CONVEX Assembly Language Reference Manual (C Series)

First Edition

CONVEX

CONVEX COMPUTER CORPORATION

CONVEX Computer Corporation
3000 Waterview Parkway
P.O. Box 833851
Richardson, TX 75083-3851
United States of America
(214)497-4000

CONVEX Assembly Language Reference Manual (C Series)

Order No. DHW-301

First Edition
December 1991

CONVEX Press
Richardson, Texas
United States of America

CONVEX
Assembly Language
Reference Manual
(C Series)

Order No. DHW-301

Printed in the United States of America

Revision information for

CONVEX Assembly Language Reference Manual (C Series)

Edition	Document No.	Description
First	081-011930-000	December 1991. Release of the First Edition of the *CONVEX Assembly Language Reference Manual (C Series)*, to separate the Assembly Language Instruction Set from the *CONVEX Architecture Reference (C100, C200 Series)*, renamed the *CONVEX Architecture Reference Manual (C Series)*.

Contents

Figures

Tables

How to use this manual

The *CONVEX Assembly Language Reference Manual (C Series)* is the definitive reference for the assembly language instruction set. It is a companion to both the *CONVEX Architecture Reference Manual (C Series)* and the *CONVEX Compiler Utilities User's Guide* (replacing the *CONVEX Assembly Language User's Guide*).

This document is a reference tool developed to help engineers and software developers make maximum use of CONVEX C Series processor's facilities.

For definitions of terms not defined in this manual, refer to the "Glossary" in the *CONVEX Architecture Reference Manual (C Series)*.

For instruction execution times, refer to the *CONVEX Assembly Language Timing Guide (C Series)*.

This document applies to all CONVEX C Series architecture supercomputers, including the C100 Series, C200 Series, C3200 Series, C3400 Series, and C3800 Series CPUs.

The C200 Series, C3200 Series, C3400 Series, and C3800 Series CPUs are sometimes refered to as the multiprocessing C Series CPUs.

Organization

The following chapters of this document examine the assembly language of CONVEX supercomputers in detail:

- **Chapter 1, "Instruction formats"**—Describes the formats of the CONVEX C Series assembly language instruction set.
- **Chapter 2, "Instruction set (topical listings)"**—Groups the CONVEX C Series assembly language instructions into the following subsets:
  - Address register instructions
  - Scalar register instructions
  - Vector register instructions
  - Vector/scalar instructions
  - Vector comparison and edit instructions
  - Vector reduction instructions
  - VL, VS, and VM instructions
  - Communication register instructions
  - Program control instructions
  - CPU control instructions
  - Privileged control/status instructions
- **Chapter 3, "Instruction set (alphabetic listing)"**— The reference for the CONVEX C Series assembly language instruction set.
- **Appendix A, "Instructions sorted by op code"**—Ordered listing of the op codes.
- **Appendix B, "Instructions sorted by mnemonic"**—Ordered listing of the op codes.

Notational conventions

Notational conventions are those characters, symbols, terminology, or abbreviated expressions used in this guide.

Text notation

Text notation conventions set apart special items in text or examples.

- `Monospace type`, represents computer output, binary or hexadecimal numbers, commands, instructions, or mnemonics.

  Example:

  ```
  ERROR: Unknown command. Reenter.
  ```

- **`Bold monospace type`**, represents your response to a program or utility prompt.

  Example:

  ```
  Do you really want to exit? y
  ```

- Bold uppercase names in a sans-serif font designate keycap names.

  Example:

  RETURN

- If two keycap names are separated by a space, they are to be pressed sequentially.

  Example:

  ESC Q

- If two keycap names are separated by a hyphen, they are to be pressed simultaneously.

  Example:

  CTRL-C

- The word "enter", followed by a command, means to type the command and then press **RETURN**.

- *Italicized words* in an example command sequence are representative of a user-supplied name, such as a file name.

  Example:

  `command` *filename*

- Angle brackets (< >) designate unprintable ASCII characters.

  Example:

  `<197>` is an em dash

- Angle brackets (< >) are used to designate fields as bits in a byte, word, register, and so forth.

  Example:

  `PSW <6...0>`

- Square brackets ([ ]) in a command sequence designate optional letters, characters, subcommands or other command elements. Brackets may be nested, indicating optional subelements. If there are two or more options they are separated by vertical slashes or pipe symbols.

  Example:

  `com[mand] [`*filename*`|`*devicename*`]`

- Braces ({ }) in a command sequence designate manditory input, which must be one of two or more possible options. These options are separated by vertical slashes or pipe symbols.

  Example:

  `com[mand] {a|b|c}`

- A vertical slash (|), also known as the pipe symbol, in a command sequence indicates "or", giving you a choice between optional elements of a command.

  Example:

  `conf[igure] [`*command* | *alias*`]`

- Horizontal ellipses (. . .) in a command sequence show that the element immediately preceding them can be repeated.

  Example:

  ad[d] [[*board*] . . . | all]

- Vertical ellipses in a command sequence show that lines of an example have been left out.

  Example:

```
Verifying image   99
Verifying image  199
.
.
.
Verifying image  999
```

Command syntax

The notational conventions above are used to define the commands in the user interface.
Example:

com[mand]{.t|.f} [-a|-b] *input_file* [...] [*output_file*]

where

- command is *required* and may be abbreviated to com (square brackets indicate optional portion).
- If a command option, indicted by a list in braces, separated by a vertical slash, is used, then either .t or .f *is required.*
- If a command option, indicted by a list in square brackets, separated by a vertical slash, is used, then either -a or -b *is optional.*
- *input_file,* indicated by italics with no square brackets, is a *required* file name *supplied by the user.*
- Additional *input_file* names, indicated by ellipses in square brackets, *may optionally be supplied* by the user .
- *output_file,* indicated by square brackets and italics, is an *optional* file name *supplied by the user.*

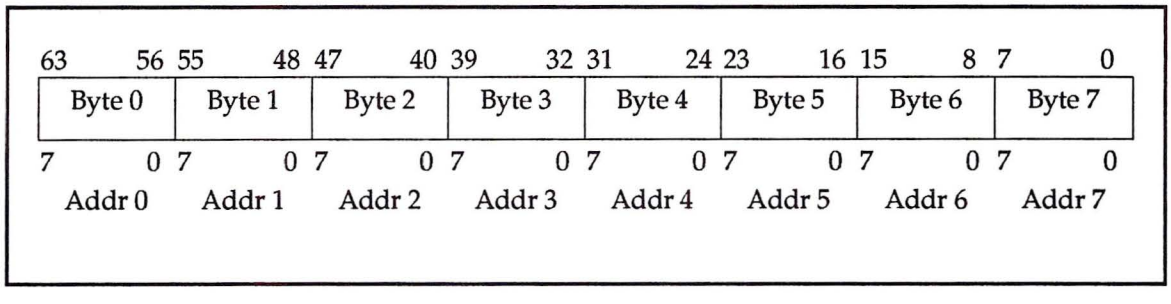
Data notation

Data notation conventions identify specific definitions in CONVEX supercomputer architecture:

- A *bit* is a single binary value or entity.
- A *nibble* is 4 bits.
- A *byte* is 8 bits.
- A *halfword* is 16 bits.
- A *word* is 32 bits.
- A *longword* is 64 bits.
- *Single precision* is a 32-bit floating point word.
- *Double precision* is a 64-bit floating point longword.
- An *instruction* is a multi-halfword operand.
- A bit is *set* when it contains a binary value of `1`.
- A bit is *clear* when it contains a binary value of `0`.
- Bit numbering is from left to right, `N-1` through `0`. The most significant numerical bit is `N-1`, the least significant is `0`. The bit numbering represents the binary weight of a position.
- Byte numbering is from left to right, `0` through `N-1`.
- Byte order in a 64-bit longword is interpreted with increasing byte addresses associated with higher order bytes within a longword. The most significant bit is associated with the least significant byte number.

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

Figure 1
Memory longword structure



- A *register* is a programmer-visible hardware storage element internal to the CPU.
- All register contents are written in hexadecimal notation, unless explicitly stated otherwise.
- Bit fields are specified with decimal numbers as:

  ```
  reg_name<x..y>
  ```

  where the bit field is `reg_name` from bits `x` through `y`.
- Individual bit positions within a register are specified as:

  ```
  reg_name<15,4,0>
  ```

  where 15, 4, and 0 are bits within `reg_name`.
- An *instruction* is a group of halfwords.
  - **C100 Series CPUs**—The first halfword is an op code and the remaining halfwords are operands.
  - **C200 Series CPUs**—The first halfword is an op code prefix, another halfword is an op code, and the remaining halfwords are operands.
- All memory and I/O addresses are written in hexadecimal notation unless explicitly stated otherwise.

- *Physical memory* is the physical storage (main memory) actually installed with the CPU.
- *Virtual memory* is the perceived amount of main memory as seen by the application programmer.
- The symbol *k* is an abbreviation for *kilo* or 1,024.
- The symbol *M* is an abbreviation for *mega* or 1,048,576.
- The symbol *G* is an abbreviation for *giga* or 1,073,741,824.
- A *stack* is a data structure in which memory is allocated and deallocated from one end, usually called the top, on a last-in, first-out basis.
- A *return block* is a collection of register contents that are pushed on or popped off a stack in response to an instruction or other event.

- *Reserved* or *undefined* indicate what, if anything, to expect from unused fields in registers, reserved memory, or reserved I/O space. Algorithm implementation based on the use of reserved fields is not recommended.

Op code representation notation

Representations of the op codes are specified with the following suffixes appended:

b	Byte (8-bit value)
h	Halfword (16-bit value)
w	Word (32-bit value)
l	Longword (64-bit value)
x	Extended (128-bit value)
s	Single-precision floating point (32 bits)
d	Double-precision floating point (64 bits)
t	Operation under mask – true (C200 Series only)
f	Operation under mask – false (C200 Series only)
#n	A 3-bit (short) immediate with binary value 0,1,..7
#N	A 32-bit two's complement signed immediate

Memory location syntax notation

The following notation describes memory locations:

- **Effective address (*effa*)**—The effective memory address of an operand.
- **Communication register effective address (*Ceffa*)**—The effective communication register address of an operand.

The following notation describes the contents of memory:

- **c(*effa*)**—The contents of the memory location specified by the memory effective address.

- c(*Ceffa*)—The contents of the communication register specified by the communication register effective address.
- L(*Ceffa*)—The contents of the lock bit associated with the communication register specified by the communication register effective address.

Register name notation

Registers in the CONVEX C Series architecture include:

BE	Broadcast enable registers
VL	Vector length
VS	Vector stride registers
VLS	Vector length and stride
VM	Vector merge
VV	Vector valid
PSW	Program status word
PC	Program counter
ICR	Interrupt control register
ITR	Interval timer register
ITSR	Interval timer scalar register
TOC	Time-of-century counter
CIR	Communication index register
CPUID	CPU identification register
TCPU	Target CPU identification register
TID	Thread identification register
TTR	Thread timer register

Warnings, cautions, and notes

The following are examples of warnings, cautions, and notes, and their typical content and location, as used in CONVEX documents:

Warning

A warning highlights procedures or information necessary to avoid injury to personnel. The warning immediately precedes the critical information and includes a description of the hazard.

Caution

A caution highlights procedures or information necessary to avoid damage to equipment, damage to software, loss of data, or invalid test results. The caution immediately precedes the critical information and includes a description of the possible damage.

Note

A note highlights information of a supplemental nature. The note immediately precedes or follows the highlighted information.

Associated documents

The following is a partial list of other manuals or books that may provide more detailed information on the topics presented in this manual:

- *CONVEX Compiler Utilities User's Guide* (replaces the *CONVEX Assembly Language User's Guide*), Order No. DSW-096—This manual is a reference guide for software engineers developing software for the CONVEX processor systems. This text contains all the individual formats and descriptions for each CONVEX processor's instruction set in addition to this document.
- *CONVEX Assembly Language Timing Guide (C Series)*, Order No. DSW-303—This guide includes the latest instruction times and information regarding the effect of instruction interaction on timing.
- *CONVEX Architecture Reference Manual (C Series)*, Order No. DHW-300—This manual includes information about the architecure of CONVEX C Series supercomputers.
- *CONVEX Guide to Writing Device Drivers*, Order No. DSW-095—This manual includes information necessary to write a device driver for CONVEX supercomputers.
- *CONVEX System Manager's Guide*, Order No. DSW-004—This manual is tailored to system managers who are responsible for administering resources on CONVEX systems. Included are descriptions for configuring devices, authorizing users, setting up mail and communications, performing backups and system accounting functions, and monitoring system resources.
- *CONVEX SPU UNIX Utilities Manual*, Order No. DHW-007—This manual contains the manual pages that describe the features of the CONVEX Service Processor Unit (SPU) Operating System utilities, including UNIX V7 commands, system call error numbers, file formats, and system management.

Ordering documents

To order the current edition of this or any other CONVEX document, send requests to:

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

Include the order number or exact title with the request. The order number is on the title page of the manual and begins with the letters "DHW-" or "DSW-."

The order number for the *CONVEX Assembly Language Reference Manual (C Series)* is DHW-301.

Technical assistance

Hardware and software support can be obtained through the CONVEX Technical Assistance Center (TAC):

- From all locations in the United States:
  - Customers call (800)952-0379.
  - CONVEX employees call (800)545-4839.
- From locations in Canada, customers and CONVEX employees call (800)345-2384.
- From all other locations, contact the nearest CONVEX office.

Acknowledgments

Many people have made contributions to this document:

- Technical contributors: Rich Adkisson, Mike Chastain, Dave Dodson, Harold Dozier, Gary Gostin, Jim Kerr, Jim Mankovich, Lee Mcfearin, David Schrodel, Randall Stiles, and Steve Wallach
- Document review team: Brad Culter, Dave Dodson, Baji Edupuganty, Robert Ellis, Tom Ford, David Gray, Al Haddix, Gil Hansen, Mark Jones, Jim Kerr, Brian Konigsburg, Jim Mankovich, Lee Mcfearin, John Moses, Brian Parks, and David Schrodel
- Editorial services: Larry Bonura, Sheri Roloff

The efforts of these people have made this document possible.

Phil Lloyd
CONVEX Hardware Documentation

Instruction formats

1

This chapter briefly describes the assembly language program and instruction formats. The assembly language addressing modes, immediate, register, direct memory, and indirect memory, are defined and described in detail.

The assembly language format and structures are described in detail in the *CONVEX Compiler Utilities User's Guide* (replaces the *CONVEX Assembly Language User's Guide*).

The assembly language execution timing is described in detail in the *CONVEX Assembly Language Timing Guide (C Series)*.

The CONVEX assembly language consists of mnemonics and op codes, which are directly related. By definition, each assembly language mnemonic translates into a unique machine language op code. These op codes give access to registers, memory, and other structures in CONVEX software.

Op codes are the focus of this manual. The op code formats of the CONVEX assembly language instruction set for the C Series architecture supercomputers are defined and described in detail.

In addition, mnemonics are used in the context of assembly language program structures.

Assembly language program format

An assembly language program is a sequence of mnemonic instructions. These can be instructions that will be translated into machine language instructions (op codes), or they can be directives to the assembler, itself. Both types of instructions follow the same basic format.

The assembly language program line is comprised of five fields: label, mnemonic, operand list, comment, and terminator. The only field that is required is the terminator field (exclamation point or newline). The format for an assembly language program instruction is shown in Figure 2.

Figure 2
Assembly language program line format

```
[label:] mnemonic [operand list] [;comment] terminator
```

Assembly language lines can be combined to form programs.

Example:

```
Arg =   4
        .
        .
        .

loop:   eq.w    #0,s0
        jmps.t  exit
        add.w   #1,s0
        ld.w    arg(a2),s1
        add.w   s1,s0
        br      loop
exit:
        .
        .
        .
```

Assembly language addressing modes

The CONVEX assembly language instruction set has eight possible operand addressing modes: immediate mode, register mode, and six modes for specifying operands in memory (direct and indirect versions of absolute, register deferred, and indexed modes). Since the CONVEX architecture is based on three sets of high-speed registers (address, scalar, and vector), most of the instructions are limited to register operands only. Each addressing mode is described in detail in the following sections.

Immediate mode

Immediate operands provide a method for referencing data in the program's instruction stream. The assembler syntax for specifying an immediate operand is written as "#" followed by the expression that defines the value of the immediate operand.

Register mode

The majority of the instructions in the instruction set require one or more register mode operands. A register mode operand specifies the register set and the particular register in the set to be used as the operand. The actual machine instruction generated by the assembler depends on both the register set used and the register within that register set.

General register operands are specified by a letter (either upper or lower case) and a number (zero to seven). The letter denotes the register set, and the number specifies the register number in the set. The general register sets for the machine are:

- The address registers (A)
- The scalar registers (S)
- The vector registers (V)

To aid in program readability, three of the address registers can also be referenced by special names or reserved words. These are:

- The stack pointer (SP = A0)
- The argument pointer (AP = A6)
- The frame pointer (FP = A7)

The assembler itself uses reserved words to denote the registers in the machine. Each register is referenced by the reserved words only. Additional symbols may not be defined to denote machine registers.

There are ten special purpose registers in the machine that are used for machine control. These registers are also specified by reserved words:

- Vector stride register (VS)
- Vector length register (VL)
- Vector stride and length combination (VLS)
- Processor status word (PSW)
- Program counter (PC)
- Interval timer register (ITR)
- Interval timer status register (ITSR)
- Interrupt target CPU register (TCPU)
- Interrupt control register (ICIR)
- Communication register set index register (CIR)
- Thread identification register (TID)
- Time-of-century clock (TOC)
- Physical CPU identification register (CPUID)
- Thread timer register (TTR)
- CPU timer register (CTR)

Direct memory addressing modes

This addressing method references the target operand directly.

Absolute addressing mode

A program can reference an address in the virtual address space by specifying the location, using an expression that evaluates to the location's address.
Example:

```
ld.w addr, reg
```

Register deferred addressing mode

The address registers can contain a pointer, or the address of a register that contains the operand, rather than the operand itself.

This addressing mode is specified by enclosing the address register that contains the pointer in parentheses.
Example:

```
ld.w (INDEX reg), reg
```

Indexed addressing mode

In this mode, an offset or base is added to the contents of the address register and the result is used as a pointer to the operand.

This addressing mode is specified by preceding a deferred register specifier with an expression.
Example:

```
ld.w offset (INDEX reg), reg
```

Indirect memory addressing modes

Indirection is an addressing method that uses an address in logical memory to reference the ultimate target operand. Indirection occurs after indexing (adding of the address register). Only one level of indirection can be specified. An indirect word is a 32-bit pointer to a virtual byte address.

Indirect absolute addressing mode

This mode provides for one level of indirection from an absolute address. The operand pointer is located at the absolute address specified by an expression.

This addressing mode is specified by preceding an absolute location operand with an ampersand.
Example:

```
ld.w @addr, reg
```

Indirect deferred addressing mode

This mode provides an additional level of indirection over the deferred mode. The register specified contains the address of the pointer to the desired operand.

This addressing mode is formed by preceding a deferred register operand with an ampersand.
Example:

```
ld.w @ (INDEX reg), reg
```

Indirect indexed addressing mode

In this mode, the value of the register and the value of the index expression are added together to form the address of a pointer to the desired operand.

This addressing mode is specified by preceding an indexed operand with an ampersand.
Example:

```
ld.w @ addr (INDEX reg), reg
```

Assembly language instruction formats

A *standard* assembly instruction is one, two, or three halfwords (16, 32, or 48 bits) in length. An *extended op code* assembly instruction is two, three, or four halfwords (32, 48, or 64 bits) in length.

Memory in the CONVEX C Series supercomputer is byte addressable. However, instructions are addressed on *even* byte boundaries. Bit <0> of the program counter (PC) is always zero.

Standard op codes

There are eight instruction formats, as shown in Table 1. The first bits of the op code are encoded using a *modified leading one's* (Huffman's code) encoding method to indicate the instruction format. The instruction format is decoded using a binary tree processing sequence. After interpreting the op code type (in column one), the number of bits remaining in the op code (column two) and the additional elements of the instruction format (column three) are then determined.

Table 1
Instruction formats

Huffman's code format (binary)	Additional op code length (bits)	Additional elements of the instruction format	Example instruction
1	6	[Ri], [Rj], [Rk]	`add.h Vi,Sj,Vk`
00	7	[L], [Aj], [Rk] [number=16/32]	`add,w #N,Ak`
	6	[@], [L], [Aj], [Rk] [number=16/32]	`call` *effa*
010	7	[Rj], [Rk]	`add.w #n,Ak`
0110	6	[Rj], [Rk]	`cprs.f Vj,Vk`
0111 0	3	[displacement <8..0>]	`br`
0111 10	4	[Rj], [Rk]	`cvtw.s Vj,Vk`
0111 110	6	[Rk]	`atan.s Sk`
0111 1110	5	[Rk]	`all Vk`

Any op code not covered by this instruction formatting is an unimplemented instruction, and its execution results in an *undefined op code* system exception.

Refer to "Op code descriptions," page 81, and the *CONVEX Architecture Reference Manual (C Series)*, Chapter 12, "Operating system exceptions."

Extended op codes

In order to add new instructions to the standard assembly language instruction set, the extended instruction set (multiprocessing C Series CPUs only) includes an *extended op code* format. In this format, a one-, two-, or three-halfword op code may be preceded by a 16-bit field known as the *op code prefix*.

The extended instruction set defines two prefixes with values `7EF0` or `7EF8`. The difference between the two prefix values is in the E flag, bit <3>, as shown in Figure 3 and Figure 4 . The last three bits of the prefix op code are not used, which means that prefixes `7EF0` through `7EF7` (E = 0) are equivalent, and `7EF8` through `7EFF` (E = 1) are equivalent.

When no prefix is encountered in the instruction stream, the instruction belongs to the standard instruction set, and is said to reside in standard (ST) space. When either of these prefixes is encountered in the instruction stream, the subsequent op code belongs to the extended instruction set, and is said to reside in extended (E0 or E1) space.

Operation under mask

A primary use for the extended op code prefix is *operation under mask* (multiprocessing C Series CPUs only). In this example, the root op code is the nonmasked version of the instruction. The prefix `7EF0` indicates *under false mask*, and the prefix `7EF8` indicates *under true mask*.
Example:

`6800`	is the standard op code for `eq.b Vj,Vk`.
`7EF0 6800`	is the extended op code for `eq.b.f Vj,Vk`.
`7EF8 6800`	is the extended op code for `eq.b.t Vj,Vk`.

The mask polarity indicated in the prefix (bit <3>) is the *only* distinction made between the two extended op code prefixes.

Operations-under-mask are explained in more detail in the "Vector operation under mask" section of Chapter 2.

The instruction set also defines some nonmasked extended instructions. These instructions have two equivalent op codes that cause their execution.
Example:

`7EF0 3700` extended op code for `snd.l Sk,` *Ceffa*

`7EF8 3700` equivalent op code for `snd.l Sk,` *Ceffa*

Op code reference modes

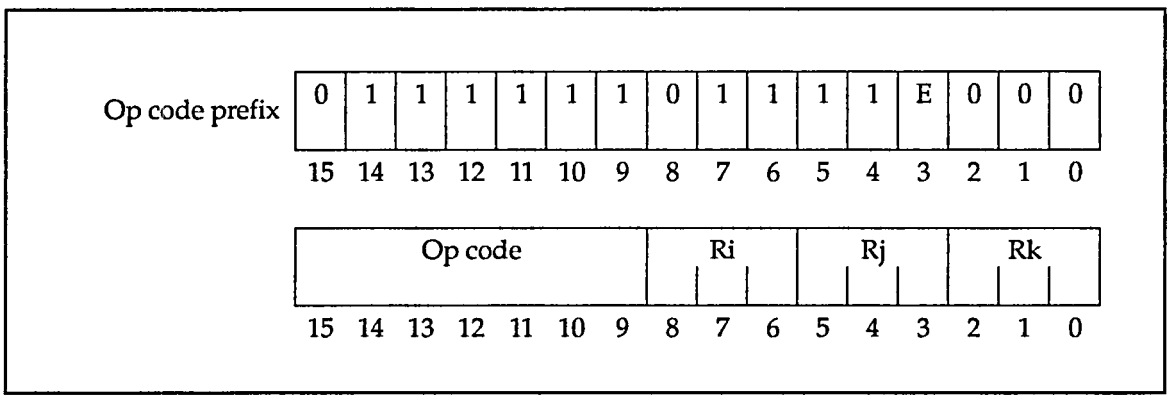
There are two generic types of addressing modes: *register-register* and *register-memory*.

Referencing registers

Register-register instructions specify none, one, two, or three unique registers and are 16 bits in length. Extended instructions also include a 16-bit op code prefix. In the register-register mode, all source operands are assumed to have been preloaded into one of the various machine registers. The destination of the result is one of the specified registers.

Figure 3 shows the structure of memory-reference instructions.

Figure 3
Register-reference instruction format



These fields are defined as follows:

- **Op code prefix** —Bits <15..0>
  The 16-bit prefix field specifies an extended instruction. The E flag (bit <3>) is the true/false mask bit. The prefix is not used for standard instructions
- **Op code** —Bits <15..9|6|3>
  The op code field specifies which operation to perform on the referenced data and selects the register class (A, S, or V) referred to by the Rx field(s), if any.
- **Ri, Rj, Rk**—Bits <8..6>, Bits <5..3>, Bits <2..0>
  Each instruction can have up to three register designation fields, Ri, Rj, and/or Rk. These fields specify the source or destination of the referenced operands, if any. The number of registers used depends on the nature of the op code.

The op code also specifies the precision of the operand. The register can be either a scalar register (Sk), an address register (Ak), or a vector register (Vk), depending on the instruction. In some instructions, the field is unused (X).

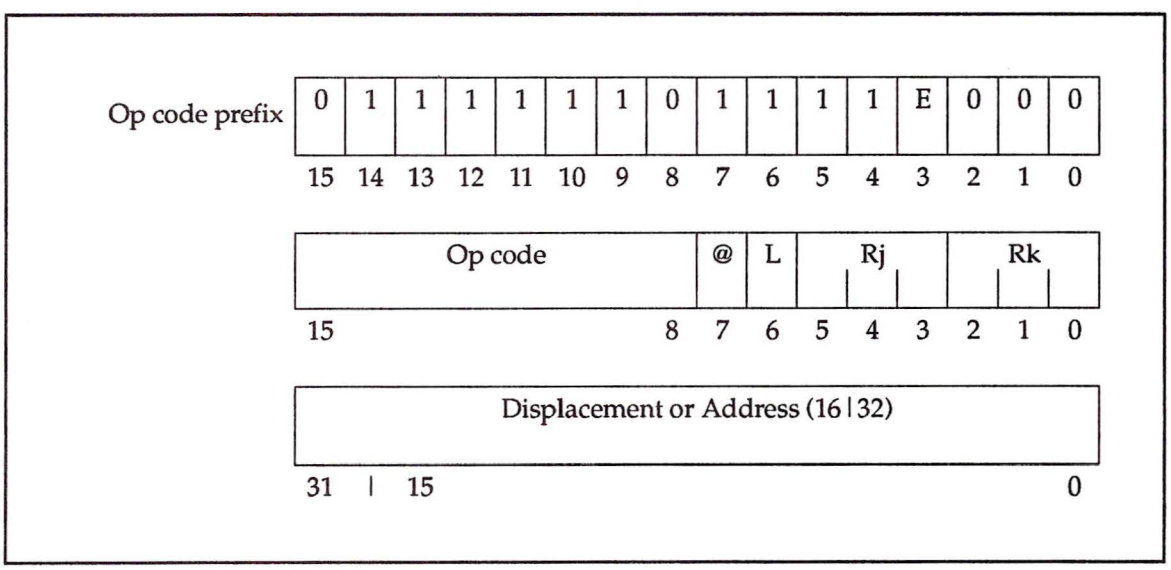
Referencing memory

The instructions that operate in the *register-to-memory* and *register-from-memory* mode use operands called an *effective address (effa)*. Communication register duals of these instructions use a *communication register effective address (Ceffa)*. Generally, instructions that reference memory to load and store operands are either 32 or 48 bits in length. Extended instructions also include a 16-bit op code prefix. The difference between these instruction lengths depends on the absence or presence of the prefix and the length of the displacement or address field.

For all instructions that reference memory, exceptions related to address translation can occur, for example, page faults or protection violations.

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

Figure 4
Memory-reference instruction format



These fields are defined as follows:

- **Op code prefix** —Bits <15..0>
  The 16-bit prefix field specifies an extended instruction. The E flag (bit <3>) is the true/false mask bit. The prefix is not used for standard instructions

- **Op code** —Bits <15..8>
  The op code field specifies which operation to perform on the referenced data and selects the register class (A, S, or V) referred to by the Rx field(s), if any.

- **Indirection (@)** —Bit <7>
  The @ field specifies the presence or absence of indirection. If @ = 0, no indirection is specified.

  If @ = 1, indirection is specified, and the effective address is the address stored in the word at the location computed by adding the contents of Aj (0, if j = 0) to the displacement.

- **Length (L)**—Bit <6>
  The L field specifies the length of the displacement field. If L = 0, the displacement field is a two's complement 16-bit value.

  If L = 1, the displacement field is a two's complement 32-bit value. This field is sign-extended to 32 bits before it is added to the effective address. The instruction length is automatically generated, based on the size of the displacement.

- **Aj**—Bits <5..3>
  The Aj field specifies the Address (A) register used to generate the logical address. If A0 is specified, the value of 0 is used (absolute addressing) and the true contents of A0 are unused. If A1, A2, . . ., A7 is specified the contents of the specified address register are added to the displacement field to generate the final effective target address.

  If indirection is not specified, the effective target address is the effective address. If indirection is specified the effective target address references byte 0 of a 32-bit word whose contents are the effective address.

- **Rj, Rk**—Bits <5..3>, <2..0>
  The Rk field specifies the source or destination of the referenced operand, if any. The op code specifies the precision of the operand. The number of registers used depends on the nature of the op code.

The op code also specifies the precision of the operand. The register can be either a scalar register (Sk), an address register (Ak), or a vector register (Vk), depending on the instruction. In some instructions, the field is unused (X).

- **Displacement | Address**—Bits <31 | 15..0>
  As a *displacement*, this field contains a 16-bit or 32-bit value that is either added to the entire 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.

  As an *address*, the field contains a 16-bit or 32-bit value that is used directly as a byte address.

Instruction set (topical listings)

2

This chapter lists by topic the assembly language instruction set for the CONVEX C Series supercomputers.

There are four register sets.

- Address registers (eight 32-bit registers)
- Scalar registers (eight 64-bit registers)
- Vector registers (8 vectors, each containing 128 64-bit elements)
- Communication registers (1024 64-bit locations for C200/C3200 Series CPUs, and 4096 64-bit locations for C3400 Series and C3800 Series CPUs)

The first three register sets are general registers, partitioned according to the type of operand to be manipulated. The fourth register set implements CPU communication and synchronization for the multiprocessing C Series CPUs.

Refer to the *CONVEX Architecture Reference Manual (C Series)*, Chapter 3, "General registers," and Chapter 6, "Communication registers," for more information.

In this chapter, the instruction set is divided into topics, first according to the type of register data the instruction manipulates, then other topics are considered:

- Process control instructions
- CPU control and status instructions
- Privileged control and status instructions
- Intrinsic instructions

Address register instructions

The instructions listed in this section manipulate operands in the address (A) registers. When these operands are less than 32 bits in length, only the specified bits of the A registers are used or modified. All other bits are unused or left unchanged.

All arithmetic operations are performed using two's complement manipulations. All overflow is generated and is stored in the processor status word (PSW). Integer traps can be disabled. Logical operations operate on words only.

The signed and unsigned compares *set* or *cleared* with C, the address carry flag (PSW<31>). Address register comparisons result in the setting or clearing of C. If the specified comparison is true, C is *set* to 1. If the comparison is false, C is *cleared* to 0.

There are two sets of comparisons: *signed* and *unsigned*. Unsigned comparisons treat both operands as positive values. Within each set of comparisons there is an instruction group that permits the specification of an immediate value.

Typically, after the execution of a compare instruction, a branch-on-carry instruction is executed.

The following are the *complementary relations* used for address compares:

<= >

< >=

== !=

The instruction set includes compare instructions for all three comparisons in the first column. If the required relation is not one of those three, the complement of the relation is used, along with the complement of the branch condition.

The first two comparisons in the second column can be achieved in either of two ways. First, A > B, *branch true* is equivalent to A <= B, *branch false*. Alternatively, B > A, *branch true* is equivalent to A < B, *branch true*.

Table 2 lists the address register instructions.

Table 2
Address register instructions

Instruction	Description
`add.h #n, Ak`	Add short immediate address halfword
`add.h #N, Ak`	Add immediate address halfword
`add.h Aj, Ak`	Add address register halfword
`add.w #n, Ak`	Add short immediate address word
`add.w #N, Ak`	Add immediate address word
`add.w Sj, Ak`	Add scalar to address word
`add.w Aj, Ak`	Add address register word
`and #N, Ak`	AND immediate to address register
`and Aj, Ak`	AND address register
`cvtb.w Aj, Ak`	Convert byte to word
`cvth.w Aj, Ak`	Convert half to word
`cvtw.b Aj, Ak`	Convert word to byte
`cvtw.h Aj, Ak`	Convert word to halfword
`div.h #n, Ak`	Divide short immediate address halfword
`div.h #N, Ak`	Divide immediate address halfword
`div.h Aj, Ak`	Divide address register halfword
`div.w #n, Ak`	Divide short immediate address word
`div.w #N, Ak`	Divide immediate address word
`div.w Aj, Ak`	Divide address register word
`eq.h #n, Ak`	Compare equal short immediate halfword
`eq.h #N, Ak`	Compare equal immediate halfword
`eq.h Aj, Ak`	Compare equal halfword
`eq.w #n, Ak`	Compare equal short immediate word
`eq.w #N, Ak`	Compare equal immediate word
`eq.w Aj, Ak`	Compare equal word
`ld.b` *effa*`, Ak`	Load address register byte
`ld.h #n, Ak`	Load short immediate halfword into Ak

Table 2 (continued)
Address register instructions

Instruction	Description
`ld.h #N, Ak`	Load immediate halfword into Ak
`ld.h` *effa*`, Ak`	Load address register halfword
`ld.w #n, Ak`	Load short immediate into Ak
`ld.w #N, Ak`	Load immediate into Ak
`ld.w` *effa*`, Ak`	Load address register word
`ldea` *effa*`, Ak`	Load effective address
`ldpa Aj, Ak`	Load a physical byte address into Ak
`le.h #n, Ak`	Compare less than or equal short immediate halfword
`le.h #N, Ak`	Compare less than or equal immediate halfword
`le.h Aj, Ak`	Compare less than or equal signed halfword
`le.w #n, Ak`	Compare less than or equal short immediate word
`le.w #N, Ak`	Compare less than or equal immediate word
`le.w Aj, Ak`	Compare less than or equal signed word
`leu.h #n, Ak`	Compare unsigned less than or equal short immediate halfword
`leu.h #N, Ak`	Compare unsigned less than immediate halfword
`leu.h Aj, Ak`	Compare unsigned less than or equal halfword
`leu.w #n, Ak`	Compare unsigned less than or equal short immediate word
`leu.w #N, Ak`	Compare unsigned less than or equal immediate word
`leu.w Aj, Ak`	Compare unsigned less than or equal word
`lt.h #n, Ak`	Compare less than short immediate halfword
`lt.h #N, Ak`	Compare less than immediate halfword

Table 2 (continued)
Address register instructions

Instruction	Description
`lt.h Aj, Ak`	Compare less than signed halfword
`lt.w #n, Ak`	Compare less than short immediate word
`lt.w #N, Ak`	Compare less than immediate word
`lt.w Aj, Ak`	Compare less than signed word
`ltu.h #n, Ak`	Compare unsigned less than short immediate halfword
`ltu.h #N, Ak`	Compare unsigned less than immediate halfword
`ltu.h Aj, Ak`	Compare unsigned less than halfword
`ltu.w #n, Ak`	Compare unsigned less than short immediate word
`ltu.w #N, Ak`	Compare unsigned less than immediate word
`ltu.w Aj, Ak`	Compare unsigned less than word
`mov Aj, Ak`	Move address register
`mov Aj, Sk`	Move an address to a scalar
`mov PSW, Ak`	Store the PSW into an address register
`mov Ak, PSW`	Load an address register into the PSW
`mov PC, Ak`	Load next PC address
`mul.h #n, Ak`	Multiply short immediate address halfword
`mul.h #N, Ak`	Multiply immediate address halfword
`mul.h Aj, Ak`	Multiply address register halfword
`mul.w #n, Ak`	Multiply short immediate address word
`mul.w #N, Ak`	Multiply immediate address word
`mul.w Aj, Ak`	Multiply address register word
`neg.h Aj, Ak`	Negate address register halfword
`neg.w Aj, Ak`	Negate address register word
`not Aj, Ak`	Complement address register
`or #N, Ak`	OR immediate to address register

Table 2 (continued)
Address register instructions

Instruction	Description
`or Aj, Ak`	OR address register
`pop.w Ak`	Pop word into address register
`psh.w Ak`	Push an address register
`pshea` *effa*	Push effective address
`shf #n, Ak`	Logical shift short immediate to address register
`shf #N, Ak`	Logical shift immediate to address register
`shf Aj, Ak`	Shift an address
`st.b Ak,` *effa*	Store address register byte
`st.h Ak,` *effa*	Store address register halfword
`st.w Ak,` *effa*	Store address register word
`sub.h #n, Ak`	Subtract short immediate address halfword
`sub.h #N, Ak`	Subtract immediate address halfword
`sub.h Aj, Ak`	Subtract address register halfword
`sub.w #n, Ak`	Subtract short immediate address word
`sub.w #N, Ak`	Subtract immediate address word
`sub.w Aj, Ak`	Subtract address register word
`xor #N, Ak`	Exclusive OR immediate to address register
`xor Aj, Ak`	Exclusive OR address register

Scalar register instructions

The instructions listed in this section manipulate operands in the eight 64-bit scalar registers. The scalar registers are used to perform computations on fixed and floating point operands.

Instructions affecting both scalar and vector registers are in the "Vector comparison and edit instructions" section.

Scalar register instructions work with byte, halfword, word, longword, single-precision floating point, and double-precision floating point operands.

Both CONVEX native format and IEEE format are supported by all floating point instructions.

The following flags are affected in the PSW:

- SIV (integer overflow), integer only
- OV (exponent overflow), floating point only
- UN (exponent underflow), floating point only
- SC (scalar carry), integer only
- RO (reserved operand), floating point only
- SDZ (divide by zero), integer only
- FDZ (divide by zero), floating point only

Scalar register comparisons result in the setting or clearing of SC, the scalar carry flag (PSW<23>). If the specified comparison is true, SC is *set* to 1. If the comparison is false, the scalar SC is *cleared* to 0.

Scalar compares are divided into two sets of comparisons: *signed* and *unsigned*. Unsigned comparisons treat both operands as positive values. Within each set of comparisons there is an instruction group that permits the specification of an immediate value.

Typically, after the execution of a scalar compare instruction, a branch-on-carry instruction is executed.

The following are the *complementary relations* used for scalar compares:

<= >

< >=

== !=

The instruction set includes compare instructions for all three comparisons in the first column. If the required relation is not one of those three, the complement of the relation is used, along with the complement of the branch condition. The first two comparisons in the second column can be achieved in either of two ways. First, A > B, *branch true* is equivalent to A <= B, *branch false*. Alternatively, B > A, *branch true* is equivalent to A < B, *branch true*.

Table 3 lists the scalar register instructions.

Table 3
Scalar register instructions

Instruction	Description
`add.b Sj, Sk`	Add scalar/scalar integer byte
`add.d Sj, Sk`	Add scalar/scalar double float
`add.h #N, Sk`	Add scalar/immediate integer halfword
`add.h Sj, Sk`	Add scalar/scalar integer halfword
`add.l Sj, Sk`	Add scalar/scalar integer longword
`add.s #N, Sk`	Add scalar/immediate single float
`add.s Sj, Sk`	Add scalar/scalar single float
`add.w #N, Sk`	Add scalar/immediate integer word
`add.w Sj, Sk`	Add scalar/scalar integer word
`and #N, Sk`	AND scalar/immediate
`and Sj, Sk`	AND scalar/scalar
`cvtb.w Sj, Sk`	Convert byte to word
`cvtd.l Sj, Sk`	Convert double float to longword
`cvtd.s Sj, Sk`	Convert double float to single float
`cvtd.w Sj, Sk`	Convert double float to word

Table 3 (continued)
Scalar register instructions

Instruction	Description
`cvth.w Sj, Sk`	Convert halfword to word
`cvtl.d Sj, Sk`	Convert longword to double float
`cvtl.s Sj, Sk`	Convert longword to single float
`cvtl.w Sj, Sk`	Convert longword to word
`cvts.d Sj, Sk`	Convert single float to double float
`cvts.l Sj, Sk`	Convert single float to longword
`cvts.w Sj, Sk`	Convert single float to word
`cvtw.b Sj, Sk`	Convert word to byte
`cvtw.d Sj, Sk`	Convert word to double float
`cvtw.h Sj, Sk`	Convert word to halfword
`cvtw.l Sj, Sk`	Convert word to longword
`cvtw.s Sj, Sk`	Convert word to single float
`div.b Sj, Sk`	Divide scalar/scalar integer byte
`div.d Sj, Sk`	Divide scalar/scalar double float
`div.h #N, Sk`	Divide scalar/scalar integer halfword
`div.h Sj, Sk`	Divide scalar/scalar integer halfword
`div.l Sj, Sk`	Divide scalar/scalar integer longword
`div.s #N, Sk`	Divide scalar/scalar single float
`div.s Sj, Sk`	Divide scalar/scalar single float
`div.w #N, Sk`	Divide scalar/scalar integer word
`div.w Sj, Sk`	Divide scalar/scalar integer word
`eq.b Sj, Sk`	Compare equal byte
`eq.d Sj, Sk`	Compare equal double float
`eq.h #N, Sk`	Compare equal halfword
`eq.h Sj, Sk`	Compare equal halfword
`eq.l Sj, Sk`	Compare equal longword
`eq.s Sj, Sk`	Compare equal single float
`eq.w #N, Sk`	Compare equal word

Table 3 (continued)
Scalar register instructions

Instruction	Description
`eq.w Sj, Sk`	Compare equal word
`frint.d Sj, Sk`	Integerize float double scalar
`frint.s Sj, Sk`	Integerize float single scalar
`ld.b` *effa*`, Sk`	Load scalar byte
`ld.d #N, Sk`	Load immediate upper 32 bits
`ld.d` *effa*`, Sk`	Load scalar double float
`ld.dl #N, Sk`	Load 64-bit floating immediate, lower half
`ld.du #N, Sk`	Load 64-bit floating immediate, upper half
`ld.h` *effa*`, Sk`	Load scalar halfword
`ld.l #N, Sk`	Load 32-bit immediate sign-extended to 64 bits
`ld.l` *effa*`, Sk`	Load scalar longword
`ld.ll #N, Sk`	Load 64-bit integer immediate, lower half
`ld.lu #N, Sk`	Load 64-bit integer immediate, upper half
`ld.s` *effa*`, Sk`	Load scalar single float
`ld.u #N, Sk`	Load immediate, upper half
`ld.w #N, Sk`	Load a 32-bit immediate
`ld.w` *effa*`, Sk`	Load scalar word
`ldb.b` *effa*`, Sk`	Load bypass scalar byte
`ldb.d` *effa*`, Sk`	Load bypass scalar double float
`ldb.h` *effa*`, Sk`	Load bypass scalar halfword
`ldb.l` *effa*`, Sk`	Load bypass scalar longword
`ldb.s` *effa*`, Sk`	Load bypass scalar single float
`ldb.w` *effa*`, Sk`	Load bypass scalar word
`ldea` *effa*`, Sk`	Load effective address/scalar
`le.b Sj, Sk`	Compare less than or equal byte

Table 3 (continued)
Scalar register instructions

Instruction	Description
`le.d Sj, Sk`	Compare less than or equal double float
`le.h #N, Sk`	Compare less than or equal halfword
`le.h Sj, Sk`	Compare less than or equal halfword
`le.l Sj, Sk`	Compare less than or equal longword
`le.s #N, Sk`	Compare less than or equal single
`le.s Sj, Sk`	Compare less than or equal single float
`le.w #N, Sk`	Compare less than or equal word
`le.w Sj, Sk`	Compare less than or equal word
`leu.b Sj, Sk`	Compare unsigned less than or equal byte
`leu.h #N, Sk`	Compare unsigned less than or equal halfword
`leu.h Sj, Sk`	Compare unsigned less than or equal halfword
`leu.l Sj, Sk`	Compare unsigned less than or equal longword
`leu.w #N, Sk`	Compare unsigned less than or equal word
`leu.w Sj, Sk`	Compare unsigned less than or equal word
`lop Sj, Sk`	Count of leading zeros in Sj
`lt.b Sj, Sk`	Compare less than byte
`lt.d Sj, Sk`	Compare less than double float
`lt.h #N, Sk`	Compare less than halfword
`lt.h Sj, Sk`	Compare less than halfword
`lt.l Sj, Sk`	Compare less than longword
`lt.s #N, Sk`	Compare less than single
`lt.s Sj, Sk`	Compare less than single float
`lt.w #N, Sk`	Compare less than word

Table 3 (continued)
Scalar register instructions

Instruction	Description
`lt.w Sj, Sk`	Compare less than word
`ltu.b Sj, Sk`	Compare unsigned less than byte
`ltu.h #N, Sk`	Compare unsigned less than halfword
`ltu.h Sj, Sk`	Compare unsigned less than halfword
`ltu.l Sj, Sk`	Compare unsigned less than longword
`ltu.w #N, Sk`	Compare unsigned less than word
`ltu.w Sj, Sk`	Compare unsigned less than word
`mov BE(Sj), Sk`	Move broadcast enable register to a scalar
`mov CIR, Sk`	Move comm index register to a scalar
`mov CPUID, Sk`	Move CPU identification to scalar
`mov ICR, Sk`	Move interrupt control register to scalar
`mov TCPU, Sk`	Move the target CPU register to scalar
`mov TID, Sk`	Load thread ID to scalar
`mov TTR, Sk`	Move thread timer/scalar
`mov TOC, Sk`	Move TOC to a scalar
`mov Sk, BE(Sj)`	Move scalar to the broadcast enable register
`mov Sk, CIR`	Move scalar to comm index register
`mov Sk, ICR`	Move scalar to interrupt control register
`mov Sk, TCPU`	Move scalar to the target CPU register
`mov Sk, TID`	Load scalar to thread ID
`mov Sk, TTR`	Move scalar to thread timer
`mov Sk, TOC`	Move scalar to TOC
`mov Sk, VMU`	Load VM<127..64> from Sk
`mov Sk, VML`	Load VM<63..0> from Sk

Table 3 (continued)
Scalar register instructions

Instruction	Description
`mov Sj, Ak`	Move 32 bits of Sj into Ak
`mov.d Sj, Sk`	Move scalar register single float
`mov.l Sj, Sk`	Move scalar register longword
`mov.s Sj, Sk`	Move scalar register double float
`mov.w Sj, Sk`	Move scalar register word
`mul.b Sj, Sk`	Multiply scalar/scalar integer byte
`mul.d Sj, Sk`	Multiply scalar/scalar double float
`mul.h #N, Sk`	Multiply scalar/immediate integer halfword
`mul.h Sj, Sk`	Multiply scalar/scalar integer halfword
`mul.l Sj, Sk`	Multiply scalar/scalar integer longword
`mul.s #N, Sk`	Multiply scalar/immediate single float
`mul.s Sj, Sk`	Multiply scalar/scalar single float
`mul.w #N, Sk`	Multiply scalar/immediate integer word
`mul.w Sj, Sk`	Multiply scalar/scalar integer word
`neg.b Sj, Sk`	Negate scalar/scalar integer byte
`neg.d Sj, Sk`	Negate scalar/scalar double float
`neg.h Sj, Sk`	Negate scalar/scalar integer halfword
`neg.l Sj, Sk`	Negate scalar/scalar integer longword
`neg.s Sj, Sk`	Negate scalar/scalar single float
`neg.w Sj, Sk`	Negate scalar/scalar integer word
`not Sj, Sk`	Complement scalar/scalar
`or #N, Sk`	OR scalar/immediate
`or Sj, Sk`	OR scalar/scalar
`plc.t Sj, Sk`	Count the number of ones in Sj
`pop.l Sk`	Pop Sk <63..0> from the stack

Table 3 (continued)
Scalar register instructions

Instruction	Description
`pop.w Sk`	Pop Sk <31..0> from the stack
`psh.l Sk`	Push Sk<63..0> onto the stack
`psh.w Sk`	Push Sk<31..0> onto the stack
`shf #N, Sk`	Shift scalar/immediate
`shf Sj, Sk`	Shift a scalar
`shf.w Sj, Sk`	Shift a scalar word
`shf.w #N, Sk`	Shift Scalar word/immediate
`st.b Sk,` *effa*	Store scalar byte
`st.d Sk,` *effa*	Store scalar double float
`st.h Sk,` *effa*	Store scalar halfword
`st.l Sk,` *effa*	Store scalar longword
`st.s Sk,` *effa*	Store scalar single float
`st.w Sk,` *effa*	Store scalar word
`ste.b Sk,` *effa*	Store an extended scalar byte
`ste.b.f Sk,` *effa*	Store extended scalar byte using not VM
`ste.b.t Sk,` *effa*	Store extended scalar byte using VM
`ste.d Sk,` *effa*	Store an extended scalar double float
`ste.d.f Sk,` *effa*	Store extended scalar double using not VM
`ste.d.t Sk,` *effa*	Store extended scalar double using VM
`ste.h Sk,` *effa*	Store an extended scalar halfword
`ste.h.f Sk,` *effa*	Store extended scalar halfword using not VM
`ste.h.t Sk,` *effa*	Store extended scalar halfword using VM
`ste.l Sk,` *effa*	Store an extended scalar longword
`ste.l.f Sk,` *effa*	Store extended scalar longword using not VM

Table 3 (continued)
Scalar register instructions

Instruction	Description
`ste.l.t Sk, effa`	Store extended scalar longword using VM
`ste.s Sk, effa`	Store an extended scalar single float
`ste.s.f Sk, effa`	Store extended scalar single using not VM
`ste.s.t Sk, effa`	Store extended scalar single using VM
`ste.w Sk, effa`	Store an extended scalar word
`ste.w.f Sk, effa`	Store extended scalar word using not VM
`ste.w.t Sk, effa`	Store extended scalar word using VM
`sub.b Sj, Sk`	Subtract scalar/scalar integer byte
`sub.d Sj, Sk`	Subtract scalar/scalar double float
`sub.h #N, Sk`	Subtract scalar/immediate integer halfword
`sub.h Sj, Sk`	Subtract scalar/scalar integer halfword
`sub.l Sj, Sk`	Subtract scalar/scalar integer longword
`sub.s #N, Sk`	Subtract scalar/immediate single float
`sub.s Sj, Sk`	Subtract scalar/scalar single float
`sub.w #N, Sk`	Subtract scalar/immediate integer word
`sub.w Sj, Sk`	Subtract scalar/scalar integer word
`tzc Sj, Sk`	Count of trailing zeros in Sj
`xor #N, Sk`	Exclusive OR scalar/immediate
`xor Sj, Sk`	Exclusive OR scalar/scalar

Vector instructions

This section defines the instructions that manipulate the vector (V) accumulators, scalar (S) accumulators, and the vector length (VL), vector stride (VS), and vector merge (VM) registers. A list of vector instructions is located at the end of the section.

The basic vector instruction set loads memory operands into an accumulator. Then, a register-to-register instruction performs the specified operation. There are some additional features applicable to the S and V registers that are not applicable to the A registers. These features are:

- Data representations manipulated
- Chaining
- Functional unit reservation
- Register unit reservation
- Register topology

The floating point data representations used by the vector/scalar instructions may be either the CONVEX native or the IEEE formats. IEEE-format floating point data representations and operations are initiated by setting PSW (IEEE), bit <13>, to 1. CONVEX native data representations and operations are initiated by clearing IEEE to 0. Refer to the *CONVEX Architecture Reference Manual (C Series)*, Chapter 2, "Data representations and operations," for more information.

Many of the multiprocessing C Series instructions are vector operations. The major architectural difference to vector processing in the multiprocessing C Series implementation is the *operation-under-mask* capability. Operation-under-mask is explained in detail in the "Vector operation under mask" section of this chapter.

Vector length specification

As discussed in the *CONVEX Architecture Reference Manual (C Series)*, Chapter 3, "General registers," a vector accumulator can hold up to 128 elements, where each element can be up to 64 bits of precision. The vector length (VL) register specifies the exact number of elements that are manipulated by vector instructions. VL applies to all vector accumulators.

Note

When VL is zero, no vector operation is performed.

Care should be taken when loading the VL register. When an operation is initiated, VL is copied into the internal machine state. Length control is generated from this internal VL value.

The first element in a vector accumulator V is V[0]. If VL is less than 128, then elements V[0] through V[VL-1] are manipulated. All other elements are not used and are left unchanged.

Chaining

Chaining is a vector processing mechanism that uses the output of one vector instruction as the input to another vector instruction. The actual chaining can occur any time the output is available. For example, the dot product operation requires a sum of a series of products. Chaining permits the sum to be initiated while the products are being produced. This form of concurrency results in significantly higher performance.

To facilitate chaining, vector register operations can specify up to three operands: two sources and one destination. This feature permits an output register to be different from either of the two possible input registers (3-operand addressing). The output register can then be used as an input for the second (chained) operation. The following is an example of chaining:

```
V3 == V2 + V1
V5 == V4 * V3
```

In the above example, the output of the addition operation that is stored into V3 can be immediately used as an input to the multiply operation. This is essentially equivalent to executing the single statement:

```
V5 == V4 * (V3 == V2 + V1)
```

Another example of chaining is:

```
V1 == Load effa
V3 == V2 + V1
V6 == V4 * V3
```

This executes as the following single expression

```
V6 == V4 * (V3 == V2 + V1 = Load effa)
```

which is essentially adding V2 to the contents of the effective address loaded into V1, and multiplying the results by V4.

The output of one functional unit may be chained into the input of a different functional unit. The output can be chained into stores, masks, reduction operations, and so on. Unless otherwise specified, there are no exceptions to this chaining criterion. In the preceding example, chaining exists across three functional units.

Functional unit reservation

As previously explained, several instructions can be executed at the same time because more than one arithmetic or functional unit is provided. Multiple functional units (for example, `add`, `multiply`, and `divide`), rather than multiple units of the same type (that is, 2 or more adders) are provided to ensure this higher performance. However, when a vector instruction is decoded that requires a functional unit currently being used, a functional unit reservation occurs. This means that the second instruction *cannot* execute simultaneously with the first.

The following is an example of functional unit reservation:

```
V2 = V1 + V0
V5 = V4 + V3
```

Both of these operations on the indicated vector registers require the add functional unit, and thus cannot function simultaneously.

The types of independent functional units that are present are implementation-dependent.

C100 Series CPUs

The possible arithmetic units for the C100 Series processors are:

- Add and subtract
- Multiply and divide
- Logical
- Compress
- Shifting
- Population count
- Mask, merge, and compress
- Load from memory
- Store to memory

The C100 Series processors structure these arithmetic units into three separate and distinct functional units, grouped as follows:

- Add, subtract, logical, compare, shift, and population count
- Multiply and divide
- Load and store from memory, mask, merge, and compress

Multiprocessing C Series CPUs

Generally, the possible arithmetic units for the C200 Series processors are:

- Add and subtract
- Multiply, divide, and square root
- Logical
- Compare
- Shift
- Type conversion
- Bit count (trailing zero count, leading zero count, population count)
- Edit (mask, merge, compress, and expand)
- Load and store from or to memory

The multiprocessing C Series processors structure these arithmetic units into three separate and distinct functional units, grouped as follows:

- Add, subtract, logical, compare, shift, bit count, type conversion
- Multiply, divide, square root, and edit
- Load and store from or to memory

Register unit reservation

To permit simultaneous instruction execution with multiple functional units, multiple registers must be provided. This means that access must be provided for all data to be manipulated. If this access is *not* provided, register reservation occurs, and instruction execution is sequential.

In the chaining example

```
V1 = V1 + V0
V4 = V3 * V1
```

the two instructions are executed sequentially since there are three references to V1 across the two instructions. This sequential execution occurs even though different functional units are specified. If the two instructions were

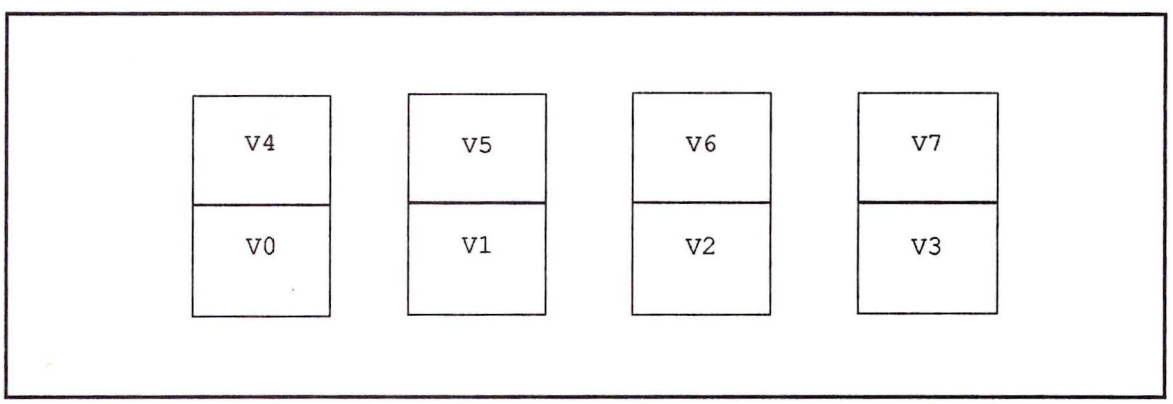
```
V2 = V1 + V0
V5 = V4 * V3
```

then both instructions would execute simultaneously.

Accumulator topology

The CONVEX C Series architecture has eight vector accumulators. The C Series processors implement these registers across four memory elements. However, the vector accumulator topology is implementation-specific and *not* part of the C Series architecture. Figure 5 shows the structure of the vector accumulators.

Figure 5
Vector accumulator structure



Each of the four register pairs shown in Figure 5 represents a high-speed memory array contained within a CPU. There are four such arrays. Each has two vector accumulators with 64 bits in each element.

Accumulator element manipulations

The following access rules govern manipulations of elements in the array.

C100 Series CPUs

- Two independent accesses may occur during each cycle for each array on the C100 Series processors. These accesses may be any combination of read and write: read/read, read/write, write/read, and write/write.
- The C100 Series processors require only one read port for two simultaneous reads of the same vector.
- If the executing instruction stream specifies more accesses at any given time than are physically possible, performance will degrade, but correct results will be obtained.
- Reads and writes to the same array during the same operation are permitted. There are no unusual side effects. The array functions exactly as one would expect. For example, the operation V0 = V0 + V1 adds V0 to V1 and stores the result in V0.

Multiprocessing C Series CPUs

- Three independent accesses can occur during each cycle for each array.
- The multiprocessing C Series processors allow a maximum of two read and one write accesses.
- If more than three accesses are specified to the same array, the new operation will wait until the operation involving the other accesses is completed.
- Reads and writes to the same array during the same operation are permitted. There are no unusual side effects. For example, the operation V0 = V0 + V1 adds V0 to V1 and stores the result in V0.

Accumulator operating speeds

The following are examples of full- and partial-speed operations as they relate to this array accessing.

C100 Series CPUs

The accumulator accessing scheme in the C100 Series processors has an impact on the level of chaining obtainable:

- The operation V2 = V1 + V0 proceeds at full speed.
- The operation V2 = V2 + V0 proceeds at full speed.
- The operation V2 = V2 + V6 proceeds at half speed, since both V2 and V6 are in the same array.

Multiprocessing C Series CPUs

The accumulator accessing scheme in multiprocessing C Series processors has an impact on the level of chaining obtainable. For example, the sequence of operations

```
V3 = V2 + V6
V4 = V6 * V1
```

would not be chained since there are more than two read accesses to the V2/V6 array.

The sequence

```
V2 = V1 + V0
V6 = V3 * V1
```

would not be chained since there is more than one write access to the V2/V6 array.

Scalar functional units

From the compiler's or programmer's viewpoint, there are an infinite number of scalar functional units. For the arithmetic expression

```
Z = A + B + C + D + E + F
```

the fastest way of evaluating the expression is:

```
Temp1 = A+B
Temp2 = C+D
Temp3 = E+F

Temp1 = Temp1 + Temp2
Temp1 = Temp1 + Temp3
```

The preceding method is faster than:

```
Temp = A +B
Temp = Temp + C
Temp = Temp + D
Temp = Temp + E
Temp = Temp + F
```

This technique is referred to as *tree height reduction.*

Vector loads and stores

The load and store with index instructions allow the use of a vector register to specify those indices of another vector which will be affected. These operations are commonly referred to as gather and scatter. None of the flags in the PSW are affected.

When a vector index is not specified, the first element is referenced by the effective address produced by evaluating the @, L, A, and address/displacement fields. The address of the next element accessed is obtained by adding the signed value in the vector stride register (VS). This signed value is the distance in bytes. The address of every successive element is obtained by adding VS to the address of the previous element.

All of the instructions in this group may be performed under mask (multiprocessing C Series CPUs only).

Vector reduction and recursion

Reduction operations reduce a vector to a scalar. There are two inputs to a reduction operator: a scalar register and a vector register. A scalar input is provided so that reduction operations can be performed for vectors greater than 128 elements.

Mathematically, reduction operations are the sum reduction (`sum`) and multiply or product reduction (`prod`). Additional reduction operations are provided to implement the FORTRAN `max` and `min` intrinsics, as well as reduction using logical operators (for example, `and (all(.t|.f) Vk)`, `or (any(.t|.f) Vk)`, and `xor (parity(.t|.f) Vk)`).

These latter reduction operations are used for a certain class of pattern recognition algorithms. For these operations, the scalar register Sk must be initialized to the identity operand of the particular operation, as listed in Table 4.

Table 4
Identity operands for vector reduction operations

Reduction operation	Identity operand
`add`	0[1]
`multiply`	1[1]
`and`	all 1s
`or`	all 0s
`xor`	all 0s
`max`	smallest representable number
`min`	largest representable number

[1] These identity operands apply to both integer and floating point operands.

Other operations are provided to manipulate the vector mask register (VM). These take the form of reductions on VM to determine the number of binary ones or zeros.

Note

The order in which the vector elements are reduced is implementation specific.

All these instructions may be performed under mask (multiprocessing C Series CPUs), that is, under the control of the VM register.

The exception condition (overflow or underflow) will always be signaled in the PSW.

- The sign of an IEEE zero result is not guaranteed to be "correct". Either positive or negative zero may be returned.
- In addition, overflows in intermediate calculations which generate reserved operand results may cause additional error bits in the PSW to be set (ROP in particular).
- If the reserved operand bit in the PSW becomes set during a vector reduction, the result of the instruction is *unpredictable.*

Vector comparison and edit instructions

The instructions in this section perform comparisons between vectors and scalars, and perform manipulations using the vector merge (VM) register. The general methodology is for a comparison to produce a bit vector, where a 1 indicates that the comparison is true and a 0 indicates that the comparison is false. This method is functionally similar to comparisons with address and scalar registers. However, the results of a vector compare (and consequently the contents of the VM register) are used quite differently from manipulations on the A and S registers.

In most processors, many common operations performed on vectors (for example, `compress`, `mask`, and `merge`) require the use of branch instructions. A full set of primitives that implement `compress`, `mask`, `merge`, and `expand` operations on vector accumulators are provided to eliminate the use of branch instructions when manipulating vectors. These primitives permit vector operations to be implemented with a sequence of chained vector instructions (for example, vector clip, where every element greater than 5 is replaced with 5). Other typical operations using these instruction set primitives include operations on sparse vectors using a compress operation, the number and location of zero crossings, and the number of successful comparisons and sorts.

Vector compares

Floating point arithmetic is performed in both CONVEX native and IEEE formats. IEEE floating point operations are initiated by setting PSW (IEEE), bit<13>. Otherwise, operations are performed in native mode.

The only flag affected is PSW (RO). If RO is set, the entire contents of the VM register are *unpredictable.*

Three forms of compare operations are provided:

```
<= , < , and =
```

If the required relation is not one of the three provided, then the complement of the relation is used, along with the complement of the VM register.

The following identity defines a complementary relation

$$\overline{\{(V\ \text{relation}\ S)\}} = \{V\ \overline{(\text{relation})}\ S\}$$

which describes the following complements:

<=	>
<	>=
=	!=

On multiprocessing C Series CPUs, all vector comparisons may be performed under mask, that is including or ignoring elements of the vector based on the state of the corresponding VM bit. Vector operation under mask is discussed in the next section.

NOTE

Comparisons under mask use the VM register to control the masking operation *before* the comparison operation occurs.

Vector operation under mask

On multiprocessing C Series CPUs, most of the vector/scalar instructions have the capability of operating under mask. A vector merge (VM) register bit is 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. A masked operand cannot cause an exception (for example, a reserved operand) or affect the processor status word (PSW) in any way.

In this mode, the bit of the VM (vector merge or vector mask) register corresponding to each vector element is examined to either enable or disable that vector element from the operation. The least significant bit of VM corresponds to vector element 0, up to the most significant bit which corresponds to vector element 127.

The operate-under-mask instructions use the extended op code format described earlier in this chapter. These instructions are created such that a prefix of `7EF0` hexadecimal indicates the following op code operates under false mask. A prefix of `7EF8` indicates the op code operates under true mask.

There are two forms of operate-under-mask:

- **True**—Elements with VM bit = 1 are included. Instructions of this type have a `.t` suffix
- **False**—Elements with VM bit = 0 are included. Instructions of this type have a `.f` suffix

For example, `add.w V0,V1,V2` adds all elements (restricted by vector length) of V0 and V1, placing the results in V2. The `add.w.t V0,V1,V2` version adds only those elements with corresponding VM bits that are 1. Elements of V2 corresponding to VM bits that are 0 remain unmodified. The opposite polarity of VM bits is used for `.f`, that is `add.w.f V0,V1,V2` operates only on those vector elements with corresponding VM bits that are 0.

