



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1046 RICHARDSON, TEXAS

POSTAGE WILL BE PAID BY ADDRESSEE

CUSTOMER SERVICE
CONVEX Computer Corp.
P.O. Box 833851
Richardson, TX 75083-3851

(Fold Here Second)

(Tape or Staple)