The following are examples of operations under mask, assuming these values before instruction execution:

```
V0[0..5] = 0 1 2 3 4 5      VL = 6
V1[0..5] = 6 7 8 9 2 3      VM<5..0> = 1 0 0 1 1 0
V2[0..5] = 5 5 5 5 5 5
```

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

```
V2[0..5] = 5 8 10 5 5 8
```

In this example, performing an `add.w.f V0,V1,V2` produces the following:

```
V2[0..5] = 6 5 5 12 6 5
```

The following code sequence is an example of using operations under mask:

```
for (i = 1; i <= 100; i++) {
    if (a[i] == b[i]) {
        c[i] = d[i];
    }
}
```

Ignoring the length and stride setup, the preceding code sequence can be vectorized with the following sequence:

```
ld.w   a,v0   ;v0 = a
ld.w   b,v1  ;v1 = b
eq.w  v0,v2 ;sets VM bits = 1 where a equals b
ld.w.t d,v2  ;load from d only where a equals b
st.w.t v2,c  ;store to c only where a equals b
```

Vector `merge`, `mask`, `compress`, and `expand`

Both `mask` and `merge` instructions take two input operands and produce a third operand as the result. These operands are referred to as Vi, Rj, and Vk, where Vk is the output. Rj may be either a vector or a scalar register.

The `merge` and `mask` instructions differ only in the way in which the indices are used to create the result vector. For the `merge` instruction, the indices of Vi and Rj are only incremented if that particular register is selected by VM. For the `mask` instruction, element n of Vk is either element n of Vi or element n of Rj.

Note

The `mask` instruction in this section should not be confused with operations-under-mask described earlier.

The `compress` instruction uses the VM register to extract elements selectively from one vector register and place them in another vector register. Either zeros or ones of VM may be used by specifying the `.f` (false) or `.t` (true) version of the instruction, respectively. Only those elements with corresponding VM bit set (clear for `.f`) are moved from the source vector to the destination vector. This creates a destination vector with a number of elements equal to the number of bits set (or cleared) in VM.

The `expand` instruction uses the VM register (multiprocessing C Series CPUs) to extract elements from one vector register and place them selectively in another. Either zeros or ones of VM may be used by specifying the `.f` or `.t` (false or true) version of the instruction, respectively. Elements from the source vector are moved to the destination vector. Only those elements with corresponding VM bit set (clear for `.f`) are loaded in the destination vector. Other elements in the destination vector, corresponding to VM bits clear (set for `.f`), are skipped. This creates a destination vector with VL elements including a number of elements of the source vector equal to the number of bits set (or clear) in the VM vector.

These instructions are most easily described by some examples. First, a simple rule is presented. Each instruction has either a single true version, or both a `.t` (true) and a `.f` (false) version. This facility allows the user to utilize either the ones of VM (`.t`) or the zeros (`.f`). Thus, in the `.t` case, when the appropriate bit of VM is a 1, the Rj operand is selected.

The combinations of VM, `.t`, and `.f` are shown in Table 5.

Table 5
VM/operand combinations

Operand	VM	
	0	1
`.t`	Vi	Rj
`.f`	Rj	Vi

For the following examples, assume:

```
V0[0..5] = 1 2 3 4 5 6      VL = 6
V1[0..5] = a b c d e f      S1 = 8
V5[0..5] = 7 7 7 7 7 7      VM<5..0> = 1 0 0 1 1 0
```

Performing a `compress` on V0 produces the following:

```
cprs.t V0,V5 = 2 3 6 7 7 7
cprs.f V0,V5 = 1 4 5 7 7 7
```

Performing an `expand` on V0 produces the following:

```
xpnd.t V0,V5 = 7 1 2 7 7 3
xpnd.f V0,V5 = 1 7 7 2 3 7
```

Performing a `mask` of V0 and V1 produces the following:

```
mask.t V0,V1,V5 = 1 b c 4 5 f
mask.t V1,V0,V5 = a 2 3 d e 6
```

Performing a `mask` of V0 and S1 produces the following:

```
mask.t V0,S1,V5 = 1 8 8 4 5 8
```

For the following examples, assume:

```
VL = 12
VM<11..0> = 1 1 1 0 0 0 1 0 0 1 1 0
```

Performing a `merge` of V0 and V1 produces the following:

```
merg.t V0,V1,V5 = 1 a b 2 3 c 4 5 6 d e f
```

Performing a `merge` of V0 and S1 produces the following:

```
merg.t V0,S1,V5 = 1 8 8 2 3 8 4 5 6 8 8 8
merg.f V0,S1,V5 = 8 1 2 8 8 3 8 8 8 4 5 6
```

Table 6 lists the vector/scalar register instructions.
Table 7 lists the vector register instructions.

Table 6
Vector/scalar register instructions

Instruction	Description
`add.b Vi, Sj, Vk`	Add vector/scalar integer byte
`add.b.f Vi, Sj, Vk`	Add vector/scalar byte using VM
`add.b.t Vi, Sj, Vk`	Add vector/scalar byte using VM
`add.d Vi, Sj, Vk`	Add vector/scalar double float
`add.d.f Vi, Sj, Vk`	Add vector/scalar double using not VM
`add.d.t Vi, Sj, Vk`	Add vector/scalar double using VM
`add.h Vi Sj, Vk`	Add vector/scalar integer halfword
`add.h.f Vi Sj, Vk`	Add vector/scalar halfword using not VM
`add.h.t Vi Sj, Vk`	Add vector/scalar halfword using VM
`add.l Vi, Sj, Vk`	Add vector/scalar integer longword
`add.l.f Vi, Sj, Vk`	Add vector/scalar longword using not VM
`add.l.t Vi, Sj, Vk`	Add vector/scalar longword using VM
`add.s Vi, Sj, Vk`	Add vector/scalar single float
`add.s.f Vi, Sj, Vk`	Add vector/scalar single using not VM
`add.s.t Vi, Sj, Vk`	Add vector/scalar single using VM
`add.w Vi, Sj, Vk`	Add vector/scalar integer word
`add.w.f Vi, Sj, Vk`	Add vector/scalar word using not VM
`add.w.t Vi, Sj, Vk`	Add vector/scalar word using VM
`all Sk`	AND reduce a vector
`all Vk`	AND reduce a vector
`all.f Sk`	AND reduce a vector using not VM
`all.f Vk`	AND reduce a vector using not VM
`all.t Sk`	AND reduce a vector using VM
`all.t Vk`	AND reduce a vector using VM

Table 6 (continued)
Vector/scalar register instructions

Instruction	Description
`and Vi, Sj, Vk`	AND vector/scalar
`and.f Vi, Sj, Vk`	AND vector/scalar using not VM
`and.t Vi, Sj, Vk`	AND vector/scalar using VM
`any Sk`	OR reduce a vector
`any Vk`	OR reduce a vector
`any.f Sk`	OR reduce a vector using not VM
`any.f Vk`	OR reduce a vector using not VM
`any.t Sk`	OR reduce a vector using VM
`any.t Vk`	OR reduce a vector using VM
`div.b Vi, Sj, Vk`	Divide vector/scalar integer byte
`div.b.f Vi, Sj, Vk`	Divide vector/scalar byte using not VM
`div.b.t Vi, Sj, Vk`	Divide vector/scalar byte using VM
`div.d Si, Vj, Vk`	Divide scalar/vector double float
`div.d Vi, Sj, Vk`	Divide vector/scalar double float
`div.d.f Si, Vj, Vk`	Divide scalar/vector double using not VM
`div.d.f Vi, Sj, Vk`	Divide vector/scalar double using not VM
`div.d.t Vi, Sj, Vk`	Divide vector/scalar double using VM
`div.d.t Si, Vj, Vk`	Divide scalar/vector double using VM
`div.h Vi, Sj, Vk`	Divide vector/scalar integer halfword
`div.h.f Vi, Sj, Vk`	Divide vector/scalar halfword using not VM
`div.h.t Vi, Sj, Vk`	Divide vector/scalar halfword using VM
`div.l Vi, Sj, Vk`	Divide vector/scalar integer longword
`div.l.f Vi, Sj, Vk`	Divide vector/scalar longword using not VM

Table 6 (continued)
Vector/scalar register instructions

Instruction	Description
`div.l.t Vi, Sj, Vk`	Divide vector/scalar longword using VM
`div.s Si, Vj, Vk`	Divide scalar/vector single float
`div.s Vi, Sj, Vk`	Divide vector/scalar single float
`div.s.f Si, Vj, Vk`	Divide scalar/vector single using not VM
`div.s.f Vi, Sj, Vk`	Divide vector/scalar single using not VM
`div.s.t Si, Vj, Vk`	Divide scalar/vector single using VM
`div.s.t Vi, Sj, Vk`	Divide vector/scalar single using VM
`div.w Vi, Sj, Vk`	Divide vector/scalar integer word
`div.w.f Vi, Sj, Vk`	Divide vector/scalar word using not VM
`div.w.t Vi, Sj, Vk`	Divide vector/scalar word using VM
`eq.b Sj, Vk`	Compare equal byte
`eq.b.f Sj, Vk`	Compare equal byte using not VM
`eq.b.t Sj, Vk`	Compare equal byte using VM
`eq.d Sj, Vk`	Compare equal double-precision
`eq.d.f Sj, Vk`	Compare equal double using not VM
`eq.d.t Sj, Vk`	Compare equal double using VM
`eq.h Sj, Vk`	Compare equal halfword
`eq.h.f Sj, Vk`	Compare equal halfword using not VM
`eq.h.t Sj, Vk`	Compare equal halfword using VM
`eq.l Sj, Vk`	Compare equal longword
`eq.l.f Sj, Vk`	Compare equal long using not VM
`eq.l.t Sj, Vk`	Compare equal long using VM
`eq.s Sj, Vk`	Compare equal single
`eq.s.f Sj, Vk`	Compare equal single using not VM
`eq.s.t Sj, Vk`	Compare equal single using VM

Table 6 (continued)
Vector/scalar register instructions

Instruction	Description
`eq.w Sj, Vk`	Compare equal word
`eq.w.f Sj, Vk`	Compare equal word using not VM
`eq.w.t Sj, Vk`	Compare equal word using VM
`le.b Sj, Vk`	Compare less than or equal byte
`le.b.f Sj, Vk`	Compare less than or equal byte using not VM
`le.b.t Sj, Vk`	Compare less than or equal byte using VM
`le.d Sj, Vk`	Compare less than or equal double float
`le.d.f Sj, Vk`	Compare less than or equal double using not VM
`le.d.t Sj, Vk`	Compare less than or equal double using VM
`le.h Sj, Vk`	Compare less than or equal halfword
`le.h.f Sj, Vk`	Compare less than or equal half using not VM
`le.h.t Sj, Vk`	Compare less than or equal half using VM
`le.l Sj, Vk`	Compare less than or equal longword
`le.l.f Sj, Vk`	Compare less than or equal long using not VM
`le.l.t Sj, Vk`	Compare less than or equal long using VM
`le.s Sj, Vk`	Compare less than or equal single
`le.s.f Sj, Vk`	Compare less than or equal single using not VM
`le.s.t Sj, Vk`	Compare less than or equal single using VM
`le.w Sj, Vk`	Compare less than or equal word
`le.w.f Sj, Vk`	Compare less than or equal word using not VM

Table 6 (continued)
Vector/scalar register instructions

Instruction	Description
`le.w.t Sj, Vk`	Compare less than or equal word using VM
`lt.b Sj, Vk`	Compare less than byte
`lt.b.f Sj, Vk`	Compare less than byte using not VM
`lt.b.t Sj, Vk`	Compare less than byte using VM
`lt.d Sj, Vk`	Compare less than double float
`lt.d.f Sj, Vk`	Compare less than double using not VM
`lt.d.t Sj, Vk`	Compare less than double using VM
`lt.h Sj, Vk`	Compare less than halfword
`lt.h.f Sj, Vk`	Compare less than halfword using not VM
`lt.h.t Sj, Vk`	Compare less than halfword using VM
`lt.l Sj, Vk`	Compare less than longword
`lt.l.f Sj, Vk`	Compare less than long using not VM
`lt.l.t Sj, Vk`	Compare less than long using VM
`lt.s Sj, Vk`	Compare less than single
`lt.s.f Sj, Vk`	Compare less than single using not VM
`lt.s.t Sj, Vk`	Compare less than single using VM
`lt.w Sj, Vk`	Compare less than word
`lt.w.f Sj, Vk`	Compare less than word using not VM
`lt.w.t Sj, Vk`	Compare less than word using VM
`mask.f Vi, Sj, Vk`	Mask vector/scalar using not VM
`mask.t Vi, Sj, Vk`	Mask vector/scalar using VM
`max.b Vk`	Max of a vector of bytes
`max.b.f Vk`	Max of vector of bytes using not VM
`max.b.t Vk`	Max of vector of bytes using VM
`max.d Vk`	Max of a vector of double float

Table 6 (continued)
Vector/scalar register instructions

Instruction	Description
`max.d.f Vk`	Max of vector of doubles using not VM
`max.d.t Vk`	Max of vector of doubles using VM
`max.h Vk`	Max of a vector of halfwords
`max.h.f Vk`	Max of vector of halfwords using not VM
`max.h.t Vk`	Max of vector of halfwords using VM
`max.l Vk`	Max of a vector of longwords
`max.l.f Vk`	Max of vector of longwords using not VM
`max.l.t Vk`	Max of vector of longwords using VM
`max.s Vk`	Max of a vector of single float
`max.s.f Vk`	Max of vector of singles using not VM
`max.s.t Vk`	Max of vector of singles using VM
`max.w Vk`	Max of a vector of words
`max.w.f Vk`	Max of vector of words using not VM
`max.w.t Vk`	Max of vector of words using VM
`merg.f Vi, Sj, Vk`	Merge vector/scalar using not VM
`merg.t Vi, Sj, Vk`	Merge vector/scalar using VM
`min.b Vk`	Min of a vector of bytes
`min.b.f Vk`	Min of vector of bytes using not VM
`min.b.t Vk`	Min of vector of bytes using VM
`min.d Vk`	Min of a vector of double float
`min.d.f Vk`	Min of vector of doubles using not VM
`min.d.t Vk`	Min of vector of doubles using VM
`min.h Vk`	Min of a vector of halfwords
`min.h.f Vk`	Min of vector of halfwords using not VM
`min.h.t Vk`	Min of vector of halfwords using VM

Table 6 (continued)
Vector/scalar register instructions

Instruction	Description
`min.l Vk`	Min of a vector of longwords
`min.l.f Vk`	Min of vector of longwords using not VM
`min.l.t Vk`	Min of vector of longwords using VM
`min.s Vk`	Min of a vector of single float
`min.s.f Vk`	Min of vector of singles using not VM
`min.s.t Vk`	Min of vector of singles using VM
`min.w Vk`	Min of a vector of words
`min.w.f Vk`	Min of vector of words using not VM
`min.w.t Vk`	Min of vector of words using VM
`mov VMU, Sk`	Load Sk from VM<127..64>
`mov VML, Sk`	Load Sk from VM<63..0>
`mov Sj, Sk, VM`	Load VM(Sj) from Sk
`mov Sj, VM, Sk`	Load Sk from VM(Sj)
`mov Vi, Sj, Sk`	Move a vector element to a scalar
`mov Si, Sj, Vk`	Move a scalar to a vector element
`mov.w Sk, VS`	Move Sk to VS
`mov.w VS, Sk`	Move VS to Sk
`mov.w VL, Sk`	Move VL to Sk
`mov.w Sk, VL`	Move Sk to VL
`mul.b Vi, Sj, Vk`	Multiply vector/scalar integer byte
`mul.b.f Vi, Sj, Vk`	Multiply vector/scalar byte using not VM
`mul.b.t Vi, Sj, Vk`	Multiply vector/scalar byte using VM
`mul.d Vi, Sj, Vk`	Multiply vector/scalar double float
`mul.d.f Vi, Sj, Vk`	Multiply vector/scalar double using not VM
`mul.d.t Vi, Sj, Vk`	Multiply vector/scalar double using VM

Table 6 (continued)
Vector/scalar register instructions

Instruction	Description
`mul.h Vi, Sj, Vk`	Multiply vector/scalar integer halfword
`mul.h.f Vi, Sj, Vk`	Multiply vector/scalar halfword using not VM
`mul.h.t Vi, Sj, Vk`	Multiply vector/scalar halfword using VM
`mul.l Vi, Sj, Vk`	Multiply vector/scalar integer longword
`mul.l.f Vi, Sj, Vk`	Multiply vector/scalar longword using not VM
`mul.l.t Vi, Sj, Vk`	Multiply vector/scalar longword using VM
`mul.s Vi, Sj, Vk`	Multiply vector/scalar single float
`mul.s.f Vi, Sj, Vk`	Multiply vector/scalar single using not VM
`mul.s.t Vi, Sj, Vk`	Multiply vector/scalar single using VM
`mul.w Vi, Sj, Vk`	Multiply vector/scalar integer word
`mul.w.f Vi, Sj, Vk`	Multiply vector/scalar word using not VM
`mul.w.t Vi, Sj, Vk`	Multiply vector/scalar word using VM
`or Vi, Sj, Vk`	OR vector/scalar
`or.f Vi, Sj, Vk`	OR vector/scalar using not VM
`or.t Vi, Sj, Vk`	OR vector/scalar using VM
`parity Vk`	Exclusive OR reduce a vector
`parity.f Vk`	Exclusive OR reduce vector using not VM
`parity.t Vk`	Exclusive OR reduce vector using VM
`plc.f VM, Sk`	Load the number of zeros in VM into Sk
`plc.t VM, Sk`	Load the number of ones in VM into Sk

Table 6 (continued)
Vector/scalar register instructions

Instruction	Description
`prod.b Vk`	Multiply reduce a vector of bytes
`prod.b.f Vk`	Multiply reduce byte vector using not VM
`prod.b.t Vk`	Multiply reduce byte vector using VM
`prod.d Vk`	Multiply reduce a vector of double float
`prod.d.f Vk`	Multiply reduce double vector using not VM
`prod.d.t Vk`	Multiply reduce double vector using VM
`prod.h Vk`	Multiply reduce a vector of halfwords
`prod.h.f Vk`	Multiply reduce halfword vector using not VM
`prod.h.t Vk`	Multiply reduce halfword vector using VM
`prod.l Vk`	Multiply reduce a vector of longwords
`prod.l.f Vk`	Multiply reduce longword vector using not VM
`prod.l.t Vk`	Multiply reduce longword vector using VM
`prod.s Vk`	Multiply reduce a vector of single float
`prod.s.f Vk`	Multiply reduce single vector using not VM
`prod.s.t Vk`	Multiply reduce single vector using VM
`prod.w Vk`	Multiply reduce a vector of words
`prod.w.f Vk`	Multiply reduce word vector using not VM
`prod.w.t Vk`	Multiply reduce word vector using VM
`shf Sj, Vk`	Shift a vector accumulator

Table 6 (continued)
Vector/scalar register instructions

Instruction	Description
`shf Vi, Sj, Vk`	Shift a vector accumulator
`shf.f Sj, Vk`	Shift vector/scalar using not VM
`shf.f Vi, Sj, Vk`	Shift vector/scalar using not VM
`shf.t Sj, Vk`	Shift vector/scalar using VM
`shf.t Vi, Sj, Vk`	Shift vector/scalar using VM
`stvi.b Sk, Vj`	Scalar index store vector byte
`stvi.b.f Sk, Vj`	Scalar index store vector byte using not VM
`stvi.b.t Sk, Vj`	Scalar index store vector byte using VM
`stvi.d Sk, Vj`	Scalar index store vector double float
`stvi.d.f Sk, Vj`	Scalar index store vector double using not VM
`stvi.d.t Sk, Vj`	Scalar index store vector double using VM
`stvi.h Sk, Vj`	Scalar index store vector halfword
`stvi.h.f Sk, Vj`	Scalar index store vector half using not VM
`stvi.h.t Sk, Vj`	Scalar index store vector half using VM
`stvi.l Sk, Vj`	Scalar index store vector longword
`stvi.l.f Sk, Vj`	Scalar index store vector long using not VM
`stvi.l.t Sk, Vj`	Scalar index store vector long using VM
`stvi.s Sk, Vj`	Scalar index store vector single float
`stvi.s.f Sk, Vj`	Scalar index store vector single using not VM
`stvi.s.t Sk, Vj`	Scalar index store vector single using VM
`stvi.w Sk, Vj`	Scalar index store vector word

Table 6 (continued)
Vector/scalar register instructions

Instruction	Description
`stvi.w.f Sk, Vj`	Scalar index store vector word using not VM
`stvi.w.t Sk, Vj`	Scalar index store vector word using VM
`sub.b Vi, Sj, Vk`	Subtract vector/scalar integer byte
`sub.b.f Vi, Sj, Vk`	Subtract vector/scalar byte using not VM
`sub.b.t Vi, Sj, Vk`	Subtract vector/scalar byte using VM
`sub.d Si, Vj, Vk`	Subtract scalar/vector double float
`sub.d Vi, Sj, Vk`	Subtract vector/scalar double float
`sub.d.f Si, Vj, Vk`	Subtract scalar/vector double using not VM
`sub.d.f Vi, Sj, Vk`	Subtract vector/scalar double using not VM
`sub.d.t Si, Vj, Vk`	Subtract scalar/vector double using VM
`sub.d.t Vi, Sj, Vk`	Subtract vector/scalar double using VM
`sub.h Vi, Sj, Vk`	Subtract vector/scalar integer halfword
`sub.h.f Vi, Sj, Vk`	Subtract vector/scalar halfword using not VM
`sub.h.t Vi, Sj, Vk`	Subtract vector/scalar halfword using VM
`sub.l Vi, Sj, Vk`	Subtract vector/scalar integer longword
`sub.l.f Vi, Sj, Vk`	Subtract vector/scalar longword using not VM
`sub.l.t Vi, Sj, Vk`	Subtract vector/scalar longword using VM
`sub.s Si, Vj, Vk`	Subtract scalar/vector single float
`sub.s Vi, Sj, Vk`	Subtract vector/scalar single float
`sub.s.f Si, Vj, Vk`	Subtract scalar/vector single using not VM

Table 6 (continued)
Vector/scalar register instructions

Instruction	Description
`sub.s.f Vi, Sj, Vk`	Subtract vector/scalar single using not VM
`sub.s.t Si, Vj, Vk`	Subtract scalar/vector single using VM
`sub.s.t Vi, Sj, Vk`	Subtract vector/scalar single using VM
`sub.w Vi, Sj, Vk`	Subtract vector/scalar integer word
`sub.w.f Vi, Sj, Vk`	Subtract vector/scalar word using not VM
`sub.w.t Vi, Sj, Vk`	Subtract vector/scalar word using VM
`sum.b Vk`	Sum a vector of bytes
`sum.b.f Vk`	Sum a vector of bytes using not VM
`sum.b.t Vk`	Sum a vector of bytes using VM
`sum.d Vk`	Sum a vector of double float
`sum.d.f Vk`	Sum a vector of double using not VM
`sum.d.t Vk`	Sum a vector of double using VM
`sum.h Vk`	Sum a vector of halfwords
`sum.h.f Vk`	Sum a vector of halfwords using not VM
`sum.h.t Vk`	Sum a vector of halfwords using VM
`sum.l Vk`	Sum a vector of longwords
`sum.l.f Vk`	Sum a vector of longwords using not VM
`sum.l.t Vk`	Sum a vector of longwords using VM
`sum.s Vk`	Sum a vector of single float
`sum.s.f Vk`	Sum a vector of single using not VM
`sum.s.t Vk`	Sum a vector of single using VM
`sum.w Vk`	Sum a vector of words
`sum.w.f Vk`	Sum a vector of words using not VM
`sum.w.t Vk`	Sum a vector of words using VM

Table 6 (continued)
Vector/scalar register instructions

Instruction	Description
`xor Vi, Sj, Vk`	Exclusive OR vector/scalar
`xor.f Vi, Sj, Vk`	Exclusive OR vector/scalar using not VM
`xor.t Vi, Sj, Vk`	Exclusive OR vector/scalar using VM

Table 7
Vector register instructions

Instruction	Description
`add.b Vi, Vj, Vk`	Add vector/vector integer byte
`add.b.f Vi, Vj, Vk`	Add vector/vector byte using not VM
`add.b.t Vi, Vj, Vk`	Add vector/vector byte using VM
`add.d Vi, Vj, Vk`	Add vector/vector double float
`add.d.f Vi, Vj, Vk`	Add vector/vector double using not VM
`add.d.t Vi, Vj, Vk`	Add vector/vector double using VM
`add.h Vi, Vj, Vk`	Add vector/vector integer halfword
`add.h.f Vi, Vj, Vk`	Add vector/vector halfword using not VM
`add.h.t Vi, Vj, Vk`	Add vector/vector halfword using VM
`add.l Vi, Vj, Vk`	Add vector/vector integer longword
`add.l.f Vi, Vj, Vk`	Add vector/vector longword using not VM
`add.l.t Vi, Vj, Vk`	Add vector/vector longword using VM
`add.s Vi, Vj, Vk`	Add vector/vector single float
`add.s.f Vi, Vj, Vk`	Add vector/vector single using not VM
`add.s.t Vi, Vj, Vk`	Add vector/vector single using VM
`add.w Vi, Vj, Vk`	Add vector/vector integer word
`add.w.f Vi, Vj, Vk`	Add vector/vector word using not VM
`add.w.t Vi, Vj, Vk`	Add vector/vector word using VM
`and Vi, Vj, Vk`	AND two vectors
`add.f Vi, Vj, Vk`	AND two vectors using not VM
`add.t Vi, Vj, Vk`	AND two vectors using VM
`cprs.f Vj, Vk`	Compress a vector using not VM
`cprs.t Vj, Vk`	Compress a vector using VM
`cvtb.w Vj, Vk`	Convert byte to word

Table 7 (continued)
Vector register instructions

Instruction	Description
`cvtb.w.f Vj, Vk`	Convert byte to word using not VM
`cvtb.w.t Vj, Vk`	Convert byte to word using VM
`cvtd.l Vj, Vk`	Convert double float to longword
`cvtd.l.f Vj, Vk`	Convert double to longword using not VM
`cvtd.l.t Vj, Vk`	Convert double to longword using VM
`cvtd.s Vj, Vk`	Convert double float to single float
`cvtd.s.f Vj, Vk`	Convert double to single using not VM
`cvtd.s.t Vj, Vk`	Convert double to single using VM
`cvtd.w Vj, Vk`	Convert double to word
`cvtd.w.f Vj, Vk`	Convert double to word using not VM
`cvtd.w.t Vj, Vk`	Convert double to word using VM
`cvth.w Vj, Vk`	Convert halfword to word
`cvth.w.f Vj, Vk`	Convert halfword to word using not VM
`cvth.w.t Vj, Vk`	Convert halfword to word using VM
`cvtl.d Vj, Vk`	Convert longword to double float
`cvtl.d.f Vj, Vk`	Convert longword to double using not VM
`cvtl.d.t Vj, Vk`	Convert longword to double using VM
`cvtl.s Vj, Vk`	Convert longword to single float
`cvtl.s.f Vj, Vk`	Convert longword to single using not VM
`cvtl.s.t Vj, Vk`	Convert longword to single using VM
`cvtl.w Vj, Vk`	Convert longword to word
`cvtl.w.f Vj, Vk`	Convert longword to word using not VM
`cvtl.w.t Vj, Vk`	Convert longword to word using VM

Table 7 (continued)
Vector register instructions

Instruction	Description
`cvts.d Vj, Vk`	Convert single float to double float
`cvts.d.f Vj, Vk`	Convert single to double using not VM
`cvts.d.t Vj, Vk`	Convert single to double using VM
`cvts.l Vj, Vk`	Convert single float to longword
`cvts.l.f Vj, Vk`	Convert single to longword using not VM
`cvts.l.t Vj, Vk`	Convert single to longword using VM
`cvts.w Vj, Vk`	Convert single float to word
`cvts.w.f Vj, Vk`	Convert single to word using not VM
`cvts.w.t Vj, Vk`	Convert single to word using VM
`cvtw.b Vj, Vk`	Convert word to byte
`cvtw.b.f Vj, Vk`	Convert word to byte using not VM
`cvtw.b.t Vj, Vk`	Convert word to byte using VM
`cvtw.d Vj, Vk`	Convert word to double
`cvtw.d.f Vj, Vk`	Convert word to double using not VM
`cvtw.d.t Vj, Vk`	Convert word to double using VM
`cvtw.h Vj, Vk`	Convert word to halfword
`cvtw.h.f Vj, Vk`	Convert word to halfword using not VM
`cvtw.h.t Vj, Vk`	Convert word to halfword using VM
`cvtw.l Vj, Vk`	Convert word to longword
`cvtw.l.f Vj, Vk`	Convert word to longword using not VM
`cvtw.l.t Vj, Vk`	Convert word to longword using VM
`cvtw.s Vj, Vk`	Convert word to single float
`cvtw.s.f Vj, Vk`	Convert word to single using not VM
`cvtw.s.t Vj, Vk`	Convert word to single using VM
`div.b Vi, Vj, Vk`	Divide vector/vector integer byte

Table 7 (continued)
Vector register instructions

Instruction	Description
`div.b.f Vi, Vj, Vk`	Divide byte vectors using not VM
`div.b.t Vi, Vj, Vk`	Divide byte vectors using VM
`div.d Vi, Vj, Vk`	Divide vector/vector double float
`div.d.f Vi, Vj, Vk`	Divide double vectors using not VM
`div.d.t Vi, Vj, Vk`	Divide double vectors using VM
`div.h Vi, Vj, Vk`	Divide vector/vector integer halfword
`div.h.f Vi, Vj, Vk`	Divide halfword vectors using not VM
`div.h.t Vi, Vj, Vk`	Divide halfword vectors using VM
`div.l Vi, Vj, Vk`	Divide vector/vector integer longword
`div.l.f Vi, Vj, Vk`	Divide longword vectors using not VM
`div.l.t Vi, Vj, Vk`	Divide longword vectors using VM
`div.s Vi, Vj, Vk`	Divide vector/vector single float
`div.s.f Vi, Vj, Vk`	Divide single vectors using not VM
`div.s.t Vi, Vj, Vk`	Divide single vectors using VM
`div.w Vi, Vj, Vk`	Divide vector/vector integer word
`div.w.f Vi, Vj, Vk`	Divide word vectors using not VM
`div.w.t Vi, Vj, Vk`	Divide word vectors using VM
`eq.b Vj, Vk`	Compare equal byte
`eq.b.f Vj, Vk`	Compare equal byte using not VM
`eq.b.t Vj, Vk`	Compare equal byte using VM
`eq.d Vj, Vk`	Compare equal double-precision
`eq.d.f Vj, Vk`	Compare equal double using not VM
`eq.d.t Vj, Vk`	Compare equal double using VM
`eq.h Vj, Vk`	Compare equal halfword
`eq.h.f Vj, Vk`	Compare equal halfword using not VM

Table 7 (continued)
Vector register instructions

Instruction	Description
`eq.h.t Vj, Vk`	Compare equal halfword using VM
`eq.l Vj, Vk`	Compare equal longword
`eq.l.f Vj, Vk`	Compare equal long using not VM
`eq.l.t Vj, Vk`	Compare equal long using VM
`eq.s Vj, Vk`	Compare equal single
`eq.s.f Vj, Vk`	Compare equal single using not VM
`eq.s.t Vj, Vk`	Compare equal single using VM
`eq.w Vj, Vk`	Compare equal word
`eq.w.f Vj, Vk`	Compare equal word using not VM
`eq.w.t Vj, Vk`	Compare equal word using VM
`frint.d Vj, Vk`	Integerize float double vector
`frint.d.f Vj, Vk`	Integerize double vector using not VM
`frint.d.t Vj, Vk`	Integerize double vector using VM
`frint.s Vj, Vk`	Integerize float single vector
`frint.s.f Vj, Vk`	Integerize single vector using not VM
`frint.s.t Vj, Vk`	Integerize single vector using VM
`ld.b` *effa*`, Vk`	Load vector byte
`ld.b.f` *effa*`, Vk`	Load vector byte using not VM
`ld.b.t` *effa*`, Vk`	Load vector byte using VM
`ld.d` *effa*`, Vk`	Load vector double float
`ld.d.f` *effa*`, Vk`	Load vector double float using not VM
`ld.d.t` *effa*`, Vk`	Load vector double float using VM
`ld.h` *effa*`, Vk`	Load vector halfword
`ld.h.f` *effa*`, Vk`	Load vector halfword using not VM
`ld.h.t` *effa*`, Vk`	Load vector halfword using VM
`ld.l` *effa*`, VLS`	Load VS and VL from memory
`ld.l` *effa*`, Vk`	Load vector longword

Table 7 (continued)
Vector register instructions

Instruction	Description
`ld.l.f` *effa*`, Vk`	Load vector longword using not VM
`ld.l.t` *effa*`, Vk`	Load vector longword using VM
`ld.s` *effa*`, Vk`	Load vector single float
`ld.s.f` *effa*`, Vk`	Load vector single float using not VM
`ld.s.t` *effa*`, Vk`	Load vector single float using VM
`ld.w #N, VL`	Load VL with an immediate
`ld.w #N, VS`	Load VS from an immediate
`ld.w` *effa*`, Vk`	Load vector word
`ld.w.f` *effa*`, Vk`	Load vector word using not VM
`ld.w.t` *effa*`, Vk`	Load vector word using VM
`ld.x` *effa*`, VM`	Load VM from memory
`ldvi.b Vj, Vk`	Index load vector byte
`ldvi.b.f Vj, Vk`	Index load vector byte using not VM
`ldvi.b.t Vj, Vk`	Index load vector byte using VM
`ldvi.d Vj, Vk`	Index load vector double float
`ldvi.d.f Vj, Vk`	Index load vector double using not VM
`ldvi.d.t Vj, Vk`	Index load vector double using VM
`ldvi.h Vj, Vk`	Index load vector halfword
`ldvi.h.f Vj, Vk`	Index load vector halfword using not VM
`ldvi.h.t Vj, Vk`	Index load vector halfword using VM
`ldvi.l Vj, Vk`	Index load vector longword
`ldvi.l.f Vj, Vk`	Index load vector longword using not VM
`ldvi.l.t Vj, Vk`	Index load vector longword using VM
`ldvi.s Vj, Vk`	Index load vector single float
`ldvi.s.f Vj, Vk`	Index load vector single using not VM

Table 7 (continued)
Vector register instructions

Instruction	Description
`ldvi.s.t Vj, Vk`	Index load vector single using VM
`ldvi.w Vj, Vk`	Index load vector word
`ldvi.w.f Vj, Vk`	Index load vector word using not VM
`ldvi.w.t Vj, Vk`	Index load vector word using VM
`le.b Vj, Vk`	Compare less than or equal byte
`le.b.f Vj, Vk`	Compare less than or equal byte using not VM
`le.b.t Vj, Vk`	Compare less than or equal byte using VM
`le.d Vj, Vk`	Compare less than or equal double float
`le.d.f Vj, Vk`	Compare less than or equal double using not VM
`le.d.t Vj, Vk`	Compare less than or equal double using VM
`le.h Vj, Vk`	Compare less than or equal halfword
`le.h.f Vj, Vk`	Compare less than or equal half using not VM
`le.h.t Vj, Vk`	Compare less than or equal half using VM
`le.l Vj, Vk`	Compare less than or equal longword
`le.l.f Vj, Vk`	Compare less than or equal long using not VM
`le.l.t Vj, Vk`	Compare less than or equal long using VM
`le.s Vj, Vk`	Compare less than or equal single
`le.s.f Vj, Vk`	Compare less than or equal single using not VM
`le.s.t Vj, Vk`	Compare less than or equal single using VM
`le.w Vj, Vk`	Compare less than or equal word
`le.w.f Vj, Vk`	Compare less than or equal word using not VM

Table 7 (continued)
Vector register instructions

Instruction	Description
`le.w.t Vj, Vk`	Compare less than or equal word using VM
`lop Vj, Vk`	Leading ones position vector
`lop.f Vj, Vk`	Leading ones position vector using not VM
`lop.t Vj, Vk`	Leading ones position vector using VM
`lt.b Vj, Vk`	Compare less than byte
`lt.b.f Vj, Vk`	Compare less than byte using not VM
`lt.b.t Vj, Vk`	Compare less than byte using VM
`lt.d Vj, Vk`	Compare less than double float
`lt.d.f Vj, Vk`	Compare less than double using not VM
`lt.d.t Vj, Vk`	Compare less than double using VM
`lt.h Vj, Vk`	Compare less than halfword
`lt.h.f Vj, Vk`	Compare less than halfword using not VM
`lt.h.t Vj, Vk`	Compare less than halfword using VM
`lt.l Vj, Vk`	Compare less than longword
`lt.l.f Vj, Vk`	Compare less than long using not VM
`lt.l.t Vj, Vk`	Compare less than long using VM
`lt.s Vj, Vk`	Compare less than single
`lt.s.f Vj, Vk`	Compare less than single using not VM
`lt.s.t Vj, Vk`	Compare less than single using VM
`lt.w Vj, Vk`	Compare less than word
`lt.w.f Vj, Vk`	Compare less than word using not VM
`lt.w.t Vj, Vk`	Compare less than word using VM
`mask.t Vi, Vj, Vk`	Mask vector/vector

Table 7 (continued)
Vector register instructions

Instruction	Description
`merg.t Vi, Vj, Vk`	Merge vector/vector
`mov VS, Ak`	Move VS to Ak
`mov Ak, VS`	Move Ak to VS
`mov VL, Ak`	Move VL to Ak
`mov Ak, VL`	Move Ak to VL
`mul.b Vi, Vj, Vk`	Multiply vector/vector integer byte
`mul.b.f Vi, Vj, Vk`	Multiply byte vectors using not VM
`mul.b.t Vi, Vj, Vk`	Multiply byte vectors using VM
`mul.d Vi, Vj, Vk`	Multiply vector/vector double float
`mul.d.f Vi, Vj, Vk`	Multiply double vectors using not VM
`mul.d.t Vi, Vj, Vk`	Multiply double vectors using VM
`mul.h Vi, Vj, Vk`	Multiply vector/vector integer halfword
`mul.h.f Vi, Vj, Vk`	Multiply halfword vectors using not VM
`mul.h.t Vi, Vj, Vk`	Multiply halfword vectors using VM
`mul.l Vi, Vj, Vk`	Multiply vector/vector integer longword
`mul.l.f Vi, Vj, Vk`	Multiply longword vectors using not VM
`mul.l.t Vi, Vj, Vk`	Multiply longword vectors using VM
`mul.s Vi, Vj, Vk`	Multiply vector/vector single float
`mul.s.f Vi, Vj, Vk`	Multiply single vectors using not VM
`mul.s.t Vi, Vj, Vk`	Multiply single vectors using VM
`mul.w Vi, Vj, Vk`	Multiply vector/vector integer word
`mul.w.f Vi, Vj, Vk`	Multiply word vectors using not VM
`mul.w.t Vi, Vj, Vk`	Multiply word vectors using VM
`neg.b Vj, Vk`	Negate vector/vector integer byte
`neg.b.f Vj, Vk`	Negate byte vector using not VM

Table 7 (continued)
Vector register instructions

Instruction	Description
`neg.b.t Vj, Vk`	Negate byte vector using VM
`neg.d Vj, Vk`	Negate vector/vector double float
`neg.d.f Vj, Vk`	Negate double using not VM
`neg.d.t Vj, Vk`	Negate double using VM
`neg.h Vj, Vk`	Negate vector/vector integer halfword
`neg.h.f Vj, Vk`	Negate halfword using not VM
`neg.h.t Vj, Vk`	Negate halfword using VM
`neg.l Vj, Vk`	Negate vector/vector integer longword
`neg.l.f Vj, Vk`	Negate longword using not VM
`neg.l.t Vj, Vk`	Negate longword using VM
`neg.s Vj, Vk`	Negate vector/vector single float
`neg.s.f Vj, Vk`	Negate single using not VM
`neg.s.t Vj, Vk`	Negate single using VM
`neg.w Vj, Vk`	Negate vector/vector integer word
`neg.w.f Vj, Vk`	Negate word using not VM
`neg.w.t Vj, Vk`	Negate word using VM
`not Vj, Vk`	Complement a vector
`not.f Vj, Vk`	Complement a vector using not VM
`not.t Vj, Vk`	Complement a vector using VM
`or Vi, Vj, Vk`	OR two vectors
`or.f Vi, Vj, Vk`	OR two vectors using not VM
`or.t Vi, Vj, Vk`	OR two vectors using VM
`plc.t Vj, Vk`	Population count of a vector
`plc.t.f Vj, Vk`	Population count of vector using not VM
`plc.t.t Vj, Vk`	Population count of vector using VM
`shf Vi, Vj, Vk`	Shift vector/vector

Table 7 (continued)
Vector register instructions

Instruction	Description
`shf.f Vi, Vj, Vk`	Shift vector/vector using not VM
`shf.t Vi, Vj, Vk`	Shift vector/vector using VM
`st.b Vk, `*effa*	Store vector byte
`st.b.f Vk, `*effa*	Store vector byte using not VM
`st.b.t Vk, `*effa*	Store vector byte using VM
`st.d Vk, `*effa*	Store vector double float
`st.d.f Vk, `*effa*	Store vector double float using not VM
`st.d.t Vk, `*effa*	Store vector double float using VM
`st.h Vk, `*effa*	Store vector halfword
`st.h.f Vk, `*effa*	Store vector halfword using not VM
`st.h.t Vk, `*effa*	Store vector halfword using VM
`st.l VLS, `*effa*	Store VS and VL to memory
`st.l Vk, `*effa*	Store vector longword
`st.l.f Vk, `*effa*	Store vector longword using not VM
`st.l.t Vk, `*effa*	Store vector longword using VM
`st.s Vk, `*effa*	Store vector single float
`st.s.f Vk, `*effa*	Store vector single float using not VM
`st.s.t Vk, `*effa*	Store vector single float using VM
`st.w Vk, `*effa*	Store vector word
`st.w.f Vk, `*effa*	Store vector word using not VM
`st.w.t Vk, `*effa*	Store vector word using VM
`st.x VM, `*effa*	Store VM into memory
`stvi.b Vk, Vj`	Index store vector byte
`stvi.b.f Vk, Vj`	Index store vector byte using not VM
`stvi.b.t Vk, Vj`	Index store vector byte using VM
`stvi.d Vk, Vj`	Index store vector double float
`stvi.d.f Vk, Vj`	Index store vector double using not VM

Table 7 (continued)
Vector register instructions

Instruction	Description
`stvi.d.t Vk, Vj`	Index store vector double using VM
`stvi.h Vk, Vj`	Index store vector halfword
`stvi.h.f Vk, Vj`	Index store vector halfword using not VM
`stvi.h.t Vk, Vj`	Index store vector halfword using VM
`stvi.l Vk, Vj`	Index store vector longword
`stvi.l.f Vk, Vj`	Index store vector longword using not VM
`stvi.l.t Vk, Vj`	Index store vector longword using VM
`stvi.s Vk, Vj`	Index store vector single float
`stvi.s.f Vk, Vj`	Index store vector single using not VM
`stvi.s.t Vk, Vj`	Index store vector single using VM
`stvi.w Vk, Vj`	Index store vector word
`stvi.w.f Vk, Vj`	Index store vector word using not VM
`stvi.w.t Vk, Vj`	Index store vector word using VM
`sub.b Vi, Vj, Vk`	Subtract vector/vector integer byte
`sub.b.f Vi, Vj, Vk`	Subtract byte vectors using not VM
`sub.b.t Vi, Vj, Vk`	Subtract byte vectors using VM
`sub.d Vi, Vj, Vk`	Subtract vector/vector double float
`sub.d.f Vi, Vj, Vk`	Subtract double vectors using not VM
`sub.d.t Vi, Vj, Vk`	Subtract double vectors using VM
`sub.h Vi, Vj, Vk`	Subtract vector/vector integer halfword
`sub.h.f Vi, Vj, Vk`	Subtract halfword vectors using not VM
`sub.h.t Vi, Vj, Vk`	Subtract halfword vectors using VM
`sub.l Vi, Vj, Vk`	Subtract vector/vector integer longword

Table 7 (continued)
Vector register instructions

Instruction	Description
`sub.l.f Vi, Vj, Vk`	Subtract longword vectors using not VM
`sub.l.t Vi, Vj, Vk`	Subtract longword vectors using VM
`sub.s Vi, Vj, Vk`	Subtract vector/vector single float
`sub.s.f Vi, Vj, Vk`	Subtract single vectors using not VM
`sub.s.t Vi, Vj, Vk`	Subtract single vectors using VM
`sub.w Vi, Vj, Vk`	Subtract vector/vector integer word
`sub.w.f Vi, Vj, Vk`	Subtract word vectors using not VM
`sub.w.t Vi, Vj, Vk`	Subtract word vectors using VM
`tzc Vj, Vk`	Trailing zero count vector
`tzc.f Vj, Vk`	Trailing zero count vector using not VM
`tzc.t Vj, Vk`	Trailing zero count vector using VM
`xor Vi, Vj, Vk`	Exclusive OR two vectors
`xor.f Vi, Vj, Vk`	Exclusive OR two vectors using not VM
`xor.t Vi, Vj, Vk`	Exclusive OR two vectors using VM
`xpnd.f Vj, Vk`	Expand a vector using not VM
`xpnd.t Vj, Vk`	Expand a vector using VM

Communication register instructions

The instructions defined in this section manipulate the communication registers and resource structures.

Table 8 lists the communication register instructions.

Table 8
Communication register instructions

Instruction	Description
`casr`	Compare and swap resource
`get.l` *Ceffa*`, Sk`	Get communication/scalar
`getr.l` *effa*`, Sk`	Get resource/scalar
`get.w` *Ceffa*`, Ak`	Get communication/address
`getr.w` *effa*`, Ak`	Get resource/address
`inc.l` *Ceffa*`, Sk`	Increment communication/scalar
`incr.l` *effa*`, Sk`	Increment long resource structure
`inc.w` *Ceffa*`, Ak`	Increment communication/address
`incr.w` *effa*`, Ak`	Increment resource structure data
`lck` *Ceffa*	Lock communication register
`mat.l Sk,` *Ceffa*	Match scalar/communication
`matr.l Sk,` *effa*	Match scalar/resource
`mat.w Ak,` *Ceffa*	Match address/communication
`matr.w Ak,` *effa*	Match address/resource
`msync`	Synchronize stores to memory
`popr Ak,` *effa*	Pop resource/address register
`pshr Ak,` *effa*	Push address register/resource
`put.l Sk,` *Ceffa*	Put scalar/communication
`putr.l Sk,` *effa*	Put scalar/resource
`put.w Ak,` *Ceffa*	Put address/communication
`putr.w Ak,` *effa*	Put address/resource
`rcv.l` *Ceffa*`, Sk`	Receive communication/scalar
`rcv.w` *Ceffa*`, Ak`	Receive communication/address
`rcvr.l` *effa*`, Sk`	Receive scalar register/resource

Table 8 (continued)
Communication register instructions

Instruction	Description
`rcvr.w` *effa*`, Ak`	Receive address register/resource
`snd.l Sk,` *Ceffa*	Send scalar/communication
`snd.w Ak,` *Ceffa*	Send address/communication
`sndr.l Sk,` *effa*	Send scalar register/resource
`sndr.w Ak,` *effa*	Send address register/resource
`tst` *Ceffa*	Test communication register lock bit
`ulk` *Ceffa*	Unlock communication register

Process control instructions

The instructions defined in this section alter the program counter (PC). Alterations of the PC may occur as a result of a branch performed after a comparison, an unconditional jump, a subroutine call or return, or an operating system call or return. A return from interrupt processing is also defined.

`branch` and `jump` instructions

These instructions include unconditional `branch` and `jump`, `branch` and `jump` on PSW bits, and `breakpoints`. The branch instructions provide a way to branch a nominal distance relative to the present value of the PC. All `branch` instructions are 16 bits in length. `Jump` instructions load the PC with an effective address that is developed as an operand address. `Branch` and `jump` instructions do not affect the PSW.

The following sequence occurs when the PC increments to reference the next sequential instruction, or when a new value is loaded into the PC as a result of a `branch` or `jump` instruction:

1. The current ring is checked.
2. The PC is loaded appropriately. If the current ring is 4, bits <30..1> are loaded. Otherwise, bits <28..1> are loaded.

In all cases, the `branch or jump` is restricted to be within the current ring.

The `process breakpoint` instruction functions differently from the `breakpoint` instruction. Refer to the *CONVEX Architecture Reference Manual (C Series)*, Chapter 12, "Operating system exceptions", and the `pbkpt` instruction page in Chapter 3 of this manual for a more detailed description of process breakpoints.

`call`, `save`, and `return` instructions

This section describes the structures and instructions provided for `call`, `save`, and `return` instructions:

- An architecturally defined stack pointer (SP), argument pointer (AP), and frame pointer (FP) within the address register space.
- A set of `call` instructions:
  - `call`—Pushes all the scalar machine state. FRL in the saved (pushed) PSW determines how much machine state is pushed.
  - `calls`—Pushes minimal machine state. FRL in the saved (pushed) PSW determines how much machine state is pushed.
  - `callq`—Pushes the address of the next instruction and branches to the subroutine. Only the program counter is saved and restored. The PSW and frame pointer are unaltered. The `callq` instruction is used for local subroutines that do not require the definition of a new stack frame. The `rtnq` instruction pops the PC value on top of the stack.
- A set of `return` instructions that return process control and status from a subroutine call:
  - `rtn`—Return from a system call (`sysc`), call (`call`), or short call (`calls`)
  - `rtnq`—Return from a quick call (`callq`)
  - `rtnc`—Return from a context switch
- A set of instructions for stack operations.

Subroutine invocation sequences

The following instructions could be generated for the two most likely subroutine invocation sequences:

1. Precompiled arguments
   a. Load the effective address (`ldea`) of the address of the packet into the argument pointer (AP).
   b. Call the subroutine.
   c. Execute `rtn` when the subroutine completes execution.
   d. Load the former AP from the stack.
2. Arguments pushed onto the stack
   a. Push arguments onto the stack.
   b. Move stack pointer (SP) to AP.
   c. Call the subroutine.
   d. Execute `rtn` when the subroutine completes execution.
   e. Remove the pushed arguments from the stack by incrementing the stack pointer (SP).
   f. Load the former AP from the stack.

Table 9 lists the process control instructions.

Table 9
Process control instructions

Instruction	Description
`bkpt`	Breakpoint
`br`	Branch always
`bra.f`	Branch on address carry false
`bra.t`	Branch on address carry true
`bri.f`	Branch on ION false
`bri.t`	Branch on ION true
`brs.f`	Branch on scalar carry false
`brs.t`	Branch on scalar carry true
`call` *effa*	Call a subroutine, long frame
`callq` *effa*	Push the PC and jump
`calls` *effa*	Call a subroutine, short frame
`exit`	Error exit instruction
`jmp` *effa*	Jump always
`jmpa.f` *effa*	Jump on address carry false
`jmpa.t` *effa*	Jump on address carry true
`jmpi.f` *effa*	Jump on ION false
`jmpi.t` *effa*	Jump on ION true
`jmps.f` *effa*	Jump on scalar carry false
`jmps.t` *effa*	Jump on scalar carry true
`nop`	No operation (branch never)
`pbkpt`	Force process breakpoint exception
`rtn`	Return from subroutine call
`rtnq`	Pop the PC and jump
`sysc #r, #g`	Perform a system call
`trap #rm, #b`	Force a trap system exception

CPU control and status instructions

The instructions listed in this section dynamically allocate CPUs to a process. The memory and timer synchronization instructions are also included.

Table 10 lists the CPU control instructions.

Table 10
CPU control and status instructions

Instruction	Description
`cfork`	Clear a fork
`join`	Join all threads at a single execution point
`pfork` *effa*`, Ak`	Post a fork (need a CPU to join a process)
`spawn` *effa*`, Ak`	Spawn a fork (need all CPUs to join a process)
`stop`	Stop CPU clocks
`tac` *effa*	Test and clear a byte in memory
`tas` *effa*	Test and set a memory byte
`tstvv`	Test value of vector valid flag
`wfork`	Wait for a fork

Privileged control and status instructions

The instructions listed in this section are the privileged instruction set. Privileged instructions are used by the operating system (OS) kernel to control process multiplexing, virtual address space management, and system clock stores.

A privileged instruction is like any other instruction in the CONVEX instruction set except that the current ring of execution must be 0. This means that PSW<31..29> must be all zeros.

An attempt to execute a privileged instruction in a ring other than 0 results in a system exception and the generation of a system call through byte address `0x0C` of page 0 of Ring 0.

Other instructions included in this section generally are used by parts of the operating system contained in a ring owned by the system other than Ring 0 (Ring 1, 2, or 3).

Table 11 lists the privileged control and status instructions.

Table 11
Privileged control and status instructions

Instruction	Description
`ctrsg`	Move scalar to CPU timer
`diag Ak`	Execute nonstandard microcode sequence
`dsi`	Disable interrupts
`enag Sj, Sk`	Enable all global CPU interrupts
`enal Sj, Sk`	Enable local CPU interrupt
`eni`	Enable interrupts
`eni_idle Sk`	Enable interrupts and idle the CPU
`eni_rtn`	Enable interrupts and return from subroutine
`halt #N, Ak`	Halt the CPU
`idle`	Idle the CPU
`ldcmr` *effa*`, Ak`	Load communication registers
`ldkdr Ak`	Load all eight SDRs
`ldsdr Ak`	Load process SDRs

Table 11 (continued)
Privileged control and status instructions

Instruction	Description
`mov Sk, BE(Sj)`	Move scalar to a broadcast enable register
`mov Sk, CIR`	Move scalar to comm index register
`mov Sk, ICR`	Move scalar to interrupt control register
`mov Sk, TCPU`	Move scalar to target CPU Register
`mov Sk, VV`	Move scalar to vector valid flag
`mov Sk, ITR`	Move scalar to NITC, ITC, ITSR
`mov Sk, ITSR`	Move scalar to ITSR
`mov Sk, TID`	Move scalar to thread ID
`mov Sk, TOC`	Move scalar to TOC
`mov ITR, Sk`	Move the ITC, ITSR, NITC into scalar
`mov BE(Sj), Sk`	Move broadcast enable register to a scalar
`mski Sk`	Mask out interrupt
`pate Ak`	Purge an ATU entry
`patu`	Purge the entire ATU
`pich`	Purge the Icache
`plch`	Purge the Lcache
`pmod`	Purge modified bits
`pref`	Purge reference bits
`rtnc`	Return from a context block
`rtni`	Return from base level interrupt
`stcmr Ak, effa`	Store communication registers
`xmti Sk`	Transmit interrupt

Intrinsic instructions

The multiprocessing C Series implementation includes a group of instructions called *intrinsics*. These instructions are microcode and hardware implementations of some of the more frequently used runtime library routines. A `.s` (single-precision) and `.d` (double-precision) version of each instruction is available. Table 12 lists the arithmetic intrinsic instructions.

Table 12
Intrinsic instructions

Instruction	Description
`atan.d Sk`	Arc-tangent of a double-precision number
`atan.s Sk`	Arc-tangent of a single-precision number
`cos.d Sk`	Cosine of a double-precision number
`cos.s Sk`	Cosine of a single-precision number
`exp.d Sk`	Exponent of a double-precision number
`exp.s Sk`	Exponent of a single-precision number
`ln.d Sk`	Natural logarithm of a double-precision number
`ln.s Sk`	Natural logarithm of a single-precision number
`sin.d Sk`	Sine of a double-precision number
`sin.s Sk`	Sine of a single-precision number
`sqrt.d Sk`	Square root of a double-precision number
`sqrt.s Sk`	Square root of a single-precision number
`sqrt.d Vj, Vk`	Square root double vector/vector
`sqrt.d.f Vj, Vk`	Square root double using not VM
`sqrt.d.t Vj, Vk`	Square root double using VM
`sqrt.s Vj, Vk`	Square root single vector/vector
`sqrt.s.f Vj, Vk`	Square root single using not VM
`sqrt.s.t Vj, Vk`	Square root single using VM

All these instructions operate in either CONVEX native or IEEE floating point format, based on the state of PSW (IEEE). Some arithmetic exception conditions are raised by these instructions and are described in the sections concerning the PSW in the *CONVEX Architecture Reference Manual (C Series)*, Chapter 3, "General registers."

Instruction set (alphabetic listing)

3

This chapter is the reference for the CONVEX C Series architecture assembly language instruction set.

The C Series architecture applies to the CONVEX C1 and C120 supercomputers with its standard instruction set op codes.

The C Series architecture applies to the C200 and C3200 Series supercomputers with its standard and extended op codes.

The C Series architecture also applies to the C3400 Series and the C3800 Series supercomputers with its standard and extended op codes and some exclusive op codes. In a general sense, a reference to the application of the C Series architecture to the C200 Series CPUs includes its application to the C3400 and C3800 Series CPUs

The assembly language execution timing is described in detail in the *CONVEX Assembly Language Timing Guide.*

The generic page layout used to describe the instruction set appears on the following page. Following the page layout is an example of the "Op code" section and a detailed description of the C language-like pseudocode used in the "Operations" section of each command description.

command mnemonic
Command name

Purpose

The purpose or intent of the instruction.

Application

The architecture to which the instruction applies.

Format

The physical format of the instruction, including field locations and use. For standard op codes, no prefix is included. The prefix should be ignored when not applicable. For extended op codes, the prefix field is included, with the E flag (bit <3> of the prefix) used to distinguish between extended-0 space (E0) and extended-1 space (E1).

Some instructions have variants that are standard and extended. For example, `cvtw.s Sj,Sk` is a nonprefixed op code that appears on the same page as `cvtw.d Sj,Sk`, which is a prefixed op code. In the Op code section (described below) it is listed whether an op code is standard (nonprefixed—ST) or extended (prefixed—E0 or E1).

Op code subfields are listed here. See the "Op code descriptions" section of this chapter.

Op code

A listing of the instruction mnemonic, op code space mnemonic, hex op code, binary op code, PSW bits affected, and op code description (see example on next page).

Additional fixed subfields within the instruction are shown graphically in the Format section (described above).

Description

A description, in text, of the functions performed by the instruction.

Operation

A description, in C language-like pseudocode, of the functions performed by the instruction. See the "Instruction pseudocode descriptions" section of this chapter.

Exceptions

A list of any operating system exceptions that are detected as a result of setting the corresponding PSW trap enable bit or generated by the hardware in response to the error condition caused by the instruction's execution. Only those exceptions listed can occur.

Notes

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

Op code descriptions

The following is an example of the format of the "Op code" section:

Example:

Mnemonic	Space	Hex	Binary	PSW	Description
add.w Aj,Ak	ST	5840	0101100001	C,AIV	Add address registers word

add.w Aj,Ak	The op code mnemonic and assembly language syntax
ST	The op code space; this is a standard (nonprefixed) instruction
5840	The hexadecimal op code representation.
0101100001	The binary op code representation. See page 6 for an explanation of the binary tree processing for determining the total op code length (in bits) and the instruction length (in bytes) and format. In this example, the first three bits (010) correspond to the third format in Table 1. In this case, the subsequent 7 bits (1100001) complete the op code, and the format calls for two 3-bit fields (Aj and Ak) that identify the registers to operate on.
C,AIV	The PSW bits affected by this op code
Add address registers word	A brief description of each op code function

Instruction pseudocode descriptions

This section describes the definitions, usage, and syntax of the C language-like pseudocode used to describe the instructions. The following elements are examined:

- Comments
- Arithmetic operators
- Relational operators
- Logical operators
- Bitwise logical operators
- Increment and decrement operators
- Grouping structure
- Conditional statement
- Looping structure
- Switch statement
- Primitive functions

Comments

All comments must begin with a slash followed by an asterisk (`/*`) and end with an asterisk followed by a slash (`*/`).
Example:

```
/* this is a comment */
```

Arithmetic operators

The arithmetic operators are:

+	Addition
-	Subtraction
*	Multiplication
/	Division
=	Assignment operator.
+=	Replacement addition. `a += b` is the same as `a = a + b`.
-=	Replacement subtraction. `a -= b` is the same as `a = a - b`.

Relational operators

The relational operators are:

>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to

Logical operators

The logical operators are:

!	Logical not
&&	Logical and
||	Logical or

Bitwise logical operators

The bitwise logical operators are:

~	Bitwise negation (one's complement). For example, if `a = 1101`, then `~a = 0010` is the bitwise one's complement of `a`.
&	Bitwise and, for example `a & b` is the and of `a` and `b`.
|	Bitwise inclusive or, for example `a | b` is the inclusive or of `a` and `b`.
^	Bitwise exclusive or, for example `a ^ b` is the exclusive or of `a` and `b`.
&=	Bitwise AND, for example `a &= b` is the same as `a = a & b`.

`|=` Bitwise inclusive OR, for example `a |= b` is the same as `a = a | b`.

`^=` Bitwise exclusive or, for example `a ^= b` is the same as `a = a ^ b`.

`::` Bitwise concatenation, that is, `A::B` will return the concatenation of `A` and `B`, where `A` is contained in the most significant location and `B` is contained in the least significant location. For example, if `A = 1010` and `B = 0011`, then `A::B` returns the value `10100011`.

`<<` Bitwise left shift, for example `a << b` will shift the bits in `a` to the left by the value in `b`.

`>>` Bitwise right shift, for example `a >> b` will shift the bits in `a` to the right by the value in `b`.

The following example illustrates the action of the `&` operator (bitwise `and`):

```
a = 0x05 (0101)        b = 0x09 (1001)

   a 0101
   b 1001
-----------
a & b 0001
```

The following example illustrates the action of the `|` operator (bitwise inclusive `or`):

```
a = 0x05 (0101)        b = 0x09 (1001)

   a 0101
   b 1001
-----------
a | b 1101
```

The following example illustrates the action of the `^` operator (bitwise exclusive `or`):

```
a = 0x05 (0101)        b = 0x09 (1001)
   a 0101
   b 1001
-----------
a ^ b 1100
```

Increment and decrement operators

The increment and decrement operators are:

++ Increment index value, for example `i++` increases the index value `i` by 1.

-- Decrement index value, for example `i--` decreases the index value `i` by 1.

Grouping structure

An open brace ({) designates the beginning of a group of instructions or statements. A close brace (}) designates the end of a group of instructions or statements.

Example:

```
{
    statement;
    statement;
    .
    .
    .
}
```

Square brackets designate an element of a structure such as a vector.

Example:

```
Vk[a] = Vj[b]
```

Angle brackets designate a bit or series of bits in a structure such as a register.

Example:

```
PSW<SC>
Sk<0..7>
```

Conditional statement

The `if` statement is the primary *conditional* statement. In the example shown below, the statements within braces are executed only if the expression evaluates *true*. If the expression evaluates *false*, the statements within braces are not executed and program execution continues with the statement following the closing brace.

Example:

```
if (expression) {
    statement;
    statement;
    .
    .
    .
}
```

The following code illustrates the use of the `if` statement:

```
if (VM<b> == 1) {      /* if VM<b> is TRUE */
    Vk[a] = Vj[b];
    a = a + 1;
}
```

Looping structure pseudocode;

The `for` statement is the primary loop statement.

The following code illustrates the *unconditional* use of the `for` statement:

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

The next example illustrates the *conditional* use of the `for` statement. The statement *initial* is executed when the `for` statement begins. Each time the loop is executed, the *test_expression* is checked. If *test_expression* is *true*, the group of statements is executed and the statement *increment* is executed. If the test is *false*, the `for` statement is exited, and program execution continues with the statement following the closing brace.
Example:

```
for (initial; test_expression; increment) {
    statement;
    statement;
    .
    .
    .
}
```

Switch statement

The `switch` statement is a *multiple conditional* statement. It is similar to a computed `goto` statement. In the following example, the value of *expression* is compared with *constant1*. If the compare evaluates *true*, program execution continues with the group of statements following *constant1*. If the compare evaluates *false*, the statements following *constant1* are not executed, and the value of *expression* is compared with *constant2*.

The `break` statement causes program execution to exit the `switch` statement and continue with the statement following the closing brace of the `switch` statement.
Example:

```
switch (expression) {
    case (constant1) :
        statement;
        statement;
        .
        .
        .
        break;
    case (constant2) :
        statement;
        statement;
        .
        .
        .
        break;
} /* end switch */
```

The following code illustrates the use of the `switch` statement:

```
a = 0
switch (E) {
    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 for loop */
        break;
    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 for loop */
        break;
} /* end switch */
```

Primitive functions

The primitive functions listed here are also used in the pseudocode descriptions of the Operation section. Many of these functions have suffixes to denote size, such as `put.l` for longword and `put.w` for word. Without a suffix, a word suffix is assumed.

`aint` (*value*)	The result of converting a single-precision floating point *value* to an integer is returned.
`atan` (*value*)	The result of calculating the trigonometric arc-tangent of a floating point *value* is returned.
`cos` (*value*)	The result of calculating the trigonometric cosine of a floating point *value* is returned.
`dint` (*value*)	The result of converting a double-precision floating point *value* to an integer is returned.
`exp` (*value*)	The result of calculating the exponent of a floating point value is returned.

`get (Ceffa)` The contents of communication register *Ceffa* is returned, using the `get` operation, described in the *CONVEX Architecture Reference Manual (C Series)*, Chapter 6, "Communication registers".

The following code illustrates the `get (Ceffa)` statement:

```
return (c(Ceffa));
```

`lck (Ceffa)` An attempt is made to lock communication register *Ceffa*, using the `lck` operation described in the *CONVEX Architecture Reference Manual (C Series)*, Chapter 6, "Communication registers." The status of the `lck` operation is returned.

The statement

```
C = lck(0x8000)
```

tries to lock register 0x8000. Status is returned and assigned to address carry (C).

The following code illustrates the `lck (Ceffa)` statement:

```
if (L(Ceffa) == 0) {
    L(Ceffa) = 1;
    return (1);
} else {
    return (0);
}
```

`ln (value)` The result of calculating the natural logarithm of a floating point *value* is returned.

`Op code_Test (p1,p2)` The first operand (*p1*) is compared to the second operand (*p2*) within the instruction operation.

`pop (value)` — The value is popped from the top of the stack, using a `pop` instruction described in Chapter 3 of this manual.

`push (value)` — The value is pushed on the stack, using a `push` instruction described in Chapter 3 of this manual.

`put (Ceffa,value)` — The *value* is put to communication register *Ceffa*, using the `put` operation described in the *CONVEX Architecture Reference Manual (C Series)*, Chapter 6, "Communication registers."

The following code illustrates the `put (Ceffa,value)` statement:

```
c(Ceffa) = value;
```

`rcv (Ceffa)` — An attempt is made to receive from communication register *Ceffa*, using the `rcv` operation described in the *CONVEX Architecture Reference Manual (C Series)*, Chapter 6 "Communication registers."

The contents of *Ceffa* and the status of the `rcv` operation is returned. The `rcv` function is used three ways.

`rcv (Ceffa)` — Tries to receive register *Ceffa* and status is returned.

`C = rcv (Ceffa)` — Tries to receive register *Ceffa*. Status is returned and assigned to address carry (C).

`C = rcv (Ceffa, Rk)` — The received contents of register *Ceffa* are returned and assigned to register `Rk`. Status is returned and assigned to address carry (C).

The following code illustrates the `rcv(`*Ceffa*`)` or `rcv(`*Ceffa*`, Rk)` statement:

```
if (argc == 2) { /*argc is the argument count*/
    Rk = c(Ceffa);
}
if (L(Ceffa) == 1) {
    L(Ceffa) = 0;
    return (1);
} else {
    return (0);
}
```

`sin(`*value*`)` — The result of calculating the trigonometric sine of a floating point value is returned.

`snd(`*Ceffa,value*`)` — An attempt is made to send *value* to communication register *Ceffa*, using an operation similar to the `snd` operation described in the *CONVEX Architecture Reference Manual (C Series)*, Chapter 6, "Communication registers." The contents of *Ceffa* and the status of the `snd` operation are returned. Therefore, the `snd` function is used three ways.

`snd(`*Ceffa*`)` — Tries to send to the *Ceffa* register and status is returned.

`C = snd(`*Ceffa*`)` — Tries to send to the *Ceffa* register. Status is returned and assigned to address carry (C).

`C = snd(`*Ceffa*`, FP::AP)` — The concatenated contents of registers `FP` and `AP` are sent to register *Ceffa*. Status is returned and assigned to address carry (C).

The following code illustrates the `snd (Ceffa,value)` statement:

```
if (L(Ceffa) == 0) {
    if (argc == 2) { /*argc is the argument count*/
        c(Ceffa) = Rk;
    }
    L(Ceffa) = 1;
    return (1);
} else {
    return (0);
}
```

`sqrt (value)` — The result of calculating the square root of a floating point value is returned.

`tac (effa)` — An attempt is made to test-and-clear memory location *effa*, using the `tac` instruction described in this chapter. The status of the `tac` operation is returned.

The following code illustrates the `tac (effa)` statement:

```
msync;
if (c(effa) == 0xFF ) {
    c(effa) = 0;
    return (1);
} else {
    c(effa) = 0;
    return (0);
}
```

`tas (effa)` — An attempt is made to test-and-set memory location *effa*, using the `tas` instruction described in this chapter. The status of the `tas` operation is returned.

The following code illustrates the `tas (effa)` statement:

```
msync;
if (c(effa) == 0) {
    c(effa) = 0xFF;
    return (1);
} else {
    c(effa) = 0xFF;
    return (0);
}
```

`tst` (*Ceffa*) — The value of the lock bit of communication register *Ceffa* is returned.

The following code illustrates the `tst` (*Ceffa*) statement:

```
return (L(Ceffa));
```

`tzc` (*value*) — The result of the *trailing-zero-bit-count* operation performed on *value* is returned. The `tzc` instruction is used.

`ulk` (*Ceffa*) — An attempt is made to unlock communication register *Ceffa* using the `ulk` operation described in the *CONVEX Architecture Reference Manual (C Series)*, Chapter 6, "Communication registers." The status of the `ulk` operation is returned.

The following code illustrates the `ulk` (*Ceffa*) statement:

```
if (L(Ceffa) == 1) {
    L(Ceffa) = 0;
    return (1);
} else {
    return (0);
}
```

add.{h|w} #{n|N},Ak

Add register (immediate to address)

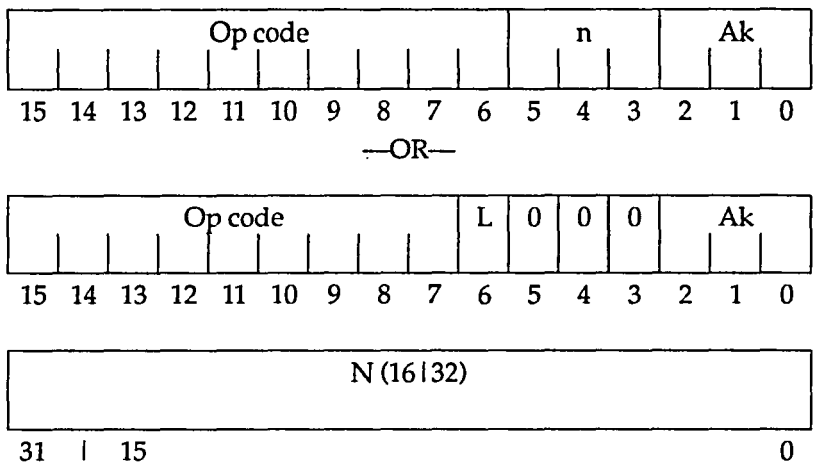
Purpose

To add an immediate operand to the contents of an address register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
add.h #n,Ak	ST	5880	0101100010	C,AIV	Add short immediate address halfword
add.w #n,Ak	ST	58C0	0101100011	C,AIV	Add short immediate address word
add.h #N,Ak	ST	1400	000101000	C,AIV	Add immediate address halfword
add.w #N,Ak	ST	1480	000101001	C,AIV	Add immediate address word

Description

The sum of either the short immediate operand (#n) or the sign-extended long immediate operand (#N, length indicated by L) and the contents of address register Ak replaces the contents of Ak.

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

Operation

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

Exceptions

`h | w` Integer overflow

add.{h|w|s} #N,Sk

Add register (immediate to scalar)

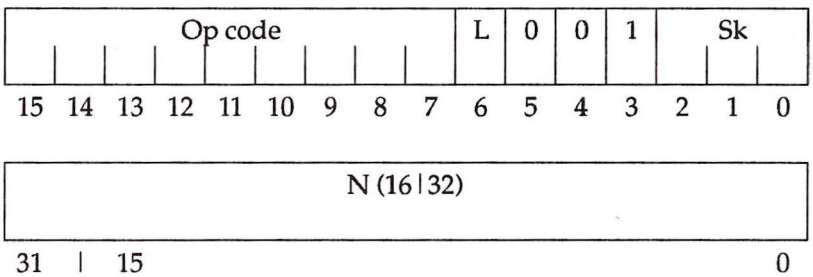
Purpose

To add an immediate operand to the contents of a scalar register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

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

Description

The sum of the sign-extended long immediate operand (#N, length indicated by L) and the contents of scalar register Sk replaces the contents of Sk.

Operation

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

Exceptions

`h	w`	Integer overflow
`s`	Reserved operand Exponent overflow Exponent underflow

add.{h|w} Aj,Ak

Add registers (address to address)

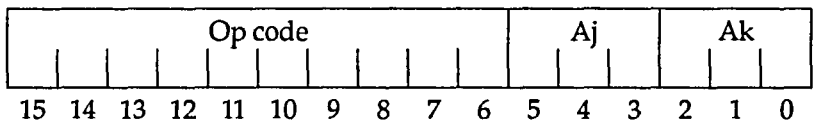
Purpose

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

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

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

Description

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

Operation

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

Exceptions

`h	w`	Integer overflow

add.{b|h|w|l|s|d} Sj,Sk
Add registers (scalar to scalar)

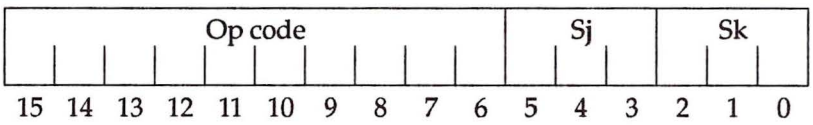
Purpose

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

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

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

Operation

```
Sk = Sk + Sj;
```

Exceptions

`b|h|w|l`	Integer overflow
`s|d`	Reserved operand Exponent overflow Exponent underflow

add.w Sj,Ak

Add registers (scalar to address)

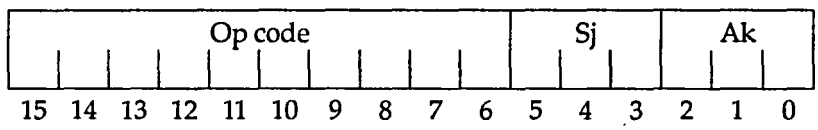
Purpose

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

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
add.w Sj,Ak	ST	5000	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 replaces the contents of Ak.

Operation

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

Exceptions

Integer overflow

add.{b|h|w|l|s|d} Vi,Sj,Vk

Add registers (vector and scalar)

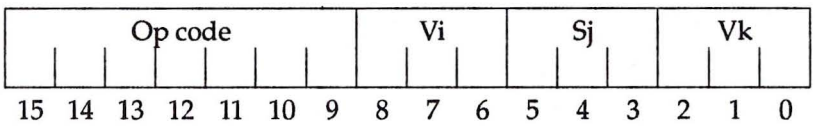
Purpose

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

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

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

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

add.{b|h|w|l|s|d}.{t|f} Vi,Sj,Vk

Add registers (vector and scalar) (masked)

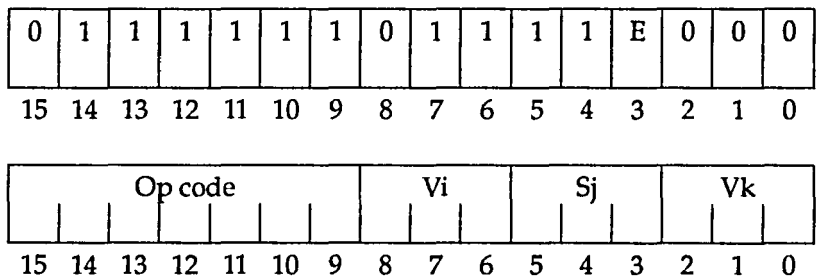
Purpose

To add a scalar to a vector under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

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

add.{b|h|w|l|s|d}.{t|f} Vi,Sj,Vk

Description

The contents of each 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, only if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

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;  } /* end of switch */
```

Exceptions

`b|h|w|l`	Integer overflow
`s|d`	Exponent overflow Exponent underflow Reserved operand

add.{b|h|w|l|s|d} Vi,Vj,Vk
Add registers (vector and vector)

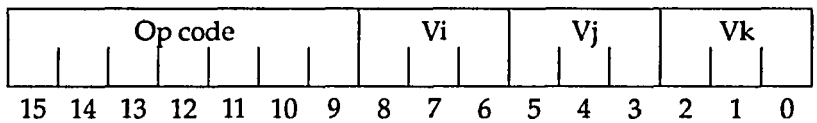
Purpose

To add the elements of two vector registers

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

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

Description

The contents of each 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.

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

add.{b|h|w|l|s|d}.{t|f} Vi,Vj,Vk

Add registers (vector and vector) (masked)

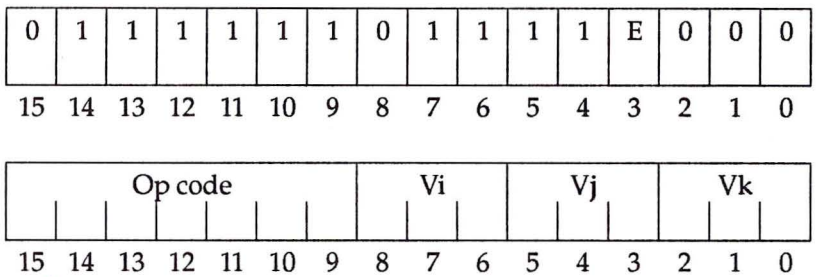
Purpose

To add two vectors under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

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

Description

The contents of each of the first VL elements of vector register Vk is replaced by the sum of contents of the corresponding elements of Vi and Vj, only if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

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; } /* end of switch */
```

Exceptions

`b|h|w|l`	Integer overflow
`s|d`	Exponent overflow Exponent underflow Reserved operand

all {Vk|Sk}
And reduce vector register

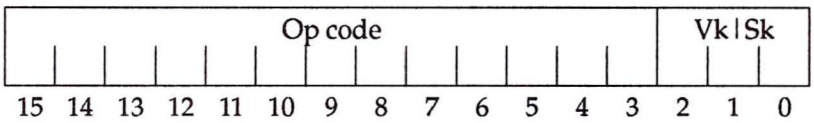
Purpose

To and reduce elements of a vector register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

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

Description

The bitwise 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 replaces the contents of Sk.

Operation

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

Notes

1. Initialize the scalar register properly for the first use of the and reduce instruction (usually 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).

all.{t|f} {Vk|Sk}
And reduce vector register (masked)

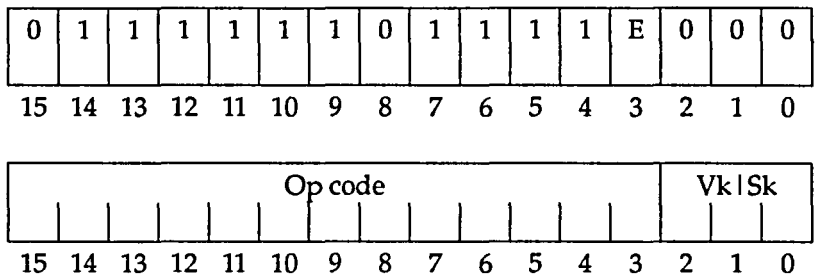
Purpose

To and reduce elements of a subset of a vector register under control of the vector merge (VM) register.

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
all.t Vk	E1	7E20	0111111000100	None	And reduce a vector (VM)
all.t Sk	E1	7E20	0111111000100	None	And reduce a vector (VM)
all.f Vk	E0	7E20	0111111000100	None	And reduce a vector (!VM)
all.f Sk	E0	7E20	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 if the corresponding vector merge (VM) bit is set (for .t) or clear (for .f).

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; } /* end of switch */
```

Notes

1. Initialize the scalar register properly for the first use of the `all` (and reduce) instruction (usually 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 register with immediate

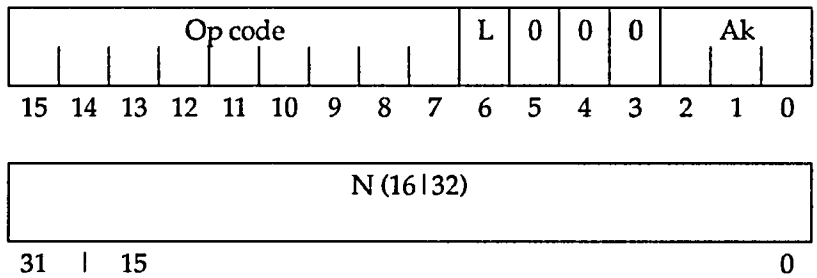
Purpose

To and an immediate operand with the contents of an address register.

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

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

Description

The bitwise and of the sign-extended long immediate operand (#N, length indicated by L) and the contents of address register Ak replaces the contents of Ak.

Operation

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

and #N,Sk

And scalar register with immediate

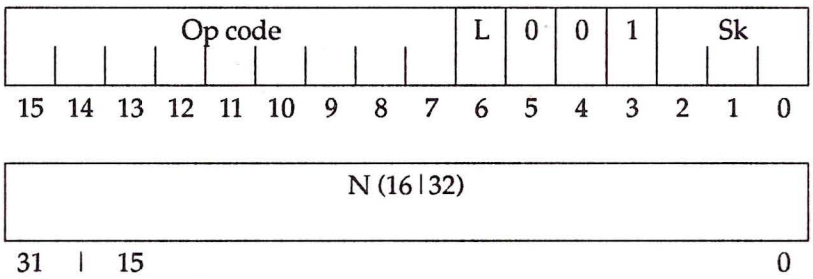
Purpose

To `and` an immediate operand and the contents of a scalar register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

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

Description

The bitwise `and` of the sign-extended long immediate operand (#N, length indicated by L) and the least significant 32 bits of scalar register Sk replaces the least significant 32 bits of Sk. The most significant 32 bits of Sk are not affected.

Operation

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

and Aj,Ak

And registers (address with address)

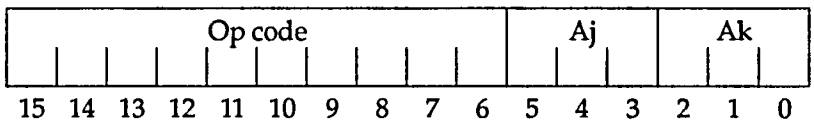
Purpose

To and the contents of two address registers

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

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

Description

The bitwise and of the contents of address registers Aj and Ak replaces the contents of Ak.

Operation

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

and Sj,Sk

And registers (scalar with scalar)

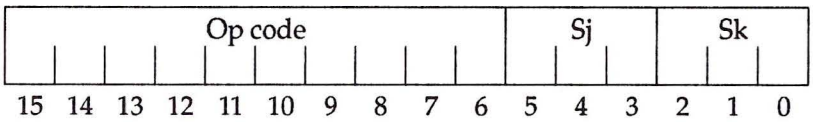
Purpose

To and the contents of two scalar registers

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

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

Description

The bitwise and of the contents of scalar registers Sj and Sk replaces the contents of Sk.

Operation

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

and Vi,Sj,Vk

And registers (vector with scalar)

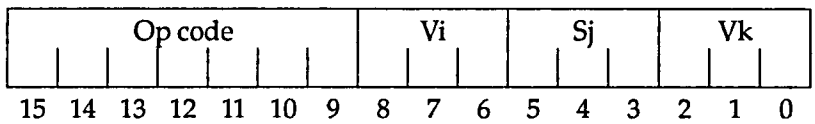
Purpose

To and the elements of a vector register with the contents of a scalar register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
and Vi,Sj,Vk	ST	A800	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.

Operation

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

and.{t|f} Vi,Sj,Vk

And registers (vector with scalar) (masked)

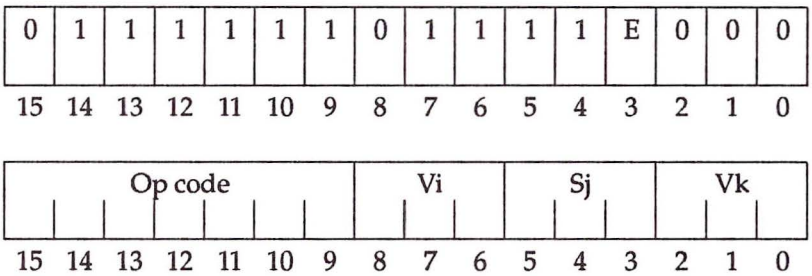
Purpose

To and the contents of a vector and a scalar under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
and.t Vi,Sj,Vk	E1	A800	1010100	None	And vector/scalar (VM)
and.f Vi,Sj,Vk	E0	A800	1010100	None	And vector/scalar (!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 scalar register Sj and the contents of the corresponding element of Vi, only if the corresponding vector merge (VM) bit is set (for `.t`) or clear (for `.f`).

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; } /* end of switch */
```

and Vi,Vj,Vk

And registers (vector with vector)

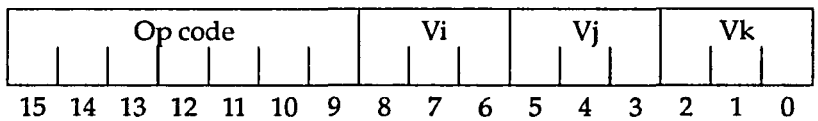
Purpose

To and the elements of two vector registers

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
and Vi,Vj,Vk	ST	A000	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.

Operation

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

Note

Copy the contents of vector register Vi to Vj with the instruction `and Vi,Vi,Vj`.

and.{t|f} Vi,Vj,Vk

And register (vector with vector) (masked)

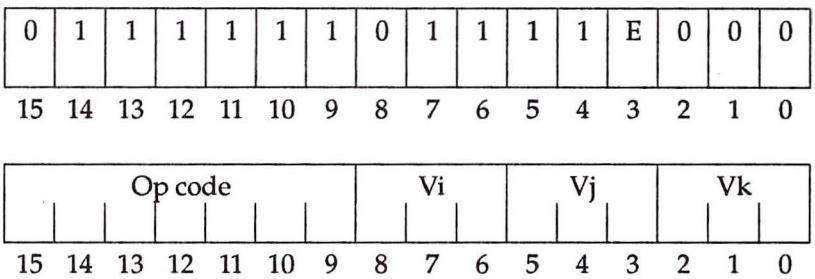
Purpose

To and the contents of two vectors under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
and.t Vi,Vj,Vk	E1	A000	1010000	None	And two vectors (VM)
and.f Vi,Vj,Vk	E0	A000	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, only if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

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; } /* end of switch */
```

any {Vk|Sk}
Or reduce vector register

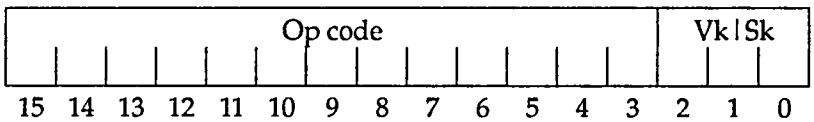
Purpose

To `or` reduce elements of a vector register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

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

Description

The bitwise `or` of the contents of scalar register Sk and the contents of each of the first VL elements of vector register Vk replaces the contents of Sk.

Operation

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

Notes

1. Initialize the scalar register properly for the first use of the `or` reduce instruction (usually 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).

any.{t|f} {Vk|Sk}
Or reduce vector register (masked)

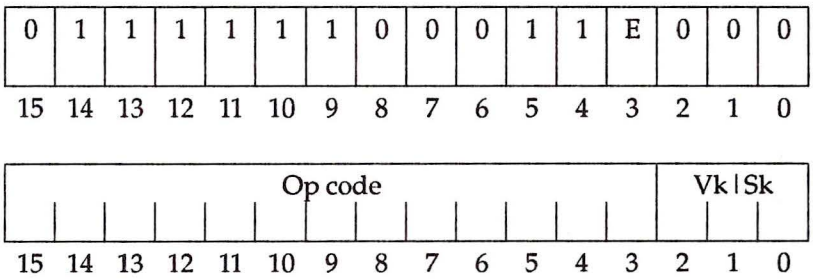
Purpose

To `or` reduce elements of a subset of a vector register under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
any.t Vk	E1	7E28	0111111000101	None	Or reduce a vector (VM)
any.t Sk	E1	7E28	0111111000101	None	Or reduce a vector (VM)
any.f Vk	E0	7E28	0111111000101	None	Or reduce a vector (!VM)
any.f Sk	E0	7E28	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 replaces the contents of Sk, only if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

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;  } /* end of switch */
```

Notes

1. Initialize the scalar register properly for the first use of the `or` reduce instruction (usually 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 (scalar)

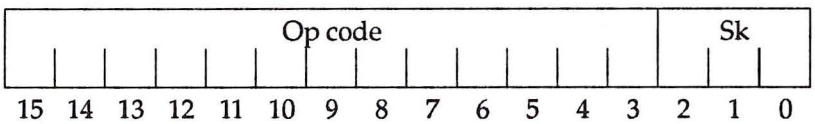
Purpose

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

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

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

Operation

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

Exceptions

s|d	Floating intrinsic error
	Reserved operand

Notes

1. The result is an angle in radians between $-\pi/2$ and $\pi/2$.
2. Intrinsic traps go through the same trap handler as other arithmetic traps (PSW (RO), PSW (FDZ), PSW (UN), etc.). If PSW (FUE) and/or PSW (FE) are set and intrinsic traps, PSW (INE) are cleared, 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 Manual (C Series)*, "Processor status word" section in Chapter 3, "General registers," for more information on the PSW (IEC) error codes and arithmetic trap conditions.

bkpt
Breakpoint

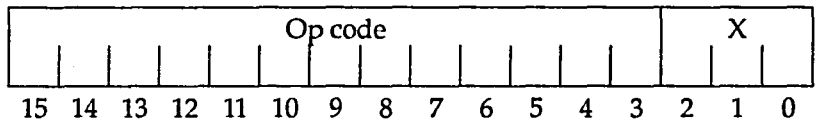
Purpose

To jump to a debugger via a breakpoint call

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

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

Description

A subroutine call is executed 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.

Operation

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

Notes

1. The following PSW flags are set to 0:
   C, AIV, ADZ, SC, SIV, SDZ, OV, UN, FDZ, RO, and FRL
2. Because the length of the `bkpt` instruction is one halfword, it can replace any instruction in the CONVEX C Series architecture.
3. The X field is unused.

br
Branch unconditionally or on PSW bit

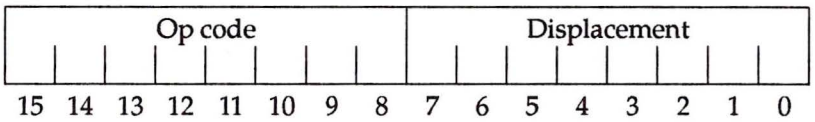
Purpose

To perform a short program counter (PC) relative branch, either unconditionally or conditionally.

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
br	ST	7100	01110001	None	Branch always
bri.f	ST	7200	01110001	None	Branch on ION false See Note 5
bri.t	ST	7300	01110001	None	Branch on ION true See Note 5
bra.f	ST	7400	01110001	None	Branch on C false
bra.t	ST	7500	01110001	None	Branch on address carry true
brs.f	ST	7600	01110001	None	Branch on scalar carry false
brs.t	ST	7700	01110001	None	Branch on scalar carry true

Description

If the branch is unconditional, or the specified condition (ION, C, or SC) is the value specified (true or false), the sum of twice the sign-extended 8-bit displacement operand and the contents of the PC (which still contains the address of the branch instruction) replace the PC. If the specified condition is not the specified value, then the next sequential instruction is executed.

Operation

```
if (condition == true) {
    PC = PC_of_branch + 2 * sign-extended_displacement); }
```

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 multiprocessing C Series CPUs. The ION flag is asynchronous to the processor. Refer to the notes for the `dsi` and `eni` instructions.

call *effa*

Call a subroutine—long call

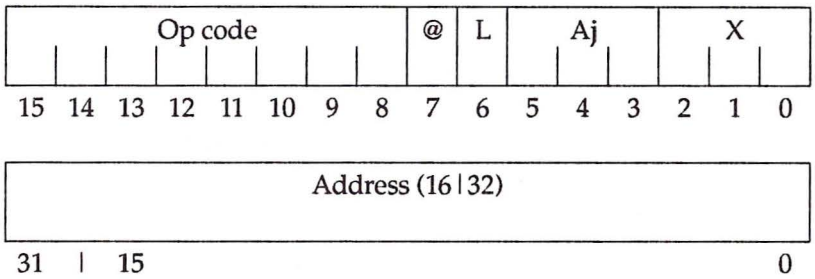
Purpose

To call a subroutine, creating a long call frame

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
call *effa*	ST	2000	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 cleared to 0.

Operation

```
PSW[FRL] = 10;      /* 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 = effa;
```

Notes

1. The following PSW flagss are set to 0:
   C, SC, AIV, ADZ, FRL, UN, OV, FDZ, RO, SIV, SDZ, and FIN
2. In typical usage, a subroutine such as `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 register S0 typically holds return values of functions.
5. The frame length bits, PSW (FRL), are always cleared to 0. To determine the type of frame created on the stack, the FRL bits 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 unused.

callq *effa*

Push PC and jump—quick call

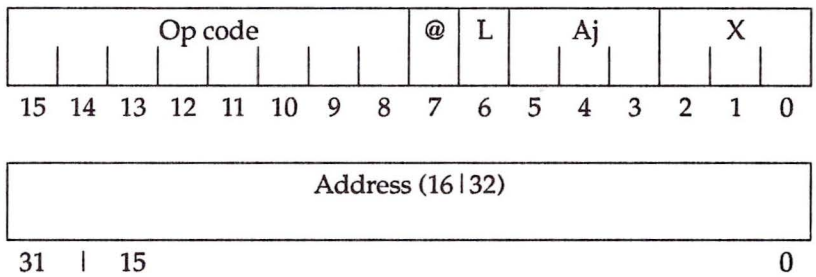
Purpose

To push the program counter (PC) onto the stack and jump

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

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

Description

The address of the instruction immediately following this one is pushed onto the stack. The effective address replaces the contents of the PC.

Operation

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

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 unused.

calls *effa*

Call a subroutine—short call

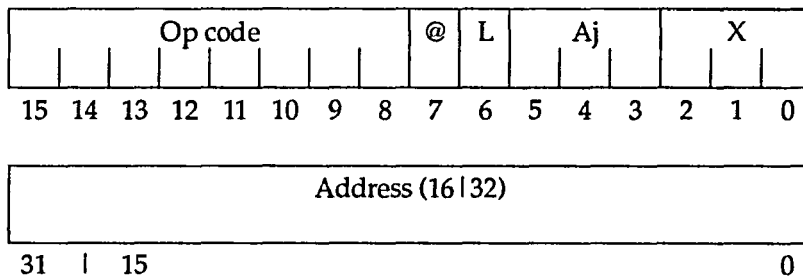
Purpose

To call a subroutine, creating a short call frame

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
calls *effa*	ST	2100	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 cleared to 0.

Operation

```
PSW[FRL] = 11;      /* short frame */
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 = effa;
```

Notes

1. The following PSW bits are set to 0:
   C, SC, AIV, ADZ, FRL, UN, OV, FDZ, RO, SIV, SDZ, and FIN
2. In typical usage, a subroutine such as `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 frame length bits, PSW (FRL), are always cleared to 0. To determine the type of frame created on the stack, the FRL bits 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 unused.

casr

Compare and swap word (resource with memory)

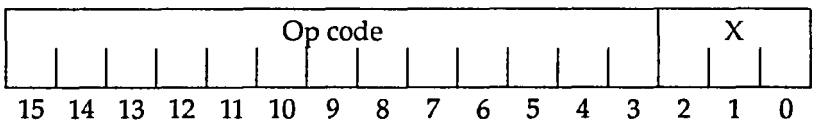
Purpose

To compare and swap a word between a resource structure and a memory location

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
casr	ST	7DE0	0111110111100	C,SC	Compare and swap

Description

A word is atomically compared and swapped between a match value in memory and a target resource structure.

If C is returned as 1 and SC is returned as 1, the operation was successful, the contents of the replace value is placed in the target resource structure, and the target resource structure is unlocked.

If C is returned as 1 and SC is returned as 0, the match failed, the current contents of the target resource structure is returned, and the target resource structure is unlocked.

If C is returned as 0, the target resource structure was in transition and was unable to be locked.

Refer to the *CONVEX Architecture Reference Manual (C Series)*, Chapter 7, "Process structures," for more information about resource structures.

casr

Operation

```
msync;
if (tas(a1.lock)) {
        C = 1;   /* tas successful */
    temp1 = c(a1.data);     /* a1 points to resource
                               data structure */
    temp2 = c(a2); /* a2 points to match value */
    temp3 = c(a3); /* a3 points to replace value */
    if (temp1 == temp2) {   /* if match */
        c(a1.data) = temp3;     /* set data to replace value*/
        SC = 1; /* store occurred */ }
    else {
        c(a2) = temp1; /* current resource data */
        SC = 0; /* store aborted */ }
    msync;
    tac(a1.lock); /* release the resource */ }
else {
    C = 0;   /* tas failed */
    SC = 0; /* store aborted */ }
```

Notes

1. This instruction is atomic.
2. The lock at (a1.lock) also protects (a2) and (a3)
3. Address carry (C) is the `tas` success status, not the carry-out, of the `casr` operation.
4. Scalar carry (SC) is the store/abort status, not the carry-out, of the `casr` operation.
5. Purge the data cache.
6. The X field is unused.

cfork
Clear a fork

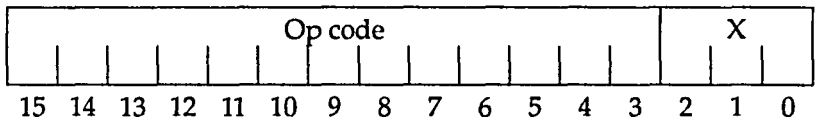
Purpose

To clear a fork

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

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

Description

Clear the hardware communication fork event registers of any outstanding forks in the current CIR. If no fork was outstanding, returns C = 0, otherwise, if a fork was cleared, returns C = 1.

Operation

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

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.

cos.{s|d} Sk
Cosine (scalar)

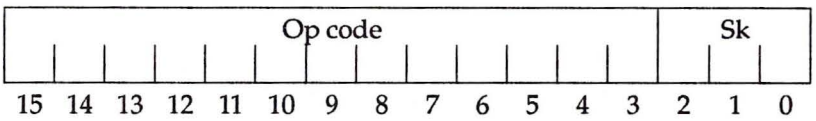
Purpose

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

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

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

Description

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

Operation

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

Exceptions

s|d	Reserved operand Floating intrinsic error

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 (PSW (RO), PSW (FDZ), PSW (UN), etc.). If PSW (FUE) and/or PSW (FE) are set and intrinsic traps, PSW (INE), are cleared, 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 Manual (C Series)*, "Processor status word" section in Chapter 3, "General registers," for more information on the PSW (IEC) error codes and arithmetic trap conditions.

cprs.{t|f} Vj,Vk
Compress register (vector)

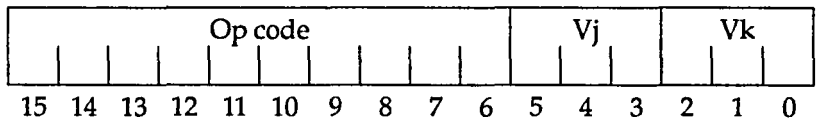
Purpose

To compress a vector using the vector merge (VM) register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

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

Description

For `.t`–The contents of the *next unused* element of vector register Vk are replaced by the next element in Vj, if the corresponding VM bit is 1.

For `.f`–The contents of the *next unused* element of vector register Vk are replaced by the next element in Vj, if the corresponding VM bit is 0.

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;  } /* end of switch */
```

Note

The `plc VM` instruction calculates the new length of Vk.

ctrsg
CPU timer synchronize (global)

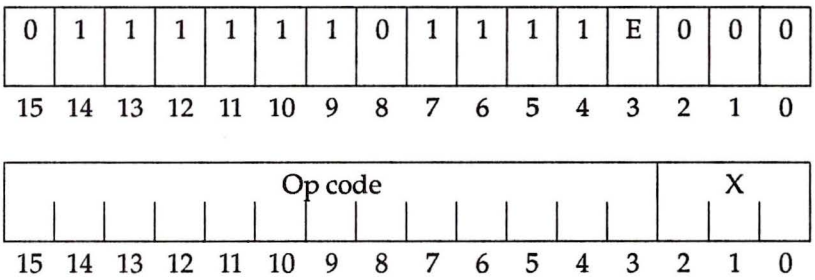
Purpose

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

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
ctrsg	E0	7C38	0111110000111	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.

Operation

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

Exceptions

Ring violation (privileged instruction)

Note

The X field is unused.

cvt{b|h|w} Aj,Ak

Convert integer in address register

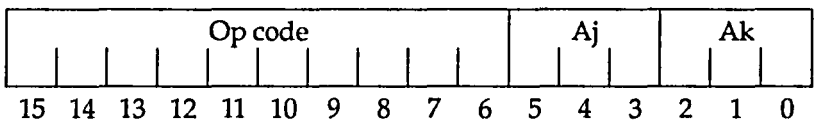
Purpose

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

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

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

Description

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

Operation

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

Exceptions

`b|h`	Integer overflow

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.

cvt{b|h|w|l}.{s|d} Sj,Sk
Convert scalar register

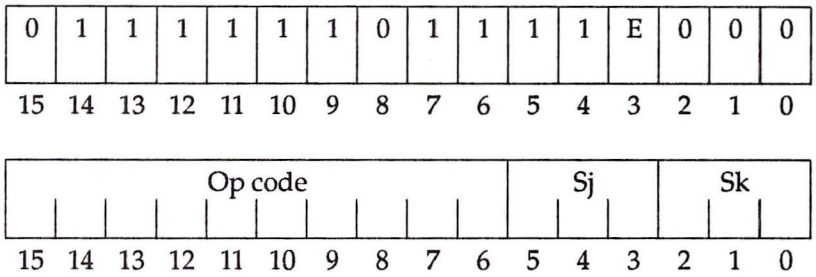
Purpose

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

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

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

Description

The converted contents of the scalar register Sj replace the contents of Sk. Conversions to smaller data sizes can cause an underflow or overflow exception. Conversions from floating-point representation to integer use truncation (rounding toward 0).

Operation

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

Exceptions

`b|h|w|l`	Integer overflow
`s|d`	Exponent overflow Exponent underflow Reserved operand

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, the destination register Sk is overwritten and the PSW (RO) flag is set to 1. On C3400 Series CPUs, the SIV flag is also 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:
   - The fixed-point number is normalized.
   - 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.
   - 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.
7. For C100 Series CPUs only, execution of the `cvtw.d Sj, Sk` and `cvtd.w Sj, Sk` instructions results in an unimplemented instruction trap.

cvt{b|h|w|l}.{s|d} Vj,Vk
Convert vector register

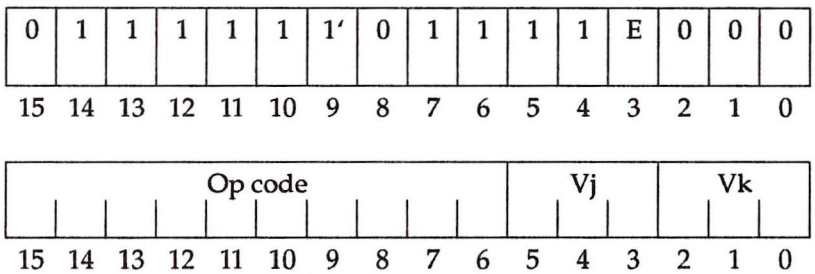
Purpose

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

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

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

Description

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

Operation

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

Exceptions

`b|h|w|l`	Integer overflow
`s|d`	Exponent overflow Exponent underflow Reserved operand

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, the PSW (RO) flag is set to 1. On C3400 Series CPUs, the PSW (SIV) flag is also set to 1.
5. Truncation from float to fix follows the following 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:
   - The fixed-point number is normalized.
   - 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.

cvt{b|h|w|l}.{s|d}.{t|f} Vj,Vk
Convert vector register (masked)

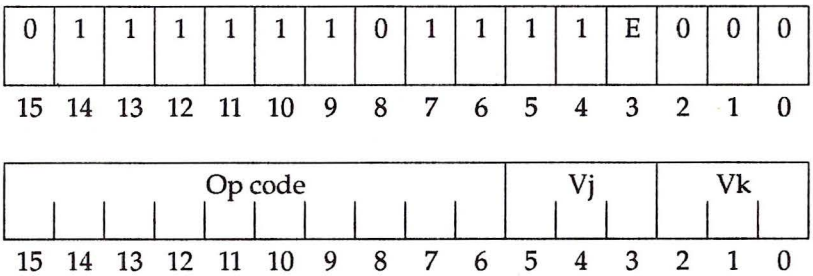
Purpose

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

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
cvtw.s.t Vj,Vk	E1	7900	0111100100	None	Convert word to single (VM)
cvtw.s.f Vj,Vk	E0	7900	0111100100	None	Convert word to single (!VM)
cvts.w.t Vj,Vk	E1	7940	0111100101	RO,SIV	Convert single to word (VM)
cvts.w.f Vj,Vk	E0	7940	0111100101	RO,SIV	Convert single to word (!VM)
cvtd.s.t Vj,Vk	E1	6000	0110000000	RO,SIV, OV,UN	Convert double to single (VM)
cvtd.s.f Vj,Vk	E0	6000	0110000000	RO,SIV, OV,UN	Convert double to single (!VM)
cvts.d.t Vj,Vk	E1	6040	0110000001	RO,SIV	Convert single to double (VM)
cvts.d.f Vj,Vk	E0	6040	0110000001	RO,SIV	Convert single to double (!VM)
cvtw.b.t Vj,Vk	E1	4100	0100000100	SIV	Convert word to byte (VM)
cvtw.b.f Vj,Vk	E0	4100	0100000100	SIV	Convert word to byte (!VM)
cvtw.h.t Vj,Vk	E1	4140	0100000101	SIV	Convert word to halfword (VM)
cvtw.h.f Vj,Vk	E0	4140	0100000101	SIV	Convert word to halfword (!VM)
cvtb.w.t Vj,Vk	E1	4180	0100000110	None	Convert byte to word (VM)
cvtb.w.f Vj,Vk	E0	4180	0100000110	None	Convert byte to word (!VM)
cvth.w.t Vj,Vk	E1	41C0	0100000111	None	Convert halfword to word (VM)
cvth.w.f Vj,Vk	E0	41C0	0100000111	None	Convert halfword to word (!VM)
cvts.l.t Vj,Vk	E1	4300	0100001100	RO,SIV	Convert single to longword (VM)
cvts.l.f Vj,Vk	E0	4300	0100001100	RO,SIV	Convert single to longword (!VM)
cvtd.l.t Vj,Vk	E1	60C0	0110000011	RO,SIV	Convert double to longword (VM)
cvtd.l.f Vj,Vk	E0	60C0	0110000011	RO,SIV	Convert double to longword (!VM)
cvtl.s.t Vj,Vk	E1	4380	0100001110	None	Convert longword to single (VM)

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

Description

The converted, possibly sign-extended, contents of the vector register Vj replace the contents of Vk, only if the corresponding vector merge (VM) bit is set (for `.t`) or clear (for `.f`).

Conversions to smaller data sizes can cause an underflow or overflow exception. Conversions from floating-point representation to integer use truncation (rounding toward 0) as the rounding algorithm.

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 op code */ }
        }   /* 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 op code */ }
        }   /* end of for loop */
        break; } /* end of switch */
```

Exceptions

`b|h|w|l`	Integer overflow
`s|d`	Exponent underflow Exponent overflow Reserved operand

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, the PSW (RO) flag is set to 1. On C3400 Series CPUs, the PSW (SIV) flag is also set to 1.
5. Truncation from float to fix follows the following 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:
   - The fixed-point number is normalized.
   - 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.

diag Ak
Execute diagnostic microcode

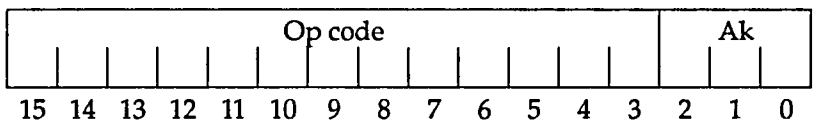
Purpose

To execute a desired sequence of nonstandard microcode

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

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

Description

One of a set of privileged operations specified by address register Ak is invoked. An illegal op code trap occurs if the sub-op codes specified by Ak are not supported.

The contents of A5 are used as an address, if one is needed. S0 is the source or destination of data, if one is needed.

Table 13 lists the subcodes implemented on all C Series CPUs.

Table 14 lists the subcodes implemented on the C100 Series CPUs only.

Table 15 lists the subcodes implemented on multiprocessing C Series CPUs only.

Table 16 lists the subcodes implemented on the C3400 Series CPUs only.

Operation

Execute microcode sequence pointed to by the contents of Ak

Exceptions

Undefined op code
Ring violation (privileged instruction [class=8, qualifier=0])

Note

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

Table 13
Subcodes implemented on all C Series CPUs

Ak	Operation
1	Disable logical cache (C100 Series) or data cache (C200 Series, C3400 Series, C3800 Series).
2	Enable logical cache (C100 Series) or data cache (C200 Series, C3400 Series, C3800 Series).
7	Pull hard error.
11	Store SDR(0-7) at (A5).
12	Enable forced faults C100 Series—based on scan control. C200 Series, C3400 Series, C3800 Series—on all references.
13	Disable forced faults.
14	Flush physical cache (C100 Series) or purge data cache (C200 Series, C3400 Series, C3800 Series)
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.

Table 14
Subcodes implemented on C100 Series CPUs

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

Table 15
Subcodes implemented on multiprocessing C Series CPUs

Ak	Operation
23	Force hits on data cache (disabled by subcode 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> (not implemented on C3400 Series and C3800 Series).
29	Read communication modified bit covered by virtual communication address in A5 into S0<63> (not implemented on C3400 Series and C3800 Series).
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.
34	Read the thread timer with no update based on the hardware delta timer. This subcode does not update TTR or CPU execution timer.

Table 16
Subcodes implemented on C3400 Series CPUs

Ak	Operation
21	Clear performance monitor counters. [1]
22	Dump performance monitor counters to memory table starting at address in A5. [1]
24	Toggle performance monitor counters ON/OFF. [1]
28	Read and cache page 0 kernel contents.

[1] Note: If performance monitor microcode is not used, the performance monitor counter remains at zero.

div.{h|w} #{n|N},Ak

Divide register (address by immediate)

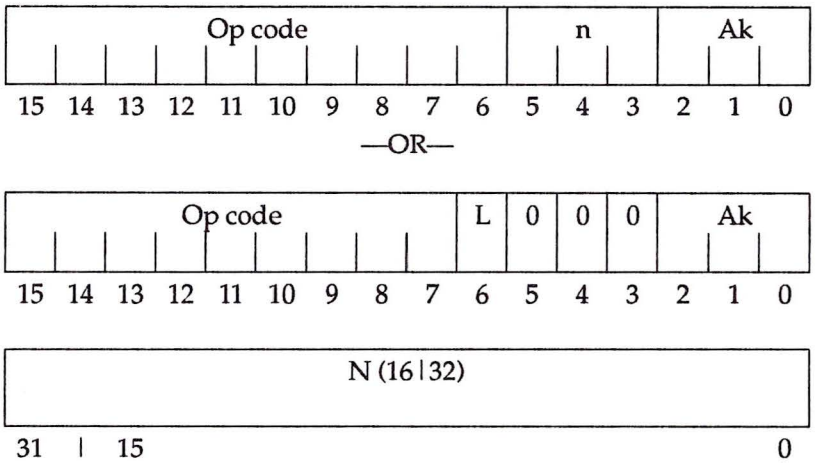
Purpose

To divide the contents of an address register by an immediate operand

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

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

Description

The quotient of the contents of address register Ak, divided by either the short immediate operand (#n) or the sign-extended long immediate operand (#N, length indicated by L), replaces the contents of Ak.

Operation

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

Exceptions

`h	w`	Integer overflow Integer divide-by-zero

Notes

1. Integer overflow occurs if the largest negative number is divided by -1.
2. Integer divide-by-zero returns the original dividend.
3. Sign extension does not occur for the 3 bits of the short immediate form.

div.{h|w|s} #N,Sk

Divide register (scalar by immediate)

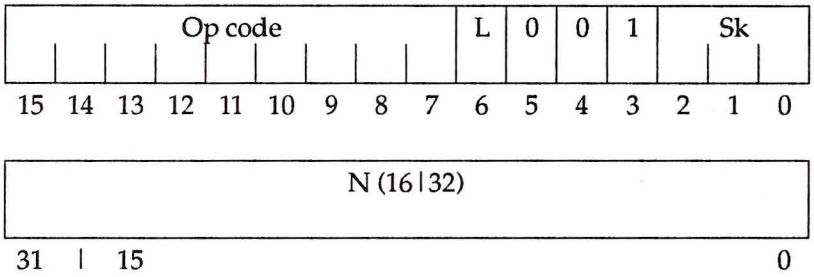
Purpose

To divide the contents of a scalar register by an immediate operand

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
div.h #N,Sk	ST	1708	000101110	SIV,SDZ	Divide scalar/scalar integer halfword
div.w #N,Sk	ST	1788	000101111	SIV,SDZ	Divide scalar/scalar integer word
div.s #N,Sk	ST	1988	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 long immediate operand (#N, length indicated by L), replaces the contents of Sk.

Operation

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

Exceptions

`h	w`	Integer overflow Integer divide-by-zero
`s`	Exponent overflow Exponent underflow Reserved operand Floating divide-by-zero

Notes

1. Integer overflow occurs if the largest negative number is divided by -1.
2. Integer divide-by-zero returns the original dividend.

div.{h|w} Aj,Ak

Divide register (address by address)

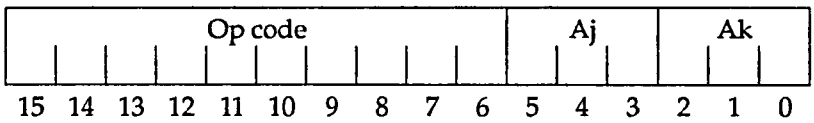
Purpose

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

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

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

Operation

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

Exceptions

h | w	Integer overflow Address divide-by-zero

Notes

1. Integer overflow occurs if the largest negative number is divided by -1.
2. Address division by zero returns the original dividend.

div.{b|h|w|l|s|d} Sj,Sk
Divide register (scalar by scalar)

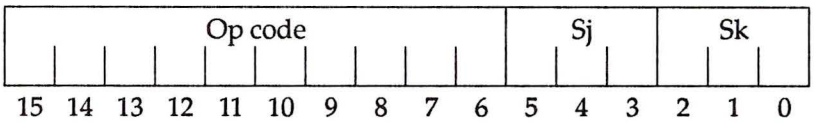
Purpose

To divide the contents of a scalar register by the contents of another scalar register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
div.b Sj,Sk	ST	5F00	0101111100	SIV,SDZ	Divide scalar/scalar integer byte
div.h Sj,Sk	ST	5F40	0101111101	SIV,SDZ	Divide scalar/scalar integer halfword
div.w Sj,Sk	ST	5F80	0101111110	SIV,SDZ	Divide scalar/scalar integer word
div.l Sj,Sk	ST	5FC0	0101111111	SIV,SDZ	Divide scalar/scalar integer longword
div.s Sj,Sk	ST	5780	0101011110	OV,UN, RO,FDZ	Divide scalar/scalar single float
div.d Sj,Sk	ST	57C0	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.

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

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.{s|d} Si,Vj,Vk

Reverse divide register (scalar by vector)

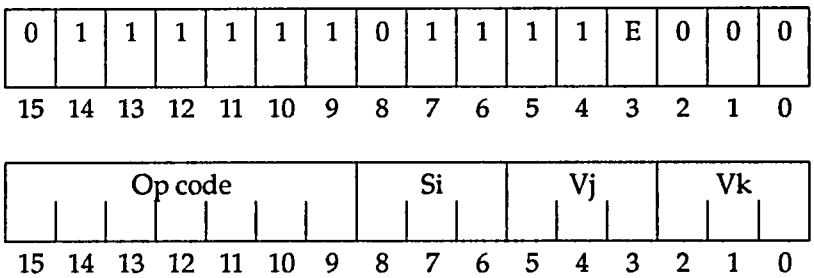
Purpose

To divide a scalar by each element of a vector

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
div.s Si,Vj,Vk	E0	8400	1000010	OV,UN, RO,FDZ	Divide scalar/vector single float
div.d Si,Vj,Vk	E0	8600	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.

Operation

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

Exceptions

Exponent overflow
Exponent underflow
Reserved operand
Floating divide-by-zero

Note

1. Floating point divide-by zero returns a reserved operand.

div.{s|d}.{t|f} Si,Vj,Vk

Reverse divide register (scalar by vector) (masked)

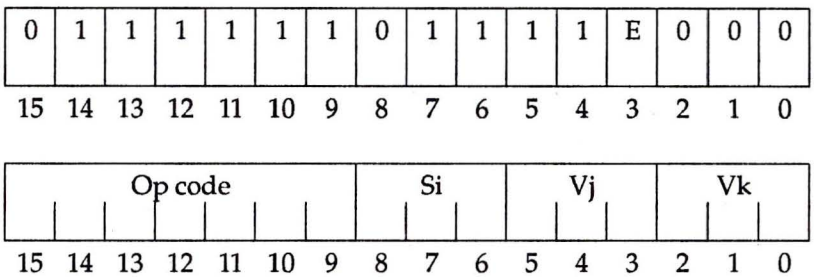
Purpose

To divide a scalar by each element of a vector under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
div.s.t Si,Vj,Vk	E1	8C00	1000110	OV,UN, RO,FDZ	Divide scalar/vector single (VM)
div.s.f Si,Vj,Vk	E0	8C00	1000110	OV,UN, RO,FDZ	Divide scalar/vector single (!VM)
div.d.t Si,Vj,Vk	E1	8E00	1000111	OV,UN, RO,FDZ	Divide scalar/vector double (VM)
div.d.f Si,Vj,Vk	E0	8E00	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, only if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

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; } /* end of switch */
```

Exceptions

s | d	Exponent overflow Exponent underflow Reserved operand Floating divide-by-zero

Note

Floating point divide-by zero returns a reserved operand.

div.{b|h|w|l|s|d} Vi,Sj,Vk
Divide register (vector by scalar)

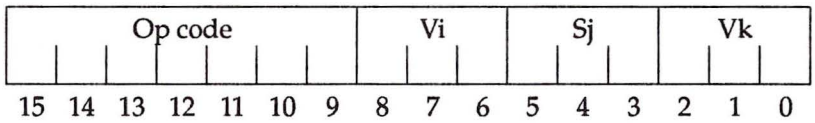
Purpose

To divide the elements of a vector register by the contents of a scalar register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
div.b Vi,Sj,Vk	ST	F800	1111100	SIV,SDZ	Divide vector/scalar integer byte
div.h Vi,Sj,Vk	ST	FA00	1111101	SIV,SDZ	Divide vector/scalar integer halfword
div.w Vi,Sj,Vk	ST	FC00	1111110	SIV,SDZ	Divide vector/scalar integer word
div.l Vi,Sj,Vk	ST	FE00	1111111	SIV,SDZ	Divide vector/scalar integer longword
div.s Vi,Sj,Vk	ST	9C00	1001110	OV,UN, RO,FDZ	Divide vector/scalar single float
div.d Vi,Sj,Vk	ST	9E00	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.

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

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 register (vector by scalar) (masked)

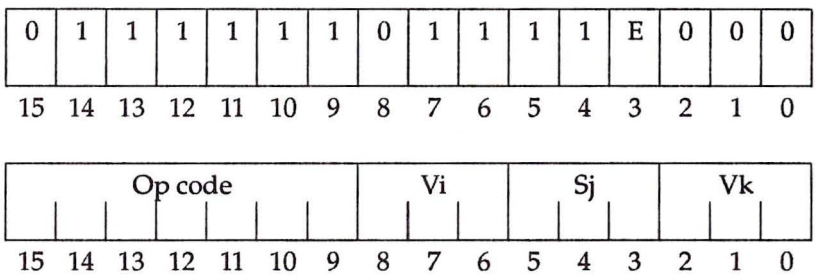
Purpose

To divide a vector by a scalar under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
div.b.t Vi,Sj,Vk	E1	F800	1111100	SIV,SDZ	Divide vector/scalar byte (VM)
div.b.f Vi,Sj,Vk	E0	F800	1111100	SIV,SDZ	Divide vector/scalar byte (!VM)
div.h.t Vi,Sj,Vk	E1	FA00	1111101	SIV,SDZ	Divide vector/scalar halfword (VM)
div.h.f Vi,Sj,Vk	E0	FA00	1111101	SIV,SDZ	Divide vector/scalar halfword (!VM)
div.w.t Vi,Sj,Vk	E1	FC00	1111110	SIV,SDZ	Divide vector/scalar word (VM)
div.w.f Vi,Sj,Vk	E0	FC00	1111110	SIV,SDZ	Divide vector/scalar word (!VM)
div.l.t Vi,Sj,Vk	E1	FE00	1111111	SIV,SDZ	Divide vector/scalar longword (VM)
div.l.f Vi,Sj,Vk	E0	FE00	1111111	SIV,SDZ	Divide vector/scalar longword (!VM)
div.s.t Vi,Sj,Vk	E1	9C00	1001110	OV,UN, RO,FDZ	Divide vector/scalar single (VM)
div.s.f Vi,Sj,Vk	E1	9C00	1001110	OV,UN, RO,FDZ	Divide vector/scalar single (!VM)
div.d.t Vi,Sj,Vk	E0	9E00	1001111	OV,UN, RO,FDZ	Divide vector/scalar double (VM)
div.d.f Vi,Sj,Vk	E1	9E00	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, only if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

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; } /* 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

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 register (vector by vector)

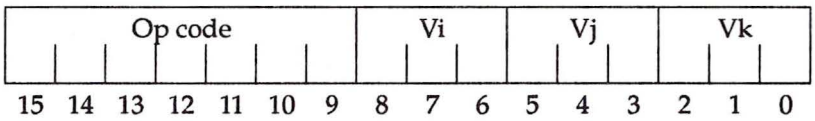
Purpose

To divide the elements of two vector registers

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
div.b Vi,Vj,Vk	ST	F000	1111000	SIV,SDZ	Divide vector/vector integer byte
div.h Vi,Vj,Vk	ST	F200	1111001	SIV,SDZ	Divide vector/vector integer halfword
div.w Vi,Vj,Vk	ST	F400	1111010	SIV,SDZ	Divide vector/vector integer word
div.l Vi,Vj,Vk	ST	F600	1111011	SIV,SDZ	Divide vector/vector integer longword
div.s Vi,Vj,Vk	ST	9400	1001010	OV,UN, RO,FDZ	Divide vector/vector single float
div.d Vi,Vj,Vk	ST	9600	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.

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

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,Vj,Vk
Divide register (vector by vector) (masked)

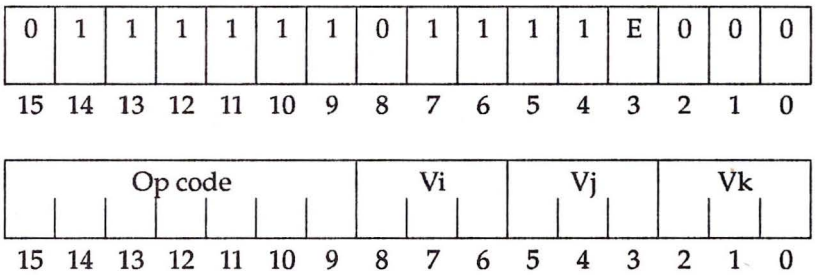
Purpose

To divide two vectors under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
div.b.t Vi,Vj,Vk	E1	F000	1111000	SIV,SDZ	Divide byte vectors (VM)
div.b.f Vi,Vj,Vk	E0	F000	1111000	SIV,SDZ	Divide byte vectors (!VM)
div.h.t Vi,Vj,Vk	E1	F200	1111001	SIV,SDZ	Divide halfword vectors (VM)
div.h.f Vi,Vj,Vk	E0	F200	1111001	SIV,SDZ	Divide halfword vectors (!VM)
div.w.t Vi,Vj,Vk	E1	F400	1111011	SIV,SDZ	Divide word vectors (VM)
div.w.f Vi,Vj,Vk	E0	F400	1111011	SIV,SDZ	Divide word vectors (!VM)
div.l.t Vi,Vj,Vk	E1	F600	1111011	SIV,SDZ	Divide longword vectors (VM)
div.l.f Vi,Vj,Vk	E0	F600	1111011	SIV,SDZ	Divide longword vectors (!VM)
div.s.t Vi,Vj,Vk	E1	9400	1001010	OV,UN, RO,FDZ	Divide single vectors (VM)
div.s.f Vi,Vj,Vk	E0	9400	1001010	OV,UN, RO,FDZ	Divide single vectors (!VM)
div.d.t Vi,Vj,Vk	E1	9600	1001011	OV,UN, RO,FDZ	Divide double vectors (VM)
div.d.f Vi,Vj,Vk	E0	9600	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, only if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

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; } /* 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

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.

dsi
Disable interrupts

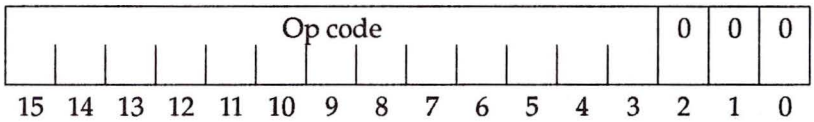
Purpose

To disable interrupts

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
dsi	ST	7D48	0111110101001	C	Disable interrupts; clear ION to 0

Description

The ION flag is atomically cleared to 0, and the previous state of ION is returned in carry (C). Interrupts are disabled when the ION flag is 0.

Operation

```
C = ION;
ION = 0;                    /* dsi—effectively tac(ION) */
```

Exceptions

Ring violation (privileged instruction)

Notes

1. This instruction is atomic.
2. The `jmpi.f`, `jmpi.t`, `bri.f`, and `bri.t` instructions test the value of ION. *Software should avoid using these instructions in the multiprocessing C Series CPUs due to the asynchronous nature of the interrupt system hardware.*
3. To ensure that the `dsi` instruction disables interrupts in the multiprocessing C Series CPUs, software should use the following code sequence:

```
foo:  dsi
      bra.f foo
```

   This requires that software be aware of the state of ION. Software should not try to disable interrupts if it has already disabled them.
4. The ION flag enables or disables all CPUs virtual channel interrupts.
5. The ION flag is the global semaphore for all other interrupt hardware. It must be cleared to 0 via `dsi` before any interrupt hardware (local and global enables, interrupt mode, or target CPU) is modified.

enag Sj,Sk
Enable global interrupts

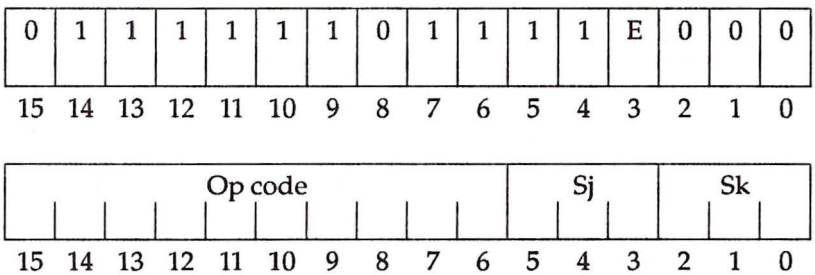
Purpose

To enable or mask interrupts from being accepted by all CPUs in the complex

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
enag Sj,Sk	E0	4480	0100010000	None	Enable all global CPU interrupts

Description

The current value of the global interrupt enable (GE) is returned in register Sk and the global interrupt enable is loaded with the value in Sj. If the bit in the register is 1, the interrupt channel is enabled. If it is 0, the channel is masked out.

Operation

```
Sk = Global_Interrupt_Enable;
Global_Interrupt_Enable = Sj;
```

Exceptions

Ring violation (privileged instruction)

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 Manual (C Series)*, Chapter 11, "Operating System interrupts," for details on the interrupt hardware.
3. The global interrupt enable is used to control interrupt delivery for the entire complex.

enal Sj,Sk
Enable local interrupts

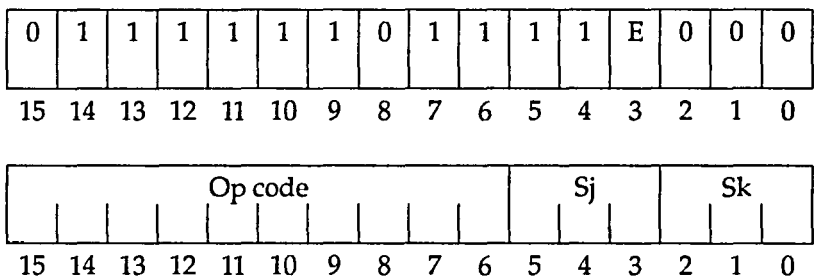
Purpose

To enable or mask interrupts from being accepted by the local CPU

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
enal Sj,Sk	E0	4400	0100010000	None	Enable local CPU interrupt

Description

The current value of the local CPU interrupt enable (LE) is retuned in register Sk and the local CPU interrupt enable is loaded with the value in Sj. If the bit in the register is 1, the interrupt channel is enabled. If it is 0, the channel is masked out.

Operation

```
Sk = Local_Interrupt_Enable;
Local_Interrupt_Enable = Sj;
```

Exceptions

Ring violation (privileged instruction)

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 Manual (C Series)*, Chapter 11, "Operating system interrupts," for details concerning the interrupt hardware.
3. The local interrupt enable is used to control interrupt delivery for the local CPU.

eni
Enable interrupts

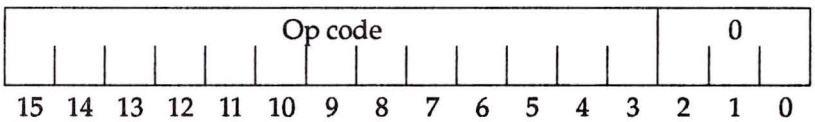
Purpose

To enable interrupts

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
eni	ST	7D40	0111110101000	C	Enable interrupts; set ION to 1

Description

The ION flag is set to 1 and the previous state of ION is returned in carry (C). Interrupts are enabled when ION is 1.

Operation

```
C = ION;
ION = 1;          /* eni—effectively "tas(ION)" */
```

Exceptions

Ring violation (privileged instruction)

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 multiprocessing C Series CPUs 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 cleared to 0 via `dsi` before any interrupt hardware (local and global enables, interrupt mode, or target CPU) is modified.

eni_idle Sk

Enable interrupts and idle the CPU

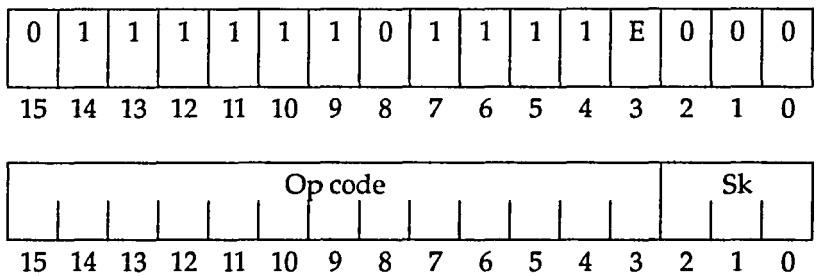
Purpose

To enable interrupts and attempt to accept a posted fork in the specified CIR. If one is not posted, idle the current CPU without deallocating the current thread.

Application

C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
eni_idle Sk	E0	7C80	0111110010000	None	Enable interrupts and idle the CPU

Description

The ION flag is set to 1 and the previous state of ION is returned in carry (C). Then, the idle instruction is executed.

Operation

```
C = ION;
ION = 1;
CIR = Sk;
Execute idle instruction;
```

Exceptions

Ring violation (privileged instruction)

Note

See also the `eni` and `idle` instructions.

eni_rtn

Enable interrupts and return from subroutine

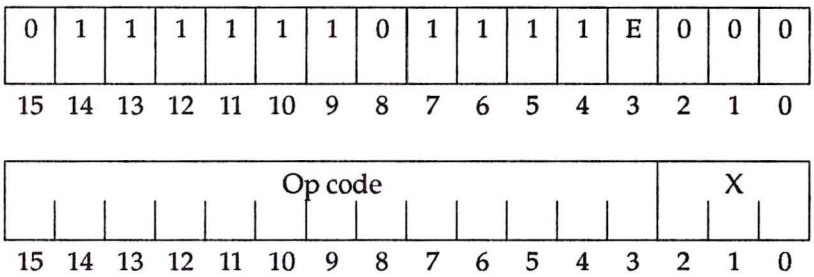
Purpose

To enable interrupts and return from a subroutine

Application

C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
eni_rtn	E0	7C88	0111110010001	All bits	Enable Interrupts and return from subroutine call

Description

Interrupts are enabled, and an `rtn` instruction is performed.

Operation

```
ION = 1;
Execute an rtn instruction;
```

Exceptions

Ring violation (privileged instruction)
See also return (`rtn`)

Notes

1. See also the `eni` and the `rtn` instructions.
2. The PSW is loaded from the current stack frame.
3. The X field is unused.

exit
Error exit

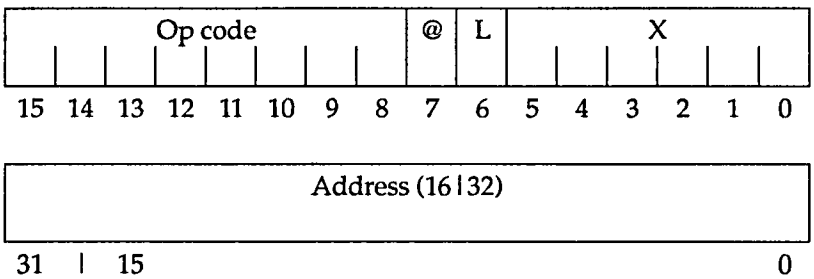
Purpose

To exit as a result of an error

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
exit	ST	0000	000000000	None	Error exit instruction

Description

A system exception of code 0 is executed.

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;
```

Notes

1. The error exit instruction op code of 0 can detect transfers to data.
2. Refer to the *CONVEX Architecture Reference Manual (C Series)*, Chapter 12 "Operating system exceptions," for a description of system exceptions.
3. The @ field is 0. Indirection never occurs for the error exit instruction.
4. The X field is unused.

exp.{s|d} Sk
Exponential

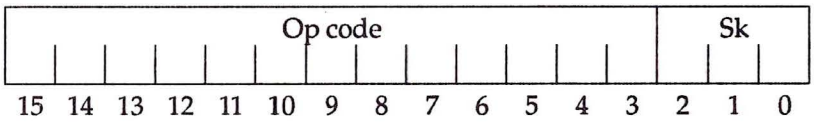
Purpose

To compute e^{Sk}, where e is the base of natural logarithms. $e = 2.718...$

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
exp.s Sk	ST	7CB0	0111110000110	RO,FIN,IEC	Exponent of a single float
exp.d Sk	ST	7CB8	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.

Operation

```
Sk = exp(Sk);
```

Exceptions

s | d	Reserved operand
	Floating intrinsic error

Notes

1. Intrinsic traps go through the same trap handler as other arithmetic traps, (PSW (RO), PSW (FDZ), and PSW (UN)). If PSW (FUE) and/or PSW (FE) are set and the intrinsic traps, PSW (INE), are cleared, these flags 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.
3. Refer to the *CONVEX Architecture Reference Manual (C Series)*, "Processor status word" section in Chapter 3, "General registers," for more information on the PSW (IEC) error codes and arithmetic trap conditions.

frint.{s|d} Sj,Sk

Integerize floating scalar register

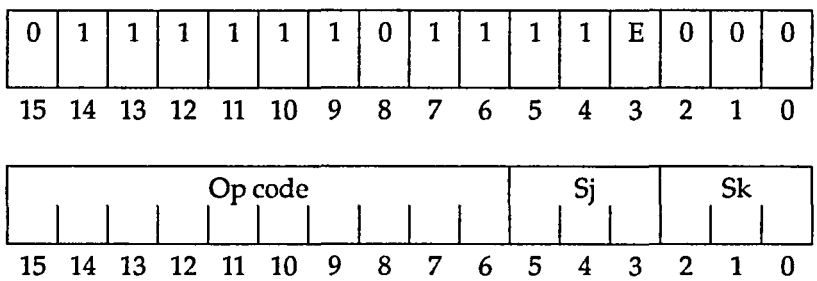
Purpose

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.

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
frint.s Sj,Sk	E0	4700	0100011100	RO	Integerizefloat single scalar
frint.d Sj,Sk	E0	4740	0100011101	RO	Integerize float double scalar

Description

The integer portion of Sj replaces Sk.

Operation

```
switch (FLOAT_LENGTH) {
    case SINGLE:    /* .s */
        Sk = aint(Sj); /*truncate from single float*/
        break;
    case DOUBLE:    /* .d */
        Sk = dint(Sj); /*truncate from double float*/
        break; } /* end of switch */
```

Exception

s | d	Reserved operand

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 register

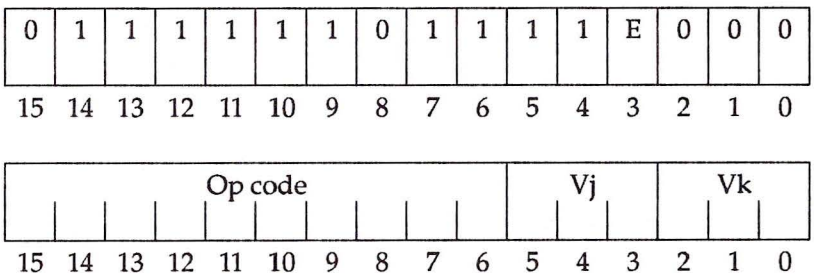
Purpose

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.

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
frint.s Vj,Vk	E0	5880	0101100010	RO	Integerize float single vector
frint.d Vj,Vk	E0	58C0	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.

Operation

```
switch (FLOAT_LENGTH) {
    case SINGLE:    /* .s */
        Vk[a] = aint(Vj[a]); /*truncate from single float*/
        break;
    case DOUBLE:    /* .d */
        Vk[a] = dint(Vj[a]); /*truncate from double float*/
        break; } /* end of switch */
```

Exception

`s|d`	Reserved operand

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}.{t|f} Vj,Vk

Integerize floating vector register (masked)

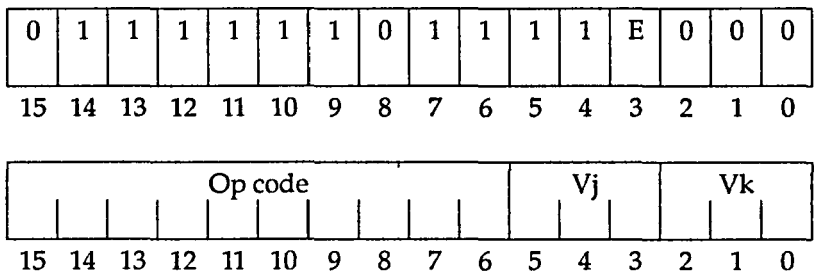
Purpose

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

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
frint.s.t Vj,Vk	E1	5980	0101100110	RO	Integerize single vector (VM)
frint.s.f Vj,Vk	E0	5980	0101100110	RO	Integerize single vector (!VM)
frint.d.t Vj,Vk	E1	59C0	0101100111	RO	Integerize double vector (VM)
frint.d.f Vj,Vk	E0	59C0	0101100111	RO	Integerize double vector (!VM)

frint.{s|d}.{t|f} Vj,Vk

Description

The integer portion of the contents of each of the first VL elements of vector register Vj replaces the corresponding elements of Vk, only if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

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;} /* end of switch */
```

Exceptions

`s|d`	Reserved operand

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.w *Ceffa*,Ak

Get register (communication as address)

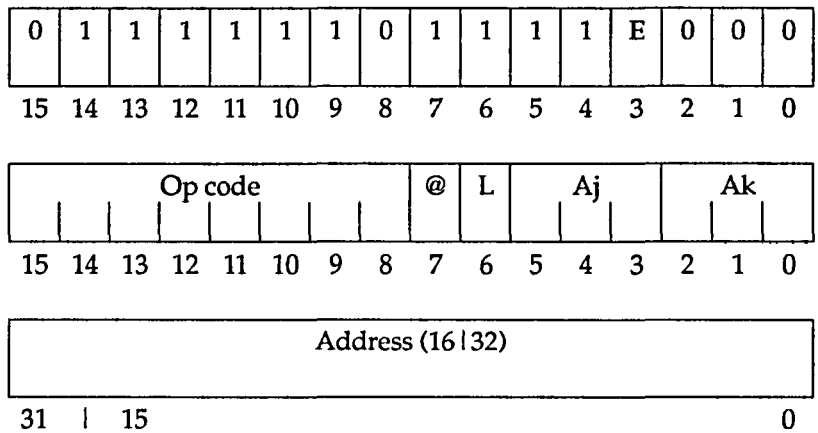
Purpose

To get the contents of a communication register into an address register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
get.w *Ceffa*,Ak	E0	2A00	0010101000	CAT	Get communication as address

Description

A word of data, bits <31..0>, is moved from c (*Ceffa*) to Ak. The lock bit, L (*Ceffa*), is not modified.

Operation

```
Ak = c(Ceffa)<31..0>;
```

Exception

Ring violation (invalid communication register address)

Notes

1. The @ and L fields are 0.
2. The memory dual of this instruction is `getr.w` *effa*,Ak

get.l *Ceffa*,Sk

Get register (communication as scalar)

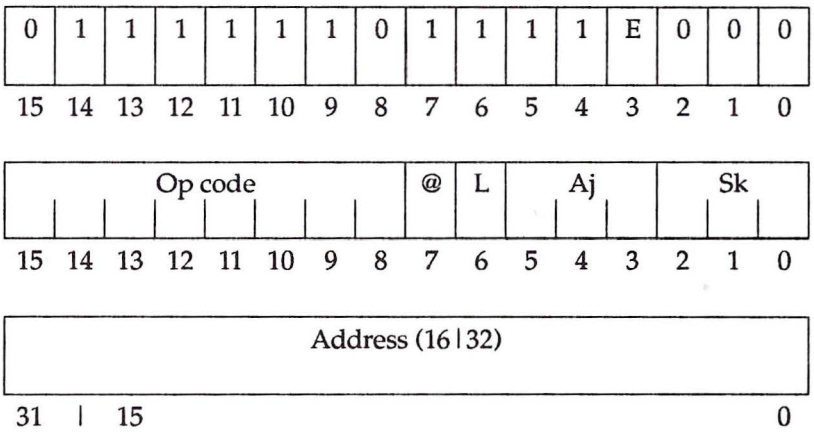
Purpose

To get the contents of a communication register into a scalar register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
get.l *Ceffa*,Sk	E0	3200	0011001000	CAT	Get communication as scalar

Description

A longword of data is moved from `c` (*Ceffa*) to Sk. The lock bit, `L` (*Ceffa*), is not modified.

Operation

```
Sk = c (Ceffa) ;
```

Exceptions

Ring violation (invalid communication register address)

Notes

1. The @ and L fields are 0.
2. The memory dual of this instruction is `getr.l` *effa*,`Sk`

getr.w *effa*,Ak

Get register (resource to address)

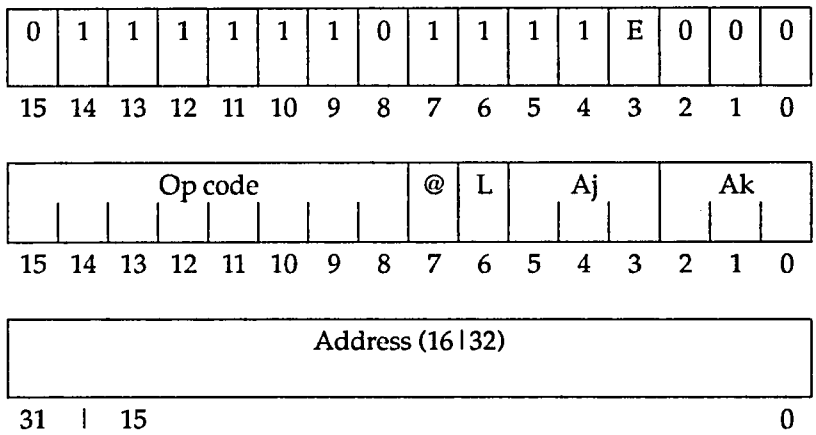
Purpose

To get the contents of a resource structure in memory into an address register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
getr.w *effa*,Ak	E0	2200	0010001000	C	Get resource to address

Description

A word of data, bits <31..0> of c(*effa*.data) is atomically moved from a resource structure to Ak. If C is returned as 0, the resource structure was in transition and was unable to be locked. If C is returned as 1, the operation was successful, and the resource structure is unlocked. c(*effa*.data) is not modified.

Operation

```
if (tas(effa.lock)) {
    Ak = c(effa.data)
    C = 1
    tac(effa.lock) }
else {
    C = 0    /* fail—structure in transition */
}
```

Exception

Ring violation (invalid communication register address)

Notes

1. This instruction is atomic.
2. The @ and L fields are 0.
3. Address carry (C) is the success status, not the carry-out, of the `getr` operation.
4. The communication register dual of this instruction is `get.w` *Ceffa*,Ak.

getr.l *effa*,Sk
Get register (resource to scalar)

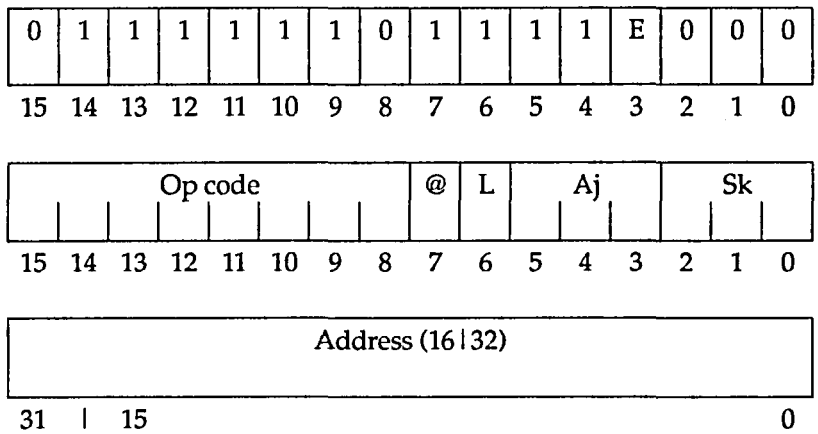
Purpose

To get the contents of a resource structure in memory into a scalar register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
getr.l *effa*,Sk	E0	2300	0010001100	SC	Get resource to scalar

Description

A longword of data, c(*effa*.data) is atomically moved from a resource structure to Sk. If SC is returned as 0, the resource structure was in transition and was unable to be locked. If SC is returned as 1, the operation was successful, and the resource structure is unlocked. c(*effa*.data) is not modified.

Operation

```
if (tas(effa.lock)) {
    Sk = c(effa.data)
    SC = 1
    tac(effa.lock) }
else {
    SC = 0   /* fail—structure in transition */
}
```

Exceptions

Ring violation (invalid communication register address)

Notes

1. This instruction is atomic.
2. The @ and L fields are 0.
3. Scalar carry (SC) is the success status, not the carry-out, of the `getr` operation.
4. The communication register dual of this instruction is `get.l` *Ceffa*,Sk

halt #N,Ak
Halt the CPU

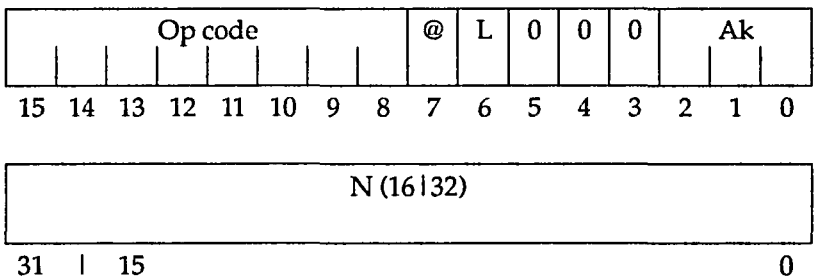
Purpose

To halt the central processing unit (CPU)

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
halt #N,Ak	ST	1000	000100000	None	Halt the CPU

Description

The sign-extended long immediate operand (#N, length indicated by L) replaces the contents of address register Ak. The CPU is halted. Further action is machine-implementation dependent.

Operation

```
Ak = Immediate;
(halt CPU);
```

Exception

Ring violation (privileged instruction)

Notes

1. The @ field is 0.
2. This instruction is typically used for diagnostic and debugging purposes.

idle Sk
Idle the CPU

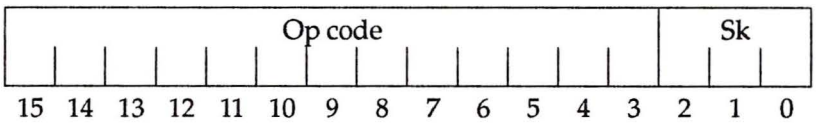
Purpose

To attempt to accept a posted fork in the specified CIR. If one is not posted, idle the current CPU without deallocating the current thread.

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
idle Sk	ST	7C58	0111110001011	None	Idle the CPU

Description

A search is begun for a posted fork in the CIR specified in Sk. If a posted fork exists, a new thread is allocated in the CIR, 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. Refer to the *CONVEX Architecture Reference Manual (C Series)*, Chapter 9, "Multiprocessor management," and Chapter 11, "Operating system interrupts," for information about the specific actions of an idle CPU.

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) */
```

Exception

Ring violation (privileged instruction)

Notes

1. This instruction may be traced, that is, if TTC or TIT 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.

inc.w *Ceffa*,Ak

Increment register (communication by address)

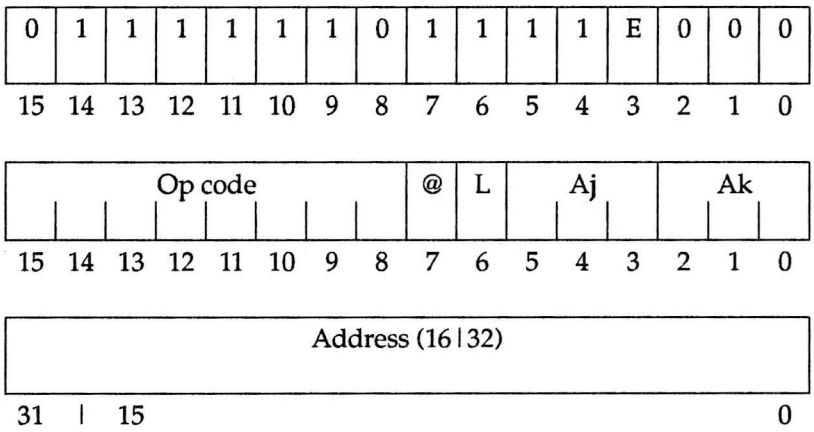
Purpose

To increment a communication register by an address register and return the new value

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Description

A communication register is atomically incremented by Ak and its value is returned in Ak. Success or failure status is returned in carry (C). The hardware lock bit is considered a *valid* bit, that is, if the bit is 1, there is valid data to be incremented.

Op code

Mnemonic	Space	Hex	Binary	PSW	Description
inc.w *Ceffa*,Ak	E0	2D00	0010110100	C,AIV,CAT	Increment communication/address

Operation

```
if (C = rcv (Ceffa,temp)) {
        /* 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

Notes

1. This instruction is atomic.
2. The @ and L fields are 0.
3. Address carry (C) is the success status, not the carry-out, of the `inc.w` *Ceffa*,Ak operation. Ak is unchanged if the operation fails (C = 0).
4. The memory dual of this instruction is `incr.w` *effa*,Ak.

inc.l *Ceffa*,Sk

Increment register (communication by scalar)

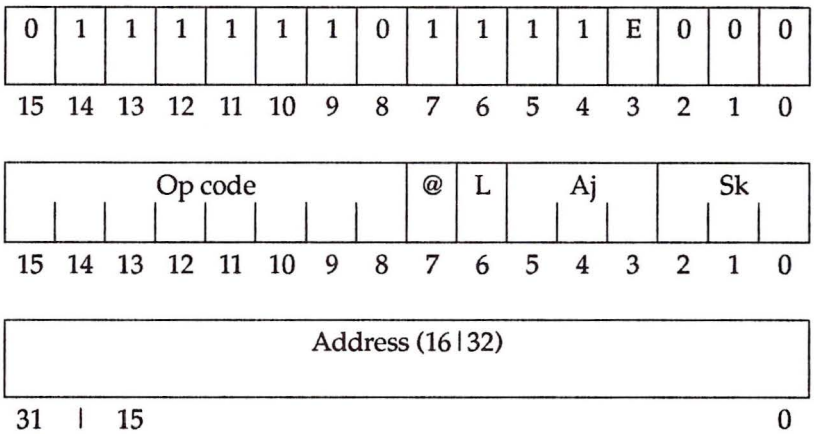
Purpose

To increment a communication register by a scalar register and return the new value

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
inc.l *Ceffa*,Sk	E0	3500	0010110100	SC,SIV,CAT	Increment communication/scalar

Description

A communication register is atomically incremented by Sk and its value is returned in Sk. Success or failure status is returned in the scalar carry (SC). The hardware lock bit is considered a *valid* bit, that is, if the bit is 1, there is valid data to be incremented.

Operation

```
if (SC = rcv (Ceffa, temp)) {
        /* 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

Notes

1. This instruction is atomic.
2. The @ and L fields are 0.
3. Scalar carry (SC) is the success status, not the carry-out, of the `inc.l` *Ceffa*,`Sk` operation. The scalar register Sk is unchanged if the operation fails (SC = 0).
4. The memory dual of this instruction is `incr.l` *effa*,Sk.

incr.w *effa*,Ak

Increment register (resource by address)

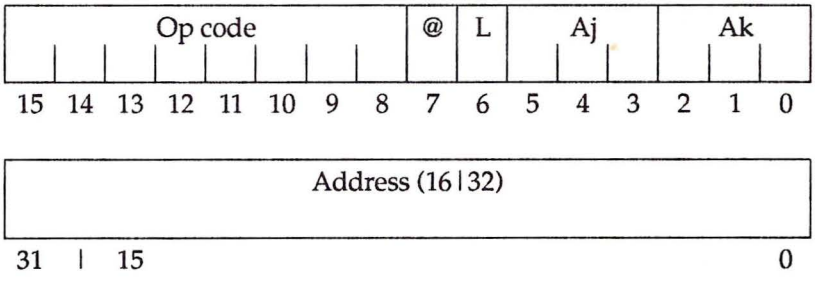
Purpose

To increment the data field of a resource structure by an address register and return the new value

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
incr.w *effa*,Ak	ST	2B00	0010101100	C,AIV	Increment resource structure data

Description

The data word of a resource structure is atomically incremented by Ak and its value is returned in Ak. Success or failure status is returned in carry (C). The hardware lock bit is considered a *valid* bit, that is, if the bit is 1, there is valid data to be incremented.

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() */
```

Exception

Deadlock exception

Notes

1. This instruction is atomic.
2. The @ and L fields are 0.
3. Address carry (C) is the status, not the carry-out, of the `incr.w` *effa*,Ak operation. Ak is unchanged if the `incr` operation fails (C = 0).
4. The communication register dual of this instruction is `inc.w` *Ceffa*, Ak.

incr.l *effa*,Sk

Increment resource by a scalar

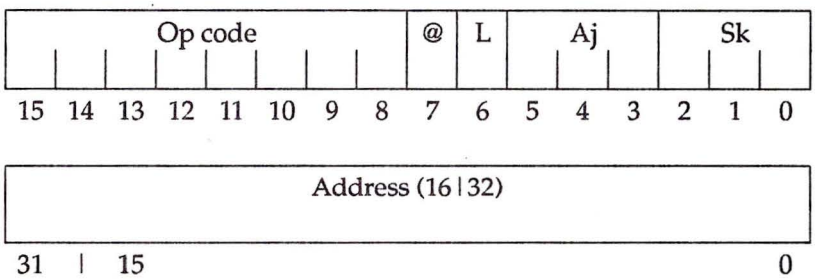
Purpose

To increment the data field of a longword resource structure by a scalar register and return the new value

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
incr.l *effa*,Sk	ST	2F00	0010111100	SC,SIV	Increment long resource structure

Description

The data longword of a resource structure is atomically incremented by Ak and its value is returned in Ak. Success or failure status is returned in scalar carry (SC). The hardware lock bit is considered a *valid* bit, that is, if the bit is 1, there is valid data to be incremented.

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() */
```

Exception

Deadlock exception

Notes

1. This instruction is atomic.
2. The @ and L fields are 0.
3. Scalar carry (SC) is affected as described by the preceding operation pseudocode, that is, 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).
4. The communication register dual of this instruction is `inc.l` *Ceffa*, Sk.

jmp *effa*
Jump unconditionally or on PSW bits

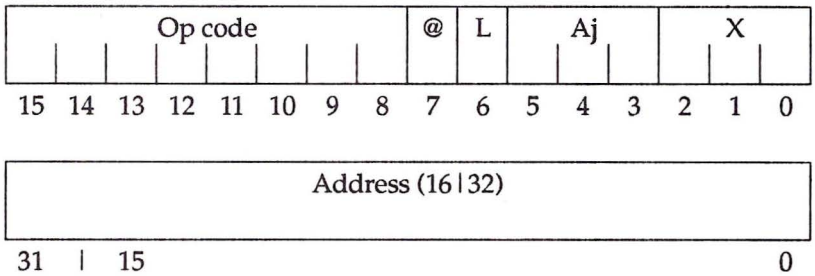
Purpose

To jump to an address conditionally or unconditionally

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
jmp *effa*	ST	0100	00000001	None	Jump always
jmpi.f *effa*	ST	0200	00000010	None	Jump on ION false
jmpi.t *effa*	ST	0300	00000011	None	Jump on ION true
jmpa.f *effa*	ST	0400	00000100	None	Jump on address carry false
jmpa.t *effa*	ST	0500	00000101	None	Jump on address carry true
jmps.f *effa*	ST	0600	00000110	None	Jump on scalar carry false
jmps.t *effa*	ST	0700	00000111	None	Jump on scalar carry true

Description

If the jump is unconditional, or the specified condition (ION, C, or SC) is the value specified (true or false), the instruction's effective address replaces the contents of the program counter (PC). If the specified condition is not the specified value, then the next sequential instruction is executed.

Operation

```
if (Op code_Cond) {
    PC = Effa; }
```

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 multiprocessing C Series CPUs. 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 unused.

join
Join all threads

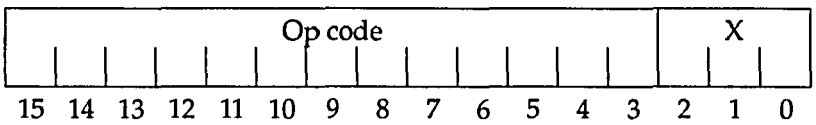
Purpose

To force all threads created by a process to join at a single execution point

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
join	ST	7CA0	0111110010100	None	Join all threads

Description

Any outstanding fork event in the hardware communication registers in the current CIR is marked "STOPPED". This signals idle CPUs that the process is joining and the fork should not be taken.

If the current thread is not the last thread to reach the `join` in the instruction stream, the CPU is relinquished. If the current thread is the last thread (all other threads have reached the `join` in the instruction stream), execution continues sequentially after the `join`.

Refer to the *CONVEX Architecture Reference Manual (C Series)*, Chapter 9, "Multiprocessor management," and Chapter 11, "Operating system interrupts," for information about the specific actions of an idle CPU.

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; }
```

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, that is, if the trace thread concurrency bit, PSW (TTC) or thread initialization trap bit, PSW (TIT) 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.

lck *Ceffa*
Lock communication register

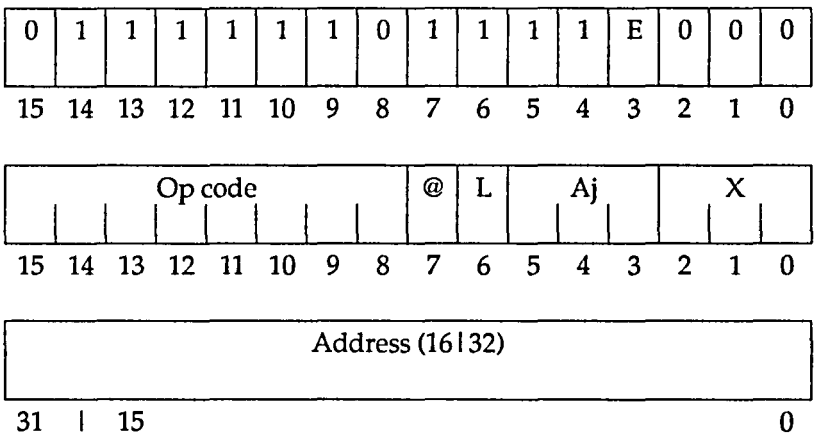
Purpose

To lock a communication register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
lck *effa*	E0	0200	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.

Operation

```
if (L(Ceffa) == 0) {
    L(Ceffa) = 1;
    C = 1; }
else {
    C = 0; }
```

Exceptions

Ring violation (invalid communication register address)
Deadlock exception

Notes

1. This is an atomic instruction.
2. The @ and L fields are 0.
3. The X field is unused.

ld.{h|w} #{n|N},Ak
Load register (address with immediate)

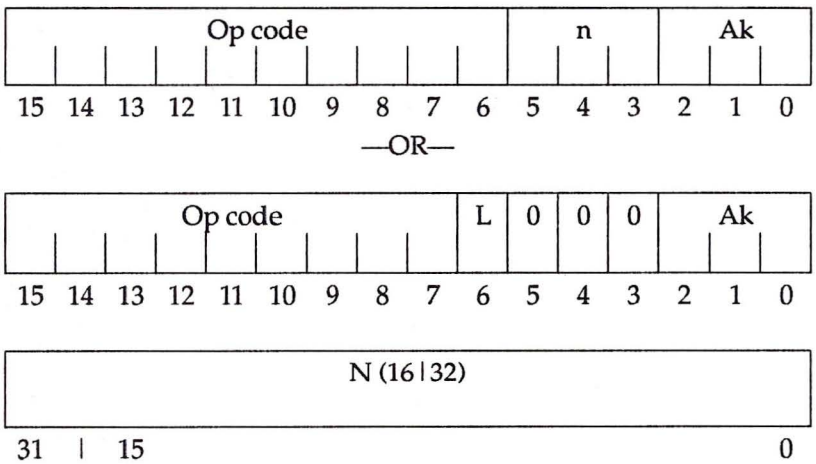
Purpose

To load an address register with an immediate operand

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
ld.h #n,Ak	ST	4480	0100010010	None	Load short immediate into Ak
ld.w #n,Ak	ST	44C0	0100010011	None	Load short immediate into Ak
ld.h #N,Ak	ST	1100	000100010	None	Load halfword immediate into Ak
ld.w #N,Ak	ST	1180	000100011	None	Load word immediate into Ak

Description

Either the short immediate operand (#n) or the sign-extended long immediate operand (#N, length indicated by L) replaces the contents of address register Ak.

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

Operation

```
Ak = Immediate;
```

ld.{w|s|l|d|u|du|dl|lu|ll} #N,Sk

Load register (scalar with immediate)

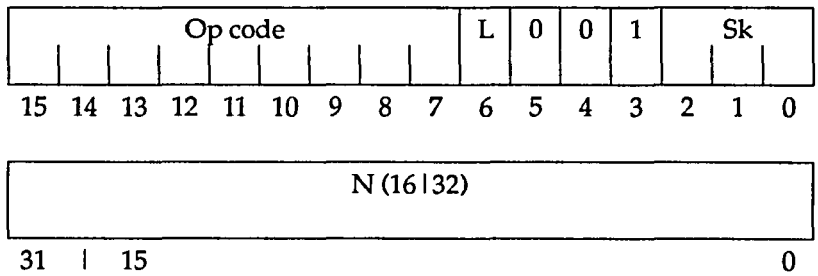
Purpose

To load an immediate operand into a scalar register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
ld.d #N,Sk	ST	1008	000100000	None	Load double float immediate upper 32 bits
ld.l #N,Sk	ST	1108	000100010	None	Load 32-bit immediate sign-extended to 64 bits
ld.w #N,Sk	ST	1188	000100011	None	Load a 32-bit immediate
ld.s #N,Sk	ST	1188	000100011	None	Load a single float immediate
ld.u #N,Sk	ST	1088	000100001	None	Load immediate, upper half
ld.du #N,Sk	ST	1088	000100001	None	Load 64-bit floating immediate, upper half
ld.dl #N,Sk	ST	1188	000100011	None	Load 64-bit floating immediate, lower half
ld.lu #N,Sk	ST	1088	000100001	None	Load 64-bit integer immediate, upper half
ld.ll #N,Sk	ST	1188	000100011	None	Load 64-bit integer immediate, lower half

ld.{w|s|l|d|u|du|dl|lu|ll} #N,Sk

Description

The sign-extended long immediate operand (#N, length indicated by L) replaces the contents of scalar register Sk.

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<32..0> = Immediate;
        break;
    case TYPE_U:    /* ld.u  */
    case TYPE_DU:   /* ld.du */
    case TYPE_LU:   /* ld.lu */
        Sk<63..32> = Immediate;
        break; }        /* end of switch */
```

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 synonymous to `ld.w #N, Sk`, as shown above.

ld.w #N,VL
Load register (VL with immediate)

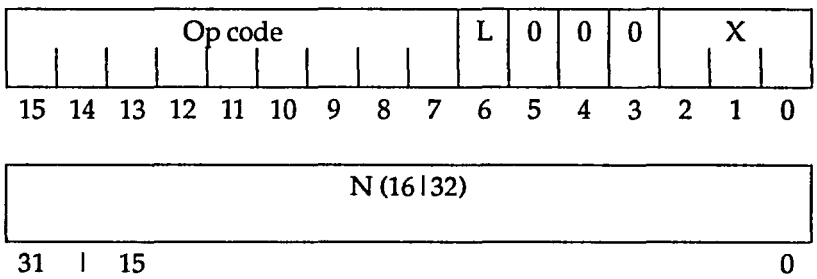
Purpose

To load the vector length (VL) register with an immediate operand

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
ld.w #N,VL	ST	1800	000110000	None	Load VL with immediate

Description

The sign-extended long immediate operand (#N, length indicated by L), range limited to [0..128], replaces the contents of the VL register.

Operation

```
if (Immediate >= 128) {
    VL = 128; }
else {
    if (Immediate <= 0) {
        VL = 0; }
    else {
        VL = Immediate; } }
```

Notes

1. The @ field is 0.
2. The X field is unused.

ld.w #N,VS

Load register (VS with immediate)

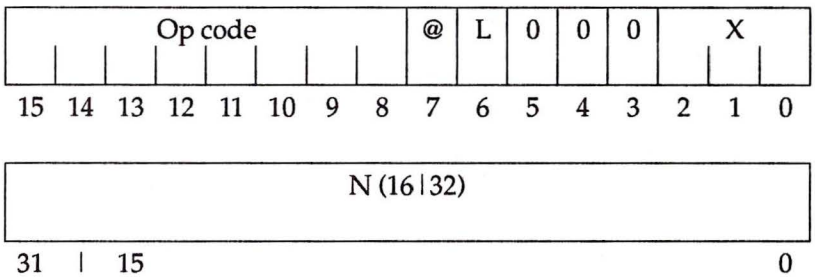
Purpose

To load the vector stride (VS) register from an immediate operand

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
ld.w #N,VS	ST	1880	000110001	None	Load VS from an immediate

Description

The sign-extended long immediate operand (#N, length indicated by L) replaces the contents of the 32-bit VS register.

Operation

```
VS = Immediate;
```

Notes

1. The @ field is 1.
2. The X field is unused.

ld.{b|h|w} *effa*,Ak
Load register (address with memory)

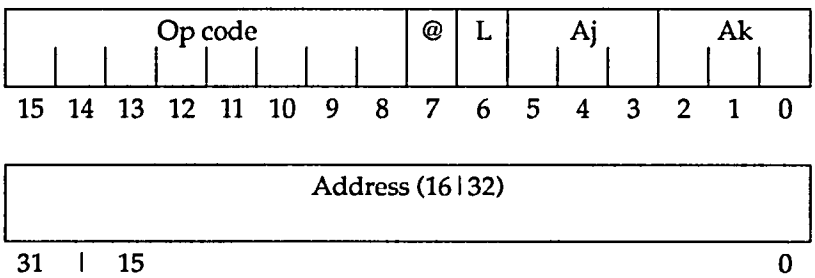
Purpose

To load memory contents into an address register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
ld.b *effa*,Ak	ST	2800	00101000	None	Load address register byte
ld.h *effa*,Ak	ST	2900	00101001	None	Load address register halfword
ld.w *effa*,Ak	ST	2A00	00101010	None	Load address register word

Description

The operand referenced by the effective address replaces the contents of address register Ak.

Operation

```
Ak = c(effa);
```

Note

Byte operands replace only bits <7..0> of address register Ak. Halfword operands replace bits <15..0> of address register Ak.

ld.{b|h|w|l|s|d} *effa*,Sk
Load register (scalar with memory)

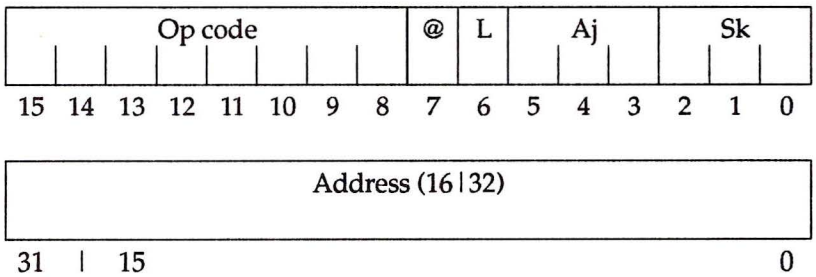
Purpose

To load memory contents into a scalar accumulator

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
ld.b *effa*,Sk	ST	3000	00110000	None	Load scalar byte
ld.h *effa*,Sk	ST	3100	00110001	None	Load scalar halfword
ld.w *effa*,Sk	ST	3200	00110010	None	Load scalar word
ld.l *effa*,Sk	ST	3300	00110011	None	Load scalar longword
ld.s *effa*,Sk	ST	3200	00110010	None	Load scalar single float
ld.d *effa*,Sk	ST	3300	00110011	None	Load scalar double float

Description

The operand referenced by the effective address replaces the contents of scalar register Sk.

Operation

```
Sk = c(effa);
```

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 register (vector with memory)

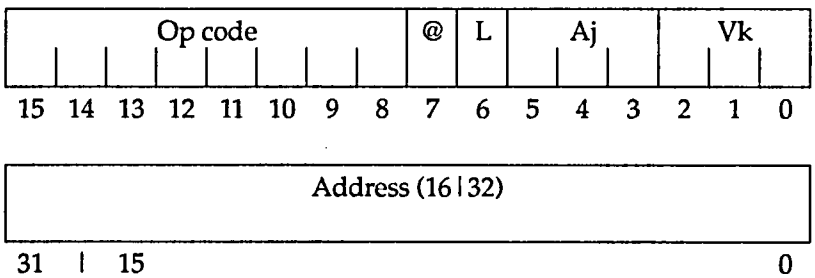
Purpose

To load memory contents into a vector accumulator

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
ld.b *effa*,Vk	ST	3800	00111000	None	Load vector byte
ld.h *effa*,Vk	ST	3900	00111001	None	Load vector halfword
ld.w *effa*,Vk	ST	3A00	00111010	None	Load vector word
ld.l *effa*,Vk	ST	3B00	00111011	None	Load vector longword
ld.s *effa*,Vk	ST	3A00	00111010	None	Load vector single float
ld.d *effa*,Vk	ST	3B00	00111011	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.

Operation

```
temp = effa;
for (a = 0; a  VL; a++) {
    Vk[a] = c(temp);    /* according to type */
    temp = temp + VS; }
```

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.

ld.{b|h|w|l|s|d}.{t|f} *effa*,Vk

Load register (vector with memory) (masked)

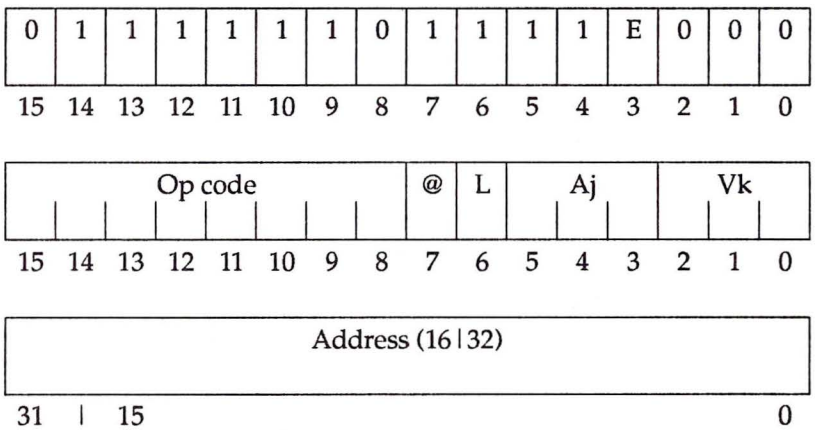
Purpose

To load a vector into a vector accumulator under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
ld.b.t *effa*,Vk	E1	3800	00111000	None	Load vector byte (VM)
ld.b.f *effa*,Vk	E0	3800	00111000	None	Load vector byte (!VM)
ld.h.t *effa*,Vk	E1	3900	00111001	None	Load vector halfword (VM)
ld.h.f *effa*,Vk	E0	3900	00111001	None	Load vector halfword (!VM)
ld.w.t *effa*,Vk	E1	3A00	00111010	None	Load vector word (VM)
ld.w.f *effa*,Vk	E0	3A00	00111010	None	Load vector word (!VM)
ld.l.t *effa*,Vk	E1	3B00	00111011	None	Load vector longword (VM)
ld.l.f *effa*,Vk	E0	3B00	00111011	None	Load vector longword (!VM)
ld.s.t *effa*,Vk	E1	3A00	00111010	None	Load vector single float (VM)
ld.s.f *effa*,Vk	E0	3A00	00111010	None	Load vector single float (!VM)
ld.d.t *effa*,Vk	E1	3B00	00111011	None	Load vector double float (VM)
ld.d.f *effa*,Vk	E0	3B00	00111011	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.

Operation

```
temp = effa;
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; }
        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; }
        break; }      /* end of switch */
```

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.

ld.l *effa*,VLS
Load registers (VS and VL)

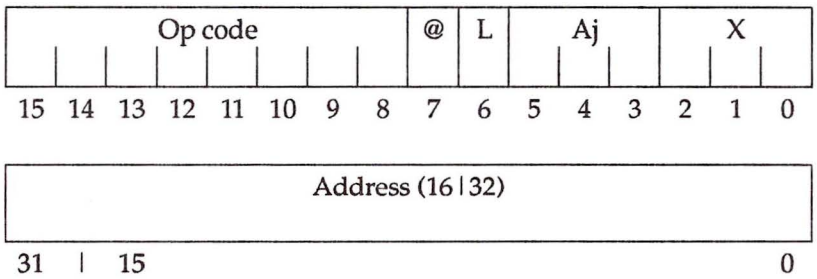
Purpose

To load the vector stride (VS) register and the vector length (VL) register from memory

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
ld.l *effa*,VLS	ST	0A00	00001010	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].

Operation

```
VS = c(effa)<63..32>;
VL = c(effa)<31..0>;
```

Note

The X field is unused.

ld.x *effa*, VM
Load register (VM)

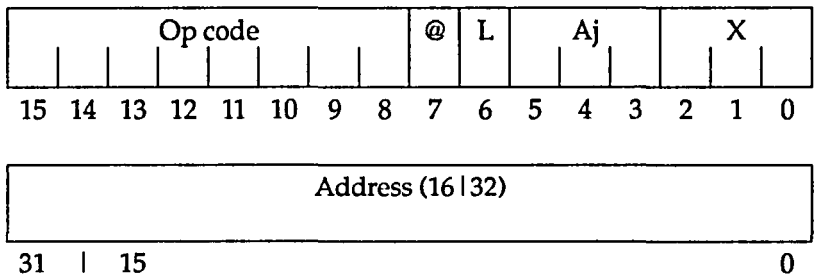
Purpose

To load the vector merge (VM) register from memory

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
ld.x *effa*, VM	ST	0B00	00001011	None	Load VM from memory

Description

The 128 bits (16 bytes) beginning at the effective address replace the value of the VM register.

Operation

```
VM = c(effa)<127..0> ;
```

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 unused.

ldb.{b|h|w|l|s|d} *effa*,Sk

Load register (scalar with memory) (cache bypass)

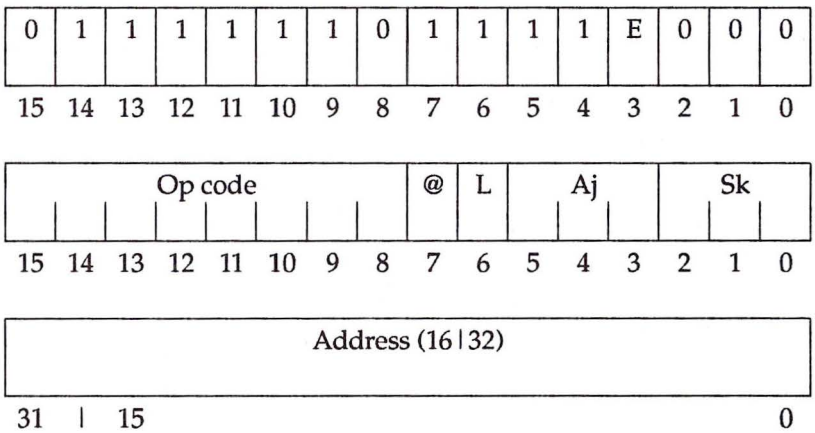
Purpose

To load memory contents into a scalar accumulator without modifying or accessing the data cache.

Application

C200/C3200 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
ldb.b *effa*,Sk	E0	0B00	00001011	None	Load bypass scalar byte
ldb.h *effa*,Sk	E0	0F00	00001111	None	Load bypass scalar halfword
ldb.w *effa*,Sk	E0	2C00	00101100	None	Load bypass scalar word
ldb.l *effa*,Sk	E0	3400	00110100	None	Load bypass scalar longword
ldb.s *effa*,Sk	E0	2C00	00101100	None	Load bypass scalar single float
ldb.d *effa*,Sk	E0	3400	00110100	None	Load bypass scalar double float

Description

The operand referenced by the effective address replaces the contents of scalar register Sk. The data cache is not affected.

Operation

```
Sk = c(effa);
```

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.
3. No indirect addressing modes may be used with this instruction (field @ is 0).
4. The data cache (Dcache) is not affected; there is no cache update or invalidate.
5. On early revisions of the C200 Series CPUs, this instruction was considered a vector instruction, and therefore caused a vector valid trap to occur. Refer to the *CONVEX Architecture Reference Manual (C Series)*, Chapter 12, "Operating system exceptions," for more information about the vector valid trap.

ldcmr *effa*,Ak

Load registers (communication with memory)

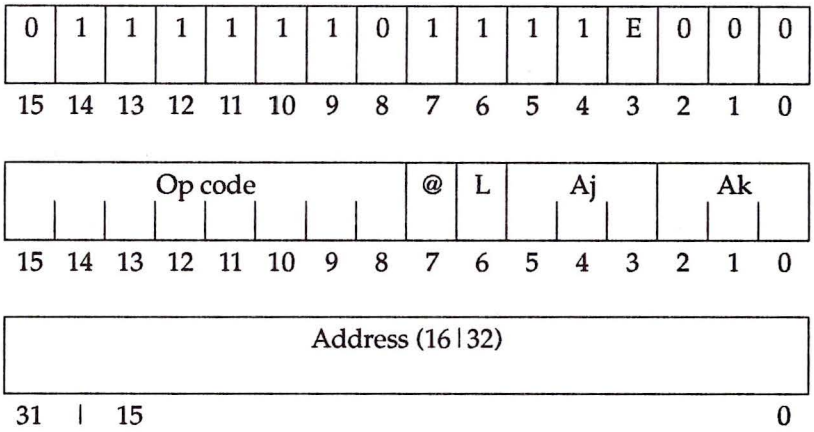
Purpose

Load a specified set of communication registers from memory

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
ldcmr *effa*,Ak	E0	0600	00000110	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 (except in C3800 Series CPUs), and if they are nonzero the associated communication register lock bit longwords and communication registers are loaded. The communication register modified bits (except in C3400 Series CPUs) are then zeroed. In C3800 Series CPUs these actions are unconditional.

The memory map that defines the exact format of the memory data is implementation-specific. Refer to the *CONVEX Architecture Reference Manual (C Series)*, Chapter 5, "Logical (virtual) address space" and Chapter 6, "Communication registers" for details.

Operation

```
/* For the CIR specified in Ak: */
if (ring 0 CMR valid bit == 1) {     /* memory copy */
    Load hardware CMRs and lock bits;
    Load ring 0 CMRs and lock bits; }
if (ring 4 CMR valid bit == 1) {     /* memory copy */
    Load ring 4 CMRs and lock bits; }
Ring 0 CMR modified bit = 0;
Ring 4 CMR modified bit = 0;
```

Exception

Ring violation (privileged instruction)

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.

Idea *effa*,Ak

Load register (address with memory)

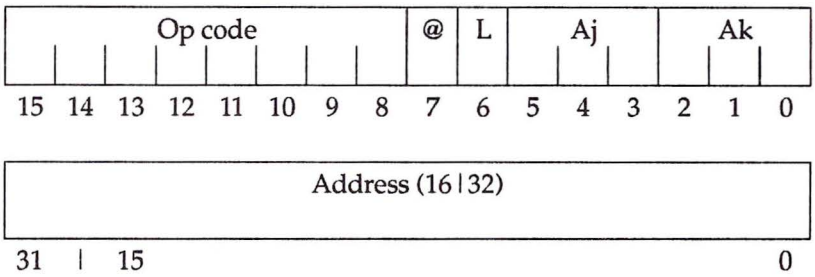
Purpose

To load an address register with a calculated effective address

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
Idea *effa*,Ak	ST	0900	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.

Operation

```
Ak = effa;
```

Exception

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.

Idea *effa*,Sk

Load register (scalar with memory)

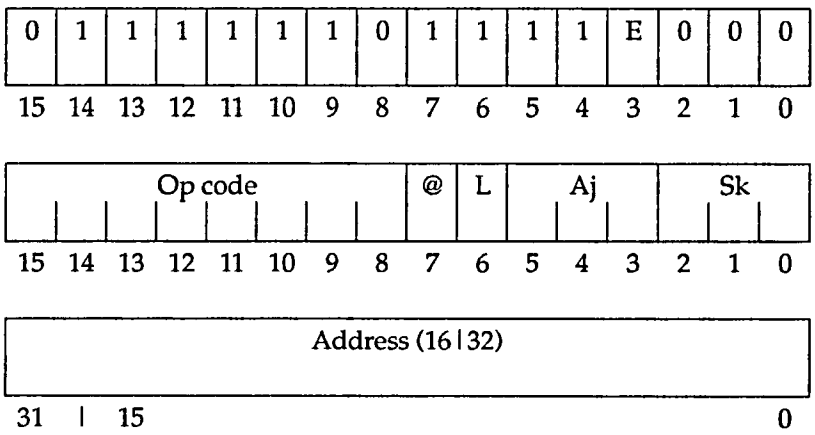
Purpose

To load a scalar register with a calculated effective address

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
Idea *effa*,Sk	E0	0400	00000100	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.

Operation

`Sk<31..0>` = *effa*;

Exception

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

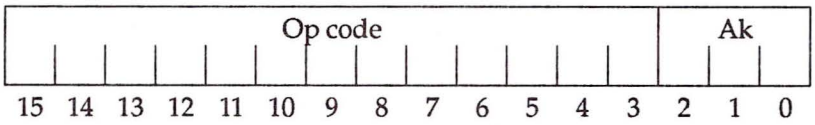
Purpose

To load values in all 8 segment descriptor registers (SDRs)

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
ldkdr Ak	ST	7C08	0111110000001	None	Load all 8 SDRs

Description

Address register Ak contains the effective address of an 8-word table. The contents of this table replaces the SDRs, beginning with SDR0.

Operation

```
SDR(0,1,...,7) = c(Ak)(0,1,...,7);
```

Exception

Ring violation (privileged instruction)

Notes

1. The data to be loaded into the ring 0 (kernel) SDRs of C100 Series CPUs 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. On C100 Series CPUs, the addressing environment must be physical; address translation must be disabled, or a machine exception results.
3. Since the multiprocessing C Series CPUs always have address translation enabled, the only safe method to execute this instruction is to use the same SDR0 that is already loaded.
4. After loading all 8 entries into SDR <0..7>, the C100 Series CPUs purge the ATU, logical, and instruction caches and enables logical address translation.
5. After loading all 8 entries into SDR <0..7>, the multiprocessing C Series CPUs purge the ATU only.
6. For C3400 Series CPUs there is no automatic purging. This must be done explicitly.

ldpa Aj,Ak
Load physical address

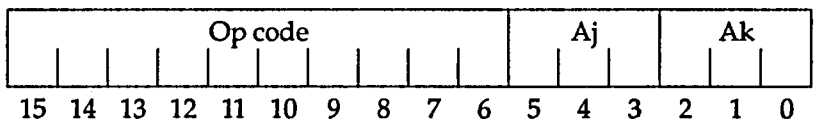
Purpose

To convert a logical address to a physical address and return the result in an address register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
ldpa Aj,Ak	ST	4400	0100010000	C	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 address carry (C) is cleared 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 Manual (C Series)*, Chapter 12, "Operating system exceptions."

Operation

```
/* The logical address in address register Aj
* is converted into a physical address and 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
* page of a multiple-CPU processor: */

    if ( (pte2[Aj].LT == 1) && (CPU = 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; }
```

Notes

1. This instruction can also be used to determine whether or not a page is resident in physical memory.
2. No access checks, that is, read, write, or execute, are performed.
3. If two or more address registers (Aj, Ak, and A5) are the same register, the results are unpredictable.

ldsdr Ak
Load process SDRs

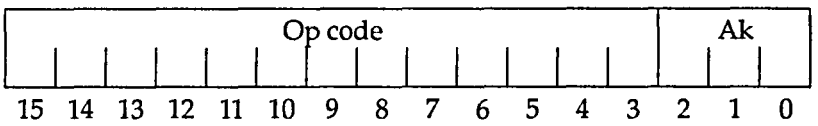
Purpose

To load values from memory into the process segment descriptor registers (SDRs)

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
ldsdr Ak	ST	7C00	0111110000000	None	Load process SDRs

Description

Address register Ak contains the effective address of an 7-word table. The contents of this table replaces the SDRs, beginning with SDR1.

Operation

```
SDR(1,2,...,7) = c(Ak)(1,2,...,7);
```

Exception

Ring violation (privileged instruction)

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 (except for C3400 Series CPUs). If the data are not resident or aligned, a machine exception occurs.
2. After the load, a C100 Series CPU purges the ATU, logical cache, and instruction cache of any entries associated with segments 1-7.
3. After the load, a multiprocessing C Series CPU purges the ATU only.
4. For C3400 Series CPUs there is no automatic purging. This must be done explicitly.

ldvi.{b|h|w|l|s|d} Vj,Vk

Load register (vector using vector index)

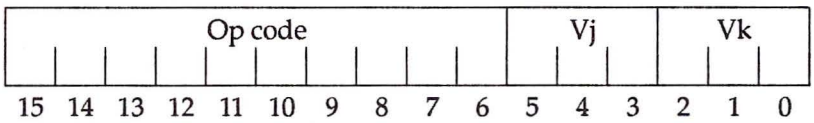
Purpose

To load a vector into a vector accumulator using a vector of indices ("vector gather")

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
ldvi.b Vj,Vk	ST	7800	0111100000	None	Index load vector byte
ldvi.h Vj,Vk	ST	7840	0111100001	None	Index load vector halfword
ldvi.w Vj,Vk	ST	7880	0111100010	None	Index load vector word
ldvi.l Vj,Vk	ST	78C0	0111100011	None	Index load vector longword
ldvi.s Vj,Vk	ST	7880	0111100010	None	Index load vector single float
ldvi.d Vj,Vk	ST	78C0	0111100011	None	Index load vector double float

Description

The lower 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.

Operation

```
for (a = 0; a  VL; a++) {
    addr = A5 + Vj[a]<31..0>;
    Vk[a] = c(addr); }  /* according to type */
```

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 register (vector using vector index) (masked)

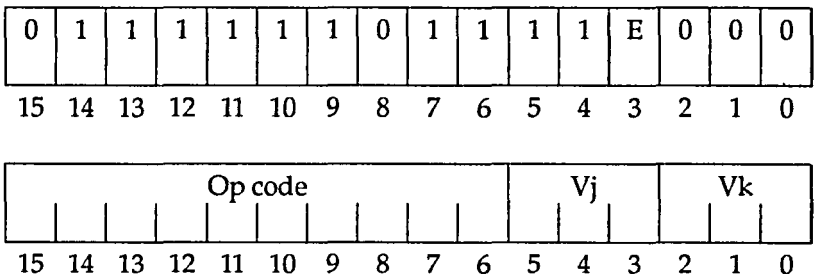
Purpose

To load a vector into a vector accumulator using a vector of indices under control of the vector merge (VM) register ("vector gather")

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
ldvi.b.t Vj,Vk	E1	7800	0111100000	None	Index load vector byte (VM)
ldvi.b.f Vj,Vk	E0	7800	0111100000	None	Index load vector byte (!VM)
ldvi.h.t Vj,Vk	E1	7840	0111100001	None	Index load vector halfword (VM)
ldvi.h.f Vj,Vk	E0	7840	0111100001	None	Index load vector halfword (!VM)
ldvi.w.t Vj,Vk	E1	7880	0111100010	None	Index load vector word (VM)
ldvi.w.f Vj,Vk	E0	7880	0111100010	None	Index load vector word (!VM)
ldvi.l.t Vj,Vk	E1	78C0	0111100011	None	Index load vector longword (VM)
ldvi.l.f Vj,Vk	E0	78C0	0111100011	None	Index load vector longword (!VM)
ldvi.s.t Vj,Vk	E1	7880	0111100010	None	Index load vector single (VM)
ldvi.s.f Vj,Vk	E0	7880	0111100010	None	Index load vector single (!VM)
ldvi.d.t Vj,Vk	E1	78C0	0111100011	None	Index load vector double (VM)
ldvi.d.f Vj,Vk	E0	78C0	0111100011	None	Index load vector double (!VM)

ldvi.{b|h|w|l|s|d}.{t|f} Vj,Vk

Description

The lower 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`).

Operation

```
switch (E) {        /* prefix bit */
    case TRUE: /* .t */
        for (a = 0; a < VL; a++) {
            if (VMa == 1) { /* if VM<a> is TRUE */
                addr = A5 + Vj[a]<31..0>;
                Vk[a] = c(addr); /* according to type */ } }
        break; /* go to end of switch */
    case FALSE:/* .f */
        for (a = 0; a < VL; a++) {
            if (VMa == 0) { /* if VM<a> is FALSE */
                addr = A5 + Vj[a]<31..0>;
                Vk[a] = c(addr); /* according to type */ } }
        break; }    /* end of switch */
```

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 register (address with immediate)

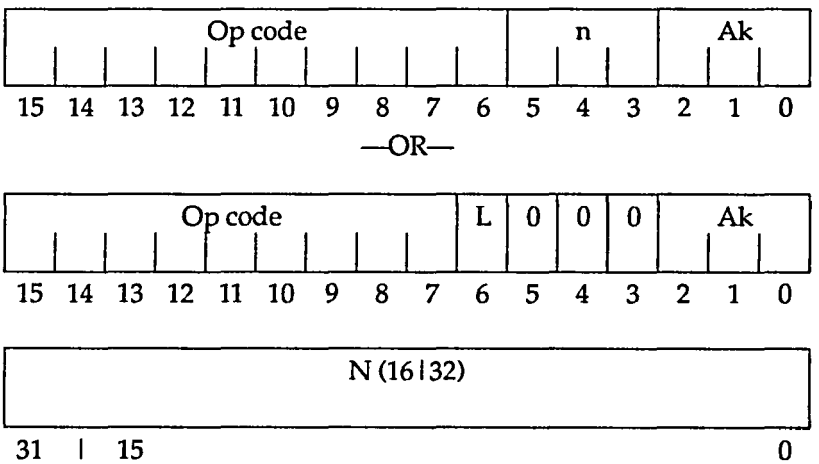
Purpose

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

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
le.h #n,Ak	ST	4C80	0100110010	C	Compare less than or equal halfword
le.h #N,Ak	ST	1E00	000111100	C	Compare less than or equal halfword
le.w #n,Ak	ST	4CC0	0100110011	C	Compare less than or equal word
le.w #N,Ak	ST	1E80	000111101	C	Compare less than or equal word
lt.h #N,Ak	ST	1F00	000111110	C	Compare less than halfword
lt.h #n,Ak	ST	4E80	0100111010	C	Compare less than halfword
lt.w #n,Ak	ST	4EC0	0100111011	C	Compare less than word
lt.w #N,Ak	ST	1F80	000111111	C	Compare less than word
eq.h #n,Ak	ST	4680	0100011010	C	Compare equal halfword
eq.h #N,Ak*	ST	1B00	000110110	C	Compare equal halfword
eq.w #n,Ak	ST	46C0	0100011011	C	Compare equal word
eq.w #N,Ak	ST	1B80	000110111	C	Compare equal word

{le|lt|eq}.{h|w} #{n|N},Ak

Description

The truth value of the comparison between the contents of the address register Ak and either the short immediate operand (#n) or the sign-extended long immediate operand (#N, length indicated by L) replaces address carry (C). If the comparison is true, C is set to 1. If the comparison is false, C bit is cleared to 0.

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

Operation

```
if (Op code_Test(Immediate,Ak) == TRUE) {
    C = 1; }
else {
    C = 0; }
```

Example

```
lt.w #1,a3
```

Sets address carry (C) to 1 when 1 is less than the contents of address register A3.

Notes

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

{le|lt|eq}.{h|w|s} #N,Sk

Compare register (scalar with immediate)

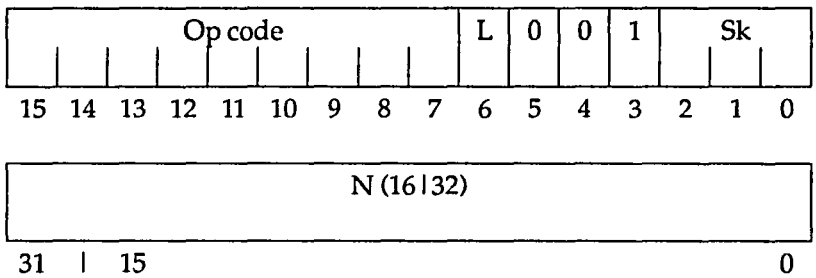
Purpose

To compare the contents of a scalar register and an immediate

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
le.h #N,Sk	ST	1E08	000111100	See note 1	Compare less than or equal halfword
le.w #N,Sk	ST	1E88	000111101	See note 1	Compare less than or equal word
le.s #N,Sk	ST	1A08	000110100	RO,SC	Compare less than or equal single
lt.h #N,Sk	ST	1F08	000111110	See note 1	Compare less than halfword
lt.w #N,Sk	ST	1F88	000111111	See note 1	Compare less than word
lt.s #N,Sk	ST	1A88	000110101	RO,SC	Compare less than single
eq.h #N,Sk	ST	1B08	000110110	See note 1	Compare equal halfword
eq.w #N,Sk	ST	1B88	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 long immediate operand (#N, length indicated by L) replaces scalar carry (SC). If the comparison is true, SC is set to 1. If the comparison is false, SC is cleared to 0.

Operation

```
if (Op code_Test(Immediate,Sk) == TRUE) {
    SC = 1; }
else {
    SC = 0; }
```

Exceptions

h | w	None
s	Reserved operand

Example

```
lt.w #1,s3
```

Sets carry (C) to 1 when 1 is less than the contents of scalar register S3.

Notes

1. Test for not equal to, greater than, and greater than or equal to by inverting the truth sense of the branch-on-carry instruction.
2. Detect floating-point reserved operands by using a compare word immediate operand (for equality with the floating point reserved operand as its immediate value).
3. The `eq.s` instruction is synonymous with `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 registers (address with address)

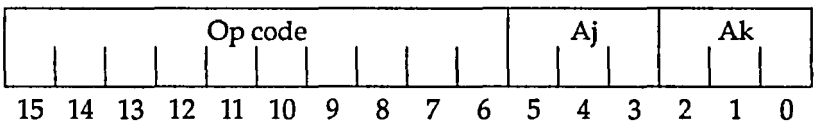
Purpose

To compare the contents of two address registers

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
le.h Aj,Ak	ST	4C00	0100110000	See note 1	Compare less than or equal signed halfword
le.w Aj,Ak	ST	4C40	0100110001	See note 1	Compare less than or equal signed word
lt.h Aj,Ak	ST	4E00	0100111000	See note 1	Compare less than signed halfword
lt.w Aj,Ak	ST	4E40	0100111001	See note 1	Compare less than signed word
eq.h Aj,Ak	ST	4600	0100011000	See note 1	Compare equal halfword
eq.w Aj,Ak	ST	4640	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 carry (C). If the comparison is true, C is set to 1. If the comparison is false, C is cleared to 0.

Operation

```
if (Op code_Test(Aj,Ak) == TRUE) {
    C = 1; }
else {
    C = 0; }
```

Example

```
lt.w a1,a3
```

Sets carry (C) to 1 when the contents of address register A1 are less than the contents of address register A3.

Notes

1. 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.
2. Unsigned equal comparison is equivalent to signed equal comparison.

{le|lt|eq}.{b|h|w|l|s|d} Sj,Sk

Compare registers (scalar with scalar)

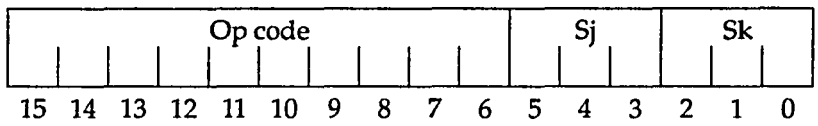
Purpose

To compare the contents of two scalar registers

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
le.b Sj,Sk	ST	4D00	0100110100	SC	Compare less than or equal byte
le.h Sj,Sk	ST	4D40	0100110101	SC	Compare less than or equal halfword
le.w Sj,Sk	ST	4D80	0100110110	SC	Compare less than or equal word
le.l Sj,Sk	ST	4DC0	0100110111	SC	Compare less than or equal longword
le.s Sj,Sk	ST	5400	0101010000	RO,SC	Compare less than or equal single float
le.d Sj,Sk	ST	5440	0101010001	RO,SC	Compare less than or equal double float
lt.b Sj,Sk	ST	4F00	0100111100	SC	Compare less than byte
lt.h Sj,Sk	ST	4F40	0100111101	SC	Compare less than halfword
lt.w Sj,Sk	ST	4F80	0100111110	SC	Compare less than word
lt.l Sj,Sk	ST	4FC0	0100111111	SC	Compare less than longword
lt.s Sj,Sk	ST	5480	0101010010	RO,SC	Compare less than single float
lt.d Sj,Sk	ST	54C0	0101010011	RO,SC	Compare less than double float
eq.b Sj,Sk	ST	4700	0100011100	SC	Compare equal byte
eq.h Sj,Sk	ST	4740	0100011101	SC	Compare equal halfword
eq.w Sj,Sk	ST	4780	0100011110	SC	Compare equal word
eq.l Sj,Sk	ST	47C0	0100011111	SC	Compare equal longword
eq.s Sj,Sk	ST	5600	0101011000	RO,SC	Compare equal single float
eq.d Sj,Sk	ST	5640	0101011001	RO,SC	Compare equal double float

Description

The truth value of the comparison between the contents of scalar registers Sk and Sj replaces scalar carry (SC). If the comparison is true, SC is set to 1. If false, SC is cleared to 0.

Operation

```
if (Op code_Test(Sj,Sk) == TRUE) {
    SC = 1; }
else {
    SC = 0; }
```

{le|lt|eq}.{b|h|w|l|s|d} Sj,Sk

Exceptions

`s|d`	Reserved operand

Example

```
lt.w S2,S4
```

Sets scalar carry (SC) to 1 when the contents of scalar register S2 are less than the contents of S4.

Notes

1. 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.
2. Unsigned equal comparison is equivalent to signed equal comparison.
3. 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 registers (vector with scalar)

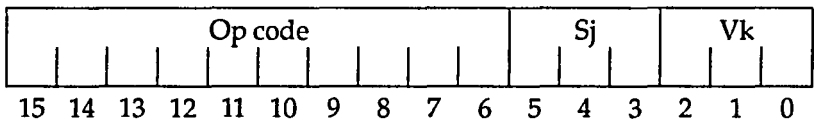
Purpose

To compare vector elements and the contents of a scalar register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
le.b Sj,Vk	ST	6B00	0110101100	None	Compare less than or equal byte
le.h Sj,Vk	ST	6B40	0110101101	None	Compare less than or equal halfword
le.w Sj,Vk	ST	6B80	0110101110	None	Compare less than or equal word
le.l Sj,Vk	ST	6BC0	0110101111	None	Compare less than or equal longword
le.s Sj,Vk	ST	6700	0110011100	RO	Compare less than or equal single
le.d Sj,Vk	ST	6740	0110011101	RO	Compare less than or equal double float
lt.b Sj,Vk	ST	6D00	0110110100	None	Compare less than byte
lt.h Sj,Vk	ST	6D40	0110110101	None	Compare less than halfword
lt.w Sj,Vk	ST	6D80	0110110110	None	Compare less than word
lt.l Sj,Vk	ST	6DC0	0110110111	None	Compare less than longword
lt.s Sj,Vk	ST	6780	0110011110	RO	Compare less than single
lt.d Sj,Vk	ST	67C0	0110011111	RO	Compare less than double float
eq.b Sj,Vk	ST	6900	0110100100	None	Compare equal byte
eq.h Sj,Vk	ST	6940	0110100101	None	Compare equal halfword
eq.w Sj,Vk	ST	6980	0110100110	None	Compare equal word
eq.l Sj,Vk	ST	69C0	0110100111	None	Compare equal longword
eq.s Sj,Vk	ST	6500	0110010100	RO	Compare equal single
eq.d Sj,Vk	ST	6540	0110010101	RO	Compare equal double precision

{le|lt|eq}.{b|h|w|l|s|d} Sj,Vk

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. If the comparison is false, the bit of VM is cleared to 0.

When VL is less than 128, the remaining bits of VM are cleared to 0.

Operation

```
for (a = 0; a < VL; a++) {
    if (Op code_Test(Sj,Vk[a]) == TRUE) {
        VM<a> = 1; }
    else {
        VM<a> = 0; }
if (VL < 128) {
    for (a = VL; a < 128; a++) {
        VM<a> = 0; } }   /* zero fill */
```

Exceptions

s|d	Reserved operand

Example

```
lt.h S4,V2
```

Sets the 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 registers (vector with scalar) (masked)

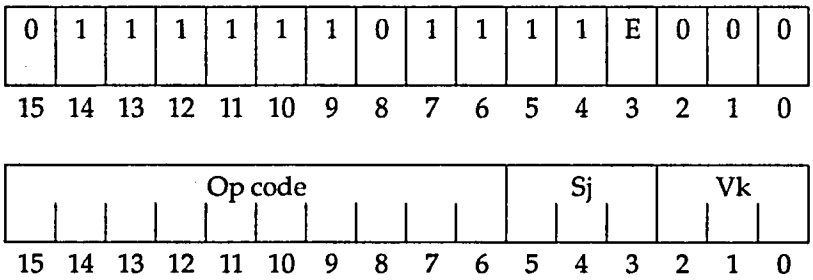
Purpose

To compare a vector and a scalar and load the vector merge (VM) under control of the VM register at the instruction entry

Application

C200/C3200, C3400, C3800 Series CPUs

Format



{le|lt|eq}.{b|h|w|l|s|d}.{t|f} Sj,Vk

lt.w.t Sj,Vk	E1	6D80	0110110110	None	Compare less than word (VM)
lt.w.f Sj,Vk	E0	6D80	0110110110	None	Compare less than word (!VM)
lt.l.t Sj,Vk	E1	6DC0	0110110111	None	Compare less than long (VM)
lt.l.f Sj,Vk	E0	6DC0	0110110111	None	Compare less than long (!VM)
lt.s.t Sj,Vk	E1	6780	0110011110	RO	Compare less than single (VM)
lt.s.f Sj,Vk	E0	6780	0110011110	RO	Compare less than single (!VM)
lt.d.t Sj,Vk	E1	67C0	0110011111	RO	Compare less than double (VM)
lt.d.f Sj,Vk	E0	67C0	0110011111	RO	Compare less than double (!VM)
eq.b.t Sj,Vk	E1	6900	0110100100	None	Compare equal byte (VM)
eq.b.f Sj,Vk	E0	6900	0110100100	None	Compare equal byte (!VM)
eq.h.t Sj,Vk	E1	6940	0110100101	None	Compare equal halfword (VM)
eq.h.f Sj,Vk	E0	6940	0110100101	None	Compare equal halfword (!VM)
eq.w.t Sj,Vk	E1	6980	0110100110	None	Compare equal word (VM)
eq.w.f Sj,Vk	E0	6980	0110100110	None	Compare equal word (!VM)
eq.l.t Sj,Vk	E1	69C0	0110100111	None	Compare equal long (VM)
eq.l.f Sj,Vk	E0	69C0	0110100111	None	Compare equal long (!VM)
eq.s.t Sj,Vk	E1	6500	0110010100	RO	Compare equal single (VM)
eq.s.f Sj,Vk	E0	6500	0110010100	RO	Compare equal single (!VM)
eq.d.t Sj,Vk	E1	6540	0110010101	RO	Compare equal double (VM)
eq.d.f Sj,Vk	E0	6540	0110010101	RO	Compare equal double (!VM)

Description

The truth values of the comparisons of the contents of scalar register Sj and the respective contents of each of the first VL elements of vector register Vj replace the respective bits of the VM register, if the corresponding initial (at beginning of the instruction) VM bit is set (for `.t`) or clear (for `.f`). If the comparison is true, the bit of VM is set to 1. If false, it is cleared to 0. When VL is less than 128, remaining bits of VM are cleared to 0.

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 (Op code_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 (Op code_Test(Sj,Vk[a] == TRUE) {
                    VM<a> = 1; }
                else {
                    VM<a> = 0; } } } /* end of for loop */
        break; }            /* end of switch */
if (VL < 128) {
    for (a = VL; a < 128; a++) {     /* zero fill */
        VM<a> = 0; } }
```

Exceptions

s|d	Reserved operand

Example

```
lt.h.t S4,V2
```

Sets the 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 registers (vector with vector)

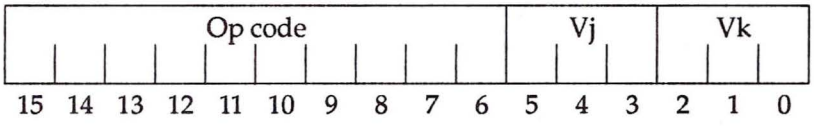
Purpose

To compare elements of two vector registers

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
le.b Vj,Vk	ST	6A00	0110101000	None	Compare less than or equal byte
le.h Vj,Vk	ST	6A40	0110101001	None	Compare less than or equal halfword
le.w Vj,Vk	ST	6A80	0110101010	None	Compare less than or equal word
le.l Vj,Vk	ST	6AC0	0110101011	None	Compare less than or equal longword
le.s Vj,Vk	ST	6600	0110011000	RO	Compare less than or equal single
le.d Vj,Vk	ST	6640	0110011001	RO	Compare less than or equal double float
lt.b Vj,Vk	ST	6C00	0110110000	None	Compare less than byte
lt.h Vj,Vk	ST	6C40	0110110001	None	Compare less than halfword
lt.w Vj,Vk	ST	6C80	0110110010	None	Compare less than word
lt.l Vj,Vk	ST	6CC0	0110110011	None	Compare less than longword
lt.s Vj,Vk	ST	6680	0110011010	RO	Compare less than single
lt.d Vj,Vk	ST	66C0	0110011011	RO	Compare less than double float
eq.b Vj,Vk	ST	6800	0110100000	None	Compare equal byte
eq.h Vj,Vk	ST	6840	0110100001	None	Compare equal halfword
eq.w Vj,Vk	ST	6880	0110100010	None	Compare equal word
eq.l Vj,Vk	ST	68C0	0110100011	None	Compare equal longword
eq.s Vj,Vk	ST	6400	0110010000	RO	Compare equal single
eq.d Vj,Vk	ST	6440	0110010001	RO	Compare equal double precision

Description

The truth values of the comparisons of the contents of the first VL elements of vector register Vj to the respective contents of elements of vector register Vj replace the respective bits of the VM register. If the comparison is true, the bit of VM is set to 1. If false, the bit of VM is cleared to 0.

When VL is less than 128, remaining bits of VM are cleared to 0.

Operation

```
for (a = 0; a < VL; a++) {
    if (Op code_Test(Vj[a],Vk[a]) == TRUE) {
        VM<a> = 1; }
    else {
        VM<a> = 0; } }
if (VL < 128) {
    for (a = VL; a < 128; a++) {     /* zero fill */
        VM<a> = 0; } }
```

Exceptions

b|h|w|l	None
s|d	Reserved operand

Example

```
lt.h V5,V1
```

Sets the VM bits 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 one'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 registers (vector with vector) (masked)

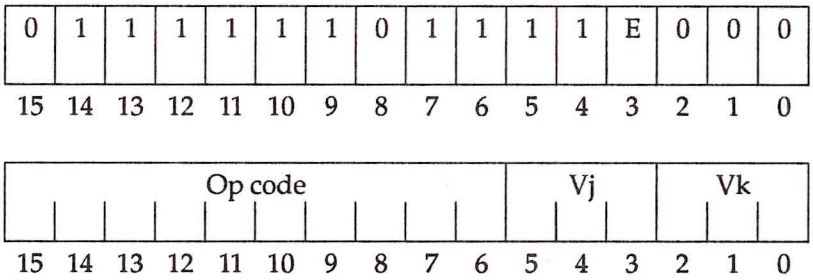
Purpose

To compare two vectors and load vector merge (VM) under control of the VM register at the instruction entry

Application

C200/C3200, C3400, C3800 Series CPUs

Format



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.w.f Vj,Vk	E0	6A80	0110101010	None	Compare less than or equal word (IVM)
le.l.t Vj,Vk	E1	6AC0	0110101011	None	Compare less than or equal long (VM)
le.l.f Vj,Vk	E0	6AC0	0110101011	None	Compare less than or equal long (IVM)
le.s.t Vj,Vk	E1	6600	0110011000	RO	Compare less than or equal single (VM)
le.s.f Vj,Vk	E0	6600	0110011000	RO	Compare less than or equal single (IVM)
le.d.t Vj,Vk	E1	6640	0110011001	RO	Compare less than or equal double (VM)
le.d.f Vj,Vk	E0	6640	0110011001	RO	Compare less than or equal double (IVM)
lt.b.t Vj,Vk	E1	6C00	0110110000	None	Compare less than byte (VM)
lt.b.f Vj,Vk	E0	6C00	0110110000	None	Compare less than byte (IVM)
lt.h.t Vj,Vk	E1	6C40	0110110001	None	Compare less than halfword (VM)
lt.h.f Vj,Vk	E0	6C40	0110110001	None	Compare less than halfword (IVM)
lt.w.t Vj,Vk	E1	6C80	0110110010	None	Compare less than word (VM)
lt.w.f Vj,Vk	E0	6C80	0110110010	None	Compare less than word (IVM)
lt.l.t Vj,Vk	E1	6CC0	0110110011	None	Compare less than long (VM)
lt.l.f Vj,Vk	E0	6CC0	0110110011	None	Compare less than long (IVM)
lt.s.t Vj,Vk	E1	6680	0110011010	RO	Compare less than single (VM)
lt.s.f Vj,Vk	E0	6680	0110011010	RO	Compare less than single (IVM)
lt.d.t Vj,Vk	E1	66C0	0110011011	RO	Compare less than double (VM)
lt.d.f Vj,Vk	E0	66C0	0110011011	RO	Compare less than double (IVM)
eq.b.t Vj,Vk	E1	6800	0110100000	None	Compare equal byte (VM)
eq.b.f Vj,Vk	E0	6800	0110100000	None	Compare equal byte (IVM)
eq.h.t Vj,Vk	E1	6840	0110100001	None	Compare equal halfword (VM)
eq.h.f Vj,Vk	E0	6840	0110100001	None	Compare equal halfword (IVM)
eq.w.t Vj,Vk	E1	6880	0110100010	None	Compare equal word (VM)
eq.w.f Vj,Vk	E0	6880	0110100010	None	Compare equal word (IVM)
eq.l.t Vj,Vk	E1	68C0	0110100011	None	Compare equal long (VM)
eq.l.f Vj,Vk	E0	68C0	0110100011	None	Compare equal long (IVM)
eq.s.t Vj,Vk	E1	6400	0110010000	RO	Compare equal single (VM)
eq.s.f Vj,Vk	E0	6400	0110010000	RO	Compare equal single (IVM)
eq.d.t Vj,Vk	E1	6440	0110010001	RO	Compare equal double (VM)
eq.d.f Vj,Vk	E0	6440	0110010001	RO	Compare equal double (IVM)

{le|lt|eq}.{b|h|w|l|s|d}.{t|f} Vj,Vk

Description

The truth value of the comparisons of the contents of the first VL respective elements of vector register Vj to the respective contents of elements of vector register Vj replace the respective bits of the VM register, if the corresponding initial (at beginning of the instruction) VM bit is set (for `.t`) or clear (for `.f`). If the comparison is true, the bit of VM is set to 1. If false, the bit of VM is cleared to 0.

When VL is less than 128, remaining bits of the VM register are cleared to 0.

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 (Op code_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 (Op code_Test(Vj[a],Vk[a]) == TRUE) {
                    VM<a> = 1; }
                else {
                    VM<a> = 0; } } } /* end of for loop */
        break; }            /* end of switch */

if (VL 128) {
    for (a = VL; a < 128; a++) {     /* zero fill */
        VM<a> = 0; } }
```

Exceptions

`b|h|w|l`	None
`s|d`	Reserved operand

Example

```
lt.h.t 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].

{le|lt}u.{h|w} #{n|N},Ak

Compare registers (address with immediate) (unsigned)

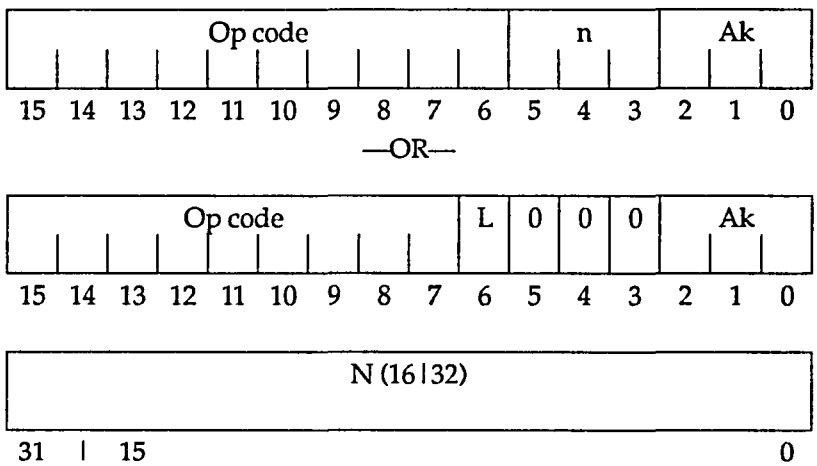
Purpose

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

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
leu.h #n,Ak	ST	4880	0100100010	See note 1	Compare unsigned less than or equal halfword
leu.h #N,Ak	ST	1C00	000111000	See note 1	Compare unsigned less than halfword
leu.w #n,Ak	ST	48C0	0100100011	See note 1	Compare unsigned less than or equal word
leu.w #N,Ak	ST	1C80	000111001	See note 1	Compare unsigned less than word
ltu.h #n,Ak	ST	4A80	0100101010	See note 1	Compare unsigned less than halfword
ltu.h #N,Ak	ST	1D00	000111010	See note 1	Compare unsigned less than halfword
ltu.w #n,Ak	ST	4AC0	0100101011	See note 1	Compare unsigned less than word
ltu.w #N,Ak	ST	1D80	000111011	See note 1	Compare unsigned less than word

{le|lt}u.{h|w} #{n|N},Ak

Description

The truth value of the unsigned comparison between the contents of the address register Ak and either the short immediate operand (#n) or the sign-extended long immediate operand (#N, length indicated by L) replaces address carry (C). If the comparison is true, C is set to 1. If false, C is cleared to 0.

Operation

```
if (Op code_Test(Immediate,Ak) == TRUE) {
    C = 1; }
else {
    C = 0; }
```

Example

```
ltu.w #1,a3
```

Sets C to 1 when 1 is less than the contents of address register A3.

Notes

1. Unsigned equal comparison is equivalent to signed equal comparison.
2. Sign extension does not occur for the 3 bits of the short immediate form.
3. 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} #N,Sk

Compare registers (scalar with immediate) (unsigned)

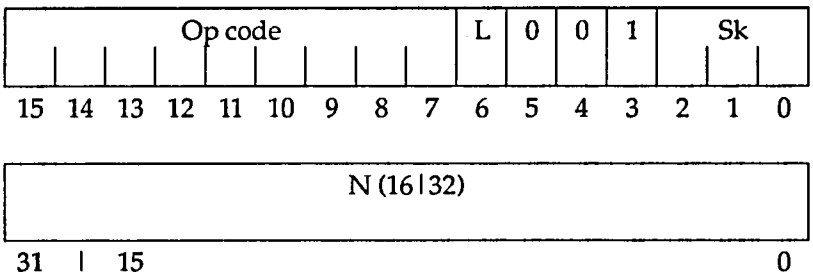
Purpose

To compare the contents of a scalar register and an immediate operand

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
leu.h #N,Sk	ST	1C08	000111000	See note 1	Compare unsigned less than or equal to halfword
leu.w #N,Sk	ST	1C88	000111001	See note 1	Compare unsigned less than or equal to word
ltu.h #N,Sk	ST	1D08	000111010	See note 1	Compare unsigned less than halfword
ltu.w #N,Sk	ST	1D88	000111011	See note 1	Compare unsigned less than word

Description

The truth value of the unsigned comparison of the contents of the scalar register Sk and the sign-extended long immediate operand (#N) replaces scalar carry (SC). If the comparison is true, SC is set to 1. If the comparison is false, SC is cleared to 0.

Operation

```
if (Op code_Test(Immediate,Sk) == TRUE) {
    SC = 1; }
else {
    SC = 0; }
```

Example

```
ltu.w #1,S5
```

Sets SC to 1 when the 1 is less than the contents of scalar register S5.

Notes

1. Use the `eq.x` instruction for unsigned compares of equality.
2. Unsigned equal comparison is equivalent to signed equal comparison.
3. 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 registers (address with address) (unsigned)

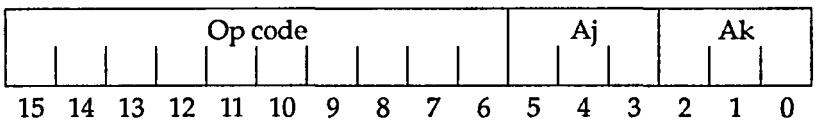
Purpose

To compare the unsigned contents of two address registers

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
leu.h Aj,Ak	ST	4800	0100100000	See note 1	Compare unsigned less than or equal to halfword
leu.w Aj,Ak	ST	4840	0100100001	See note 1	Compare unsigned less than or equal to word
ltu.h Aj,Ak	ST	4A00	0100101000	See note 1	Compare unsigned less than halfword
ltu.w Aj,Ak	ST	4A40	0100101001	See note 1	Compare unsigned less than word

Description

The truth value of the comparison of the contents of address register Ak and the contents of address register Aj replaces C. If the comparison is true, C is set to 1. If the comparison is false, C is cleared to 0.

Operation

```
if (Op code_Test(Aj,Ak) == TRUE) {
    C = 1; }
else {
    C = 0; }
```

Example

```
ltu.h a2,a1
```

Sets address carry (C) to 1 if the halfword contents of address register A2 are less than the contents of address register A1.

Notes

1. Use the `eq.x` instruction for unsigned compares of equality.
2. Unsigned equal comparison is equivalent to signed equal comparison.
3. 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.

{le|lt}u.{b|h|w|l} Sj,Sk

Compare registers (scalar with scalar) (unsigned)

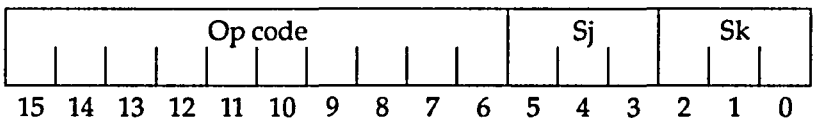
Purpose

To compare the unsigned contents of two scalar registers

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
leu.b Sj,Sk	ST	4900	0100100100	SC	Compare less than or equal to byte
leu.h Sj,Sk	ST	4940	0100100101	SC	Compare less than or equal to halfword
leu.w Sj,Sk	ST	4980	0100100110	SC	Compare less than or equal to word
leu.l Sj,Sk	ST	49C0	0100100111	SC	Compare less than or equal to longword
ltu.b Sj,Sk	ST	4B00	0100101100	SC	Compare less than byte
ltu.h Sj,Sk	ST	4B40	0100101101	SC	Compare less than halfword
ltu.w Sj,Sk	ST	4B80	0100101110	SC	Compare less than word
ltu.l Sj,Sk	ST	4BC0	0100101111	SC	Compare less than longword

Description

The truth value of the comparison of the contents of scalar register Sk and the contents of scalar register Sj replaces scalar carry (SC). If the comparison is true, the SC is set to 1. If false, SC is cleared to 0.

Operation

```
if (Op code_Test(Sj,Sk) == TRUE) {
    SC = 1; }
else {
    SC = 0; }
```

Example

```
ltu.h s3,s1
```

Sets scalar carry (SC) to 1 if the halfword contents of scalar register S3 are less than the contents of S1.

Notes

1. Use the `eq.x` instruction for unsigned compares of equality.
2. Unsigned equal comparison is equivalent to signed equal comparison.
3. 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

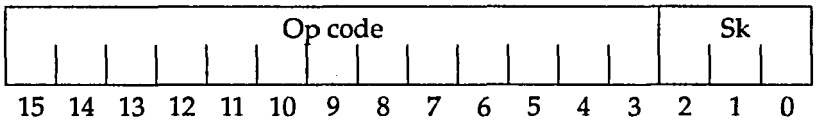
Purpose

To compute the natural logarithm of the contents of a scalar register.

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
ln.s Sk	ST	7C10	0111110000010	FIN,IEC	Natural logarithm of a single precision number
ln.d Sk	ST	7C18	0111110000011	FIN,IEC	Natural logarithm of a double precision number

Description

The natural logarithm of the contents of Sk replaces the contents of Sk.

Operation

```
Sk = ln(Sk)
```

Exceptions

`s|d`	Floating intrinsic error

Notes

1. Intrinsic traps go through the same trap handler as other arithmetic traps (RO, FDZ, UN, etc.). If FUE and/or FE are set and intrinsic traps are not (INE clear), these bits must be examined to determine the type of the current trap.
2. When FIN is set, the IEC bits contain a code that specifies the type of error encountered by the intrinsic instruction. Refer to the *CONVEX Architecture Reference Manual (C Series)*, "Processor status word" section in Chapter 3, "General registers," for more information on the IEC error codes and arithmetic trap conditions.

lop Sj,Sk
Leading one's position (scalar)

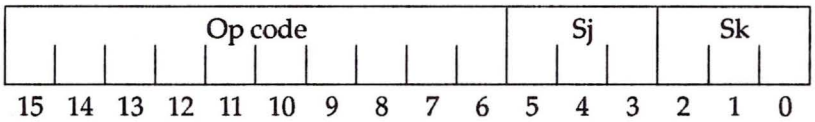
Purpose

To calculate the leading one's position in a scalar register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
lop Sj,Sk	ST	61C0	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 in the operation pseudocode).

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;
```

Note

For compatibility with previous releases, the CONVEX assembler recognizes the mnemonic `lzc` as equivalent to `lop`.

lop Vj,Vk
Leading one's position (vector)

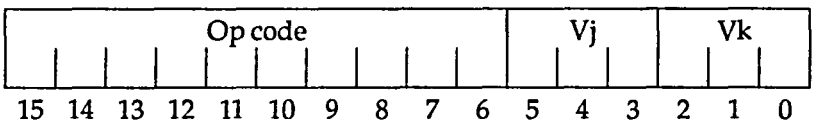
Purpose

To calculate the leading one's position in each element of a vector register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
lop Vj,Vk	ST	6240	0110001001	None	Leading one's 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 element 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 in the operation pseudocode).

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; }
```

lop.{t|f} Vj,Vk

Leading one's position (vector) (masked)

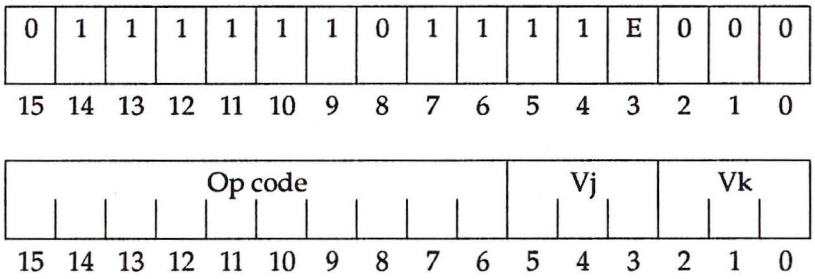
Purpose

To calculate the leading one's position in each element of a vector register under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
lop.t Vj,Vk	E1	6240	0110001001	None	Leading one's position vector (VM)
lop.f Vj,Vk	E0	6240	0110001001	None	Leading one's position vector (!VM)

Description

The index number of the leftmost 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 element 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 in the Operation pseudocode).

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; } } /* leftmost 1 */
                Vk[a]<7..0> = b;
                Vk[a]<63..8> = 0; } } /* end if TRUE */
        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; } } /* leftmost 1 */
                Vk[a]<7..0> = b;
                Vk[a]<63..8> = 0; } } /* end if FALSE */
        break; }    /* end of switch */
```

mask.t Vi,Vj,Vk

Mask (selectively copy) registers (vector or vector) (masked)

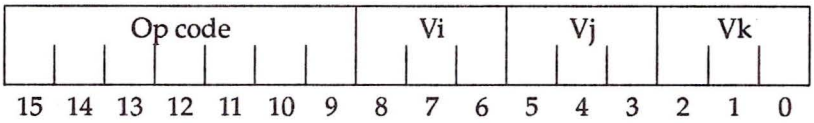
Purpose

To mask the elements of two vectors to the elements of a third vector

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mask.t Vi,Vj,Vk	ST	8600	1000011	None	Mask vector/vector

Description

The contents of each of the first VL elements of vector register Vk are replaced by the corresponding element of Vj (if the corresponding VM bit is 1) or Vi (if the corresponding VM bit is 0).

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]; } }
```

Note

Interchanging Vi and Vj is equivalent to using a complemented VM.

mask.{t|f} Vi,Sj,Vk

Mask (selectively copy) registers (vector or scalar)

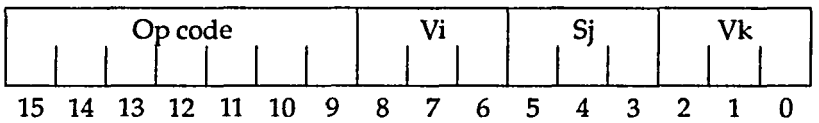
Purpose

To mask a scalar to a vector

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mask.t Vi,Sj,Vk	ST	8E00	1000111	None	Mask vector/scalar (VM)
mask.f Vi,Sj,Vk	ST	8A00	1000101	None	Mask vector/scalar (!VM)

Description

For `.t`–The contents of each of the first VL elements of vector register Vk are replaced by the 64 bits of scalar register Sj (if the corresponding VM bit is 1) or the 64 bits of the *corresponding* element in Vi (if the corresponding VM bit is 0).
For `.f`–The contents of each of the first VL elements of vector register Vk are replaced by the 64 bits of the *corresponding* element in Vi (if the corresponding VM bit is 1) or the 64 bits of scalar register Sj (if the corresponding VM bit is 0).
Compare this instruction to merge.{t | f} Vi,Sj,Vk

Operation

```
switch (op code<10>) { /* op code 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; } /* end of switch */
```

Note

Load a scalar into all elements of a vector register by setting VM to all ones and using `mask.t` (or use `mask.f` with a VM of all zeros).

mat.w Ak,*Ceffa*

Match registers (address with communication)

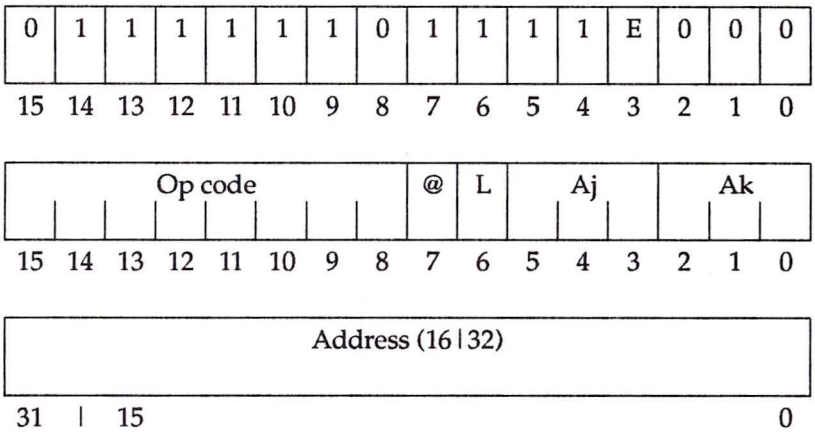
Purpose

To compare the address register contents with the communication register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mat.w Ak,*Ceffa*	E0	2900	0010100100	C,CAT	Match address/communication

Description

Bits <31..0> of c (*Ceffa*) are compared to Ak. C is set to 1 if the two values are equal. C is cleared to 0 if the two bits are not equal. Bits <63..32> of c (*Ceffa*) are not used. c (*Ceffa*) and L (*Ceffa*) are not modified.

Operation

```
if ( c(Ceffa) == Ak ) {
    C = 1; }
else {
    C = 0; }
```

Exceptions

Ring violation (invalid communication register address)
Deadlock exception

Notes

1. This instruction provides the same functionality as an `equal` test followed by a branch on carry except that `mat.w` provides deadlock detection capability.
2. The memory dual of this instruction is `matr.w Ak,effa`.

mat.l Sk,*Ceffa*

Match registers (scalar with communication)

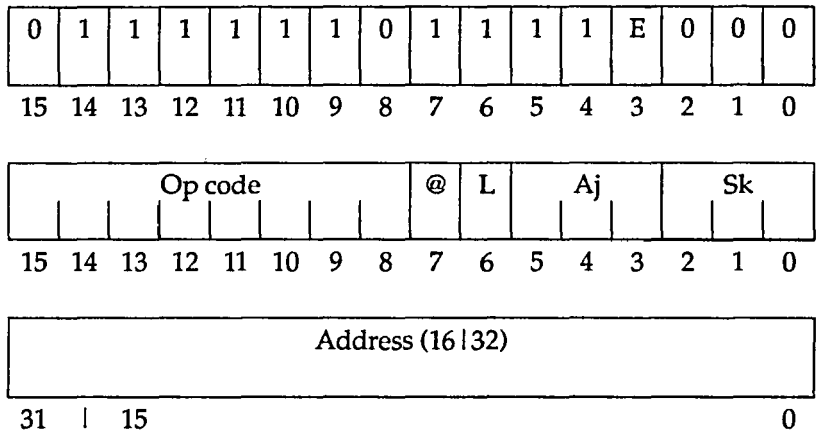
Purpose

To compare the scalar register contents with the communication register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mat.l Sk,*Ceffa*	E0	3100	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.

Operation

```
if ( c(Ceffa) == Sk ) {
    SC = 1; }
else {
    SC = 0; }
```

Exceptions

Ring violation (invalid communication register address)
Deadlock exception

Notes

1. This instruction provides the same functionality as an equal test followed by a branch on carry except that `mat.l` provides deadlock detection capability.
2. The memory dual of this instruction is `matr.l Sk,` *effa*.

matr.w Ak,*effa*

Match registers (address with resource)

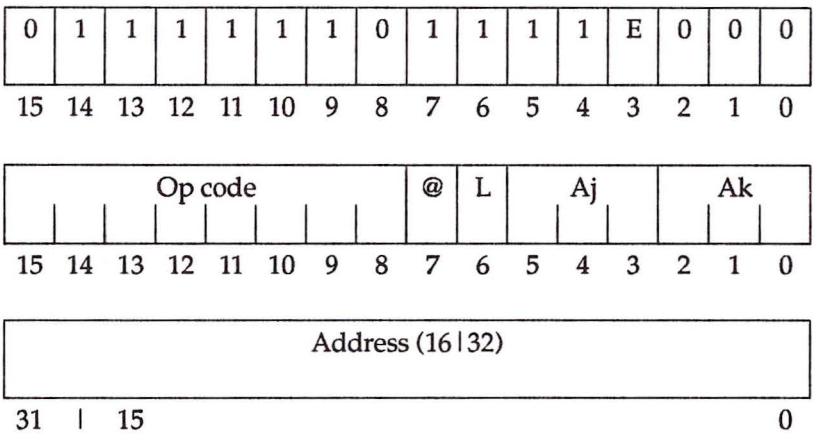
Purpose

To compare the contents of an address register with the contents of a resource structure in memory

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
matr.w Ak,*effa*	E0	2800	0010100000	C	Match address/resource

Description

A word from a resource structure (`C (`*effa*`.data)`) is atomically compared to the contents of Ak. If the resource structure was in transition (unable to lock the resource structure) or the two compared values are not equal, C is cleared to 0 . If the operation was successful (the two compared values are equal and the resource structure is unlocked) C is set to 1. `C (`*effa*`.data)` is not modified.

Operation

```
if (tas(effa.lock)) {
    if (Ak == c(effa.data))
        C = 1
    else
        C = 0    /* fail - data does not match */
        tac(effa.lock) }
else {
    C = 0 }  /* fail - structure in transition */
```

Exceptions

Ring violation (invalid communication register address)
Deadlock exception

Notes

1. This instruction provides the same functionality as an equal test followed by a branch on carry except that `matr.w` provides deadlock detection capability.
2. This instruction is atomic.
3. The communication register dual of this instruction is `mat.w Ak,` *Ceffa*.

matr.l Sk,*effa*

Match registers (scalar with resource)

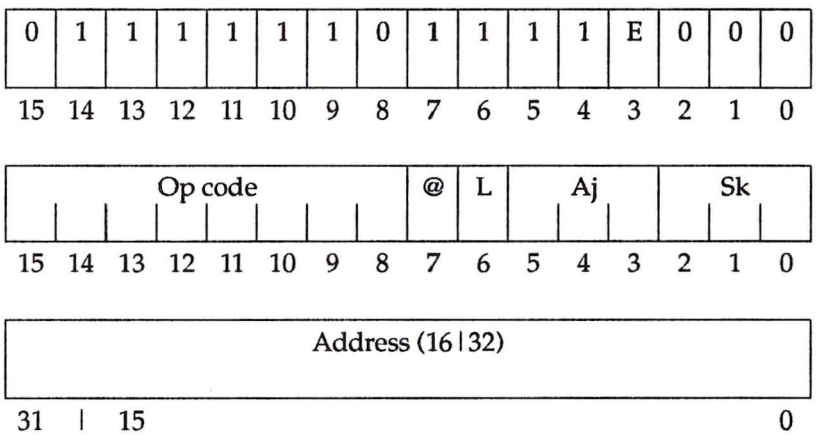
Purpose

To compare the contents of a scalar register with the contents of a resource structure in memory

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
matr.l Sk,*effa*	E0	3000	0011000000	SC	Match scalar/resource

Description

A longword from a resource structure (`C(`*effa*`.data)`) is atomically compared to the contents of Sk. If the resource structure was in transition (unable to lock the resource structure) or the two compared values are not equal, SC is cleared to 0. If the operation was successful (the two compared values are equal and the resource structure is unlocked) SC is set to 1. `C(`*effa*`.data)` is not modified.

Operation

```
if (tas(effa.lock)) {
    if (Sk == c(effa.data)) {
        SC = 1 }
    else {
        SC = 0 } /* fail - data does not match */
    tac(effa.lock) }
else {
    SC = 0 } /* fail - structure in transition */
```

Exceptions

Ring violation (invalid communication register address)
Deadlock exception

Notes

1. This instruction provides the same functionality as an equal test followed by a branch on carry except that `matr.l` provides deadlock detection capability.
2. This instruction is atomic.
3. The communication register dual of this instruction is `mat.l Sk,` *Ceffa*.

max.{b|h|w|l|s|d} {Vk|Sk}

Max register (vector)

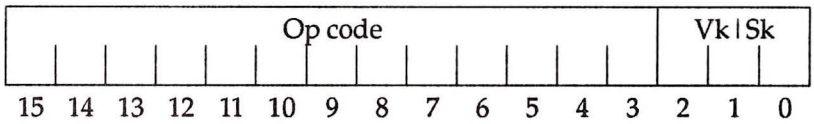
Purpose

To find the maximum element of a vector register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
max.b Vk	ST	7E40	0111111001000	None	Maximum of a vector of bytes
max.h Vk	ST	7E48	0111111001001	None	Maximum of a vector of halfwords
max.w Vk	ST	7E50	0111111001010	None	Maximum of a vector of words
max.l Vk	ST	7E58	0111111001011	None	Maximum of a vector of longwords
max.s Vk	ST	7EA0	0111111010100	RO	Maximum of a vector of single float
max.d Vk	ST	7EA8	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 replaces Sk.

Operation

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

Exceptions

`b|h|w|l`	None
`s|d`	Reserved operand

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.{b|h|w|l|s|d}.{t|f} {Vk|Sk}
Max register (vector) (masked)

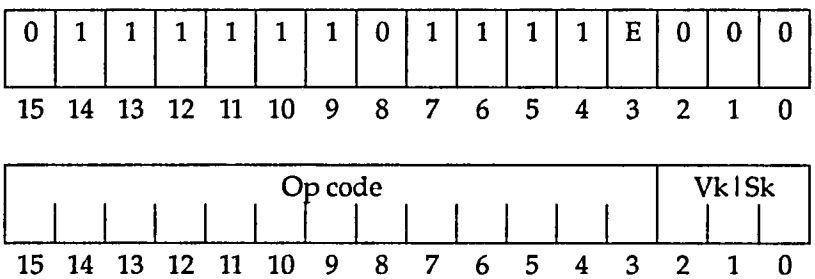
Purpose

To find the maximum element of a subset of a vector register under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
max.b.t Vk	E1	7E40	0111111001000	None	Maximum of vector of bytes (VM)
max.b.f Vk	E0	7E40	0111111001000	None	Maximum of vector of bytes (!VM)
max.h.t Vk	E1	7E48	0111111001001	None	Maximum of vector of halfwords (VM)
max.h.f Vk	E0	7E48	0111111001001	None	Maximum of vector of halfwords (!VM)
max.w.t Vk	E1	7E50	0111111001010	None	Maximum of vector of words (VM)
max.w.f Vk	E0	7E50	0111111001010	None	Maximum of vector of words (!VM)
max.l.t Vk	E1	7E58	0111111001011	None	Maximum of vector of longwords (VM)
max.l.f Vk	E0	7E58	0111111001011	None	Maximum of vector of longwords (!VM)
max.s.t Vk	E1	7EA0	0111111010100	RO	Maximum of vector of singles (VM)
max.s.f Vk	E0	7EA0	0111111010100	RO	Maximum of vector of singles (!VM)
max.d.t Vk	E1	7EA8	0111111010101	RO	Maximum of vector of doubles (VM)
max.d.f Vk	E0	7EA8	0111111010101	RO	Maximum of vector of doubles (!VM)

max.{b|h|w|l|s|d}.{t|f} {Vk|Sk}

Description

The maximum of scalar register Sk and the first VL elements of vector register Vk replaces Sk, only if the corresponding VM bit set (for `.t`) or clear (for `.f`).

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;  } /* end of switch */
```

Exceptions

`b|h|w|l`	None
`s|d`	Reserved operand

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).

merg.{t|f} Vi,Sj,Vk

Merge registers (vector with scalar)

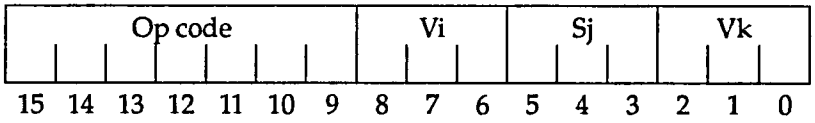
Purpose

To merge the contents of a scalar register and a vector register into the contents of a vector register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
merg.t Vi,Sj,Vk	ST	8C00	1000110	None	Merge vector/scalar
merg.f Vi,Sj,Vk	ST	8800	1000100	None	Merge vector/scalar (IVM)

merg.{t|f} Vi,Sj,Vk

Description

For `.t`–The contents of each of the first VL elements of vector register Vk are replaced by the 64 bits of scalar register Sj (if the corresponding VM bit is 1) or the 64 bits of the *next uncopied* element in Vi (if the corresponding VM bit is 0).

For `.f`–The contents of each of the first VL elements of vector register Vk are replaced by the 64 bits of the *next uncopied* element in Vi (if the corresponding VM bit is 1) or the 64 bits of scalar register Sj (if the corresponding VM bit is 0).

Compare this instruction to mask.{t | f} Vi,Sj,Vk

Operation

```
b = 0;
switch (op code<10>) { /* op code bit<10> */
    case TRUE:      /* .t */
        for (a = 0; a < VL; a++) {
            if (VM<a> == 1) { /* 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) { /* VM<a> is FALSE */
                Vk[a] = Sj; }
            else {
                Vk[a] = Vi[b];
                b = b + 1; } } /* end of for loop */
        break; } /* end of switch */
```

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 registers (vector with vector)

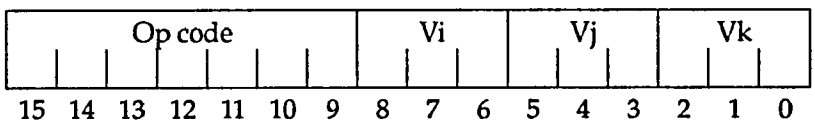
Purpose

To merge the contents of two vector registers into a third vector register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
merg.t Vi,Vj,Vk	ST	8400	1000010	None	Merge vector/vector

Description

The first VL elements of vector register Vk are replaced by either the *next uncopied* element of Vi (when VM<a> is 0) or the *next uncopied* element of Vj (when VM<a> is 1).

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 */
```

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.{b|h|w|l|s|d} {Vk|Sk}
Min register (vector)

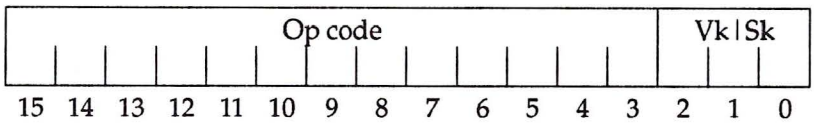
Purpose

To find the minimum element of a vector

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
min.b Vk	ST	7E60	0111111001100	None	Minimum of a vector of bytes
min.h Vk	ST	7E68	0111111001101	None	Minimum of a vector of halfwords
min.w Vk	ST	7E70	0111111001110	None	Minimum of a vector of words
min.l Vk	ST	7E78	0111111001111	None	Minimum of a vector of longwords
min.s Vk	ST	7EB0	0111111010110	RO	Minimum of a vector of single float
min.d Vk	ST	7EB8	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 replaces Sk.

Operation

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

Exceptions

`b|h|w|l`	None
`s|d`	Reserved operand

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 register (vector) (masked)

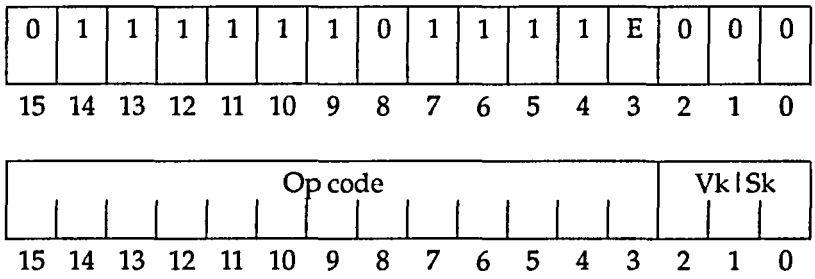
Purpose

To find the minimum element of a subset of a vector register under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
min.b.t Vk	E1	7E60	0111111001100	None	Minimum of vector of bytes (VM)
min.b.f Vk	E0	7E60	0111111001100	None	Minimum of vector of bytes (IVM)
min.h.t Vk	E1	7E68	0111111001101	None	Minimum of vector of halfwords (VM)
min.h.f Vk	E0	7E68	0111111001101	None	Minimum of vector of halfwords (IVM)
min.w.t Vk	E1	7E70	0111111001110	None	Minimum of vector of words (VM)
min.w.f Vk	E0	7E70	0111111001110	None	Minimum of vector of words (IVM)
min.l.t Vk	E1	7E78	0111111001111	None	Minimum of vector of longwords (VM)
min.l.f Vk	E0	7E78	0111111001111	None	Minimum of vector of longwords (IVM)
min.s.t Vk	E1	7EB0	0111111010110	RO	Minimum of vector of singles (VM)
min.s.f Vk	E0	7EB0	0111111010110	RO	Minimum of vector of singles (IVM)
min.d.t Vk	E1	7EB8	0111111010111	RO	Minimum of vector of doubles (VM)
min.d.f Vk	E0	7EB8	0111111010111	RO	Minimum of vector of doubles (IVM)

min.{b|h|w|l|s|d}.{t|f} {Vk|Sk}

Description

The minimum of scalar register Sk and the first VL elements of vector register Vk replaces Sk, only if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

Operation

```
switch (E) {    /* prefix bit */
    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; } /* end of switch */
```

Exceptions

`b|h|w|l`	None
`s|d`	Reserved operand

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 register (address to address)

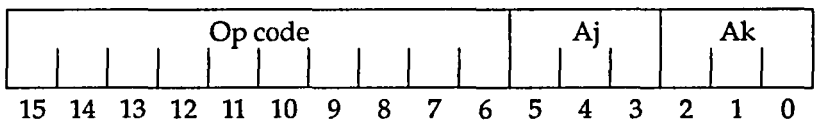
Purpose

To copy the contents of one address register to another address register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov Aj,Ak	ST	5080	0101000010	None	Move address register

Description

The contents of address register Aj replace the contents of address register Ak.

Operation

```
Ak = Aj;
```

mov Ak,PSW
Move register (address to PSW)

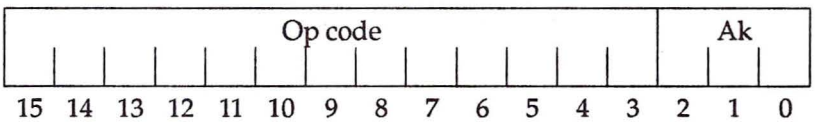
Purpose

To copy an address register to the processor status word (PSW)

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov Ak,PSW	ST	7C48	0111110001001	All bits	Load an address register into the PSW

Description

The contents of address register Ak replace the contents of the PSW.

Operation

```
PSW = Ak;
```

Exceptions

Any exceptions as dictated by the new 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 register (address to scalar)

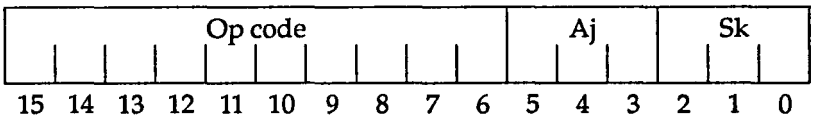
Purpose

To copy the contents of an address register into a scalar register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov Aj,Sk	ST	51C0	0101000111	None	Move an address to a scalar

Description

The contents of address register Aj replace the least significant 32 bits of scalar register Sk. The most significant 32 bits of Sk remain unchanged.

Operation

```
Sk<31..0> = Aj;
```

mov Ak,VL
Move register (address to VL)

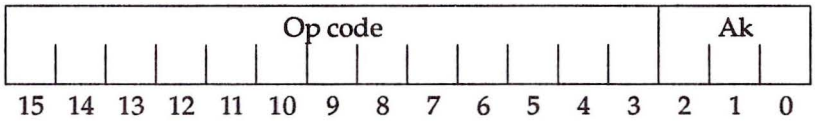
Purpose

To copy the contents of an address register to the vector length (VL) register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov Ak,VL	ST	7D98	0111110110011	None	Move Ak to VL

Description

The contents of the VL register are replaced with the contents of address register Ak, bracketed to the range <0..128>.

Operation

```
if (Ak < 128) {
    VL = 128; }
else {
    if (Ak < 0) {
        VL = 0; }
    else {
        if Ak > 128 {
            VL = 128 }
        else {
            VL = Ak; } } }
```

mov Ak,VS
Move register (address to VS)

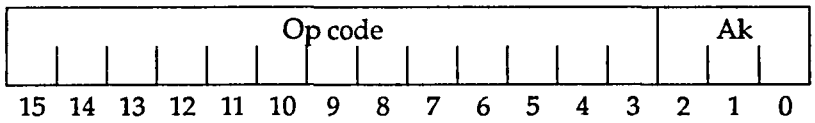
Purpose

To copy the contents of an address register to the (VS) register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov Ak,VS	ST	7D88	0111110110001	None	Move Ak to VS

Description

The contents of address register Ak replace the contents of VS.

Operation

```
VS = Ak;
```

mov BE(Sj),Sk

Move register (BE(scalar) to scalar)

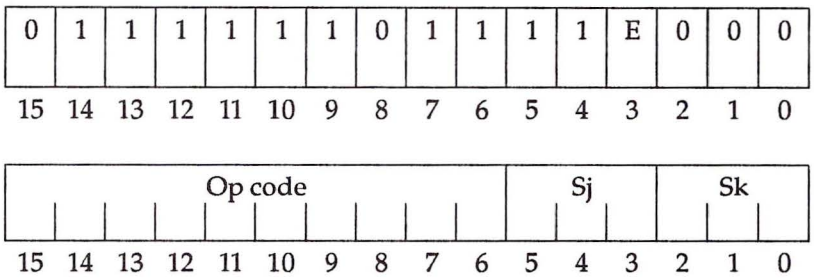
Purpose

To copy the contents of a broadcast enable register (BE) to a scalar register

Application

C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov BE(Sj),Sk	E0	4580	0100010110	None	Move brodcast enable register to a scalar register

Description

The lower 8 bits of the scalar register Sk are replaced by the broadcast enable register identified by Sj <2..0>. The upper 56 bits of Sk are set to zero.

Operation

```
Sk<63..8> = 0;
Sk<7..0> = BE for channel Sj<2..0>;
```

Exceptions

Ring violation (privileged instruction)

Notes

Refer to the *CONVEX Architecture Reference Manual (C Series)*, Chapter 11, "Operating system interrupts," for more information on the broadcast enable registers.

mov CIR,Sk

Move register (CIR to scalar)

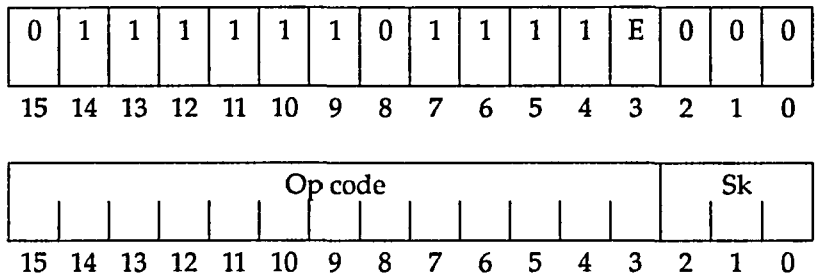
Purpose

To move communication index register (CIR) to a scalar register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov CIR,Sk	E0	7C08	0111110000001	None	Move CIR a scalar

Description

The zero-extended CIR for this CPU is copied to Sk.

Operation

```
Sk = CIR;
```

mov CPUID,Sk

Move register (CPUID to scalar)

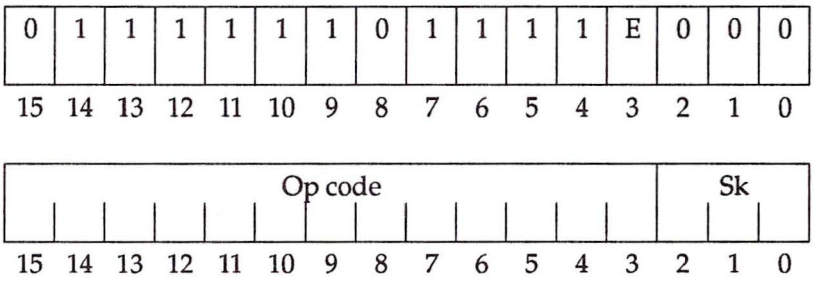
Purpose

To move the read only CPUID register to a scalar register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov CPUID,Sk	E0	7C18	0111110000011	None	Move CPU ID to scalar

Description

The CPUID for this CPU, a zero-extended longword between 0 and the maximum configurable CPU number, is copied into Sk.

Operation

```
Sk = CPUID;
```

mov ICR,Sk

Move register (ICR to scalar)

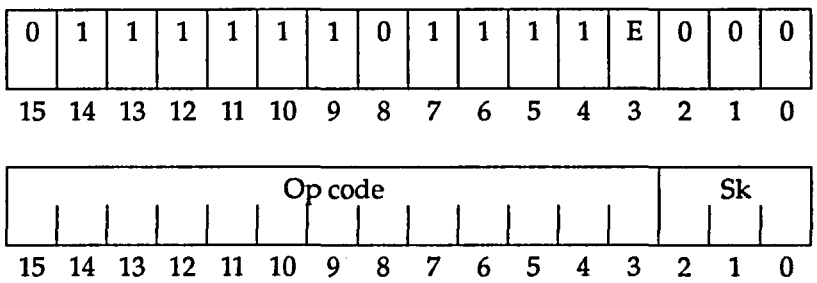
Purpose

To move the interrupt control register (ICR) to a scalar register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov ICR,Sk	E0	7C68	0111110001101	None	Move ICR to scalar

Description

The interrupt control register (ICR), which contains global interrupt control information for all CPUs, is copied to Sk.

Operation

```
Sk = ICR;
```

Notes

The format of the ICR is described in the *CONVEX Architecture Reference Manual (C Series)*, Chapter 11, "Operating system interrupts."

mov ITR,Sk
Move register (ITR to scalar)

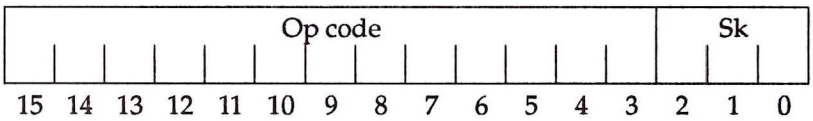
Purpose

To copy the contents of the interval timer registers (ITC, NITC, and ITSR) to a scalar register

Application

C100, C3400 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov ITR,Sk	ST	7C60	0111110001100	None	Move the ITC, ITSR, NITC into Sk

Description

The contents of the interval timer status register (ITSR) replace Sk <63..60>. The contents of the next interval timer counter (NITC) replace Sk <59..40>. The contents of the interval timer counter (ITC) replace Sk <27..8>.

Operation

```
Sk<63..60> = ITSR;
Sk<59..40> = NITC;
Sk<27..8> = ITC;
```

Exceptions

Ring violation (privileged instruction)

mov PC,Ak
Move register (PC to address)

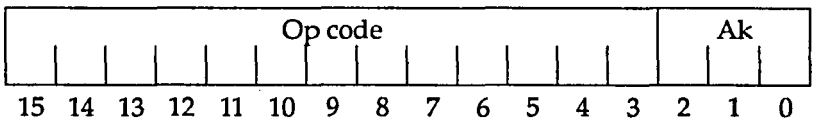
Purpose

To copy the address of the next instruction (PC) into an address register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov PC,Ak	ST	7C50	0111110001010	None	Load the next PC address

Description

The address of the instruction following this one replaces the contents of address register Ak.

Operation

```
Ak = Current_Address + 2;
```

mov PSW,Ak

Move register (PSW to address)

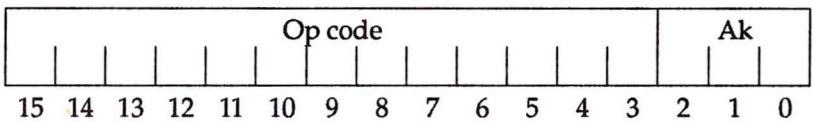
Purpose

To copy the processor status word (PSW) to an address register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov PSW,Ak	ST	7C40	0111110001000	None	Store the PSW into an address register

Description

The contents of the PSW replace the contents of address register Ak.

Operation

```
Ak = PSW;
```

Note

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.

mov.{w|l|s|d} Sj,Sk
Move register (scalar to scalar)

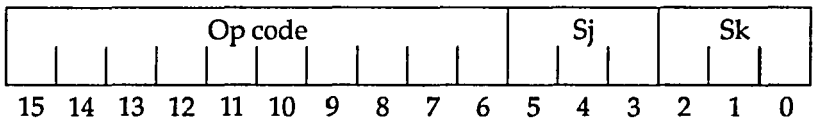
Purpose

To copy the contents of one scalar register to another

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov.l Sj,Sk	ST	5180	0101000110	None	Move scalar register longword
mov.w Sj,Sk	ST	5100	0101000100	None	Move scalar register word
mov.d Sj,Sk	ST	5180	0101000110	None	Move scalar register double float
mov.s Sj,Sk	ST	5100	0101000100	None	Move scalar register single float

Description

The specified portion (word, longword, single float, double float) of scalar register Sj replaces the corresponding portion of Sk. The remaining bits are unchanged.

Operation

```
Sk<63..0> = Sj<63..0>; /* mov.l, mov.d */
Sk<31..0> = Sj<31..0>; /* mov.s, mov.w */
```

Note

The `.s` and `.d` forms rename the `.w` and `.l` forms, respectively, for convenience.

mov Sj,Ak

Move register (scalar to address)

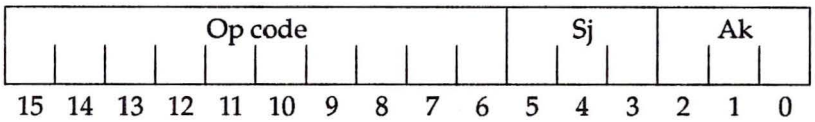
Purpose

To copy the contents of a scalar register into an address register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov Sj,Ak	ST	50C0	0101000011	None	Move 32 LSBs of Sj into Ak

Description

The least significant 32 bits of scalar register Sj replace the contents of address register Ak.

Operation

```
Ak = Sj<31..0>;
```

mov Sk,BE(Sj)

Move register (scalar to BE(scalar))

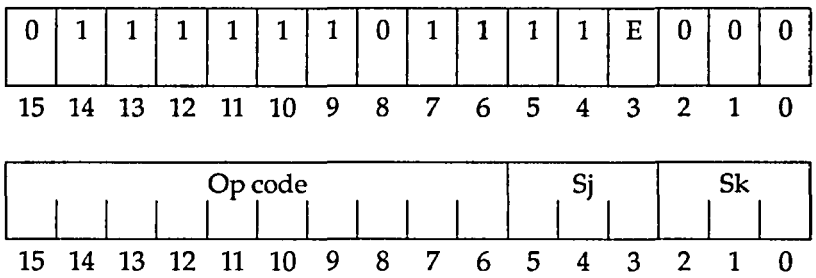
Purpose

To copy the contents of a scalar register to a broadcast enable (BE) register.

Application

C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov Sk,BE(Sj)	E0	45C0	0100010111	None	Move a scalar register to a brodcast enable register

Description

The contents of the broadcast enable register identified by Sj <2..0> are replaced by Sk <7..0>.

Operation

```
BE for channel Sj <2..0> = Sk <7..0>;
```

Exception

Ring violation (privileged instruction)

Notes

Refer to the *CONVEX Architecture Reference Manual (C Series)*, Chapter 11 "Operating system interrupts," for more information on the broadcast enable registers.

mov Sk,CIR

Move register (scalar to CIR)

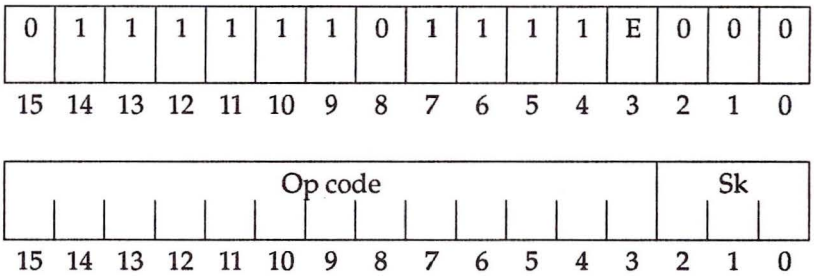
Purpose

To move scalar register to the communication index register (CIR)

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov Sk,CIR	E0	7C00	0111110000000	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 (CIR index) to one of the communication register sets.

Operation

```
CIR = Sk <4..0>;
```

Exceptions

Ring violation (privileged instruction)

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
Move register (scalar to ICR)

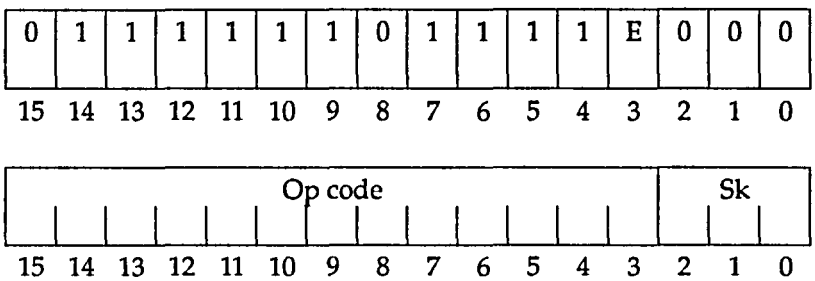
Purpose

To move scalar register to the interrupt control register (ICR)

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov Sk,ICR	E0	7C60	0111110001100	None	Move Scalar to ICR

Description

This instruction is used to set the interrupt control register (ICR), which contains global interrupt control information for all CPUs.

Operation

```
ICR = Sk;
```

Exceptions

Ring violation (privileged instruction)

Notes

1. The format of the ICR is detailed in the *CONVEX Architecture Reference Manual (C Series)*, Chapter 11, "Operating system interrupts."
2. Interrupts must be successfully disabled using `dsi` before this instruction is executed. Otherwise, unpredictable results can occur.

mov Sk,ITR

Move register (scalar to ITR)

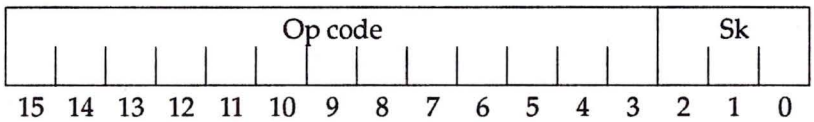
Purpose

To move the contents of a scalar register to the interval timer registers (NITC, ITSR, and ITC)

Application

C100, C3400 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov Sk,ITR	ST	7C68	0111110001101	None	Load NITC, ITC, ITSR from Sk

Description

Scalar register bits Sk<63..60> replace the contents of the interval timer status register (ITSR). Bits Sk<59..40> replace the contents of the next iteration counter (NITC). Bits Sk<27..8> replace the contents of the interval timer counter (ITC).

Operation

```
ITSR = Sk<63..60>;
NITC = Sk<59..40>;
ITC = Sk<27..8>;    /* C100 Series only */
```

Exceptions

Ring violation (privileged instruction)

mov Sk,ITSR

Move register (scalar to ITSR)

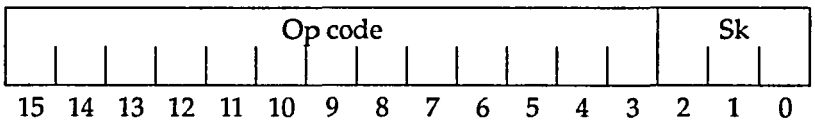
Purpose

To copy the 4 most significant bits of a scalar register to the interval timer status register (ITSR)

Application

C100, C3400 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov Sk,ITSR	ST	7C78	0111110001111	None	Load ITSR with a scalar

Description

The most significant 4 bits of scalar register Sk replace the interval timer status register (ITSR).

Operation

```
ITSR = Sk<63..60>;
```

Exceptions

Ring violation (privileged instruction)

mov Sj,Sk,VM
Move register (scalar to VM)

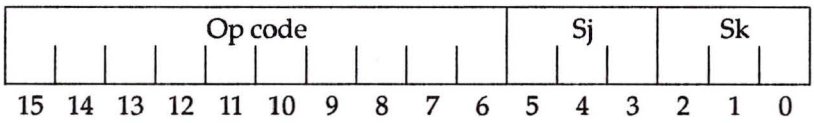
Purpose

To copy the contents from a scalar register to the vector merge (VM) register.

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov Sj,Sk,VM	ST	6100	0110000100	None	Load VM(Sj) from Sk

Description

If Sj is zero, the contents of the vector merge register (VM<63..0>) are replaced with scalar register Sk. If Sj is one, the contents of VM <127..64> are replaced with the contents of scalar register Sk.

Operation

```
if (Sj == 0) {
    VM<63..0> = Sk; }
else {
    VM<127..64> = Sk; }
```

mov Sk,TCPU
Move register (scalar to TCPU)

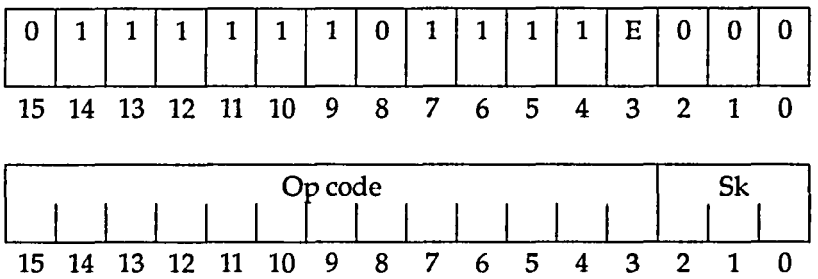
Purpose

To move scalar register to the interrupt target CPU (TCPU) register

Application

C200/C3200 Series CPUs
(undefined code on C3400 and C3800 Series CPUs)

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov Sk,TCPU	E0	7C70	0111110001110	None	Move scalar to TCPU register

Description

This instruction sets the interrupt target CPU (TCPU) register. The TCPU register can be set to either all ones, to indicate any CPU may be selected for interrupt delivery, or to a valid CPUID to vector all interrupts to the specified CPU.

Operation

```
TCPU = Sk;
```

Exceptions

Ring violation (privileged instruction)

Notes

Interrupts must be successfully disabled using `dsi` before this instruction is executed. Otherwise, unpredictable results can occur.

mov Sk,TID

Move register (scalar to thread ID)

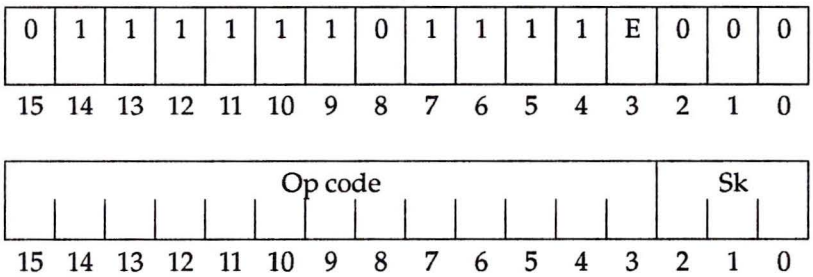
Purpose

To move the contents of a scalar to the thread ID (TID) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov Sk,TID	E0	7CB0	0111110010100	None	Load TID from scalar

Description

This instruction sets the current thread ID (TID) register to the value in Sk.

Operation

```
THREAD_ID = Sk <4..0>;
```

Exceptions

Ring violation (privileged instruction)

Notes

The TID being loaded should not be allocatable in the thread allocation mask.

mov Sk,TOC
Move register (scalar to TOC)

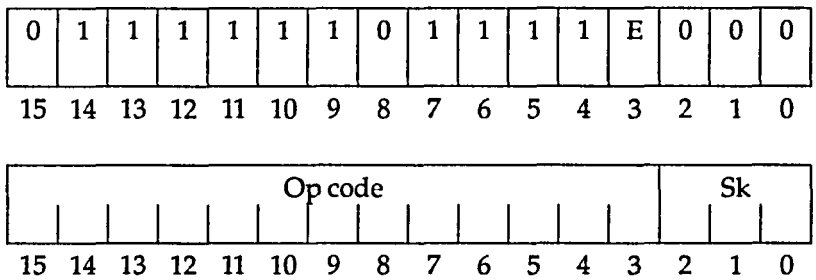
Purpose

To move the contents of a scalar register to the time-of-century counter (TOC).

Application

C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov Sk,TOC	E0	7C30	0111110000110	None	Move a scalar to TOC

Description

This instruction loads the time-of-century counter with the value of the scalar register Sk.

Operation

```
TOC = Sk <>;
```

Exceptions

Ring violation (privileged instruction)

Notes

Refer to the *CONVEX Architecture Reference Manual (C Series)*, Chapter 10, "Timers," for more information on the TOC.

mov Sk,TTR

Move register (scalar to thread timer)

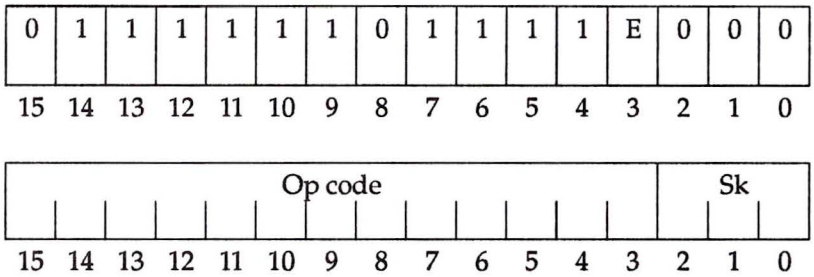
Purpose

To move scalar register to thread timer register (TTR)

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov Sk,TTR	E0	7C20	0111110000100	None	Move scalar to TTR

Description

This instruction is used to initialize the current thread timer register (TTR) from a scalar register.

Operation

```
THREAD_TIMER = Sk;
```

mov Si,Sj,Vk

Move register (scalar to vector element)

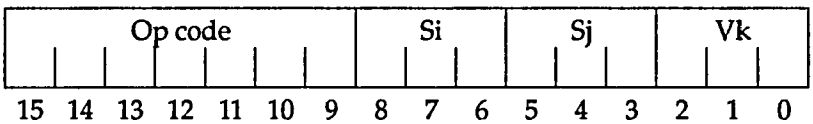
Purpose

To copy the contents of a scalar register to a vector register element

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov Si,Sj,Vk	ST	8200	1000001	None	Move a scalar to a vector element

Description

The contents of scalar register Si replace the contents of element Vk pointed to by Sj<6..0>. Bits Sj<63..7> are ignored.

Operation

```
Vk[Sj<6..0>] = Si;
```

mov.w Sk,VL

Move register (scalar to VL)

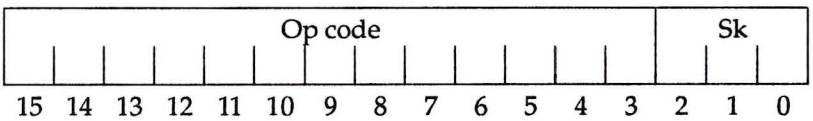
Purpose

To copy the contents of a scalar register (Sk) to the vector length (VL) register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov.w Sk,VL	ST	7DB8	0111110110111	None	Move Sk to VL

Description

The `mov Sk,VL` instruction replaces the contents of the VL register with the contents of scalar vector register, under control of the vector merge (VM) register.

Operation

```
if (Ak < 128) {
    VL = 128; }
else {
    if (Ak < 0) {
        VL = 0; }
    else {
        if Ak > 125 {
            VL = 128 }
        else {
            VL = Ak; } } }
```

mov Sj,VM,Sk
Move register (VM to scalar)

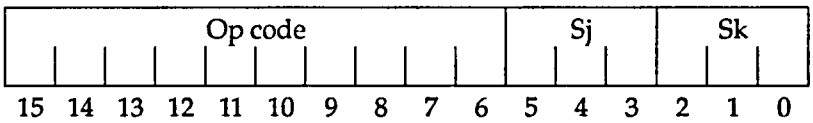
Purpose

To copy half of the contents from the vector merge (VM) register to a scalar register.

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov Sj,VM,Sk	ST	6140	0110000101	None	Load Sk from VM(Sj)

Description

If Sj is 0, the contents of scalar register Sk are replaced with the contents of VM<63..0>. If Sj is 1, the contents of scalar register Sk are replaced with the contents of VM<127..64>.

Operation

```
if (Sj == 0) {
    Sk = VM<63..0>; }
else {
    Sk = VM<127..64>; }
```

mov Sk,VML

Move register (scalar to VM lower)

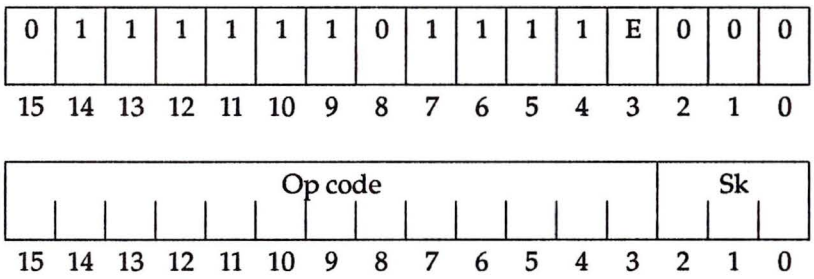
Purpose

To move a scalar register to the lower longword of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov Sk,VML	E0	7C50	0111110001010	None	Load VM<63..0> from Sk

Description

The contents of the lower longword of VM are replaced with the contents of Sk.

Operation

```
VM<63..0> = Sk;
```

mov Sk,VMU

Move register (scalar to VM upper)

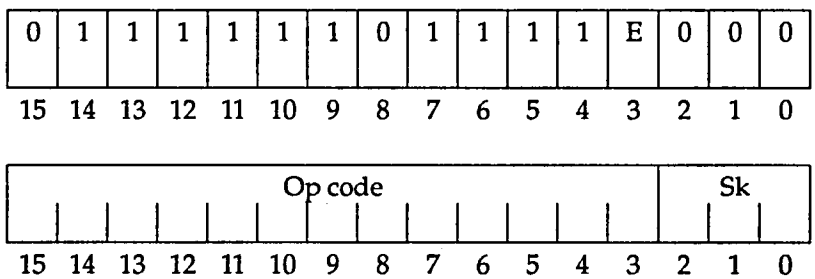
Purpose

To move a scalar register to the upper longword of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov Sk,VMU	E0	7C40	0111110001000	None	Load VM<127. .64> from Sk

Description

The contents of the upper longword of VM are replaced with the contents of Sk.

Operation

```
VM<127..64> = Sk;
```

mov.w Sk,VS
Move register (scalar to VS)

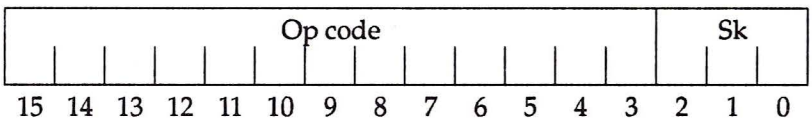
Purpose

To copy the contents of a scalar register Sk to the vector stride (VS) register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov.w Sk,VS	ST	7DA8	0111110110101	None	Move Sk to VS

Description

The least significant 32 bits of scalar register Sk replace the contents of VS.

Operation

```
VS = Sk <31..0>;
```

mov Sk,VV
Move register (scalar to VV)

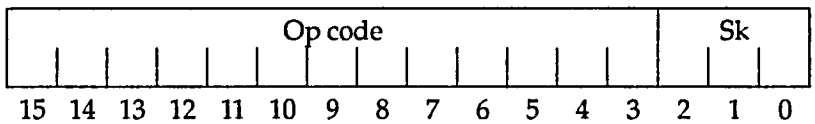
Purpose

To copy the least significant bit of a scalar register to the vector valid (VV) flag

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov Sk,VV	ST	7D70	0111110101110	None	Move scalar to vector valid flag

Description

The least significant bit of scalar register Sk replaces the VV flag.

Operation

```
VV = Sk<0>;
```

Exceptions

Ring violation (privileged instruction)

Notes

1. For C100 Series CPUs only, the current ring of execution must be 0 or a privileged instruction exception will occur.
2. For multiprocessing C Series CPUs, if the current ring of execution is ring 0, the `mov Sk, VV` operation is performed as described by the preceding operation pseudocode. If the current ring of execution is ring 1-4, the `mov Sk, VV` operation is not performed, that is, a nop (no operation) is performed.

mov TCPU,Sk

Move register (TCPU to a scalar)

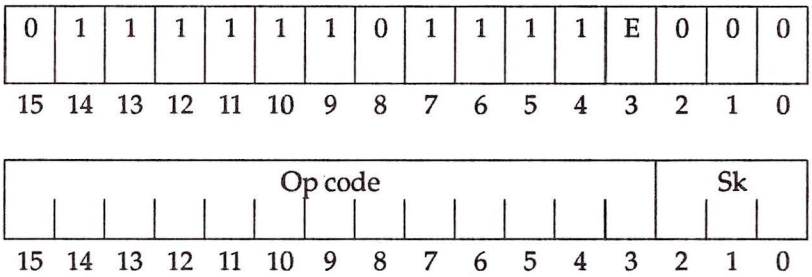
Purpose

To move the interrupt target CPU register (TCPU) to a scalar register

Application

C200/C3200 Series CPUs
(undefined code on C3400 and C3800 Series CPUs)

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov TCPU,Sk	E0	7C78	0111110001111	None	Move TCPU register to a scalar

Description

This instruction reads the interrupt target CPU (TCPU) register, which contains either a valid target CPUID or all ones.

Operation

```
Sk = TCPU;
```

Exceptions

Ring violation (privileged instruction)

mov TID,Sk

Move register (thread ID to a scalar)

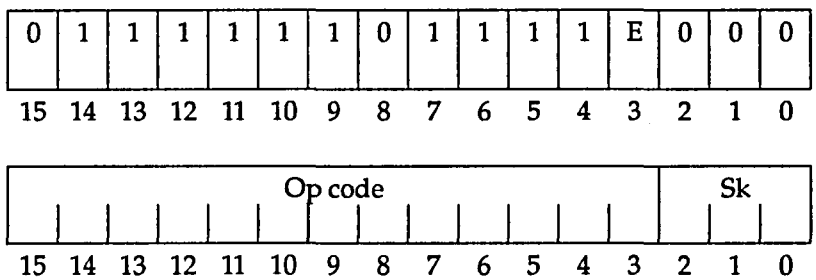
Purpose

To move the thread ID (TID) register to a scalar register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov TID,Sk	E0	7CB8	0111110010101	None	Move TID register to a scalar

Description

This instruction copies the thread ID (TID) for this CPU into Sk. The value moved to Sk is a zero-extended longword between 0 and the maximum number of threads.

Operation

```
Sk = THREAD_ID;
```

mov TOC,Sk

Move register (TOC to a scalar)

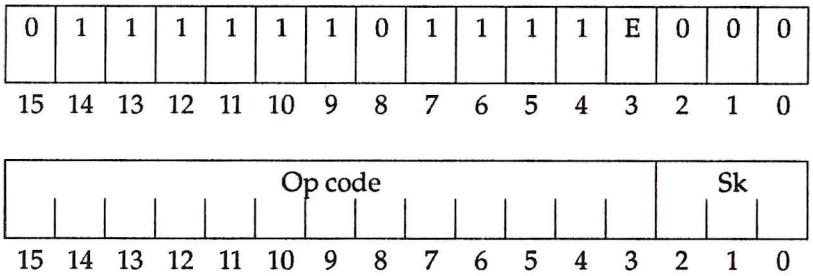
Purpose

To move the time-of-century (TOC) counter to a scalar register

Application

C130, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov TOC,Sk	E0	7C10	0111110000010	None	Move TOC counter to a scalar

Description

This instruction reads a pointer to the TOC counter from ring 0 page 0, and then combines the 4 16-bit I/O locations that define the clock into scalar register Sk.

Operation

```
Sk = TOC;
```

Notes

Refer to the *CONVEX Architecture Reference Manual (C Series)*, Chapter 10, "Timers," for more information on the TOC.

mov TTR,Sk

Move register (thread timer to a scalar)

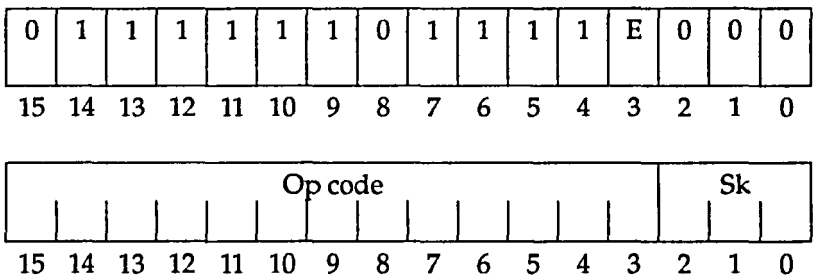
Purpose

To move the thread timer register (TTR) to a scalar register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov TTR,Sk	E0	7C28	0111110000101	None	Move TTR to a scalar

Description

This instruction copies the value in the thread timer register (TTR) to a scalar register.

Operation

```
Sk = THREAD_TIMER;
```

Notes

This instruction has no effect on the timer register.

mov VL,Ak

Move register (VL to an address)

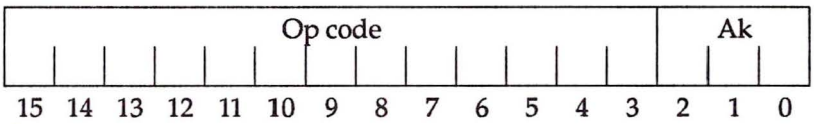
Purpose

To copy the contents of the vector length (VL) register to an address register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov VL,Ak	ST	7D90	0111110110010	None	Move VL to Ak

Description

The contents of Ak are replaced with the contents of VL.

Operation

```
Ak = VL;
```

mov.w VL,Sk

Move register (VL to a scalar)

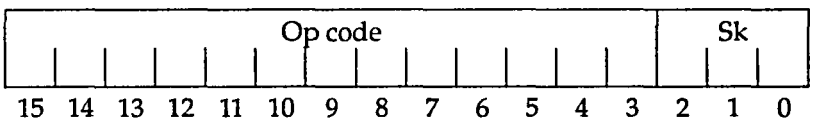
Purpose

To copy the contents of the vector length (VL) register to a scalar register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov.w VL,Sk	ST	7DB0	0111110110110	None	Move VL to Sk

Description

The contents of scalar register Sk are replaced with the contents of VL.

Operation

```
Sk <31..0> = VL;
```

mov VML,Sk

Move register (VM lower to a scalar)

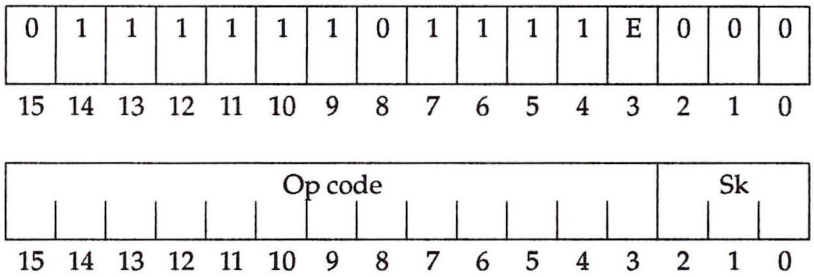
Purpose

To move the lower longwords of the vector merge (VM) register to a scalar register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov VML,Sk	E0	7C58	0111110001011	None	Load Sk from VM<63..0>

Description

The contents of Sk are replaced by the lower longword of the VM register.

Operation

```
Sk = VM<63..0>;
```

mov VMU,Sk

Move register (VM upper to a scalar)

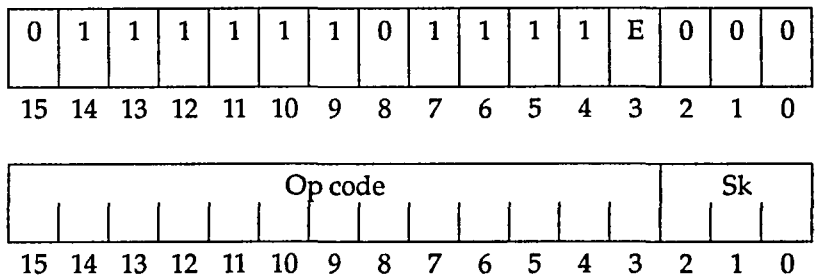
Purpose

To move the upper longword of the vector merge (VM) register to a scalar register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov VMU,Sk	E0	7C48	0111110001001	None	Load Sk from VM<127. .64>

Description

The contents of Sk are replaced by the upper longword of the VM register.

Operation

```
Sk = VM<127..64>;
```

mov Vi,Sj,Sk

Move register (vector element to a scalar)

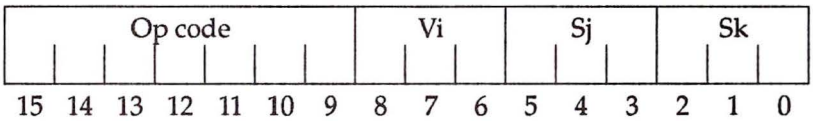
Purpose

To copy the contents of a vector element into a scalar register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov Vi,Sj,Sk	ST	8000	1000000	None	Move a vector element to a scalar

Description

The element of vector register Vi specified by Sj <6..0> replaces the contents of Sk.

Operation

```
Sk = Vi[Sj<6..0>];
```

mov VS,Ak

Move register (VS to an address)

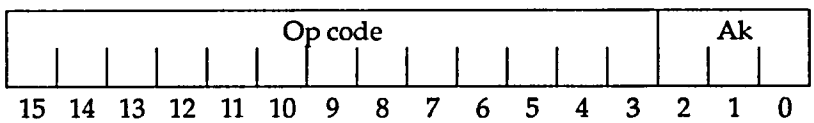
Purpose

To copy the contents of the vector stride (VS) register to an address register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov VS,Ak	ST	7D80	0111110110000	None	Move VS to Ak

Description

The contents of the VS register replace the contents of address register Ak.

Operation

```
Ak = VS;
```

mov.w VS,Sk

Move register (VS to a scalar)

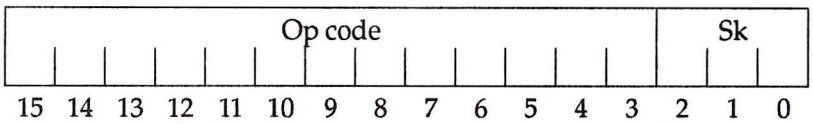
Purpose

To copy the contents of the vector stride (VS) register to a scalar register Sk

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mov.w VS,Sk	ST	7DA0	0111110110100	None	Move VS to Sk

Description

The contents of VS replace the least significant 32 bits of scalar register Sk.

Operation

```
Sk<31..0> = VS;
```

mski Sk
Mask interrupt

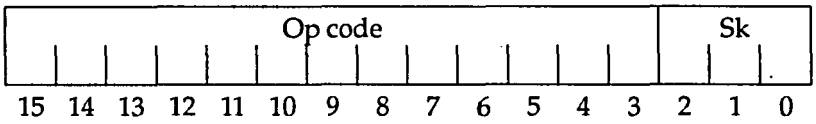
Purpose

To mask the virtual channels

Application

C100 Series CPUs only

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mski Sk	ST	7D60	0111110101100	None	Mask out interrupt

Description

Sk <7..0> replaces 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.

Operation

```
Set the interrupt mask to Sk<7..0>
```

Exceptions

Ring violation (privileged instruction)

Notes

The operating system must explicitly perform `mski` during interrupt service. This may require the saving of any previous mask values.

msync
Synchronize memory

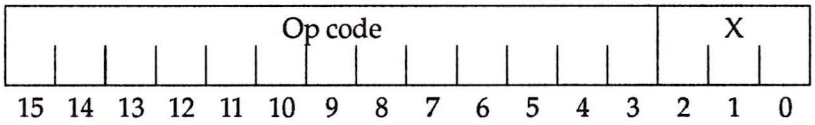
Purpose

To wait for all previous memory stores and loads to complete

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
msync	ST	7D58	0111110101011	None	Synchronize stores/loads to memory

Description

This instruction will wait (not complete) if the CPU has store data or load 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. In addition, this allows a consumer process to complete loading all data prior to informing a producer process of load completion.

Operation

```
while (memory_pipeline != EMPTY) {
    wait(); }
```

Note

The X field is unused.

mul.{h|w} #{n|N},Ak

Multiply registers (address by immediate)

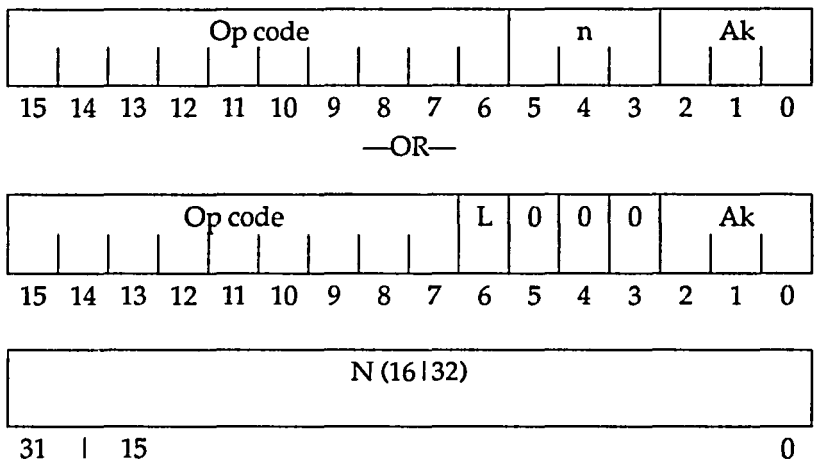
Purpose

To multiply the contents of an address register by an immediate operand

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mul.h #n,Ak	ST	5C80	0101110010	AIV	Multiply short immediate address halfword
mul.w #n,Ak	ST	5CC0	0101110011	AIV	Multiply short immediate address word
mul.h #N,Ak	ST	1600	000101100	AIV	Multiply immediate address halfword
mul.w #N,Ak	ST	1680	000101101	AIV	Multiply immediate address word

Description

The product of the contents of address register Ak and either the short immediate operand (#n) or the sign- extended long immediate operand (#N, length indicated by L) replaces the contents of Ak.
Sign extension does not occur for the 3 bits of the short immediate form.

Operation

```
Ak = Ak * Immediate;
```

Exceptions

h | w	Integer overflow

Note

The precision of the result is equal to the precision of the specified register and immediate.

mul.{h|w|s} #N,Sk

Multiply registers (scalar by immediate)

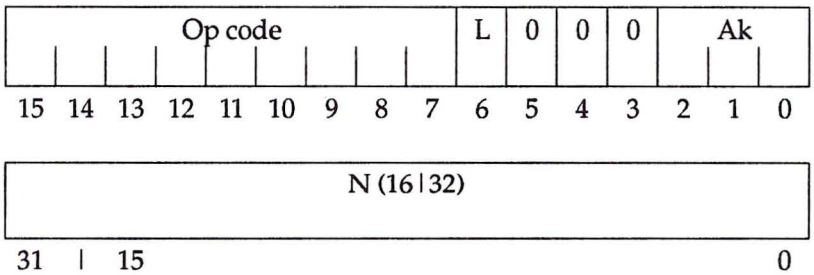
Purpose

To multiply the contents of a scalar register by an immediate operand

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mul.h #N,Sk	ST	1608	000101100	SIV	Multiply scalar/immediate integer halfword
mul.w #N,Sk	ST	1688	000101101	SIV	Multiply scalar/immediate integer word
mul.s #N,Sk	ST	1908	000110010	RO,OV,UN	Multiply scalar/immediate single float

Description

The product of the contents of the sign-extended long immediate operand (#N, length indicated by L) and the contents of scalar register Sk replaces the contents of Sk.

Operation

```
Sk = Sk * Immediate;
```

Exceptions

`h	w`	Integer overflow
`s`	Exponent overflow Exponent underflow Reserved operand

Note

The precision of the result is equal to the precision of the specified register and immediate operand.

mul.{h|w} Aj,Ak

Multiply registers (address by address)

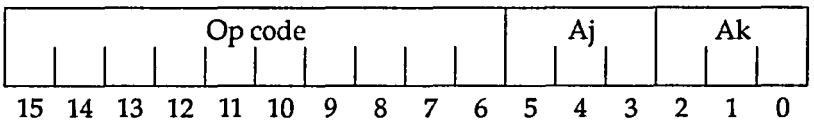
Purpose

To multiply the contents of two address registers

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mul.h Aj,Ak	ST	5C00	0101110000	AIV	Multiply address register halfword
mul.w Aj,Ak	ST	5C40	0101110001	AIV	Multiply address register word

Description

The product of the contents of address registers Aj and Ak replaces the contents of Ak.

Operation

```
Ak = Ak * Aj;
```

Exceptions

`h | w`	Integer overflow

Note

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 registers (scalar by scalar)

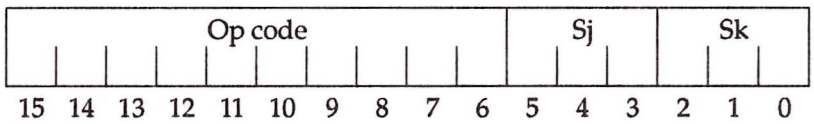
Purpose

To multiply the contents two scalar registers

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mul.b Sj,Sk	ST	5D00	0101110100	SIV	Multiply scalar/scalar integer byte
mul.h Sj,Sk	ST	5D40	0101110101	SIV	Multiply scalar/scalar integer halfword
mul.w Sj,Sk	ST	5D80	0101110110	SIV	Multiply scalar/scalar integer word
mul.l Sj,Sk	ST	5DC0	0101110111	SIV	Multiply scalar/scalar integer longword
mul.s Sj,Sk	ST	5700	0101011100	OV,UN,RO	Multiply scalar/scalar single float
mul.d Sj,Sk	ST	5740	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.

Operation

```
Sk = Sk * Sj;
```

Exceptions

`b|h|w|l`	Integer overflow
`s|d`	Exponent overflow Exponent underflow Reserved operand

Note

The precision of the result is equal to the precision of the two specified registers.

mul.{b|h|w|l|s|d} Vi,Sj,Vk

Multiply registers (vector by scalar)

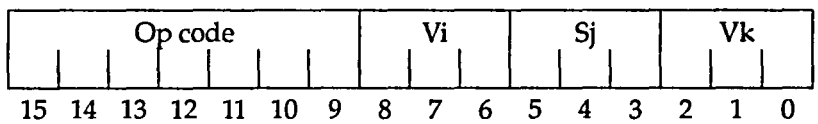
Purpose

To multiply the contents of a vector register by the contents of a scalar register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mul.b Vi,Sj,Vk	ST	E800	1110100	SIV	Multiply vector/scalar integer byte
mul.h Vi,Sj,Vk	ST	EA00	1110101	SIV	Multiply vector/scalar integer halfword
mul.w Vi,Sj,Vk	ST	EC00	1110110	SIV	Multiply vector/scalar integer word
mul.l Vi,Sj,Vk	ST	EE00	1110111	SIV	Multiply vector/scalar integer longword
mul.s Vi,Sj,Vk	ST	9800	1001100	OV,UN,RO	Multiply vector/scalar single float
mul.d Vi,Sj,Vk	ST	9A00	1001101	OV,UN,RO	Multiply vector/scalar double float

Description

The contents of each 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.

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

Note

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 registers (vector by scalar) (masked)

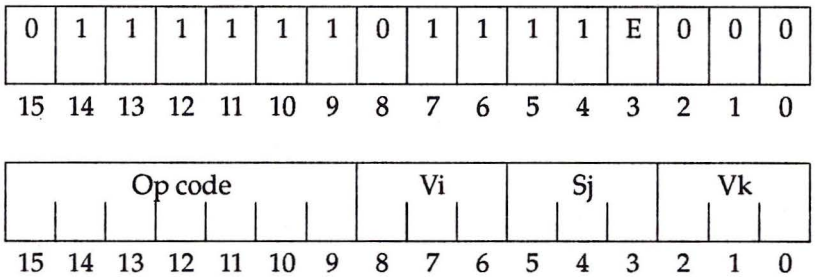
Purpose

To multiply a vector by a scalar under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mul.b.t Vi,Sj,Vk	E1	E800	1110100	SIV	Multiply vector/scalar byte (VM)
mul.b.f Vi,Sj,Vk	E0	E800	1110100	SIV	Multiply vector/scalar byte (!VM)
mul.h.t Vi,Sj,Vk	E1	EA00	1110101	SIV	Multiply vector/scalar halfword (VM)
mul.h.f Vi,Sj,Vk	E0	EA00	1110101	SIV	Multiply vector/scalar halfword (!VM)
mul.w.t Vi,Sj,Vk	E1	EC00	1110110	SIV	Multiply vector/scalar word (VM)
mul.w.f Vi,Sj,Vk	E0	EC00	1110110	SIV	Multiply vector/scalar word (!VM)
mul.l.t Vi,Sj,Vk	E1	EE00	1110111	SIV	Multiply vector/scalar longword (VM)
mul.l.f Vi,Sj,Vk	E0	EE00	1110111	SIV	Multiply vector/scalar longword (!VM)
mul.s.t Vi,Sj,Vk	E1	9800	1001100	OV,UN,RO	Multiply vector/scalar single (VM)
mul.s.f Vi,Sj,Vk	E0	9800	1001100	OV,UN,RO	Multiply vector/scalar single (!VM)
mul.d.t Vi,Sj,Vk	E1	9A00	1001101	OV,UN,RO	Multiply vector/scalar double (VM)
mul.d.f Vi,Sj,Vk	E0	9A00	1001101	OV,UN,RO	Multiply vector/scalar double (!VM)

Description

The contents of each 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, only if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

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; } /* end of switch */
```

Exceptions

`b|h|w|l`	Integer overflow
`s|d`	Exponent overflow Exponent underflow Reserved operand

Note

The precision of the result is equal to the precision of the specified registers.

mul.{b|h|w|l|s|d} Vi,Vj,Vk
Multiply registers (vector by vector)

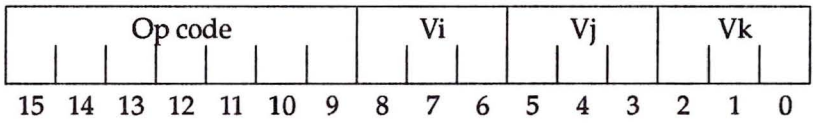
Purpose

To multiply the corresponding elements of two vector registers

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mul.b Vi,Vj,Vk	ST	E000	1110000	SIV	Multiply vector/vector integer byte
mul.h Vi,Vj,Vk	ST	E200	1110001	SIV	Multiply vector/vector integer halfword
mul.w Vi,Vj,Vk	ST	E400	1110010	SIV	Multiply vector/vector integer word
mul.l Vi,Vj,Vk	ST	E600	1110011	SIV	Multiply vector/vector integer longword
mul.s Vi,Vj,Vk	ST	9000	1001000	OV,UN,RO	Multiply vector/vector single float
mul.d Vi,Vj,Vk	ST	9200	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.

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

Note

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 registers (vector by vector) (masked)

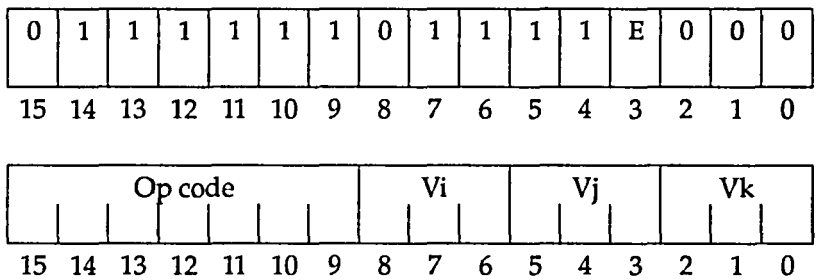
Purpose

To multiply two vectors under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
mul.b.t Vi,Vj,Vk	E1	E000	1110000	SIV	Multiply byte vectors (VM)
mul.b.f Vi,Vj,Vk	E0	E000	1110000	SIV	Multiply byte vectors (IVM)
mul.h.t Vi,Vj,Vk	E1	E200	1110001	SIV	Multiply halfword vectors (VM)
mul.h.f Vi,Vj,Vk	E0	E200	1110001	SIV	Multiply halfword vectors (IVM)
mul.w.t Vi,Vj,Vk	E1	E400	1110010	SIV	Multiply word vectors (VM)
mul.w.f Vi,Vj,Vk	E0	E400	1110010	SIV	Multiply word vectors (IVM)
mul.l.t Vi,Vj,Vk	E1	E600	1110011	SIV	Multiply longword vectors (VM)
mul.l.f Vi,Vj,Vk	E0	E600	1110011	SIV	Multiply longword vectors (IVM)
mul.s.t Vi,Vj,Vk	E1	9000	1001000	OV,UN, RO	Multiply single vectors (VM)
mul.s.f Vi,Vj,Vk	E0	9000	1001000	OV,UN, RO	Multiply single vectors (IVM)
mul.d.t Vi,Vj,Vk	E1	9200	1001001	OV,UN, RO	Multiply double vectors (VM)
mul.d.f Vi,Vj,Vk	E0	9200	1001001	OV,UN, RO	Multiply double vectors (IVM)

mul.{b|h|w|l|s|d}.{t|f} Vi,Vj,Vk

Description

The contents of each of the first VL elements of vector register Vk is replaced by the product of the contents of the corresponding elements of Vi and Vj, only if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

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;  } /* end of switch */
```

Exceptions

`b|h|w|l`	Integer overflow
`s|d`	Exponent overflow Exponent underflow Reserved operand

Notes

The precision of the result is equal to the precision of the specified registers.

neg.{h|w} Aj,Ak
Negate register (address)

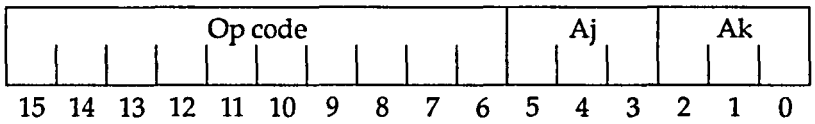
Purpose

To negate arithmetically the contents of an address register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
neg.h Aj,Ak	ST	5680	0101011010	C,AIV	Negate address register halfword
neg.w Aj,Ak	ST	56C0	0101011011	C,AIV	Negate address register word

Description

The two's complement of address register Aj replaces the contents of Ak.

Operation

```
Ak = 0 - Aj;
```

Exceptions

`h | w`	Integer overflow

Note

Overflow can occur for the negation of the most negative integer.

neg.{b|h|w|l|s|d} Sj,Sk
Negate register (scalar)

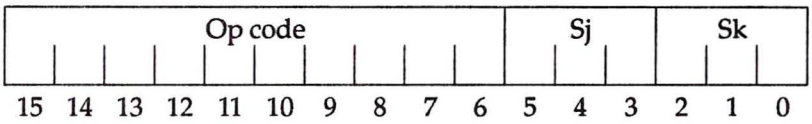
Purpose

To negate arithmetically the contents of a scalar register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
neg.b Sj,Sk	ST	6F00	0110111100	SIV	Negate scalar/scalar integer byte
neg.h Sj,Sk	ST	6F40	0110111101	SIV	Negate scalar/scalar integer halfword
neg.w Sj,Sk	ST	6F80	0110111110	SIV	Negate scalar/scalar integer word
neg.l Sj,Sk	ST	6FC0	0110111111	SIV	Negate scalar/scalar integer longword
neg.s Sj,Sk	ST	6580	0110010110	RO	Negate scalar/scalar single float
neg.d Sj,Sk	ST	65C0	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.

Operation

```
Sk = 0 - Sj;
```

Exceptions

`b|h|w|l`	Integer overflow
`s|d`	Reserved operand

Note

Overflow can occur for the negation of the most negative integer.

neg.{b|h|w|l|s|d} Vj,Vk
Negate register (vector)

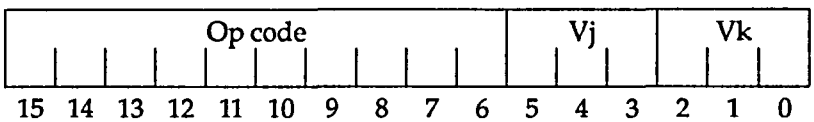
Purpose

To negate arithmetically the contents of a vector register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
neg.b Vj,Vk	ST	6E00	0110111000	SIV	Negate vector/vector integer byte
neg.h Vj,Vk	ST	6E40	0110111001	SIV	Negate vector/vector integer halfword
neg.w Vj,Vk	ST	6E80	0110111010	SIV	Negate vector/vector integer word
neg.l Vj,Vk	ST	6EC0	0110111011	SIV	Negate vector/vector integer longword
neg.s Vj,Vk	ST	6480	0110010010	RO	Negate vector/vector single float
neg.d Vj,Vk	ST	64C0	0110010011	RO	Negate vector/vector double float

Description

The contents of each of the first VL elements of vector register Vk is replaced by zero minus the contents of the corresponding element of Vj.

Operation

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

Exceptions

`b|h|w|l`	Integer overflow
`s|d`	Reserved operand

Note

Overflow can occur for the negation of the most negative integer.

neg.{b|h|w|l|s|d}.{t|f} Vj,Vk
Negate register (vector) (masked)

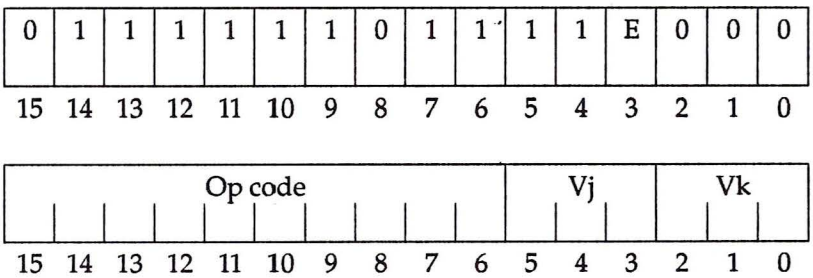
Purpose

To negate a vector under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
neg.b.t Vj,Vk	E1	6E00	0110111000	SIV	Negate byte vector (VM)
neg.b.f Vj,Vk	E0	6E00	0110111000	SIV	Negate byte vector (!VM)
neg.h.t Vj,Vk	E1	6E40	0110111001	SIV	Negate halfword (VM)
neg.h.f Vj,Vk	E0	6E40	0110111001	SIV	egate halfword (!VM)
neg.w.t Vj,Vk	E1	6E80	0110111010	SIV	Negate word (VM)
neg.w.f Vj,Vk	E0	6E80	0110111010	SIV	Negate word (!VM)
neg.l.t Vj,Vk	E1	6EC0	0110111011	SIV	Negate longword (VM)
neg.l.f Vj,Vk	E0	6EC0	0110111011	SIV	Negate longword (!VM)
neg.s.t Vj,Vk	E1	6480	0110010010	RO	Negate single (VM)
neg.s.f Vj,Vk	E0	6480	0110010010	RO	Negate single (!VM)
neg.d.t Vj,Vk	E1	64C0	0110010011	RO	Negate double (VM)
neg.d.f Vj,Vk	E0	64C0	0110010011	RO	Negate double (!VM)

Description

The contents of each of the first VL elements of vector register Vk is replaced by zero minus the contents of the corresponding element of Vj, only if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

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;  } /* end of switch */
```

Exceptions

`b|h|w|l`	Integer overflow
`s|d`	Reserved operand

Note

Overflow can occur for the negation of the most negative integer.

nop
No operation

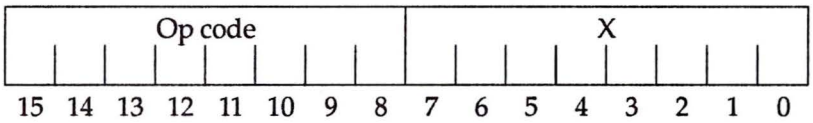
Purpose

To perform no operation

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
nop	ST	7000	01110000	None	No operation (branch never)

Description

This instruction performs no operation.

Operation

```
Perform no operation;
```

Notes

The X field is unused.

not Aj,Ak
Complement register (address)

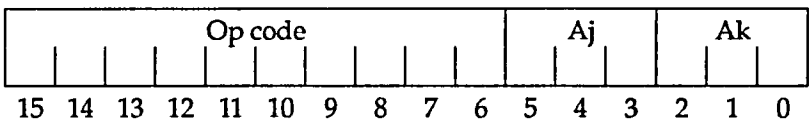
Purpose

To complement the contents of an address register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
not Aj,Ak	ST	52C0	0101001011	None	Complement address register

Description

The one's complement of the contents of address register Aj replaces the contents of Ak.

Operation

```
Ak = ~Aj;
```

not Sj,Sk
Complement register (scalar)

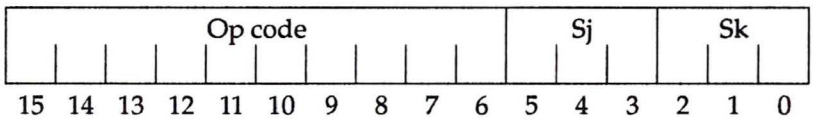
Purpose

To complement the contents of a scalar register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
not Sj,Sk	ST	53C0	0101001111	None	Complement scalar/scalar

Description

The one's complement of scalar register Sj replaces the contents of Sk.

Operation

```
Sk = ~Sj;
```

not Vj,Vk
Complement register (vector)

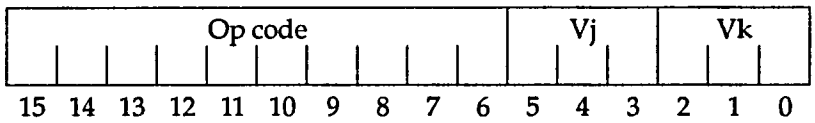
Purpose

To complement the elements of a vector

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
not Vj,Vk	ST	62C0	0110001011	None	Complement a vector

Description

The one's complement of the contents of the contents of each of the first VL elements of vector register Vj replaces the corresponding elements of Vk.

Operation

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

not.{t|f} Vj,Vk
Complement register (vector) (masked)

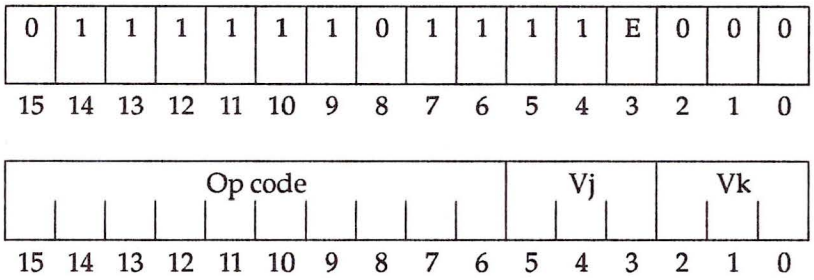
Purpose

To complement the contents of a vector under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
not.t Vj,Vk	E1	62C0	0110001011	None	Complement a vector (VM)
not.f Vj,Vk	E0	62C0	0110001011	None	Complement a vector (!VM)

Description

The one's complement of the contents of each of the first VL elements of vector register Vj replaces the corresponding elements of Vk, only if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

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;  } /* end of switch */
```

or #N,Ak

Or register (address with immediate)

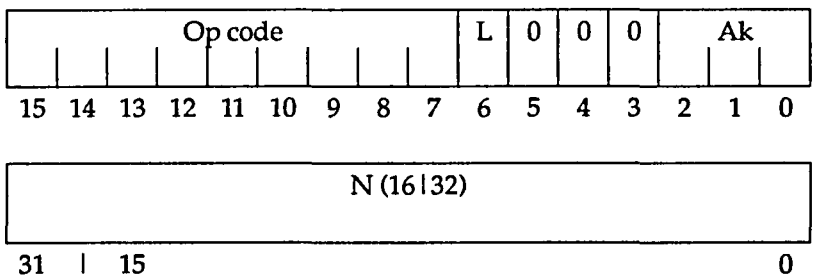
Purpose

To `or` the contents of an address register and an immediate operand

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
or #N,Ak	ST	1280	000100101	None	Or immediate to address register

Description

The logical `or` of the sign-extended long immediate operand (#N, length indicated by L) and the contents of address register Ak replace the contents of Ak.

Operation

```
Ak = Ak | Immediate;
```

or #N,Sk

Or register (scalar with immediate)

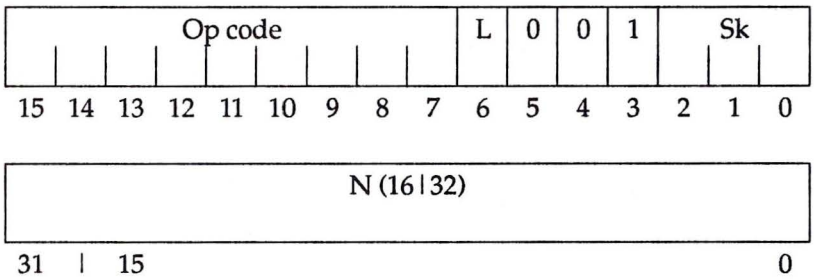
Purpose

To `or` the contents of a scalar register and an immediate operand

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
or #N,Sk	ST	1288	000100101	None	Or scalar/immediate

Description

The logical `or` of the sign-extended long immediate operand (#N, length indicated by L) and the contents of the least significant 32 bits of scalar register Sk replaces the least significant 32 bits of Sk. The most significant 32 bits of Sk are not affected.

Operation

```
Sk<31..0> = Sk<31..0> | Immediate;
```

or Aj,Ak

Or registers (address with address)

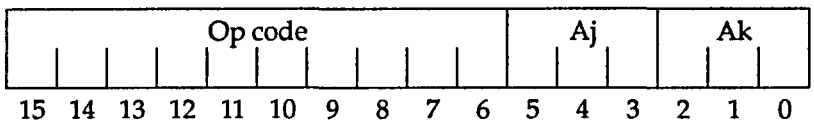
Purpose

To `or` the contents of two address registers

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
or Aj,Ak	ST	5240	0101001001	None	Or address register

Description

The logical `or` of the contents of address registers Aj and Ak replaces the contents of Ak.

Operation

```
Ak = Ak | Aj;
```

or Sj,Sk

Or registers (scalar with scalar)

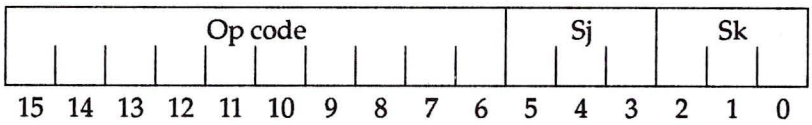
Purpose

To `or` the contents of two scalar registers

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
or Sj,Sk	ST	5340	0101001101	None	Or scalar/scalar

Description

The logical `or` of the contents of scalar registers Sj and Sk replaces the contents of Sk.

Operation

```
Sk = Sk | Sj;
```

or Vi,Sj,Vk

Or registers (vector with scalar)

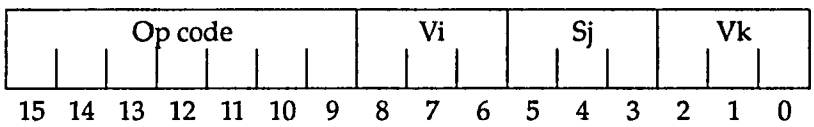
Purpose

To `or` the elements of a vector register and the contents of a scalar register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
or Vi,Sj,Vk	ST	AA00	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.

Operation

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

or.{t|f} Vi,Sj,Vk

Or register (vector with scalar) (masked)

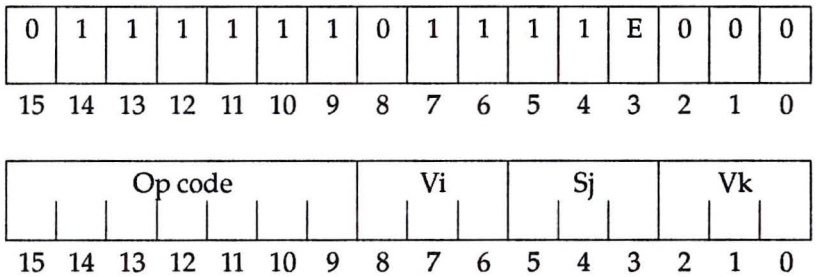
Purpose

To `or` the contents of a vector and a scalar under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
or.t Vi,Sj,Vk	E1	AA00	1010101	None	Or vector/scalar (VM)
or.f Vi,Sj,Vk	E0	AA00	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 (for `.t`) or clear (for `.f`).

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; } /* end of switch */
```

or Vi,Vj,Vk

Or registers (vector with vector)

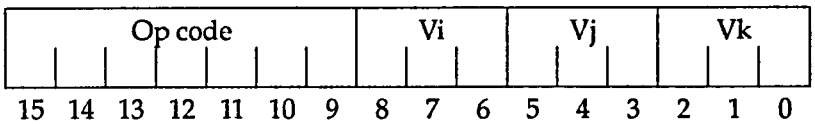
Purpose

To `or` the elements of two vector registers

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
or Vi,Vj,Vk	ST	A200	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.

Operation

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

or.{t|f} Vi,Vj,Vk

Or registers (vector with vector) (masked)

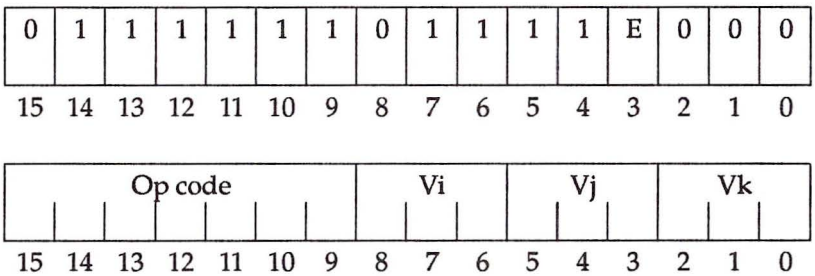
Purpose

To `or` the contents of two vectors under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
or.t Vi,Vj,Vk	E1	A200	1010001	None	Or two vectors (VM)
or.f Vi,Vj,Vk	E0	A200	1010001	None	Or two vectors (!VM)

Description

The contents of 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, only if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

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; } /* end of switch */
```

parity {Vk|Sk}
Exclusive or reduce register (vector)

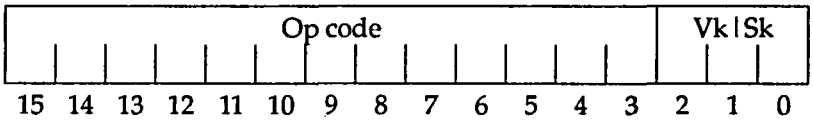
Purpose

To `exclusive or` reduce all the elements of a vector register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
parity Vk	ST	7E30	0111111000110	None	Exclusive or reduce a vector

Description

The `exclusive or` of all 64 bits of the contents of scalar register Sk and the first VL elements of vector register Vk replace Sk. r reduce a vector

Operation

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

Notes

1. Initialize the scalar register properly for the first use of this instruction (usually 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).

parity.{t|f} {Vk|Sk}

Exclusive or reduce register (vector) (masked)

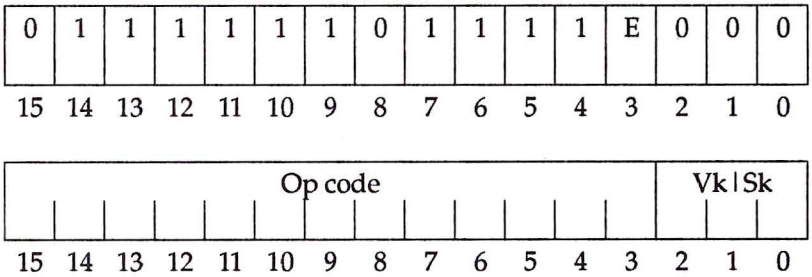
Purpose

To `exclusive or` reduce the elements of a subset of a vector register under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
parity.t Vk	E1	7E30	0111111000110	None	Exclusive or reduce vector (VM)
parity.f Vk	E0	7E30	0111111000110	None	Exclusive or reduce vector (!VM)

Description

The `exclusive or` of all 64 bits of the contents of scalar register Sk and the first VL elements of vector register Vk replace Sk, only if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

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;  } /* end of switch */
```

Notes

1. Initialize the scalar register properly for the first use of this instruction (usually 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

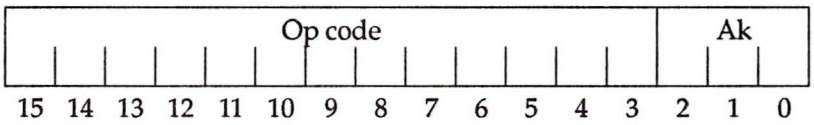
Purpose

To purge an address translation unit (ATU) entry

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
pate Ak	ST	7C28	0111110000101	None	Purge ATU entry

Description

If there is an ATU entry associated with the address contained in address register Ak, that ATU entry is purged (marked invalid). All other ATU entries are left unchanged.

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.

Operation

```
if (CPU == C200) { /* multiprocessing C Series */
    ATU_valid_bit(Ak) = 0; /* purge All CPU(s) */ }
else {              /* C100 Series */
    ATU_valid_bit(Ak) = 0; /* purge All CPU(s) */
    (Purge L_cache);
    (Purge I_cache); }
```

Exceptions

Ring violation (privileged instruction)

Notes

For C200/C3200, C3400, C3800 Series CPUs:

1. All the entries in each CPU's ATU are purged.
2. The logical cache (Lcache) or the instruction cache (Icache) are not purged.

patu
Purge ATUs

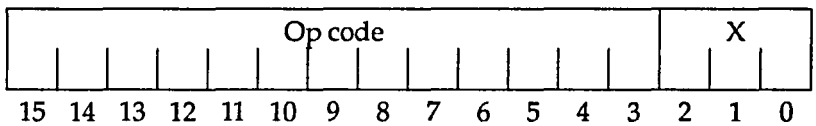
Purpose

To purge all address translation unit (ATU) entries

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
patu	ST	7C20	0111110000100	None	Purge the entire ATU

Description

All entries in the ATU are purged, including the logical and instruction caches.

Operation

```
if (CPU == C200) { /* multiprocessing C Series */
    All_CPUs_ATU_valid_bits = 0; }
else {                    /* C100 Series only */
    ATU_valid_bits = 0;
    (Purge L_cache);
    (Purge I_cache); }
```

Exception

Ring violation (privileged instruction)

Notes

1. For multiprocessing C Series CPUs:
   - All the entries in each CPU's ATU are purged.
   - The logical cache (Lcache) or the instruction cache (Icache) are not purged.
2. The X field is unused.

pbkpt
Process breakpoint

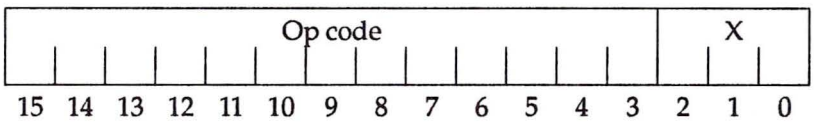
Purpose

To force all threads associated with the current process to stop execution

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
pbkpt	ST	7DC8	0111110111001	See note 1	Force process breakpoint exception

Description

A ring 0 system exception occurs 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 `0x14` and qualifier of 0.

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;
```

Notes

1. The PSW is cleared to all zero after the PSW is pushed on the stack.
2. 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.
3. Refer to the *CONVEX Architecture Reference Manual (C Series)*, Chapter 12. "Operating system exceptions," for a more detailed explanation of the `pbkpt` instruction.
4. The X field is unused.

pfork *effa*,Ak
Post a fork

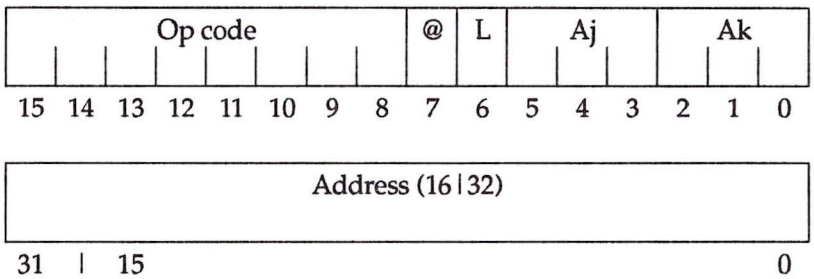
Purpose

To post the need for a CPU to join in an ongoing process

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
pfork *effa*,Ak	ST	2300	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 cleared 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, that is, there is no currently posted fork. Similarly, the `snd` operation to forkposted loads the constant "PFORKED" into `fork.type` and Ak into `fork.SP`.

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); }
```

Notes

1. At most, `pfork` will add one CPU to the process. Use `spawn` to attempt to add multiple CPUs to a process.
2. Once a fork is posted, the fork must either be taken by a CPU or cleared with `cfork P` before another fork can be successfully posted.
3. Refer to the *CONVEX Architecture Reference Manual (C Series)*, Chapter 9, "Multiprocessor management," for more information on `pfork` and forking operations.

pich
Purge instruction cache

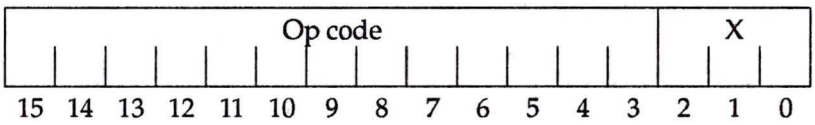
Purpose

To purge the entire instruction cache (Icache)

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
pich	ST	7C30	0111110000110	None	Purge the Icache

Description

All the entries in the instruction cache (Icache) are purged.

Operation

```
Icache_valid_bits = 0;
```

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 unused.

plc.t Sj,Sk
Population count (scalar)

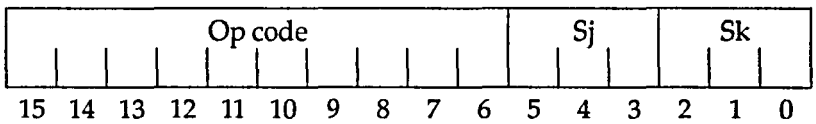
Purpose

To determine the number of ones in a scalar register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
plc.t Sj,Sk	ST	4580	0100010110	None	Count the number of ones in Sj

Description

The number of 1 bits in the 64 bits of scalar register Sj replaces the contents of Sk.

Operation

```
temp = 0;
for (a = 0; a <= 63; a++) {
    if (Sj<a> == 1) {       temp = temp + 1; } }
Sk = temp;
```

plc.t Vj,Vk
Population count (vector)

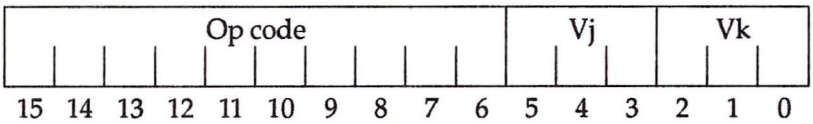
Purpose

To count the number of ones in each vector element

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
plc.t Vj,Vk	ST	6340	0110001101	None	Population count of a vector

Description

The number of one bits contained in the 64 bits of the corresponding element of Vj replaces each of the first VL elements of Vk.

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; }
```

plc.t.{t|f} Vj,Vk
Population count (vector) (masked)

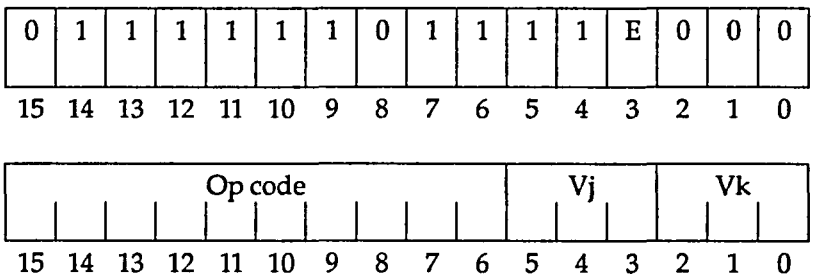
Purpose

To count the number of ones in each vector element under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
plc.t.t Vj,Vk	E1	6340	0110001101	None	Population count of vector (VM)
plc.t.f Vj,Vk	E0	6340	0110001101	None	Population count of vector (!VM)

plc.t.{t|f} Vj,Vk

Description

The contents of each of the first VL elements of Vk is replaced by the number of one bits contained in the 64 bits of the corresponding element of Vj, only if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

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; } /* end of switch */
```

plc.{t|f} VM,Sk
Population count (VM)

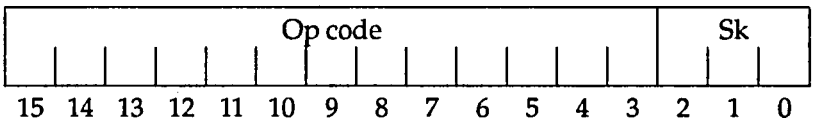
Purpose

To load the number of ones or zeros in vector merge (VM) into a scalar register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
plc.f VM,Sk	ST	7EE0	0111111011100	None	Load the number of zeros in VM into Sk
plc.t VM,Sk	ST	7EE8	0111111011101	None	Load the number of ones in VM into Sk

Description

The number of one bits (for `plc.t`) or zeros bits (for `plc.f`) in the VM register replaces the contents of Sk.

Operation

```
temp = 0;
switch (E) {    /* op code 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;  } /* end of switch */
```

plch
Purge logical cache

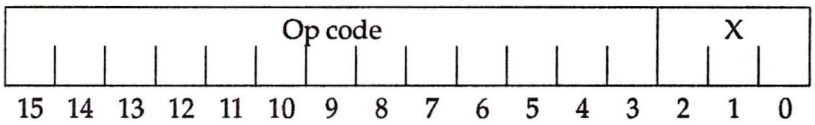
Purpose

To purge the logical cache (Lcache)

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
plch	ST	7C38	0111110000111	None	Purge the Lcache

Description

All the entries in the logical cache (Lcache) are purged.

Operation

```
Lcache_valid_bits = 0;
```

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 unused.

pmod
Purge modified bits

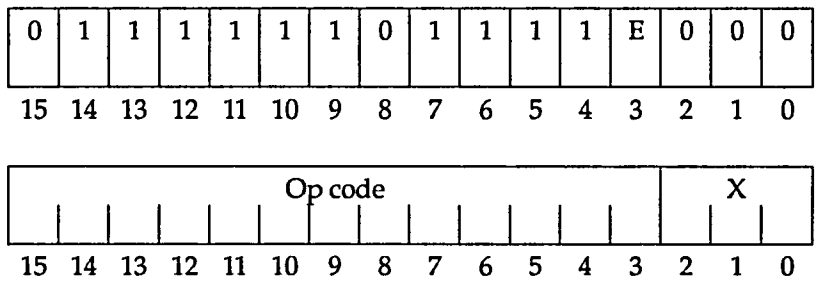
Purpose

To purge the encached modified bits

Application

C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
pmod	E0	7C98	0111110010011	None	Purge the modified bits

Description

All CPUs' local encachements of the modified bits are purged.

Operation

```
Modified bits = 0;
```

Exception

Ring violation (privileged instruction)

Note

Refer to the *CONVEX Architecture Reference Manual (C Series)*, Chapter 4, "Physical address space," for more information on the modified bits.

pop.w Ak
Pop to address register

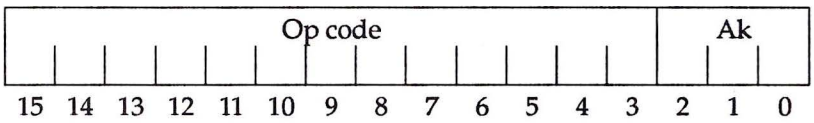
Purpose

To pop a word from the stack into an address register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
pop.w Ak	ST	7D10	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 4.

Operation

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

pop.{w|l} Sk
Pop to scalar register

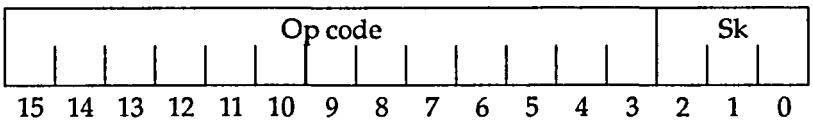
Purpose

To pop one element from the stack into a scalar register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
pop.w Sk	ST	7D30	0111110100110	None	Pop Sk<31..0> from the stack
pop.l Sk	ST	7D38	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 lower 32 bits of scalar register Sk. A0 is then incremented by 4.

`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 8.

Operation

```
Sk<31..0> = c(A0);      /* pop.w */
A0 = A0 + 4;

Sk = c(A0);             /* pop.l */
A0 = A0 + 8;
```

popr *effa*,Ak

Pop resource to address register

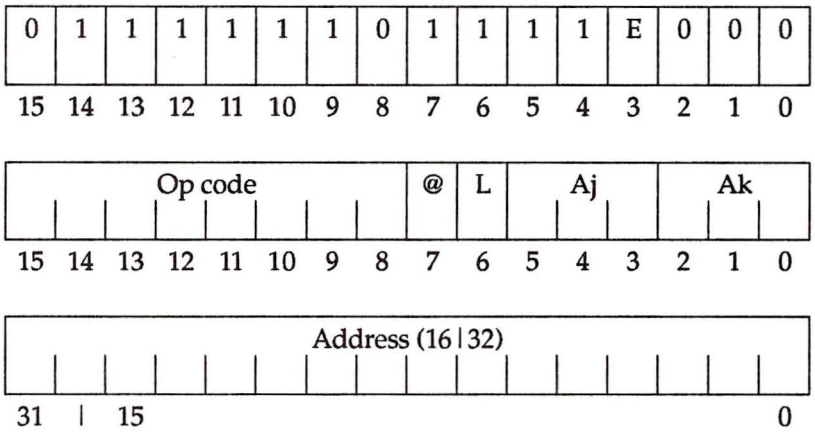
Purpose

To pop a word from a resource structure into an address register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
popr Ak,*effa*	E0	0800	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 SC reflects the status of the operation. SC is 0 for a resource structure underflow, and SC is 1 for a successful pop operation.

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; }
```

Notes

1. This instruction is atomic.
2. Address carry (C) and scalar carry (SC) are affected as described by the preceding operation pseudocode.
3. When an underflow occurs (that is, 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 Manual (C Series)*, Chapter 7, "Process structures," for more information on the `popr` instruction and resource structures.

pref
Purge reference bits

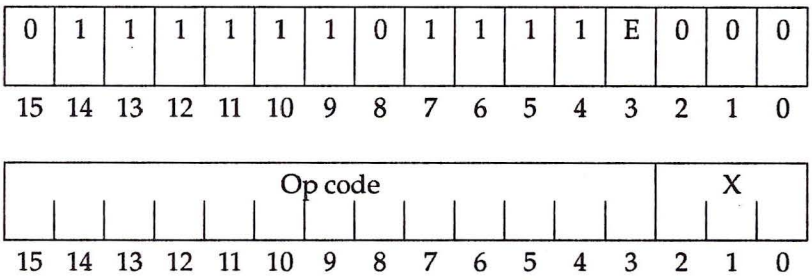
Purpose

To purge the encached reference bits

Application

C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
pref	E0	7C90	0111110010010	None	Purge the reference bits

Description

All CPUs' local encachements of the reference bits are purged.

Operation

```
Reference bits = 0;
```

Exceptions

Ring violation (privileged instruction)

Note

Refer to the *CONVEX Architecture Reference Manual (C Series)*, Chapter 4, "Physical address space," for more information on the reference bits.

prod.{b|h|w|l|s|d} {Vk|Sk}
Product vector register

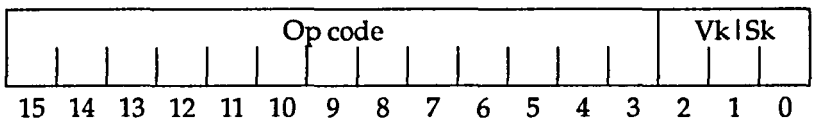
Purpose

To obtain the product of all elements of a vector register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
prod.b Vk	ST	7EC0	0111111011000	SIV	Multiply reduce a vector of bytes
prod.h Vk	ST	7EC8	0111111011001	SIV	Multiply reduce a vector of halfwords
prod.w Vk	ST	7ED0	0111111011010	SIV	Multiply reduce a vector of words
prod.l Vk	ST	7ED8	0111111011011	SIV	Multiply reduce a vector of longwords
prod.s Vk	ST	7E90	0111111010010	OV,UN,RO	Multiply reduce a vector of single float
prod.d Vk	ST	7E98	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.

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

Notes

1. Initialize the scalar register properly for the first use of the `prod` reduce instruction (usually to 1).
2. The sequence of the products performed by the hardware is `not` identical to the pseudocode sequence as noted above, that is, it is implementation-specific. The C100 Series and multiprocessing C Series CPUs execute a different order of reduction and may get different results. For more information, refer to the discussion of vector operations in Chapter 2.
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 register (masked)

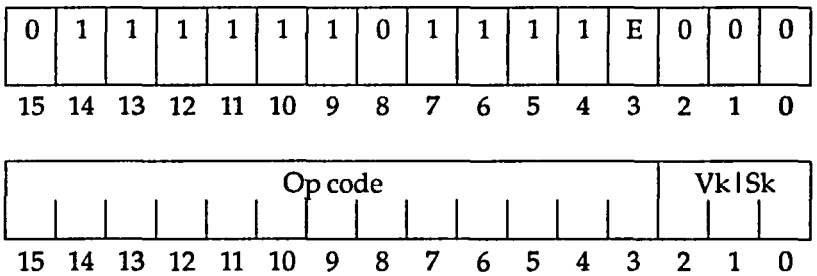
Purpose

To obtain the product of a subset of the elements of a vector register under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
prod.b.t Vk	E1	7EC0	0111111011000	SIV	Multiply reduce byte vector (VM)
prod.b.f Vk	E0	7EC0	0111111011000	SIV	Multiply reduce byte vector (IVM)
prod.h.t Vk	E1	7EC8	0111111011001	SIV	Multiply reduce halfword vector (VM)
prod.h.f Vk	E0	7EC8	0111111011001	SIV	Multiply reduce halfword vector (IVM)
prod.w.t Vk	E1	7ED0	0111111011010	SIV	Multiply reduce word vector (VM)
prod.w.f Vk	E0	7ED0	0111111011010	SIV	Multiply reduce word vector (IVM)
prod.l.t Vk	E1	7ED8	0111111011011	SIV	Multiply reduce longword vector (VM)
prod.l.f Vk	E0	7ED8	0111111011011	SIV	Multiply reduce longword vector (IVM)
prod.s.t Vk	E1	7E90	0111111010010	OV,UN,RO	Multiply reduce single vector (VM)
prod.s.f Vk	E0	7E90	0111111010010	OV,UN,RO	Multiply reduce single vector (IVM)
prod.d.t Vk	E1	7E98	0111111010011	OV,UN,RO	Multiply reduce double vector (VM)
prod.d.f Vk	E0	7E98	0111111010011	OV,UN,RO	Multiply reduce double vector (IVM)

prod.{b|h|w|l|s|d}.{t|f} {Vk|Sk}

Description

The product of the contents of scalar register Sk and the first VL elements of vector register Vk replaces Sk, only if elements the corresponding VM bit is set (for `.t`) or clear (for `.f`).

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 */
        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 */
        break;  } /* end of switch */
```

Exceptions

`b|h|w|l`	Integer overflow
`s|d`	Exponent overflow Exponent underflow Reserved operand

Notes

1. Initialize the scalar register properly for the first use of the `prod` reduce instruction (usually to 1).
2. The sequence of the products performed by the hardware is `not` identical to the pseudocode sequence as noted above, that is, it is implementation-specific. The C100 Series and multiprocessing C Series CPUs execute the reduction in a different order and may get different results. For more information, refer to the discussion of vector operations in Chapter 2.
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

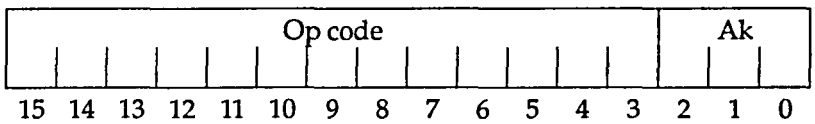
Purpose

To push the contents of an address register onto the stack

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Descriptionp
psh.w Ak	ST	7D00	0111110100000	None	Push an address register

Description

After A0 is decremented by 4, the contents of address register Ak replace the contents of the memory addressed by A0.

Operation

```
A0 = A0 - 4; /* See notes below */
c(A0) = Ak;
```

Notes

1. The `psh.w Ak` operation performed by the hardware is not identical to the pseudocode sequence as noted above, that is, it is implementation-specific.
2. C100 Series CPUs execute the decrement of A0 and the contents of address register Ak replace the contents of the memory addressed by A0.
3. Multiprocessing C Series CPUs execute the decrement of A0 and the contents of address register Ak replace the contents of the memory addressed by A0 (A0 - 4), prior to the decrement of A0.

psh.{w|l} Sk
Push scalar register

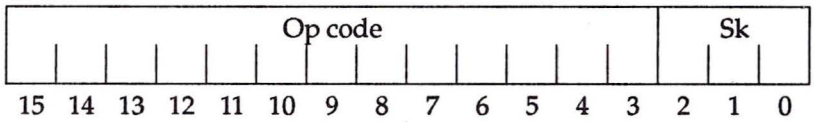
Purpose

To push the contents of a scalar register onto the stack

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
psh.w Sk	ST	7D20	0111110100100	None	Push Sk<31..0> onto the stack
psh.l Sk	ST	7D28	0111110100101	None	Push Sk<63..0> onto the stack

Description

psh.w—After A0 is decremented by 4, the lower 32 bits of scalar register Sk replace the contents of the word of memory addressed by A0.

psh.l—After A0 is decremented by 8, the contents of scalar register Sk replace the contents of the longword of memory addressed by A0.

Operation

```
A0 = A0 - 4;      /* psh.w */
c(A0) = Sk<31..0>;

A0 = A0 - 8;      /* psh.l */
c(A0) = Sk<63..0>;
```

pshea *effa*

Push effective address to stack

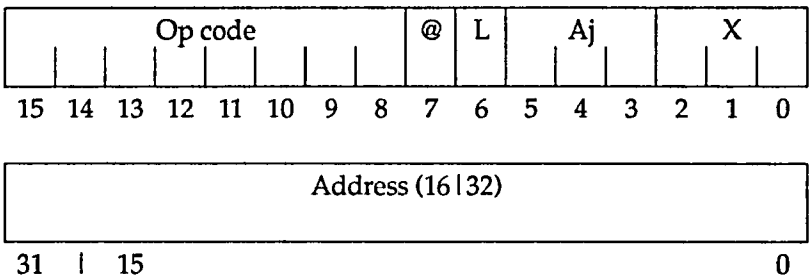
Purpose

To copy an effective address onto the stack

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
pshea *effa*	ST	0D00	00001101	None	Push effective address

Description

The effective address is determined by evaluating the @, L, Aj, and address fields. Address register A0 is decremented by 4 and the effective address replaces the word of memory specified by the contents of A0.

Operation

```
c(A0-4) = effa;
A0 = A0 - 4;
```

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 as an index register are ignored. Refer to the *CONVEX Architecture Reference Manual (C Series)*, Chapter 3, "General registers," for more information on address registers.
4. The X field is unused.

pshr Ak,*effa*

Push address register to resource

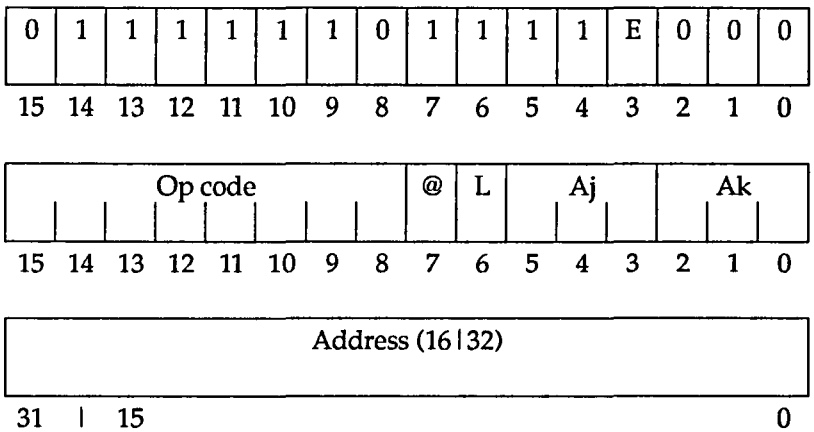
Purpose

To copy the address register onto a resource structure

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
pshr Ak,*effa*	E0	0900	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).

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; }
```

Notes

1. This instruction is atomic.
2. The "valid" byte in the resource structure is ignored.
3. Refer to the *CONVEX Architecture Reference Manual (C Series)*, Chapter 7, "Process structures," for more information on the `pshr` instruction and resource structures.

put.w Ak,*Ceffa*

Put register (address to communication)

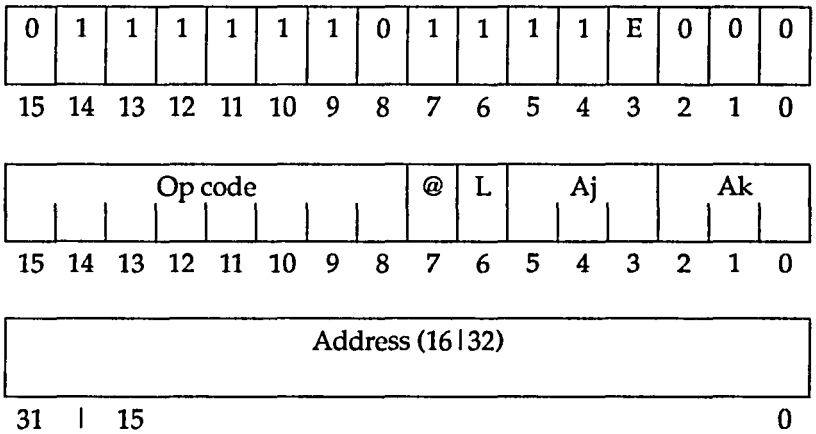
Purpose

To copy the contents of the address register into a communication register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
put.w Ak,*Ceffa*	E0	2E00	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.

Operation

```
c(Ceffa)<31..0> = Ak;
```

Exceptions

Ring violation (invalid communication register address)

Note

The memory dual of this instruction is `putr.w Ak,` *effa*

put.l Sk,*Ceffa*

Put register (scalar to communication)

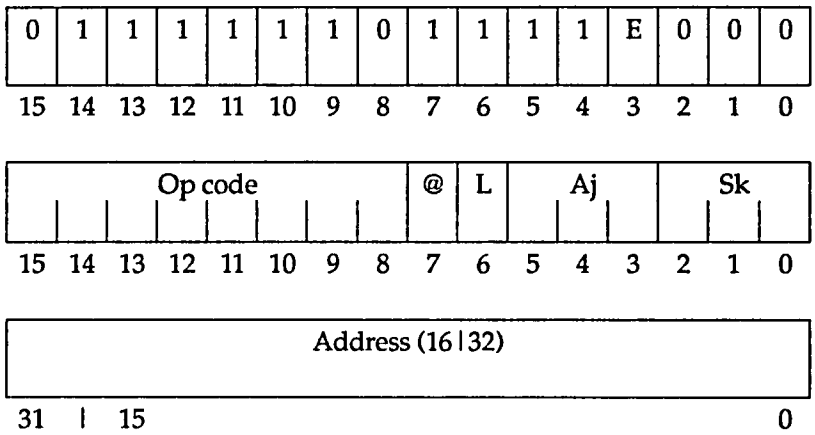
Purpose

To copy the contents of a scalar register into a communication register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
put.l Sk,*Ceffa*	E0	3600	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.

Operation

```
c(Ceffa) = Sk;
```

Exceptions

Ring violation (invalid communication register address)

Note

The memory dual of this instruction is `putr.l Sk,effa`.

putr.w Ak,*effa*

Put register (address to resource)

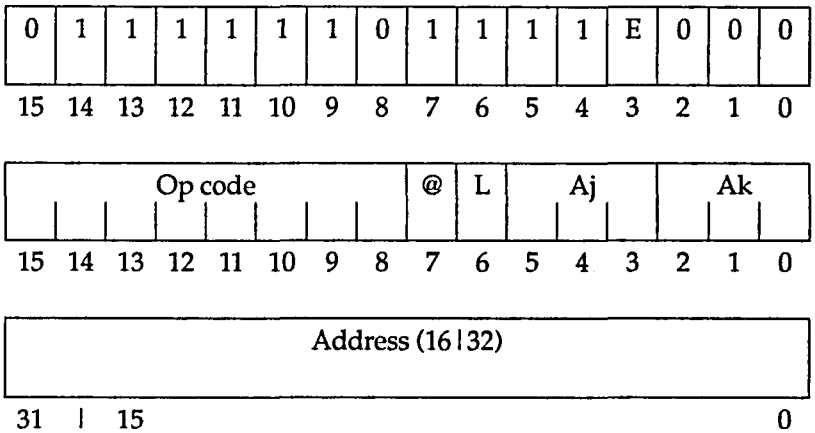
Purpose

To copy the contents of an address register into a resource structure in memory

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
putr.w Ak,*effa*	E0	2000	0010000000	C	Put address/resource

Description

The `putr.w` instruction atomically moves a word of data, bits <31..0> of `c(`*effa*`.data)`, from Ak to a resource structure in memory. If C is returned as 0, `putr.w` is unable to lock the resource structure (the resource structure is in transition). If C is returned as 1, the operation is successful, and the resource structure is unlocked.

Operation

```
if (tas(effa.lock)) {
    c(effa.data) = Ak
    C = 1
    msync
    tac(effa.lock) }
else {
    C = 0 }  /* fail-structure in transition */
```

Exceptions

Ring violation (invalid communication register address)

Notes

1. This instruction is atomic.
2. The communication register dual of this instruction is `put.w Ak,` *Ceffa*.

putr.l Sk,*effa*

Put register (scalar to resource)

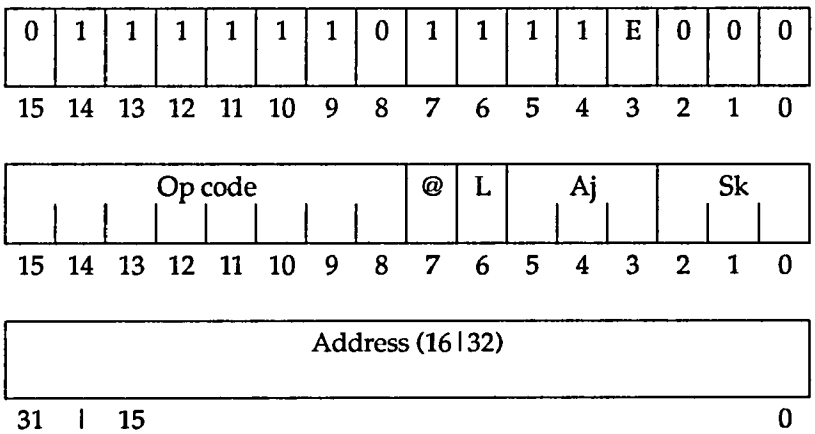
Purpose

To copy the contents of a scalar register into a resource structure in memory

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
putr.l Sk,*effa*	E0	2100	0010000100	SC	Put scalar/resource

Description

The `putr.l` instruction atomically moves a longword of data c(*effa*`.data`) from Sk to a resource structure in memory. If SC is returned as 0, `putr.l` was unable to lock the resource structure (the resource structure was in transition). If SC is returned as 1, the operation was successful, and the resource structure is unlocked.

Operation

```
if (tas(effa.lock)) {
    c(effa.data) = Sk
    SC = 1
    msync
    tac(effa.lock) }
else {
    SC = 0 } /* fail—structure in transition */
```

Exceptions

Ring violation (invalid communication register address)

Notes

1. This instruction is atomic.
2. The communication register dual of this instruction is `put.l Sk,`*Ceffa*.

rcv.w *Ceffa*,Ak

Receive register (communication to address)

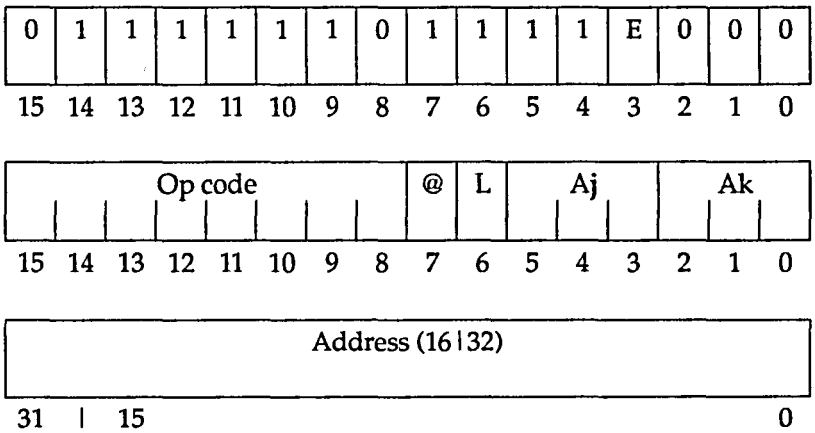
Purpose

To copy the contents of a communication register into an address register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
rcv.w *Ceffa*,Ak	E0	2b00	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, that is, if the lock bit was clear when the operation started.

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

Notes

1. This instruction is atomic.
2. The memory dual of this instruction is `rcvr.w` *effa*`,Ak`.

rcv.l *Ceffa*,Sk

Receive register (communication to scalar)

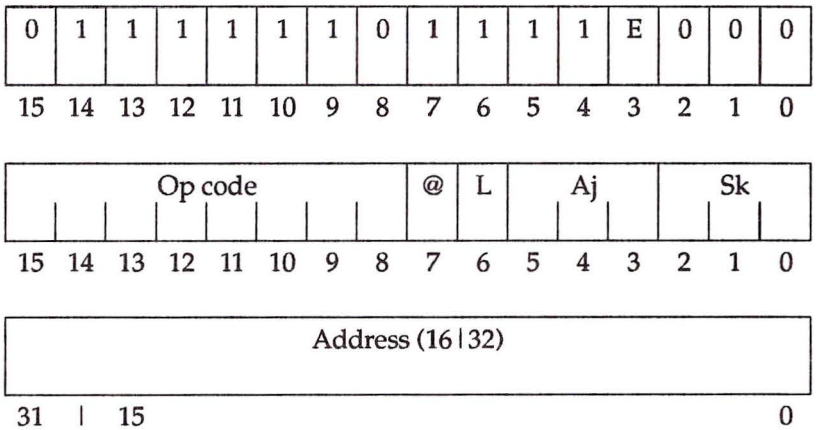
Purpose

To copy the contents of a communication register into a scalar register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
rcv.l *Ceffa*,Sk	E0	3300	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, that is, if the lock bit was clear when the operation started.

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

Notes

1. This instruction is atomic.
2. The memory dual of this instruction is `rcvr.l` *effa*`, Sk.`

rcvr.w *effa*,Ak

Receive register (resource to address)

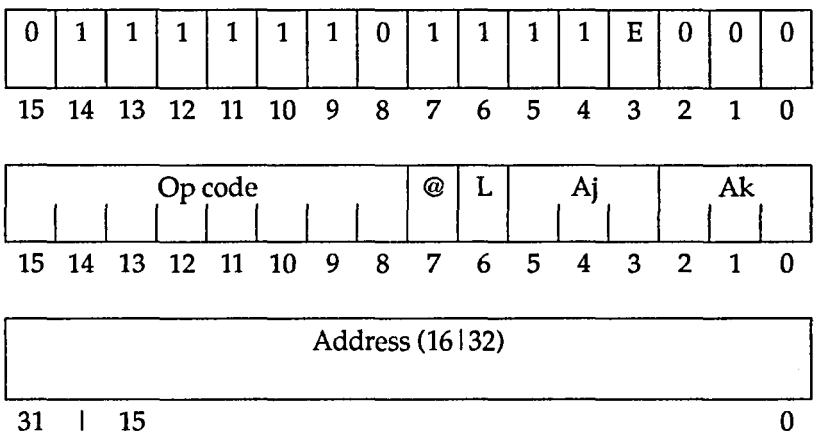
Purpose

To copy the contents of a synchronized resource structure in memory into an address register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
rcvr.w *effa*,Ak	E0	0A00	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, that is, if the lock bit was clear when the operation started.

Operation

```
msync;                              /* not if C3400 */
if (tas(effa.lock)) {
    Ak = c(effa.data);
    if (effa.valid == 0xFF) {
        c(effa.valid) = 0x00;
        C = 1; }
        msync;                      /* only if C3400 */
    else {
        C = 0; }     /* fail—no valid data there */
    msync;                          /* not if C3400 */
    tac(effa.lock); }
else {
    C = 0; }         /* fail—structure in transition */
```

Exceptions

Deadlock exception

Notes

1. This instruction is atomic.
2. The communication dual of this instruction is `rcv.w` C*effa*`,Ak`.

rcvr.l *effa*,Sk

Receive register (resource to scalar)

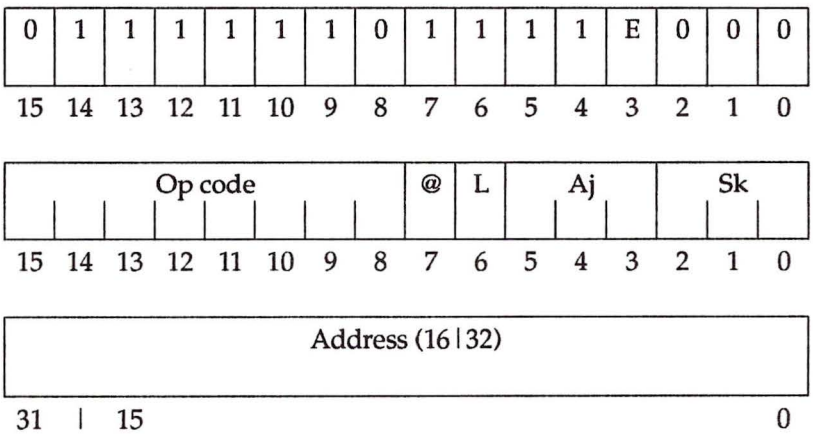
Purpose

To copy the contents of a synchronized resource structure in memory into a scalar register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
rcvr.l *effa*,Sk	E0	0E00	0000111000	SC	Receive scalar register/resource

Description

A word is atomically received from a resource structure into a scalar register. If SC is returned as 0, the resource structure was in transition and was unable to be locked 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 Sk are undefined if the communication register contains no valid data, that is, if the lock bit was clear when the operation started.

Operation

```
msync;                          /* not if C3400 */
if (tas(effa.lock)) {
    Sk = c(effa.data);
    if (effa.valid == 0xFF) {
        c(effa.valid) = 0x00;
        SC = 1;
        msync;                  /* only if C3400 */ }
    else {
        SC = 0; }    /* fail—no valid data there */
    msync;                      /* not if C3400 */
    tac(effa.lock); }
else {
    SC = 0; }        /* fail—structure in transition */
```

Exceptions

Deadlock exception

Notes

1. This instruction is atomic.
2. The communication dual of this instruction is `rcv.l` *Ceffa*`, Sk.`

rtn
Return from subroutine

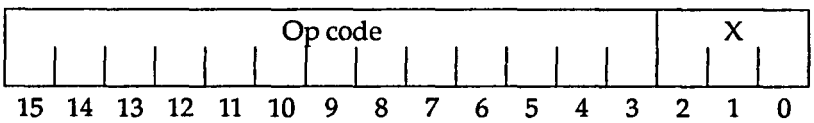
Purpose

To return from a subroutine

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
rtn	ST	7C90	0111110010010	All bits	Return from subroutine call

Description

A subroutine exit is performed. The contents of the FP replace the SP to reestablish 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 4 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:

1. A check for outward ring crossing is made.
2. If the ring crossing is inward, a system exception condition occurs.
   - **C100 Series CPUs**—The stack pointer after the pop is stored in bytes <72..75> of page 0 of the ring containing the `rtn` instruction.
   - **Multiprocessing C Series CPUs**—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.

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)

Notes

1. The PSW is loaded from the current stack frame.
2. The frame length bits, PSW (FRL), in the stack indicate the type of return block saved:

FRL	Return block type
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 popped off 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 Manual (C Series)*, Chapter 7, "Process structures," for more information on process return blocks, stack management, and subroutine calls and returns.
6. The X field is unused.

rtnc
Return from context block

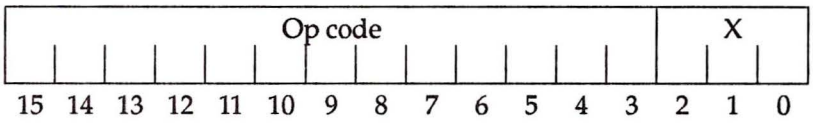
Purpose

To return from a context block

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
rtnc	ST	7CA8	0111110010101	None	Return from a context block

Description

The context block on the ring 0 stack is popped.

C100 Series CPUs—After the pop the new value of the stack pointer (A0) replaces the context stack pointer in the word at byte address `0x36` of ring 0.

Multiprocessing C Series CPUs—After the pop 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.

Operation

```
Processor_State = Context_Block;  /* FRL = 00 */
if (CPU == C200 ) {  /* multiprocessing C Series */ }
    Push A0 on system resource structure;
else {               /* C100 Series */ }
    ring 0, bytes <36..39> = A0; }
A0 = (stack pointer from context block);
```

Exception

Ring violation (privileged instruction)

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 bits, PSW (FRL), are checked. If the FRL bits are not clear (00), then a hard error is generated. Refer to the *CONVEX Architecture Reference Manual (C Series)*, Chapter 12, "Operating system exceptions," 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 Manual (C Series)*, Chapter 7, "Process structures," for more information on process return blocks, stack management, and subrouting calls and returns.
5. The X field is unused.

rtnq
Pop PC and jump

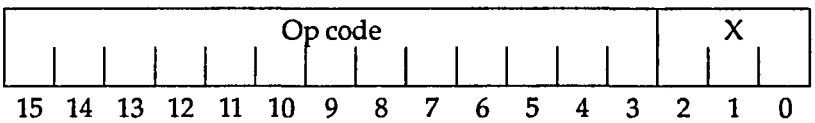
Purpose

To copy the top element of the stack into the program counter (PC) and jump to that address

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
rtnq	ST	7C80	0111110010000	None	Pop the PC and jump

Description

The previously pushed PC value at the top of the stack replaces the present PC (adjusted to stay within the current ring). The stack is adjusted. Execution continues at the instruction referenced by the popped PC.

Operation

```
temp = c(A0);
A0 = A0 + 4;
if (PC<31> == 1) {
    PC<30..0> = temp<30..0>; }
else {
    PC<28..0> = temp<28..0>; }
```

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 unused.

shf #{n|N},Ak
Logical shift (address by immediate)

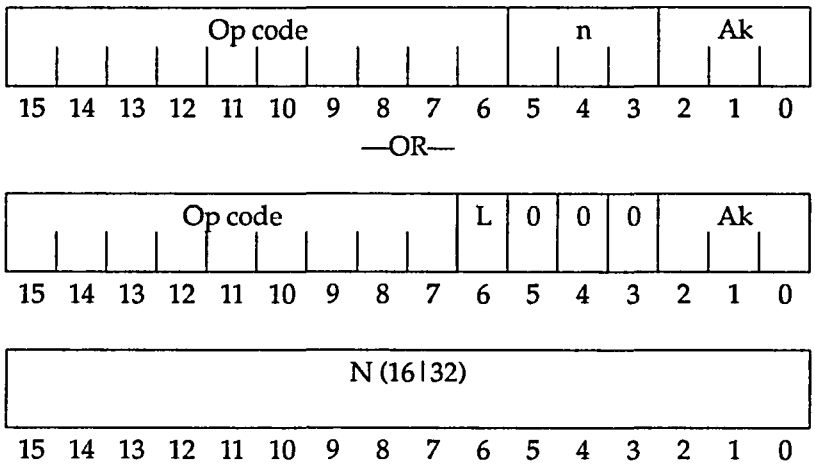
Purpose

To shift logically the contents of an address register by an immediate operand

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
shf #n,Ak	ST	4440	0100010001	None	Logical shift short immediate
shf #N,Ak	ST	1380	000100111	None	Logical shift immediate

Description

The contents of address register Ak are logically shifted left the number of bits specified by either the short immediate operand (#n) or the sign-extended bits <7..0> of the long immediate operand (#N, length indicated by L) and replace the contents of Ak.
Logical right shifts occur only when the sign-extended long immediate operand is negative. Logical shifts always zero-fill.
Sign extension does not occur for the 3 bits of the short immediate form.

Operation

```
if (Immediate >= 0) {
    Ak = Ak << Immediate<7..0>;          /* shift left */ }
else {
    Ak = Ak >> -Immediate<7..0>; } /* shift right */
```

Note

Use multiplies or divides to implement arithmetic shifts.

shf #N,Sk

Logical shift (scalar by immediate)

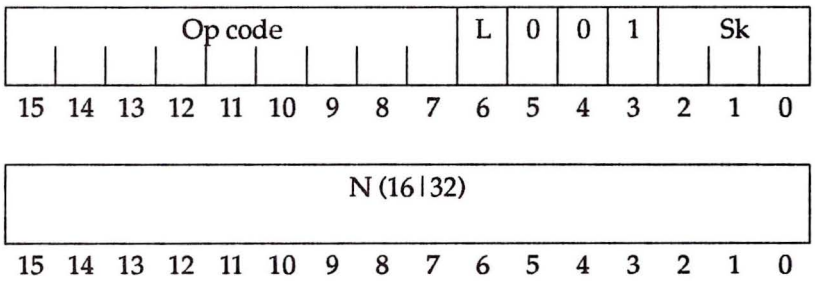
Purpose

To shift logically the contents of a scalar register by an immediate operand

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
shf #N,Sk	ST	1388	000100111	None	Shift scalar/immediate

Description

The contents of scalar register Sk are logically shifted left the number of bits specified by the sign-extended bits <7..0> of the long immediate operand (#N) and replace the contents of Sk.

Logical right shifts occur only when the sign-extended long immediate operand is negative. Logical shifts always zero-fill.

Operation

```
if (Immediate<7..0> >= 0) {
    Sk = Sk << Immediate<7..0>;          /* shift left */ }
else {
    Sk = Sk >> -Immediate<7..0>; } /* shift right */
```

Note

Use multiplies and divides to implement arithmetic shifts.

shf.w #N,Sk

Logical shift (scalar by immediate)

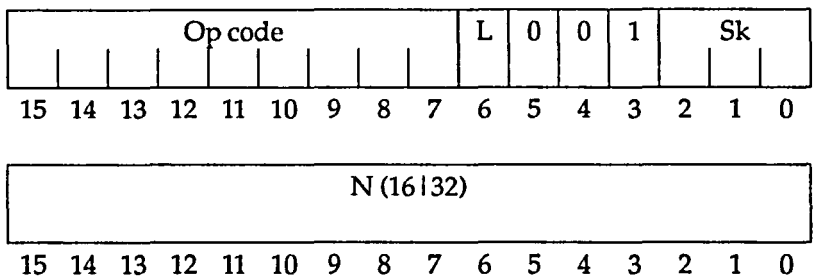
Purpose

To shift logically the contents of the low order word of a scalar register by an immediate operand

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
shf.w #N,Sk	ST	1980	000110011	None	Shift scalar word/immediate

Description

The contents of scalar register Sk are logically shifted left the number of bits specified by the sign-extended bits <7..0> of the long immediate operand (#N) and replace the contents of Sk.

Logical right shifts occur only when the sign-extended long immediate operand is negative. Logical shifts always zero-fill.

Operation

```
if (Immediate<7..0> > 0) {
    Sk<31..0> = Sk<31..0> << Immediate<7..0>;
                                    /* shift left */ }
else {
    Sk<31..0> = Sk<31..0> >> -Immediate<7..0>; }
                                    /* shift right */
```

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 op code in a C1 or C120 processor will result in an unimplemented instruction trap.

shf Aj,Ak
Logical shift (address by address)

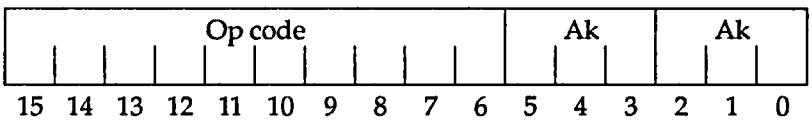
Purpose

To shift logically the contents of an address register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
shf Aj,Ak	ST	5040	0101000001	None	Shift an address

Description

The contents of address register Ak are logically shifted left the number of bits specified by the sign-extended bits <7..0> of Aj and replace the contents of Ak.

Logical right shifts occur only when the sign-extension is negative. Logical shifts always zero-fill.

Operation

```
if (Aj<7..0> > 0) {
    Ak = Ak << Aj<7..0>;    /* shift left */ }
else {
    Ak = Ak >> -Aj<7..0>; } /* shift right */
```

Note

Use multiplies or divides to implement arithmetic shifts.

shf Sj,Sk

Logical shift (scalar by scalar)

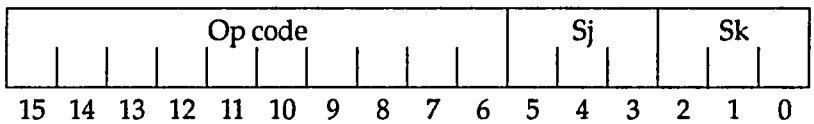
Purpose

To shift logically the contents of a scalar register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
shf Sj,Sk	ST	5140	0101000101	None	Shift a scalar

Description

The contents of scalar register Sk are logically shifted left the number of bits specified by the sign-extended bits <7..0> of Sj and replace the contents of Sk.

Logical right shifts occur only when the sign-extension is negative. Logical shifts always zero-fill.

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 */
```

Notes

Use multiplies or divides to implement arithmetic shifts.

shf.w Sj,Sk

Logical shift (scalar word by scalar)

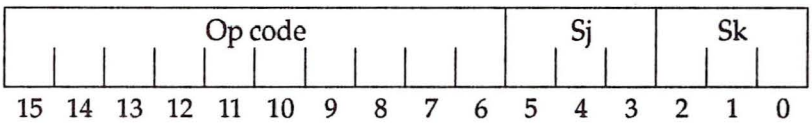
Purpose

To shift logically the contents of the lower order word of a scalar register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
shf.w Sj,Sk	E0	4440	0100010001	None	Shift a scalar word

Description

The contents of scalar register Sk are logically shifted left the number of bits specified by the sign-extended bits <7..0> of Sj and replace the contents of Sk.

Logical right shifts occur only when the sign-extension is negative. Logical shifts always zero-fill.

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 */
```

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 by scalar)

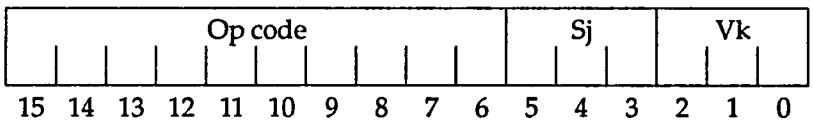
Purpose

To shift logically the elements of a vector register by the contents of a scalar register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
shf Sj,Vk	ST	6300	0110001100	None	Shift a vector accumulator

Description

The contents of each of the first VL elements of vector register Vk are logically shifted left the number of bits specified by the sign-extended bits <7..0> of Sj and replace the contents of Vk[i], where i=1,VL.

Logical right shifts occur only when the sign-extension is negative. Logical shifts always zero-fill.

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 */
```

Note

Use multiplies or divides to implement arithmetic shifts.

shf.{t|f} Sj,Vk

Logical shift (vector by scalar) (masked)

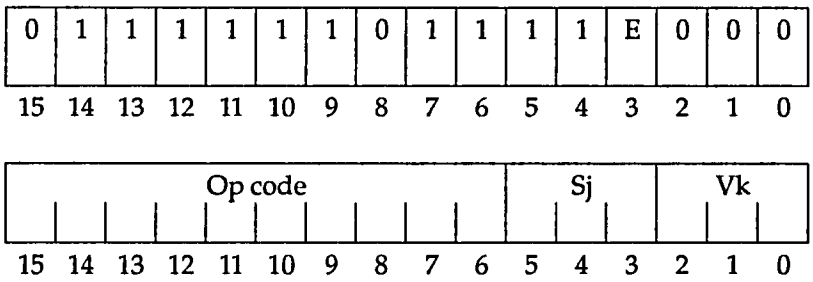
Purpose

To logically shift the contents of a vector register by a scalar register under control of the vector merge (VM) registerts

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
shf.t Sj,Vk	E1	6300	0110001100	None	Shift vector/scalar (VM)
shf.f Sj,Vk	E0	6300	0110001100	None	Shift vector/scalar (IVM)

Description

The contents of each of the first VL elements of vector register Vk are logically shifted left the number of bits specified by the sign-extended bits <7..0> of Sj and replace the contents of Vk[i], where i=1,VL, only if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

Logical right shifts occur only when the sign-extension is negative. Logical shifts always zero-fill.

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.0 = 0) {
                    Vk[a] = Vk[a] < Sj.0; /* left */ }
                else {
                    Vk[a] = Vk[a] > -Sj.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.0 = 0) {
                    Vk[a] = Vk[a] < -Sj.0; /* left */ }
                else {
                    Vk[a] = Vk[a] > Sj.0; /* right */ } } }
                            /* end of for loop */
        break; } /* end of switch */
```

Note

Use multiplies or divides to implement arithmetic shifts.

shf Vi,Sj,Vk

Logical shift (vector by scalar)

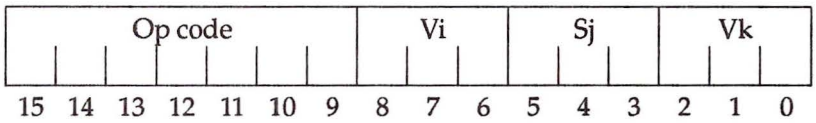
Purpose

To logically shift the contents of a vector register by a scalar register and store the result in another vector register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
shf Vi,Sj,Vk	ST	AE00	1010111	None	Shift a vector accumulator

Description

The contents of each of the first VL elements of vector register Vi are logically shifted left the number of bits specified by the sign-extended bits <7..0> of Sj and replace the contents of Vk[i], where i=1,VL.

Logical right shifts occur only when the sign-extension is negative. Logical shifts always zero-fill.

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 */ } }
```

Note

Use multiplies or divides to implement arithmetic shifts.

shf.{t|f} Vi,Sj,Vk

Logical shift (vector by scalar) (masked)

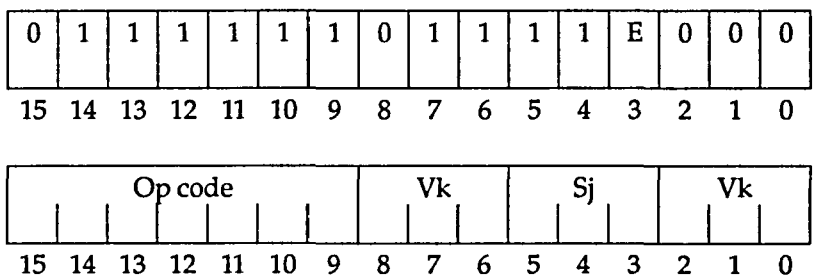
Purpose

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

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
shf.t Vi,Sj,Vk	E1	AE00	1010111	None	Shift vector/scalar (VM)
shf.f Vi,Sj,Vk	E0	AE00	1010111	None	Shift vector/scalar (IVM)

shf.{t|f} Vi,Sj,Vk

Description

The contents of each of the first VL elements of vector register Vi are logically shifted left the number of bits specified by the sign-extended bits <7..0> of Sj and replace the contents of Vk[i], where i=1,VL, only if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

Logical right shifts occur only when the sign-extension is negative. Logical shifts always zero-fill.

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.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.0 = 0) {
                    Vk[a] = Vi[a] << Sj.0; } /* left */
                else {
                    Vk[a] = Vi[a] >> -Sj.0; } } } /* right */
                                /* end of for loop */
        break;  } /* end of switch */
```

Note

Use multiplies or divides to implement arithmetic shifts.

shf Vi,Vj,Vk

Logical shift (vector by vector)

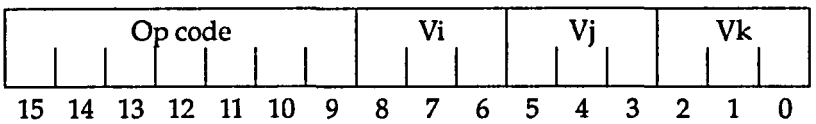
Purpose

To logically shift the contents of one vector register by another vector register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
shf Vi,Vj,Vk	ST	A600	1010011	None	Shift vector/vector

Description

The contents of each of the first VL elements of vector register Vi are logically shifted left the number of bits specified by the sign-extended bits <7..0> of the corresponding elements of Vj and replace the contents of Vk[i], where i=1,VL.

Logical right shifts occur only when the sign-extension is negative. Logical shifts always zero-fill.

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 */
```

Note

Use multiplies or divides to implement arithmetic shifts.

shf.{t|f} Vi,Vj,Vk

Logical shift (vector by vector) (masked)

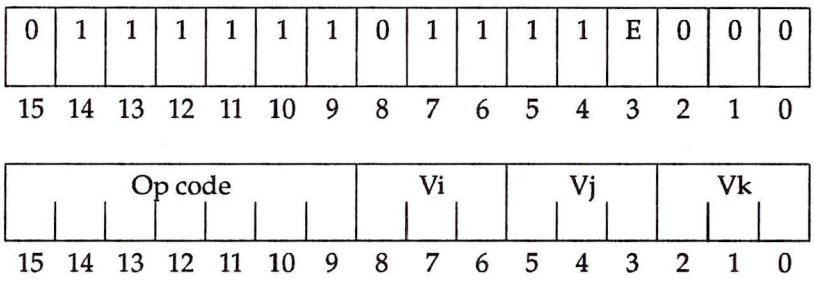
Purpose

To logically shift the contents of one vector register by another vector register under control of the Vector Merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
shf.t Vi,Vj,Vk	E1	A600	1010011	None	Shift vector/vector (VM)
shf.f Vi,Vj,Vk	E0	A600	1010011	None	Shift vector/vector (!VM)

Description

The contents of each of the first VL elements of vector register Vi are logically shifted left the number of bits specified by the sign-extended bits <7..0> of the corresponding elements of Vj and replace the contents of Vk[i], where i=1,VL, only if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

Logical right shifts occur only when the sign-extension is negative. Logical shifts always zero-fill.

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;  } /* end of switch */
```

Note

Use multiplies or divides to implement arithmetic shifts.

sin.{s|d} Sk
Sine

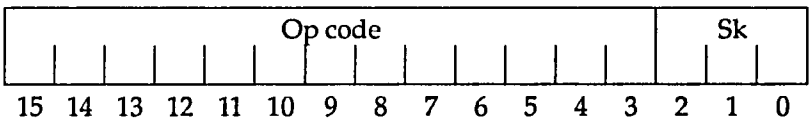
Purpose

To compute take the trigonometric sine of the contents of a scalar register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
sin.s Sk	ST	7CC0	0111110011000	RO,FIN, IEC	Sine of a single precision number
sin.d Sk	ST	7CC8	0111110011001	RO,FIN, IEC	Sine of a double precision number

Description

The sine of the contents of Sk replaces the contents of Sk.

Operation

```
Sk = sin(Sk)
```

Exception

s|d	Reserved operand Floating intrinsic error

Notes

1. The input operand is interpreted as an angle in radians.
2. Intrinsic traps go through the same trap handler as other PSW arithmetic traps (RO, FDZ, UN, etc.). If PSW (FUE) and/or PSW (FE) are set and intrinsic traps, PSW (INE), are cleared, 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 Manual (C Series)*, "Processor status word" section in Chapter 3, "General registers," for more information on the PSW (IEC) error codes and arithmetic trap conditions.

snd.w Ak,*Ceffa*

Send register (address to communication)

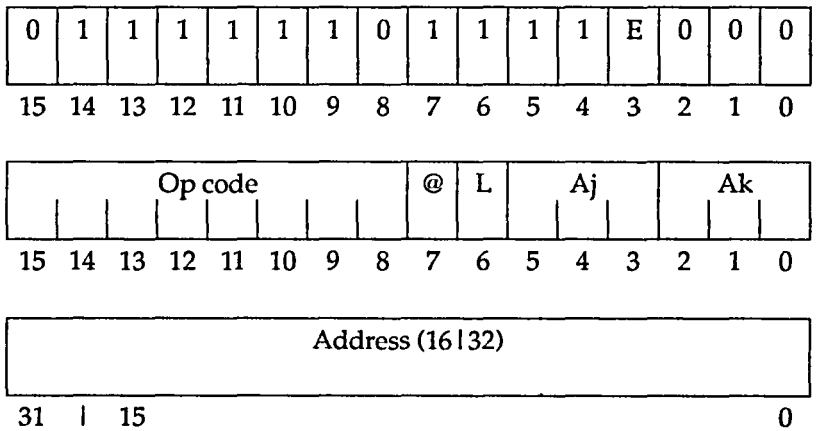
Purpose

To copy the contents of an address register to a communication register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
snd.w Ak,*Ceffa*	E0	2F00	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.

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

Notes

1. This instruction is atomic.
2. The memory dual of this instruction is `sndr.w Ak,` *effa*.

snd.l Sk,*Ceffa*

Send register (scalar to communication)

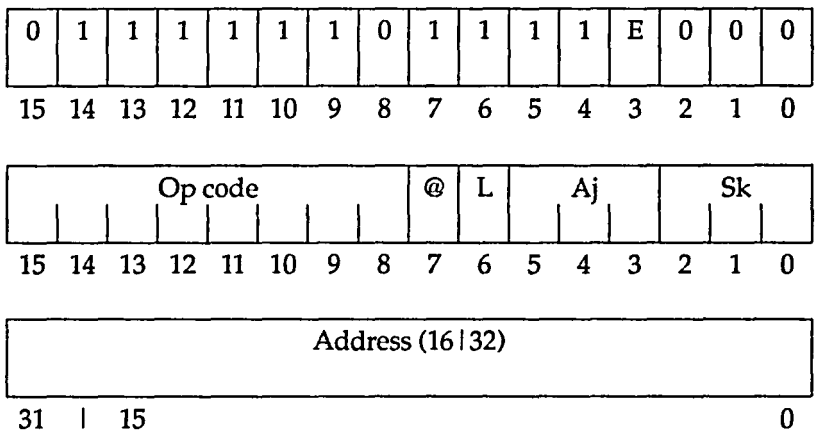
Purpose

To copy the contents of a scalar register to a communication register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
snd.l Sk,*Ceffa*	E0	3700	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 a "fail" status (SC=0) is returned. If the lock bit is clear, a word of data is moved from Sk to c (*Ceffa*), the lock bit is set, and a "success" status (SC=1) is returned.

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

Notes

1. This instruction is atomic.
2. The memory dual of this instruction is `sndr.l Sk,` *effa*.

sndr.w Ak,*effa*

Send register (address to resource)

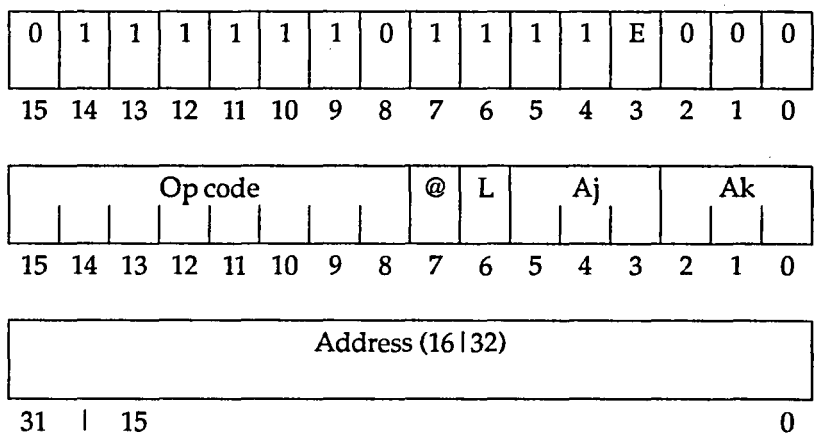
Purpose

To copy the contents of an address register to a synchronized resource structure in memory

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
sndr.w Ak,*effa*	E0	0C00	00001100	C	Send address register/resource

sndr.w Ak,*effa*

Description

A word is atomically sent from an address register to a resource structure if it previously contained no valid data. If C is returned as 0, the resource structure could not be locked 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.

Operation

```
msync;                                  /* not if C3400 */
if (tas(effa.lock)) {
    if (effa.valid == 0x00) {
        c(effa.data) = Ak;
        c(effa.valid) = 0xFF;
        C = 1;
        msync; }                        /* only if C3400 */
    else {
        C = 0; }     /* fail—already valid data there */
    msync;                              /* not if C3400 */
    tac(effa.lock); }
else {
    C = 0; }         /* fail—structure in transition */
```

Exception

Deadlock exception

Notes

1. This instruction is atomic.
2. The communication register dual of this instruction is `snd.w Ak,`*Ceffa*.

sndr.l Sk,*effa*

Send register (scalar to resource)

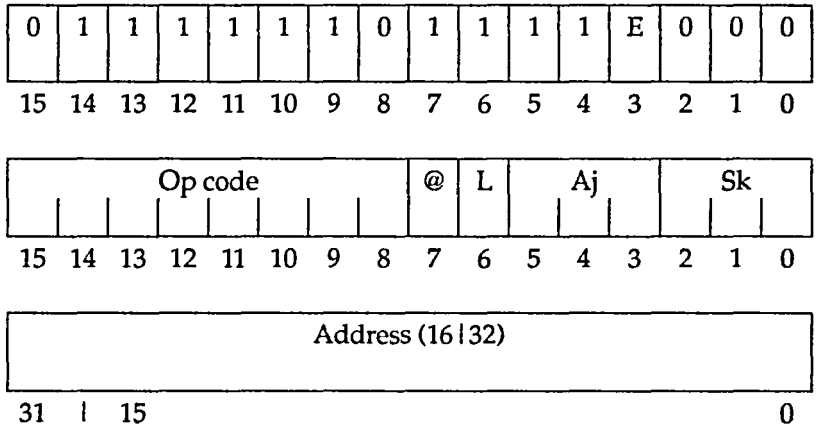
Purpose

To copy the contents of a scalar register to a synchronized long resource structure in memory

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
sndr.l Sk,*effa*	E0	0D00	0000110100	SC	Send scalar register/resource

sndr.l Sk,*effa*

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.

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 */
```

Exception

Deadlock exception

Notes

1. This instruction is atomic.
2. The communication register dual of this instruction is `snd.l Sk,`*Ceffa*.

spawn *effa*,Ak
Spawn a fork

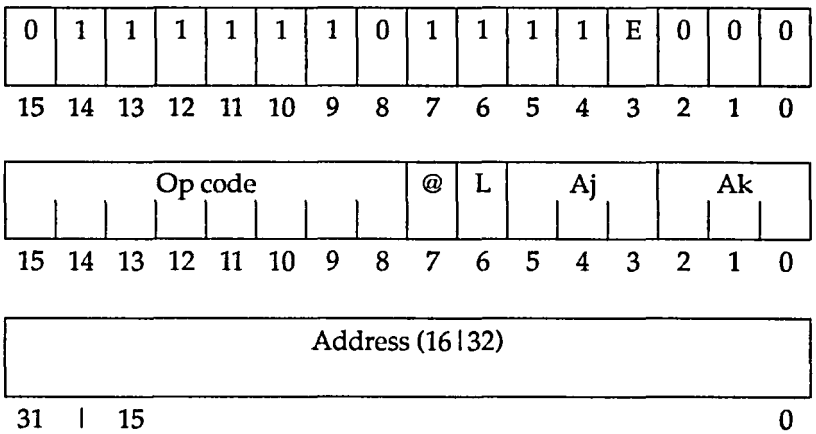
Purpose

To post the need for as many CPUs as possible to join in a process computation

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
spawn *effa*,Ak	E0	0500	00000101	C	Spawn a fork

spawn *effa*,Ak

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.

The `snd` to `forklck` loads FP into `fork.FP` and AP into `fork.AP`. The communication registers are loaded only if `forklck` is zero, that is, there is no currently posted fork. The `snd` to `forkposted` loads the constant "SPAWNED" into `fork.type` and Ak into `fork.SP`.

Address carry (C) is set to 1 if the fork is successfully posted, otherwise C is cleared to 0.

Operation

```
if (C = snd.l(forklck, FP::AP)) {
                        /* C = 1 if snd.l() succeeds */
    fork.PC  = a;
    fork.PSW = PSW;
    fork.source_PC = PC;
    snd.l(forkposted, SPAWNED::Ak); }
```

Notes

1. A `spawn` will potentially add multiple processors to the process. Use `pfork` to add a single processor.
2. Once a fork is posted, it must either be taken by a CPU or cleared with `cfork` before another fork can be successfully posted.
3. Refer to the *CONVEX Architecture Reference Manual (C Series)*, Chapter 9, "Multiprocessor management," for more information on `pfork` and forking operations.

sqrt.{s|d} Sk
Square root (scalar)

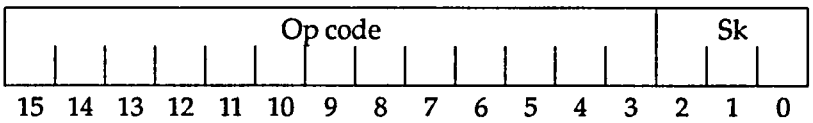
Purpose

To compute the square root of the contents of a scalar register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
sqrt.s Sk	ST	7DD0	0111110111010	RO,FIN, IEC	Square root of a single float
sqrt.d Sk	ST	7DD8	0111110111011	RO,FIN, IEC	Square root of a double float

Description

The square root of the contents of Sk replaces the contents of Sk.

Operation

```
Sk = sqrt(Sk);
```

Exception

s|d	Reserved operand Floating intrinsic error

Notes

1. Intrinsic traps go through the same trap handler as other arithmetic traps (PSW (RO), PSW (FDZ), PSW (UN), etc.). If PSW (FUE) and/or PSW (FE) are set and and intrinsic traps, PSW (INE), are cleared, 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 Manual (C Series)*, "Processor status word" section in Chapter 3, "General registers," for more information on the PSW (IEC) error codes and arithmetic trap conditions.

sqrt.{s|d} Vj,Vk
Square root (vector)

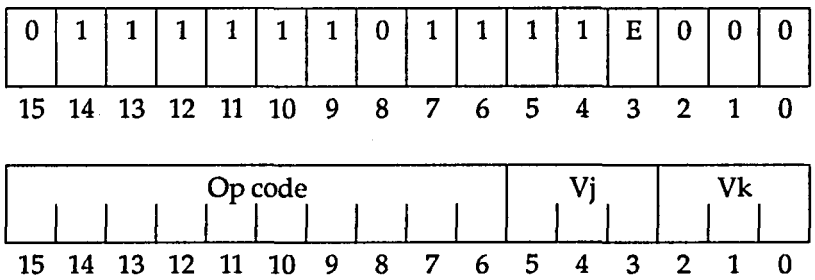
Purpose

To compute the square root of the contents a vector register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
sqrt.s Vj,Vk	E0	5D00	0101110100	RO,FIN, IEC	Square root single vector/vector
sqrt.d Vj,Vk	E0	5D40	0101110101	RO,FIN, IEC	Square root double vector/vector

Description

The contents of each of the first VL elements of vector register Vk are replaced by the square root of the corresponding element of vector register Vj. If any of the first VL elements of Vj is negative, PSW (FIN) is set, 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.

Operation

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

Exception

s | d	Reserved operand
	Floating intrinsic error

Notes

1. Intrinsic traps go through the same trap handler as other arithmetic traps (PSW (RO), PSW (FDZ), PSW (UN), etc.). If PSW (FUE) and/or PSW (FE) are set, and intrinsic traps, PSW (INE), are not, 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 Manual (C Series)*, "Processor status word" section in Chapter 3, "General registers," for more information on the PSW (IEC) error codes and arithmetic trap conditions.

sqrt.{s|d}.{t|f} Vj,Vk
Square root (vector) (masked)

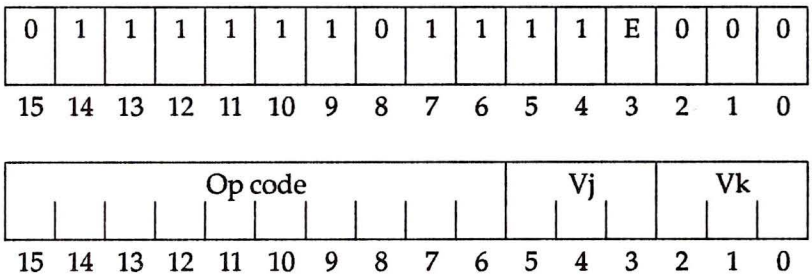
Purpose

To compute the square root of the contents of a vector register under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
sqrt.s.t Vj,Vk	E1	5F00	0101111100	RO,FIN, IEC	Square root single (VM)
sqrt.s.f Vj,Vk	E0	5F00	0101111100	RO,FIN, IEC	Square root single (!VM)
sqrt.d.t Vj,Vk	E1	5F40	0101111101	RO,FIN, IEC	Square root double (VM)
sqrt.d.f Vj,Vk	E0	5F40	0101111101	RO,FIN, IEC	Square Root double (!VM)

Description

The contents of each of the first VL elements of vector register Vk are replaced by the square root of the corresponding element of vector register Vj, if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

If any of the first VL elements of Vj that are enabled for the operation by VM is negative, PSW (FIN) is set, 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 the corresponding element of Vj.

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; } /* end of switch */
```

Exception

`s|d`	Reserved operand Floating intrinsic error

Notes

1. Intrinsic traps go through the same trap handler as other arithmetic traps (PSW (RO), PSW (FDZ), PSW (UN), etc.). If PSW (FUE) and/or PSW (FE) are set and intrinsic traps, PSW (INE), are cleared, 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 Manual (C Series)*, "Processor status word" section in Chapter 3 "General registers," for more information on the PSW (IEC) error codes and arithmetic trap conditions.

st.{b|h|w} Ak,*effa*

Store register (address to memory)

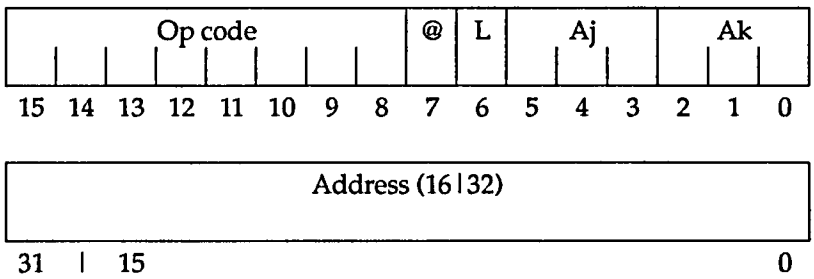
Purpose

To copy the contents of an address register into memory

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
st.b Ak,*effa*	ST	2C00	00101100	None	Store address register byte
st.h Ak,*effa*	ST	2D00	00101100	None	Store address register halfword
st.w Ak,*effa*	ST	2E00	00101110	None	Store address register word

Description

The contents of the address register Ak replace the contents of the memory location referenced by the effective address.

Operation

```
c(effa) = Ak;
```

Note

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

st.{b|h|w|l|s|d} Sk,*effa*

Store register (scalar to memory)

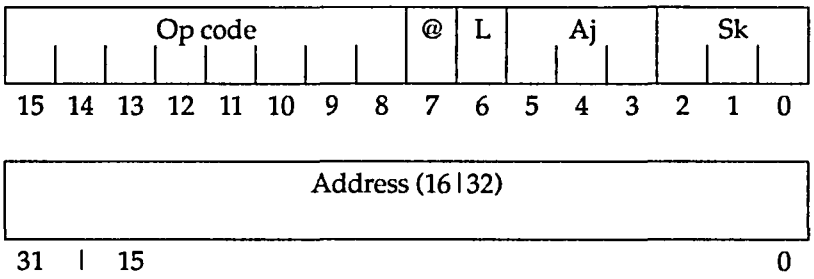
Purpose

To copy the contents of a scalar register into memory

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
st.b Sk,*effa*	ST	3400	00110100	None	Store scalar byte
st.h Sk,*effa*	ST	3500	00110100	None	Store scalar halfword
st.w Sk,*effa*	ST	3600	00110100	None	Store scalar word
st.l Sk,*effa*	ST	3700	00110111	None	Store scalar longword
st.s Sk,*effa*	ST	3600	00110110	None	Store scalar single float
st.d Sk,*effa*	ST	3700	00110111l	None	Store scalar double float

Description

The contents of scalar register Sk replace the contents of the memory location referenced by the effective address.

Operation

```
c(effa) = Sk;
```

Notes

1. Single precision floating point data and 32-bit integer data occupy the same bit positions within a scalar register.
2. The `.s` and `.d` forms rename the `.w` and `.l` forms, respectively, for convenience.

st.{b|h|w|l|s|d} Vk,*effa*

Store register (vector to memory)

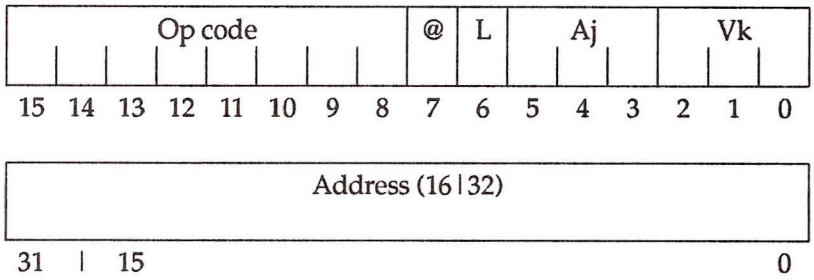
Purpose

To copy the elements of a vector register into memory

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
st.b Vk,*effa*	ST	3C00	001111000	None	Store vector byte
st.h Vk,*effa*	ST	3D00	001111010	None	Store vector halfword
st.w Vk,*effa*	ST	3E00	001111100	None	Store vector word
st.l Vk,*effa*	ST	3F00	001111110	None	Store vector longword
st.s Vk,*effa*	ST	3E00	001111100	None	Store vector single float
st.d Vk,*effa*	ST	3F00	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.

Operation

```
temp = effa
for (a = 0; a < VL; a++) {
    c(temp) = Vk[a];
    temp = temp + VS; }
```

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 register (vector to memory) (masked)

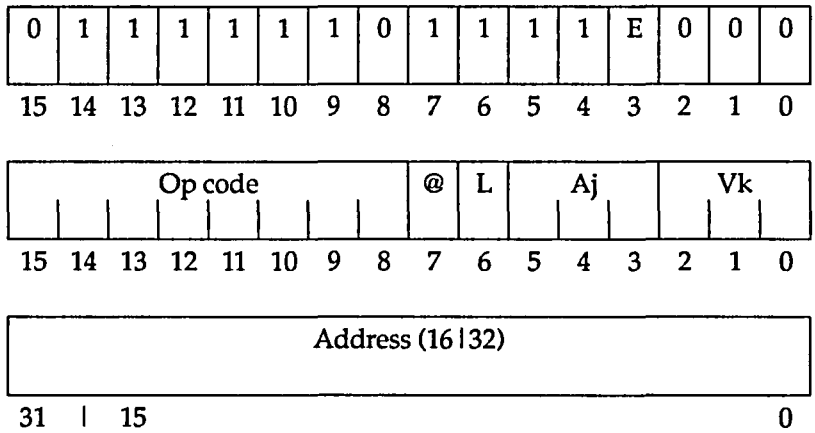
Purpose

To copy a vector from a vector register under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
st.b.t Vk,*effa*	E1	3C00	001111000	None	Store vector byte (VM)
st.b.f Vk,*effa*	E0	3C00	001111000	None	Store vector byte (IVM)
st.h.t Vk,*effa*	E1	3D00	001111010	None	Store vector halfword (VM)
st.h.f Vk,*effa*	E0	3D00	001111010	None	Store vector halfword (IVM)
st.w.t Vk,*effa*	E1	3E00	001111100	None	Store vector word (VM)
st.w.f Vk,*effa*	E0	3E00	001111100	None	Store vector word (IVM)
st.l.t Vk,*effa*	E1	3F00	001111110	None	Store vector longword (VM)
st.l.f Vk,*effa*	E0	3F00	001111110	None	Store vector longword (IVM)
st.s.t Vk,*effa*	E1	3E00	001111100	None	Store vector single float (VM)
st.s.f Vk,*effa*	E0	3E00	001111100	None	Store vector single float (IVM)
st.d.t Vk,*effa*	E1	3F00	001111110	None	Store vector double float (VM)
st.d.f Vk,*effa*	E0	3F00	001111110	None	Store vector double float (IVM)

st.{b|h|w|l|s|d}.{t|f} Vk,*effa*

Description

VL elements of vector Vj replace VL memory elements if the corresponding VM bit is set (for `.t`) or 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.

Operation

```
temp = effa;
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; } /* end of switch */
```

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 registers (VS and VL to memory)

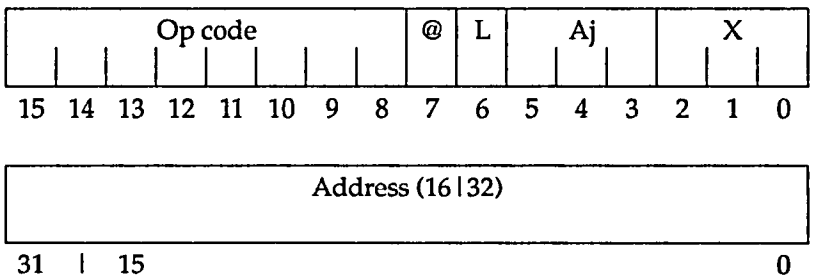
Purpose

To store the contents of the vector stride (VS) register and the vector length (VL) register into memory

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
st.l VLS,*effa*	ST	0E00	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 4 more than the effective address.

Operation

```
c(effa)<63..32> = VS;
c(effa)<31..0> = VL;
```

Note

The X field is unused.

st.x VM,*effa*

Store register (VM to memory)

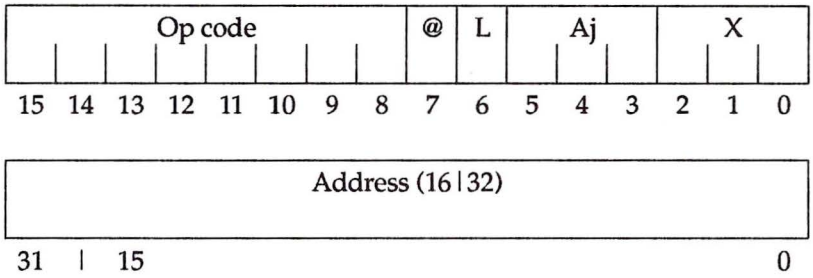
Purpose

To store the contents of the vector merge (VM) register into memory

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
st.x VM,*effa*	ST	0F00	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.

Operation

```
c(effa)<127..0> = VM
```

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 unused.

stcmr Ak,*effa*
Store registers (communication set)

Purpose

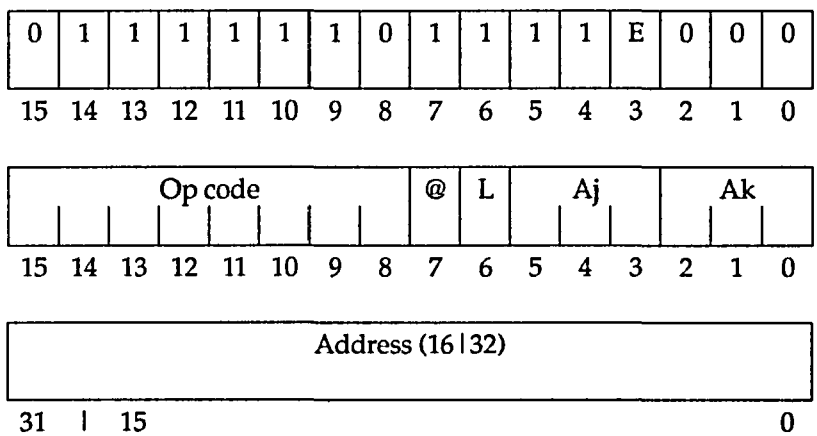
To store a specified set of communication registers in memory

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 Manual (C Series)*, Chapter 6, "Communication registers," for more information concerning the `stcmr/ldcmr` memory map. If the communication registers have not been modified since last loaded via the `ldcmr` instruction, the communication registers are not stored into memory.

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
stcmr Ak,*effa*	E0	0700	0000011100	None	Store communication registers

stcmr Ak,*effa*

Description

The communication register set index contained in Ak specifies a communication register set that is stored in memory.

Operation

```
/* For the CIR specified in Ak: */
/* assume current CIR = Ak */
Update CPU timer register for current ring;
if (ring 0 CMR modified bit == 1 ) {
    Store the hardware CMRs;
    Store the hardware lock bits;
    Store the ring 0 CMRs;
    Store the ring 0 lock bits; }
if (ring 4 CMR modified bit == 1 ) {
    Store the ring 4 CMRs;
    Store the ring 4 lock bits; }
ring 0 CMR valid bit = ring 0 CMR modified bit;
ring 4 CMR valid bit = ring 4 CMR modified bit;
```

Exception

Ring violation (privileged instruction)

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 register (scalar to memory) (extended)

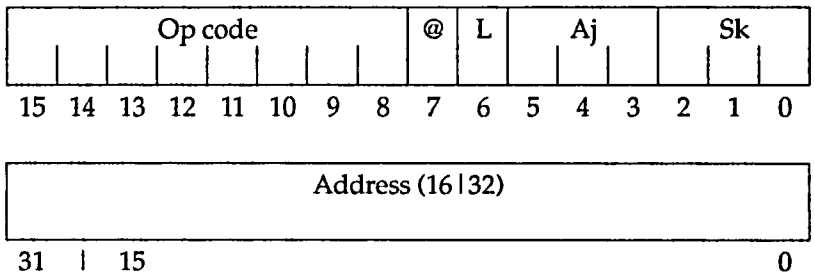
Purpose

To store the contents of a scalar register repetitively into memory

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
ste.b Sk,*effa*	ST	2400	001001000	None	Store an extended scalar byte
ste.h Sk,*effa*	ST	2500	001001010	None	Store an extended scalar halfword
ste.w Sk,*effa*	ST	2600	001001100	None	Store an extended scalar word
ste.l Sk,*effa*	ST	2700	001001110	None	Store an extended scalar longword
ste.s Sk,*effa*	ST	2600	001001100	None	Store an extended scalar single float
ste.d Sk,*effa*	ST	2700	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.

Operation

```
temp = effa;
for (a = 0; a < VL; a++) {
    c<temp> = Sk;
    temp = temp + VS; }
```

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.

ste.{b|h|w|l|s|d}.{t|f} Sk,*effa*

Store register (scalar to memory) (extended) (masked)

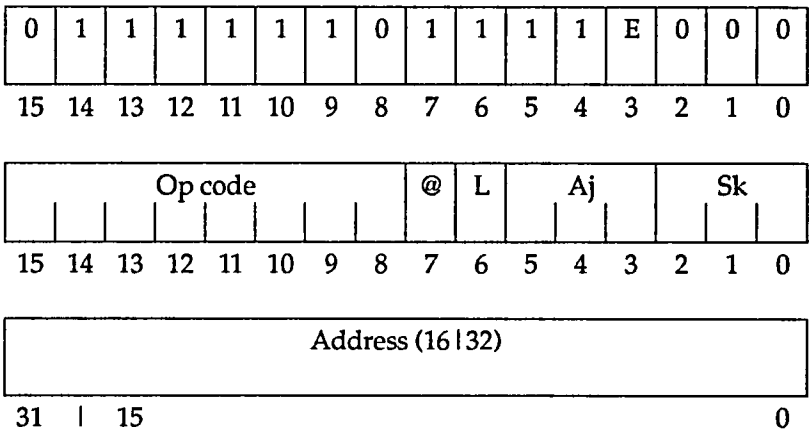
Purpose

To store a scalar register repetitively into memory under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
ste.b.t Sk,*effa*	E1	2400	001001000	None	Store extended scalar byte (VM)
ste.b.f Sk,*effa*	E0	2400	001001000	None	Store extended scalar byte (!VM)
ste.h.t Sk,*effa*	E1	2500	001001010	None	Store extended scalar halfword (VM)
ste.h.f Sk,*effa*	E0	500	001001010	None	Store extended scalar halfword (!VM)
ste.w.t Sk,*effa*	E1	2600	001001100	None	Store extended scalar word (VM)
ste.w.f Sk,*effa*	E0	2600	001001100	None	Store extended scalar word (!VM)
ste.l.t Sk,*effa*	E1	2700	001001110	None	Store extended scalar longword (VM)
ste.l.f Sk,*effa*	E0	2700	001001110	None	Store extended scalar longword (!VM)
ste.s.t Sk,*effa*	E1	2600	001001100	None	Store extended scalar single (VM)
ste.s.f Sk,*effa*	E0	2600	001001100	None	Store extended scalar single (!VM)
ste.d.t Sk,*effa*	E1	2700	001001110	None	Store extended scalar double (VM)
ste.d.f Sk,*effa*	E0	2700	001001110	None	Store extended scalar double (!VM)

ste.{b|h|w|l|s|d}.{t|f} Sk,*effa*

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 (for `.t`) or clear (for `.f`). The signed value of VS is the distance in bytes.

Operation

```
temp = effa;
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; } /* end of switch */
```

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.

stvi.{b|h|w|l|s|d} Sk,Vj
Store register (scalar with vector index)

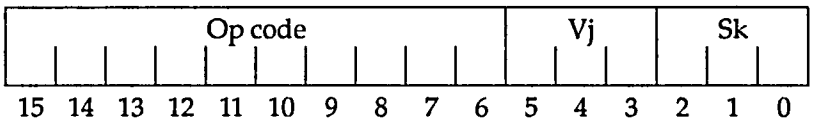
Purpose

To store the contents of a scalar register using the contents of a vector register as a set of indices

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
stvi.b Sk,Vj	ST	7B00	0111101100	None	Scalar index store vector byte
stvi.h Sk,Vj	ST	7B40	0111101101	None	Scalar index store vector halfword
stvi.w Sk,Vj	ST	7B80	0111101110	None	Scalar index store vector word
stvi.l Sk,Vj	ST	7BC0	0111101111	None	Scalar index store vector longword
stvi.s Sk,Vj	ST	7B80	0111101110	None	Scalar index store vector single float
stvi.d Sk,Vj	ST	7BC0	0111101111	None	Scalar index store vector double float

Description

The lower 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.

Operation

```
for (a = 0; a < VL; a++) {
    c(Vj[a]<31..0> + A5) = 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 register (scalar with vector index) (masked)

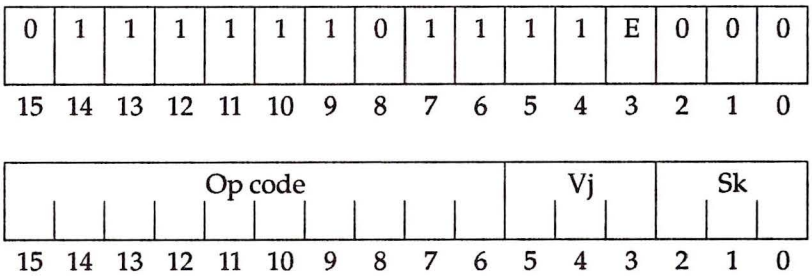
Purpose

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

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
stvi.b.t Sk,Vj	E1	7B00	0111101100	None	Scalar index store vector byte (VM)
stvi.b.f Sk,Vj	E0	7B00	0111101100	None	Scalar index store vector byte (!VM)
stvi.h.t Sk,Vj	E1	7B40	0111101101	None	Scalar index store vector half (VM)
stvi.h.f Sk,Vj	E0	7B40	0111101101	None	Scalar index store vector half (!VM)
stvi.w.t Sk,Vj	E1	7B80	0111101110	None	Scalar index store vector word (VM)
stvi.w.f Sk,Vj	E0	7B80	0111101110	None	Scalar index store vector word (!VM)
stvi.l.t Sk,Vj	E1	7BC0	0111101111	None	Scalar index store vector long (VM)
stvi.l.f Sk,Vj	E0	7BC0	0111101111	None	Scalar index store vector long (!VM)
stvi.s.t Sk,Vj	E1	7B80	0111101110	None	Scalar index store vector single (VM)
stvi.s.f Sk,Vj	E0	7B80	0111101110	None	Scalar index store vector single (!VM)
stvi.d.t Sk,Vj	E1	7BC0	0111101111	None	Scalar index store vector double (VM)
stvi.d.f Sk,Vj	E0	7BC0	0111101111	None	Scalar index store vector double (!VM)

Description

The lower 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 (for `.t`) or clear (for `.f`).

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; } /* end of switch */
```

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 register (vector with vector index)

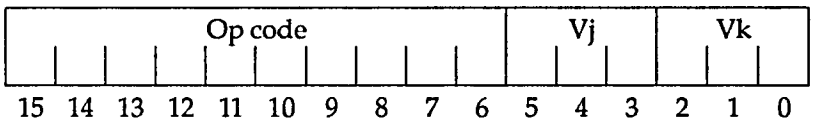
Purpose

To store the elements of a vector register using the contents of a vector register as set of indices ("vector scatter")

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
stvi.b Vk,Vj	ST	7A00	0111101000	None	Index store vector byte
stvi.h Vk,Vj	ST	7A40	0111101001	None	Index store vector halfword
stvi.w Vk,Vj	ST	7A80	0111101010	None	Index store vector word
stvi.l Vk,Vj	ST	7AC0	0111101011	None	Index store vector longword
stvi.s Vk,Vj	ST	7A80	0111101010	None	Index store vector single
stvi.d Vk,Vj	ST	7AC0	0111101011	None	Index store vector double

Description

The lower 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.

Operation

```
for (a = 0; a < VL; a++) {
    c(Vj[a]<31..0> + A5) = Vk[a]; }
```

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} Vk,Vj

Store register (vector with vector index) (masked)

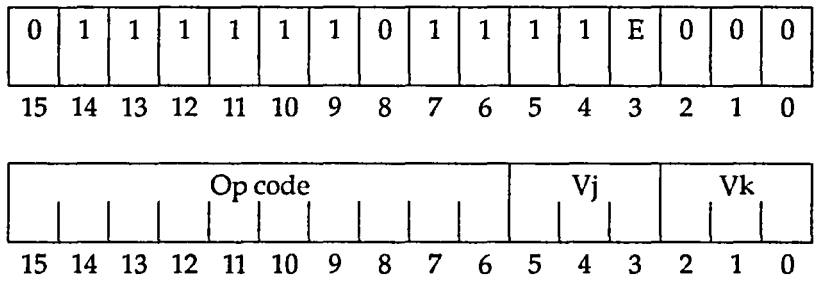
Purpose

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")

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
stvi.b.t Vk,Vj	E1	7A00	0111101000	None	Index store vector byte (VM)
stvi.b.f Vk,Vj	E0	7A00	0111101000	None	Index store vector byte (!VM)
stvi.h.t Vk,Vj	E1	7A40	0111101001	None	Index store vector halfword (VM)
stvi.h.f Vk,Vj	E0	7A40	0111101001	None	Index store vector halfword (!VM)
stvi.w.t Vk,Vj	E1	7A80	0111101010	None	Index store vector word (VM)
stvi.w.f Vk,Vj	E0	7A80	0111101010	None	Index store vector word (!VM)
stvi.l.t Vk,Vj	E1	7AC0	0111101011	None	Index store vector longword (VM)
stvi.l.f Vk,Vj	E0	7AC0	0111101011	None	Index store vector longword (!VM)
stvi.s.t Vk,Vj	E1	7A80	0111101010	None	Index store vector single (VM)
stvi.s.f Vk,Vj	E0	7A80	0111101010	None	Index store vector single (!VM)
stvi.d.t Vk,Vj	E1	7AC0	0111101011	None	Index store vector double (VM)
stvi.d.f Vk,Vj	E0	7AC0	0111101011	None	Index store vector double (!VM)

stvi.{b|h|w|l|s|d}.{t|f} Vk,Vj

Description

The lower 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 (for `.t`) or clear (for `.f`).

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; } /* end of switch */
```

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.

sub.{h|w} #{n|N},Ak

Subtract register (immediate from address)

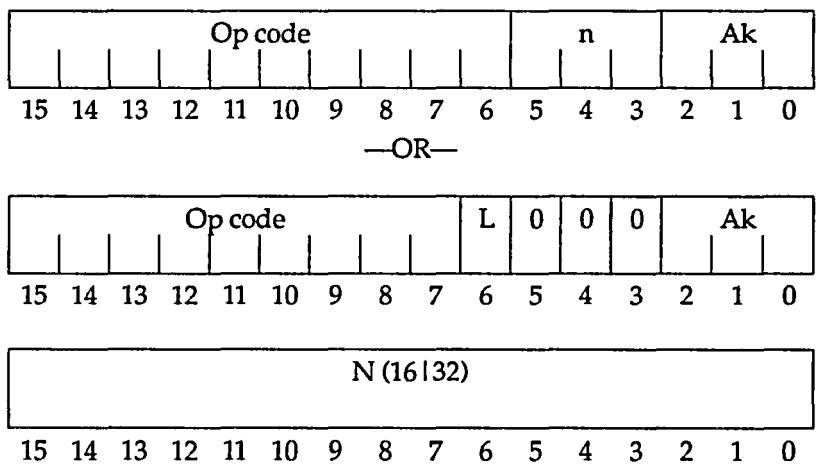
Purpose

To subtract an immediate operand from the contents of an address register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
sub.h #n,Ak	ST	5A80	0101101010	C,AIV	Subtract short immediate address halfword
sub.w #n,Ak	ST	5AC0	0101101011	C,AIV	Subtract short immediate address word
sub.h #N,Ak	ST	1500	000101010	C,AIV	Subtract immediate address halfword
sub.w #N,Ak	ST	1580	000101011	C,AIV	Subtract immediate address word

Description

The contents of address register Ak minus either the short immediate operand (#n) or the sign-extended long immediate operand (#N) replace the contents of Ak.

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

Operation

```
Ak = Ak - Immediate;
```

Exception

`h | w` Integer overflow

sub.{h|w|s} #N,Sk

Subtract register (immediate from scalar)

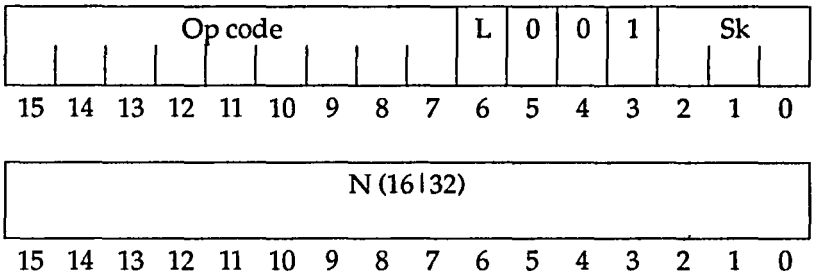
Purpose

To subtract an immediate operand from the contents of a scalar register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
sub.h #N,Sk	ST	1508	000101010	SC,SIV	Subtract scalar/immediate integer halfword
sub.w #N,Sk	ST	1588	000101011	SC,SIV	Subtract scalar/immediate integer word
sub.s #N,Sk	ST	1888	000110001	OV,UN,RO	Subtract scalar/immediate single float

Description

The contents of scalar register Sk minus the sign-extended long immediate operand (#N, length indicated in L) replace the contents of Sk.

The carry out of the subtraction is returned in scalar carry (SC).

Operation

```
Sk = Sk - Immediate;
```

Exceptions

`h|w`	Integer overflow
`s`	Exponent overflow Exponent underflow Reserved operand

sub.{h|w} Aj,Ak

Subtract registers (address from address)

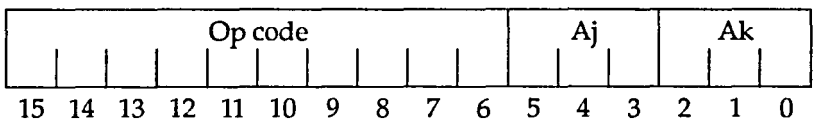
Purpose

To subtract the contents of one address register from the contents of another address register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
sub.h Aj,Ak	ST	5A00	0101101000	C,AIV	Subtract address register halfword
sub.w Aj,Ak	ST	5A40	0101101001	C,AIV	Subtract address register word

Description

The contents of address register Ak minus the contents of Aj replace the contents of Ak.

Operation

```
Ak = Ak - Aj;
```

Exception

h | w	Integer overflow

sub.{b|h|w|l|s|d} Sj,Sk

Subtract registers (scalar from scalar)

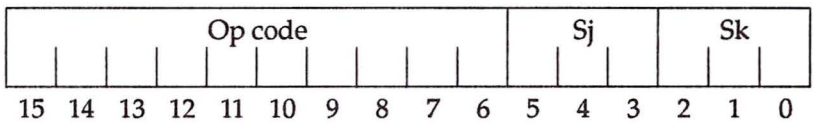
Purpose

To subtract the contents of one scalar register from the contents of another scalar register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
sub.b Sj,Sk	ST	5B00	0101101100	SC,SIV	Subtract scalar/scalar integer byte
sub.h Sj,Sk	ST	5B40	0101101101	SC,SIV	Subtract scalar/scalar integer halfword
sub.w Sj,Sk	ST	5B80	0101101110	SC,SIV	Subtract scalar/scalar integer word
sub.l Sj,Sk	ST	5BC0	0101101111	SC,SIV	Subtract scalar/scalar integer longword
sub.s Sj,Sk	ST	5580	0101010110	OV,UN,RO	Subtract scalar/scalar single float
sub.d Sj,Sk	ST	55C0	0101010111	OV,UN,RO	Subtract scalar/scalar double float

Description

The contents of scalar register Sk minus the contents of Sj replace the contents of Sk.

Operation

```
Sk = Sk - Sj;
```

Exceptions

`b|h|w|l`	Integer overflow
`s|d`	Exponent overflow Exponent underflow Reserved operand

sub.{s|d} Si,Vj,Vk

Reverse subtract registers (vector from scalar)

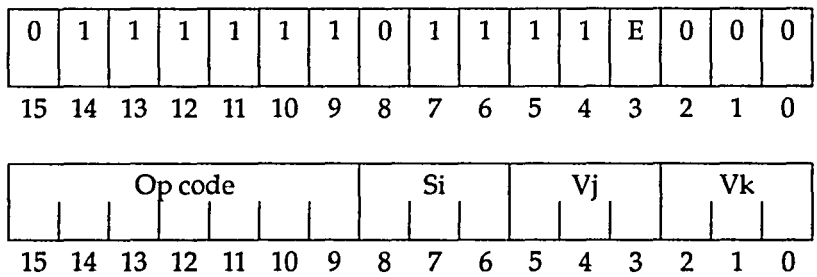
Purpose

To subtract each element of a vector register from the contents of a scalar register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
sub.s Si,Vj,Vk	E0	8000	1000000	OV,UN,RO	Subtract scalar/vector single float
sub.d Si,Vj,Vk	E0	8200	1000001	OV,UN,RO	Subtract 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 minus the contents of the corresponding element of Vj.

Operation

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

Exception

s | d	Exponent overflow Exponent underflow Reserved operand

sub.{s|d}.{t|f} Si,Vj,Vk

Reverse subtract registers (vector from scalar) (masked)

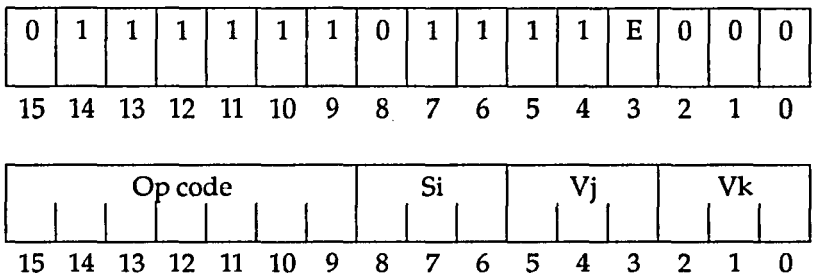
Purpose

To subtract each element of a vector register from the contents of a scalar register under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
sub.s.t Si,Vj,Vk	E1	8800	1000100	OV,UN,RO	Subtract scalar/vector single (VM)
sub.s.f Si,Vj,Vk	E0	8800	1000100	OV,UN,RO	Subtract scalar/vector single (IVM)
sub.d.t Si,Vj,Vk	E1	8A00	1000101	OV,UN,RO	Subtract scalar/vector double (VM)
sub.d.f Si,Vj,Vk	E0	8A00	1000101	OV,UN,RO	Subtract scalar/vector double (IVM)

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 minus the contents of the corresponding element of Vj if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

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;  } /* end of switch */
```

Exceptions

s | d	Exponent overflow Exponent underflow Reserved operand

sub.{b|h|w|l|s|d} Vi,Sj,Vk

Subtract registers (scalar from vector)

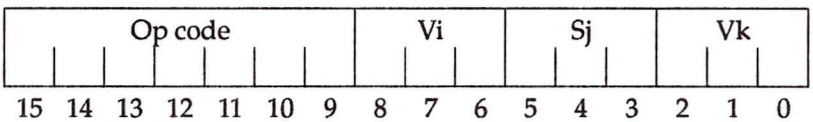
Purpose

To subtract the contents of a scalar register from the elements of a vector register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
sub.b Vi,Sj,Vk	ST	D800	1101100	SIV	Subtract vector/scalar integer byte
sub.h Vi,Sj,Vk	ST	DA00	1101101	SIV	Subtract vector/scalar integer halfword
sub.w Vi,Sj,Vk	ST	DC00	1101110	SIV	Subtract vector/scalar integer word
sub.l Vi,Sj,Vk	ST	DE00	1101111	SIV	Subtract vector/scalar integer longword
sub.s Vi,Sj,Vk	ST	BC00	1011110	OV,UN,RO	Subtract vector/scalar single float
sub.d Vi,Sj,Vk	ST	BE00	1011111	OV,UN,RO	Subtract vector/scalar 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 the corresponding element of Vi minus the contents of scalar register Sj.

Operation

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

Exception

`b|h|w|l`	Integer overflow
`s|d`	Exponent overflow Exponent underflow Reserved operand

sub.{b|h|w|l|s|d}.{t|f} Vi,Sj,Vk

Subtract registers (scalar from vector) (masked)

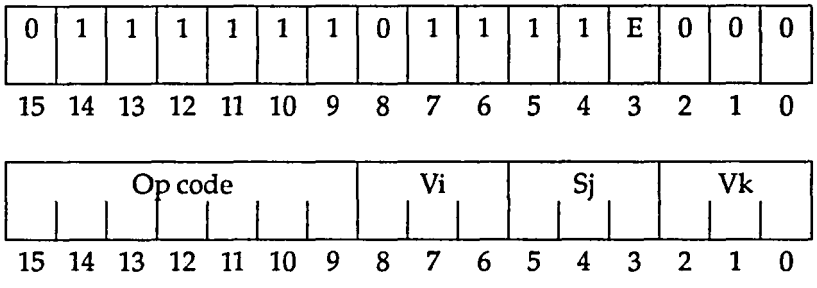
Purpose

To subtract the contents of a scalar register from the elements of a vector register under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
sub.b.t Vi,Sj,Vk	E1	D800	1101100	SIV	Subtract vector/scalar byte (VM)
sub.b.f Vi,Sj,Vk	E0	D800	1101100	SIV	Subtract vector/scalar byte (!VM)
sub.h.t Vi,Sj,Vk	E1	DA00	1101101	SIV	Subtract vector/scalar halfword (VM)
sub.h.f Vi,Sj,Vk	E0	DA00	1101101	SIV	Subtract vector/scalar halfword (!VM)
sub.w.t Vi,Sj,Vk	E1	DC00	1101110	SIV	Subtract vector/scalar word (VM)
sub.w.f Vi,Sj,Vk	E0	DC00	1101110	SIV	Subtract vector/scalar word (!VM)
sub.l.t Vi,Sj,Vk	E1	DE00	1101111	SIV	Subtract vector/scalar longword (VM)
sub.l.f Vi,Sj,Vk	E0	DE00	1101111	SIV	Subtract vector/scalar longword (!VM)
sub.s.t Vi,Sj,Vk	E1	BC00	1011110	OV,UN,RO	Subtract vector/scalar single (VM)
sub.s.f Vi,Sj,Vk	E0	BC00	1011110	OV,UN,RO	Subtract vector/scalar single (!VM)
sub.d.t Vi,Sj,Vk	E1	BE00	1011111	OV,UN,RO	Subtract vector/scalar double (VM)
sub.d.f Vi,Sj,Vk	E0	BE00	1011111	OV,UN,RO	Subtract vector/scalar double (!VM)

sub.{b|h|w|l|s|d}.{t|f} Vi,Sj,Vk

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 scalar register Sj, only if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

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;  } /* end of switch */
```

Exceptions

`b|h|w|l`	Integer overflow
`s|d`	Exponent overflow Exponent underflow Reserved operand

sub.{b|h|w|l|s|d} Vi,Vj,Vk

Subtract registers (vector from vector)

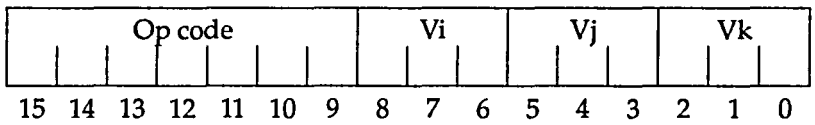
Purpose

To subtract the elements of two vector registers

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
sub.b Vi,Vj,Vk	ST	D000	1101000	SIV	Subtract vector/vector integer byte
sub.h Vi,Vj,Vk	ST	D200	1101001	SIV	Subtract vector/vector integer halfword
sub.w Vi,Vj,Vk	ST	D400	1101010	SIV	Subtract vector/vector integer word
sub.l Vi,Vj,Vk	ST	D600	1101011	SIV	Subtract vector/vector integer longword
sub.s Vi,Vj,Vk	ST	B400	1011010	OV,UN,RO	Subtract vector/vector single float
sub.d Vi,Vj,Vk	ST	B600	1011011	OV,UN,RO	Subtract vector/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 corresponding element of Vi minus the contents of the corresponding element of Vj.

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

sub.{b|h|w|l|s|d}.{t|f} Vi,Vj,Vk

Subtract registers (vector from vector) (masked)

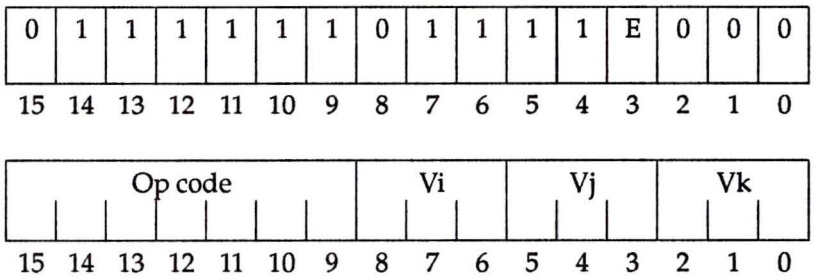
Purpose

To subtract the elements of two vector registers under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
sub.b.t Vi,Vj,Vk	E1	D000	1101000	SIV	Subtract byte vectors (VM)
sub.b.f Vi,Vj,Vk	E0	D000	1101000	SIV	Subtract byte vectors (!VM)
sub.h.t Vi,Vj,Vk	E1	D200	1101001	SIV	Subtract halfword vectors (VM)
sub.h.f Vi,Vj,Vk	E0	D200	1101001	SIV	Subtract halfword vectors (!VM)
sub.w.t Vi,Vj,Vk	E1	D400	1101010	SIV	Subtract word vectors (VM)
sub.w.f Vi,Vj,Vk	E0	D400	1101010	SIV	Subtract word vectors (!VM)
sub.l.t Vi,Vj,Vk	E1	D600	1101011	SIV	Subtract longword vectors (VM)
sub.l.f Vi,Vj,Vk	E0	D600	1101011	SIV	Subtract longword vectors (!VM)
sub.s.t Vi,Vj,Vk	E1	B400	1011010	OV,UN,RO	Subtract single vectors (VM)
sub.s.f Vi,Vj,Vk	E0	B400	1011010	OV,UN,RO	Subtract single vectors (!VM)
sub.d.t Vi,Vj,Vk	E1	B600	1011011	OV,UN,RO	Subtract double vectors (VM)
sub.d.f Vi,Vj,Vk	E0	B600	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, only if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

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; } /* end of switch */
```

Exceptions

`b|h|w|l`	Integer overflow
`s|d`	Exponent overflow Exponent underflow Reserved operand

sum.{b|h|w|l|s|d} {Vk|Sk}
Sum register (vector)

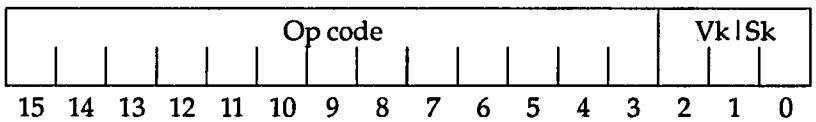
Purpose

To sum all the elements of a vector register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
sum.b Vk	ST	7E00	0111111000000	SIV	Sum a vector of bytes
sum.h Vk	ST	7E08	0111111000001	SIV	Sum a vector of halfwords
sum.w Vk	ST	7E10	0111111000010	SIV	Sum a vector of words
sum.l Vk	ST	7E18	0111111000011	SIV	Sum a vector of longwords
sum.s Vk	ST	7E80	0111111010000	OV,UN,RO	Sum a vector of single float
sum.d Vk	ST	7E88	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 replaces Sk.

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

Notes

1. Initialize the scalar register properly for the first use of the `sum` reduce instruction (usually to 0).
2. The sequence of the sum executed by the hardware is *not* identical to the pseudocode sequence in the "Operation" section, that is, the execution sequence is implementation-specific. For more information, refer to the discussion of vector operations in Chapter 2.
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 register (vector) (masked)

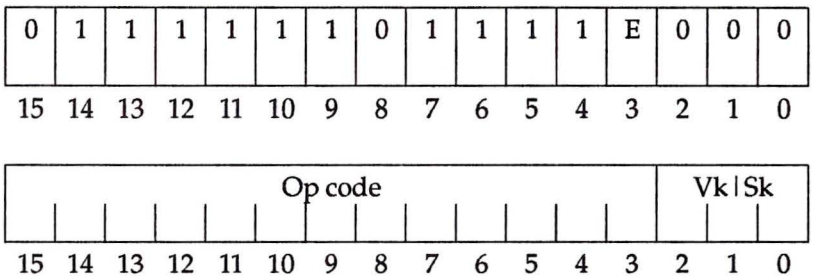
Purpose

To sum a subset of the elements of a vector under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
sum.b.t Vk	E1	7E00	0111111000000	SIV	Sum a vector of bytes (VM)
sum.b.f Vk	E0	7E00	0111111000000	SIV	Sum a vector of bytes (!VM)
sum.h.t Vk	E1	7E08	0111111000001	SIV	Sum a vector of halfwords (VM)
sum.h.f Vk	E0	7E08	0111111000001	SIV	Sum a vector of halfwords (!VM)
sum.w.t Vk	E1	7E10	0111111000010	SIV	Sum a vector of words (VM)
sum.w.f Vk	E0	7E10	0111111000010	SIV	Sum a vector of words (!VM)
sum.l.t Vk	E1	7E18	0111111000011	SIV	Sum a vector of longwords (VM)
sum.l.f Vk	E0	7E18	0111111000011	SIV	Sum a vector of longwords (!VM)
sum.s.t Vk	E1	7E80	0111111010000	OV,UN,RO	Sum a vector of single (VM)
sum.s.f Vk	E0	7E80	0111111010000	OV,UN,RO	Sum a vector of single (!VM)
sum.d.t Vk	E1	7E88	0111111010001	OV,UN,RO	Sum a vector of double (VM)
sum.d.f Vk	E0	7E88	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 replaces Sk, only if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

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;  } /* end of switch */
```

Exceptions

`b|h|w|l`	Integer overflow
`s|d`	Exponent overflow Exponent underflow Reserved operand

Notes

1. Initialize the scalar register properly for the first use of the `sum` reduce instruction (usually to 0).
2. The sequence of the sum executed by the hardware is *not* identical to the pseudocode sequence in the "Operation" section above, that is, the execution sequence is implementation-specific. For more information, refer to the discussion of vector operations in Chapter 2.
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

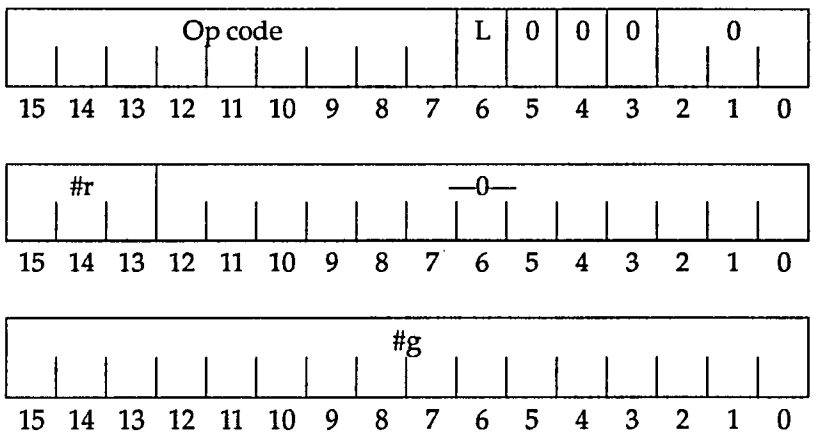
Purpose

To perform a system call to ring specified in `#r`

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
sysc #r,#g	ST	1080	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.

For C100 Series CPUs the SP contained in page 0 of the inner ring is used to define the base of the inner ring stack.

For multiprocessing C Series CPUs:

- 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.

Operation

```
PSW[FRL] = 01;      /* extended frame */
if (CPU == C200) { /* multiprocessing C Series */
    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 (CPU == C200) { /* multiprocessing C Series */
    SP = SP - 112;  /* Extended return block */ }
else {              /* C100 Series */
    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)

Notes

1. All PSW bits are cleared to zero after the extended frame is pushed.
2. Refer to the *CONVEX Architecture Reference Manual (C Series)*, Chapter 7, "Process structures," 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 op code are sign-extended to 32 bits. These 32 bits are interpreted as the `#r`, `#g` fields.

tac *effa*
Test and clear byte

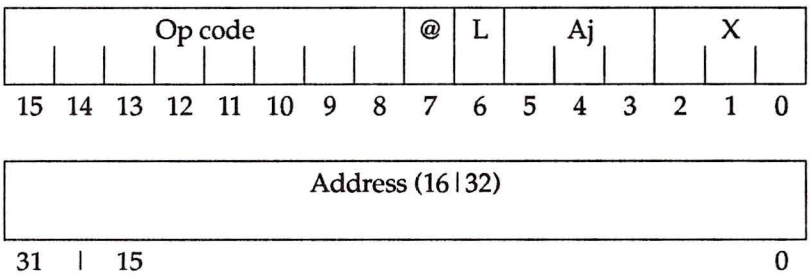
Purpose

To indivisibly test and clear a byte in memory

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
tac *effa*	ST	0800	00001000	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`, that is, it waits for all previous stores to reach memory before proceeding. Next, it clears the "lock" byte. C is set to 1 if the lock byte was set, otherwise, C is cleared to 0.

Operation

```
msync;
if ( c(effa) == 0xFF ) {
    C = 1; }
else {
    C = 0; }
c(effa) = 0;
```

Exception

Deadlock exception

Notes

1. 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.
2. The X field is unused.

tas *effa*
Test and set byte

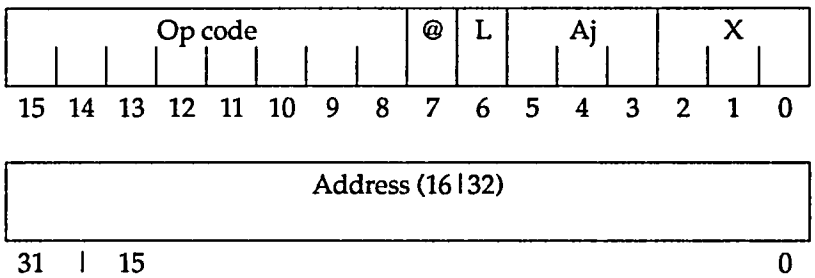
Purpose

To indivisibly test and set a byte in memory

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
tas *effa*	ST	0C00	00001100	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`, that is, waits for all previous stores to reach memory before proceeding. It then sets the "lock" byte. C is set to 1 if the lock byte was clear, otherwise C is cleared to 0.

Operation

```
msync;
if ( c(effa) == 0 ) {
    C = 1; }
else {
    C = 0; }
c(effa) = 0xFF;
```

Notes

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

trap #rm,#b
Trap process

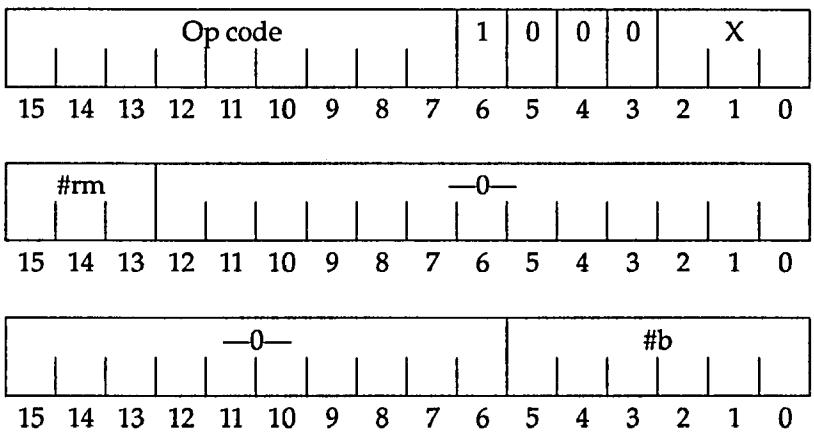
Purpose

To selectively force all threads sharing the same communication register set to enter the exception handler

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
trap #rm,#b	ST	1A00	0001101000	All bits	Force a trap system exception

Description

The ring number `#rm` and bit number `#b` are validated. The most significant bit of `#rm` specifies ring 4. The least significant bit specifies ring 0. If they are valid, bit `#b` (a number from 0-63) in trap instruction registers specified by the ring mask `#rm` is set. If `#rm` specifies the current ring, then a trap on this CPU is taken in the manner described in the pseudocode in the "Operation" section.

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.

Operation

```
if (crossing_rings) {
    Allocate a ring 0 stack from the system resource
structur; }
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

Notes

1. The PSW is cleared to all zeros.
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 Manual (C Series)*, Chapter 12, "Operating system exceptions," for a detailed explanation of the `trap` instruction.
4. The X field is unused.

tst *Ceffa*

Test communication register lock

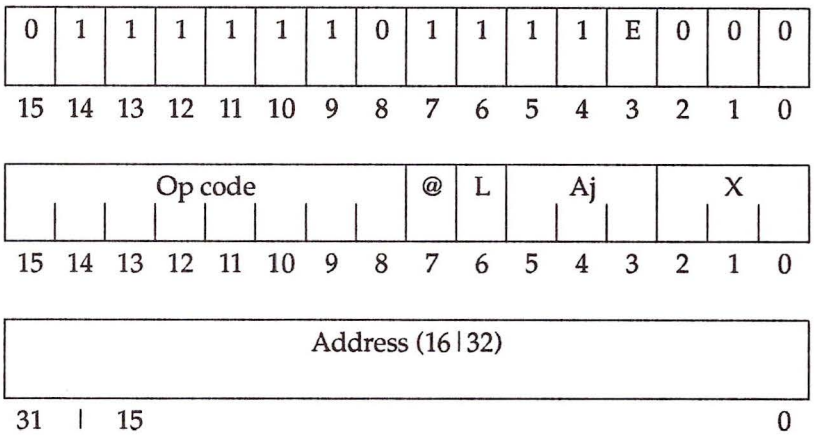
Purpose

To test the value of the lock bit associated with the communication register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
tst a	E0	0100	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.

Operation

```
C = L(Ceffa);
```

Exception

Ring violation (invalid communication register address)

Note

The X field is unused.

tstvv
Test vector valid

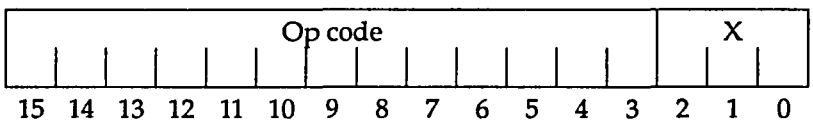
Purpose

To test the value of the vector valid (VV) flag

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
tstvv	ST	7D78	0111110101111	SC	Test value of vector valid flag

Description

The value of the VV flag replaces the SC bit.

Operation

```
if (VV == 1) {
    SC = 1; }
else {
    SC = 0; }
```

Notes

1. This instruction is not privileged.
2. The X field is unused.

tzc Sj,Sk
Trailing zero count (scalar)

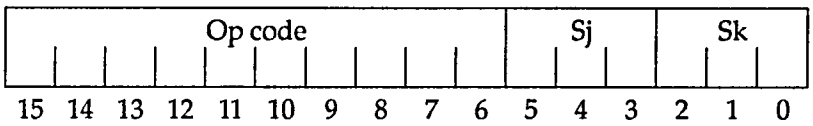
Purpose

To determine the number of trailing zero bits in a scalar register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
tzc Sj,Sk	ST	45C0	0100010111	None	Count of trailing zeros in Sj

Description

The number of trailing zero bits in scalar register Sj replaces the contents of Sk. If Sj contains all zeros, Sk becomes 64.

Operation

```
for (a = 0; a < 63; a++) {
    if (Sj<a> != 0) {
        break; } } /* found rightmost 1, so break loop */
Sk = a;
```

tzc Vj,Vk
Trailing zero count (vector)

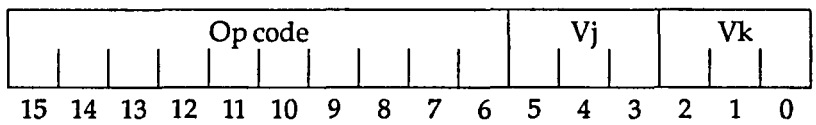
Purpose

To count the number of trailing zero bits in each element of a vector register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
tzc Vj,Vk	ST	6200	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.

Operation

```
for (a = 0; a < VL; a++) {
    for (b = 0; b < 63; b++) {
        if (Vj[a]<b> != 0) {
            break; } } /* found rightmost 1, so break loop */
    Vk[a] = b; }
```

tzc.{t|f} Vj,Vk

Trailing zero count (vector) (masked)

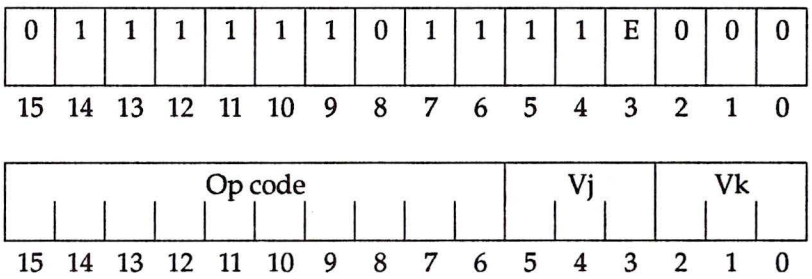
Purpose

To count the number of trailing zero bits in each element of a vector register under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
tzc.t Vj,Vk	E1	6200	0110001000	None	Trailing zero count vector (VM)
tzc.f Vj,Vk	E0	6200	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, only if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

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 */
        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 */
        break; } /* end of switch */
```

ulk *Ceffa*
Unlock communication register

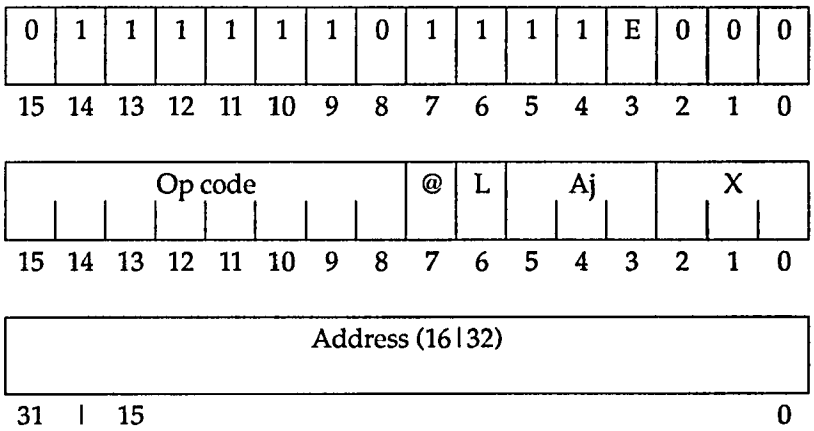
Purpose

To unlock a communication register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
ulk *Ceffa*	E0	0300	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.

Operation

```
if (L(Ceffa) == 1) {
    L(Ceffa) = 0;
    C = 1; }
else {
    C = 0; }
```

Exceptions

Ring violation (invalid communication register address)
Deadlock exception

Notes

1. This instruction is atomic.
2. The X field is unused.

wfork
Wait for a fork

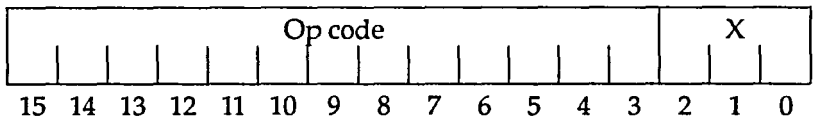
Purpose

To terminate a thread, idle the current CPU and look for any posted forks

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
wfork	ST	7C98	0111110010011	None	Wait for a fork

Description

The CPU is returned to the idle state and is disassociated from any process context. Then an attempt is made to find other posted forks. If a posted fork is found in the current CIR, the fork is taken directly and the CPU is not idled. If there is no fork in the current CIR, the CPU is idled and forks are accepted from other CIRs.

Refer to the *CONVEX Architecture Reference Manual (C Series)*, Chapter 11, "Operating system interrupts," for a description of specific actions of an idle CPU.

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`)

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, that is, if either PSW (TTR) or PSW (TIT) is set and the CPU attempts to go idle, an instruction trace trap will occur.
3. The X field is unused.

xmti Sk
Transmit interrupt

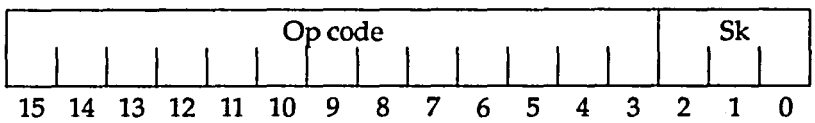
Purpose

To interrupt a channel

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
xmti Sk	ST	7D68	0111110101101	C	Transmit interrupt

Description

An interrupt is asserted to the virtual interrupt channel specified by Sk<7..0>.

Operation

```
Assert the channel interrupt line of virtual
   channel c(Sk)<7..0>);
```

Exception

Ring violation (privileged instruction)

Notes

1. Channels <0..7> are associated with the CPU and are maskable (refer to the `mski` instruction).
2. Address carry (C) is cleared to 0 if the interrupt is successfully transmitted, and C is set to 1 if the operation times out.

xor #N,Ak

Exclusive or (address register with immediate)

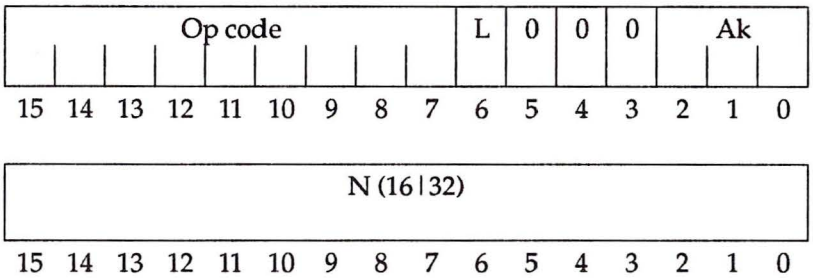
Purpose

To `exclusive or` an immediate operand with an address register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
xor #N,Ak	ST	1300	000100110	None	Exclusive or (immediate with address register)

Description

The `exclusive or` of the sign-extended long immediate operand (#N, length indicated by L) and the contents of address register Ak replaces the contents of Ak.

Operation

```
Ak = Ak ^ Immediate;
```

xor #N,Sk

Exclusive or (immediate with scalar)

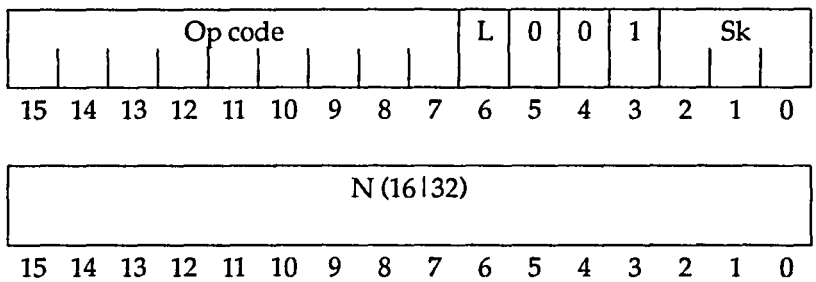
Purpose

To `exclusive or` an immediate operand with the contents of a scalar register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
xor #N,Sk	ST	1308	000100110	None	Exclusive or (scalar with immediate)

Description

The `exclusive or` of the sign-extended long immediate operand (#N, length indicated by L) and the least significant 32 bits of scalar register Sk replaces the least significant 32 bits of Sk.

The most significant 32 bits of Sk are not affected.

Operation

```
Sk<31..0> = Sk<31..0> ^ Immediate;
```

xor Aj,Ak

Exclusive or registers (address with address)

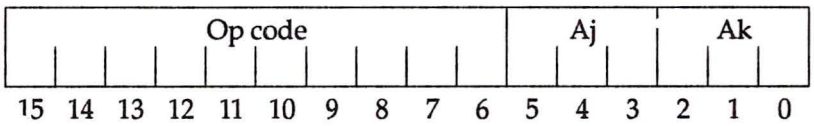
Purpose

To `exclusive or` the contents of two address registers

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
xor Aj,Ak	ST	5280	0101001010	None	Exclusive or address register

Description

The `exclusive or` of the contents of address registers Aj and Ak replaces the contents of Ak.

Operation

```
Ak = Ak ^ Aj;
```

xor Sj,Sk

Exclusive or registers (scalar with scalar)

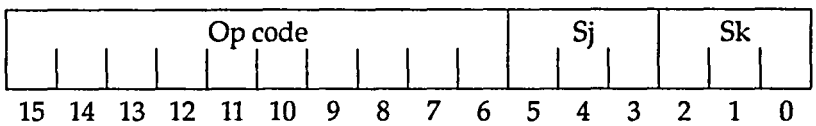
Purpose

To `exclusive or` the contents of two scalar registers

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
xor Sj,Sk	ST	5380	0101001110	None	Exclusive or scalar/scalar

Description

The `exclusive or` of the contents of scalar registers Sj and Sk replaces the contents of Sk.

Operation

```
Sk = Sk ^ Sj;
```

xor Vi,Sj,Vk

Exclusive or registers (vector with scalar)

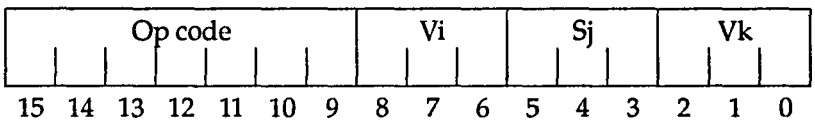
Purpose

To `exclusive or` the elements of a vector register and the contents of a scalar register

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
xor Vi,Sj,Vk	ST	AC00	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.

Operation

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

xor.{t|f} Vi,Sj,Vk

Exclusive or registers (vector with scalar) (masked)

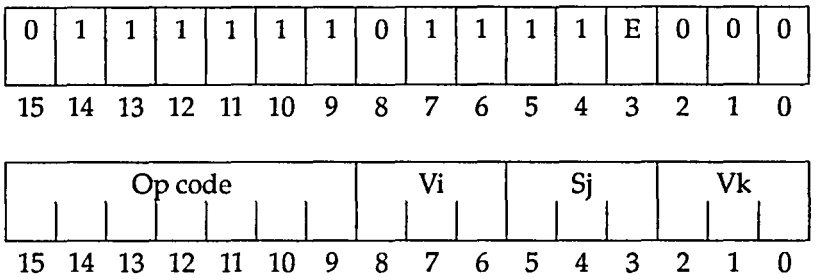
Purpose

To `exclusive or` the contents of a vector and a scalar under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
xor.t Vi,Sj,Vk	E1	AC00	1010110	None	Exclusive or vector/scalar (VM)
xor.f Vi,Sj,Vk	E0	AC00	1010110	None	Exclusive or vector/scalar (!VM)

Description

The `exclusive or` of the contents of the corresponding element of Vi and the contents of scalar register Sj, replaces the contents each of the first VL elements of vector register Vk, only if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

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;  } /* end of switch */
```

xor Vi,Vj,Vk

Exclusive or registers (vector with vector)

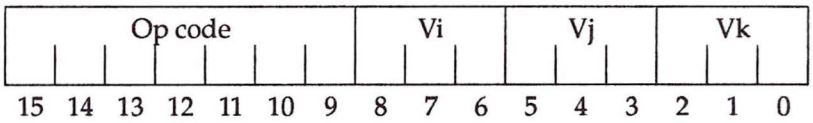
Purpose

To `exclusive or` the elements of two vector registers

Application

C100, C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
xor Vi,Vj,Vk	ST	A400	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.

Operation

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

xor.{t|f} Vi,Vj,Vk

Exclusive or registers (vector with vector) (masked)

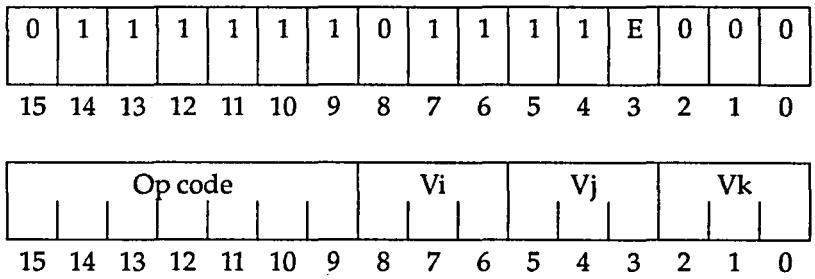
Purpose

To `exclusive or` the contents of two vector registers under control of the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
xor.t Vi,Vj,Vk	E1	A400	1010010	None	Exclusive or two vectors (VM)
xor.f Vi,Vj,Vk	E0	A400	1010010	None	Exclusive or two vectors (IVM)

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, only if the corresponding VM bit is set (for `.t`) or clear (for `.f`).

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; } /* end of switch */
```

xpnd.{t|f} Vj,Vk
Expand (copy) register (vector) (masked)

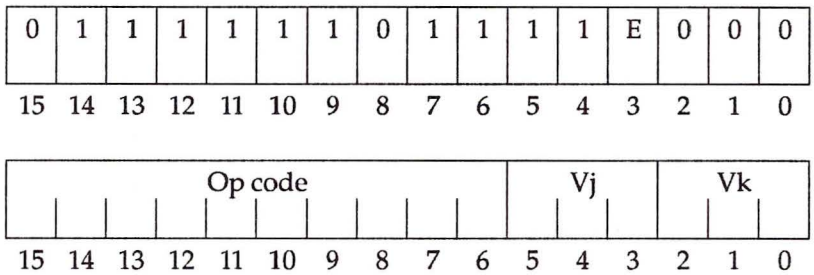
Purpose

To expand (copy) a vector using the vector merge (VM) register

Application

C200/C3200, C3400, C3800 Series CPUs

Format



Op code

Mnemonic	Space	Hex	Binary	PSW	Description
xpnd.t Vj,Vk	E1	6280	0110001010	None	Expand a vector (VM)
xpnd.f Vj,Vk	E0	6280	0110001010	None	Expand a vector (!VM)

Description

Each element of vector register Vj is copied to vector register Vk, only if the corresponding VM register bit is set (for `.t`) or clear (for `.f`).

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;  } /* end of switch */
```

Note

The `plc VM` instruction calculates the number of elements copied from Vj.

Instructions sorted by op code

Table 17
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
ST 0000	`exit`	Error exit instruction
ST 0100	`jmp` *effa*	Jump always
ST 0200	`jmpi.f` *effa*	Jump on ION false
ST 0300	`jmpi.t` *effa*	Jump on ION true
ST 0400	`jmpa.f` *effa*	Jump on address carry false
ST 0500	`jmpa.t` *effa*	Jump on address carry true
ST 0600	`jmps.f` *effa*	Jump on scalar carry false
ST 0700	`jmps.t` *effa*	Jump on scalar carry true
ST 0800	`tac` *effa*	Test and clear a byte in memory
ST 0900	`ldea` *effa*`,Ak`	Load effective address
ST 0A00	`ld.l` *effa*`,VLS`	Load VS and VL from memory
ST 0B00	`ld.x` *effa*`,VM`	Load VM from memory
ST 0C00	`tas` *effa*	Test and set a memory byte
ST 0D00	`pshea` *effa*	Push effective address
ST 0E00	`st.l VLS,` *effa*	Store VS and VL to memory
ST 0F00	`st.x VM,` *effa*	Store VM into memory
ST 1000	`halt #N,Ak`	Halt the CPU
ST 1008	`ld.d #N,Sk`	Load double float immediate upper 32 bits

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
ST 1080	`sysc #r,#g`	Perform a system call
ST 1088	`ld.du #N,Sk`	Load 64-bit floating immediate
ST 1088	`ld.lu #N,Sk`	Load 64-bit integer immediate
ST 1088	`ld.u #N,Sk`	Load immediate
ST 1100	`ld.h #N,Ak`	Load halfword immediate into Ak
ST 1108	`ld.l #N,Sk`	Load 32-bit immediate sign-extended to 64 bits
ST 1180	`ld.w #N,Ak`	Load immediate into Ak
ST 1188	`ld.dl #N,Sk`	Load 64-bit floating immediate
ST 1188	`ld.ll #N,Sk`	Load 64-bit integer immediate
ST 1188	`ld.s #N,Sk`	Load a single float immediate
ST 1188	`ld.w #N,Sk`	Load a 32-bit immediate
ST 1200	`and #N,Ak`	AND immediate to address register
ST 1208	`and #N,Sk`	AND scalar/immediate
ST 1280	`or #N,Ak`	OR immediate to address register
ST 1288	`or #N,Sk`	OR scalar/immediate
ST 1300	`xor #N,Ak`	Exclusive OR immediate to address register
ST 1308	`xor #N,Sk`	Exclusive OR scalar/immediate
ST 1380	`shf #N,Ak`	Logical shift immediate
ST 1388	`shf #N,Sk`	Shift scalar/immediate
ST 1400	`add.h #N,Ak`	Add immediate address halfword
ST 1408	`add.h #N,Sk`	Add scalar/immediate integer halfword
ST 1480	`add.w #N,Ak`	Add immediate address word
ST 1488	`add.w #N,Sk`	Add scalar/immediate integer word
ST 1500	`sub.h #N,Ak`	Subtract immediate address halfword
ST 1508	`sub.h #N,Sk`	Subtract scalar/immediate integer halfword
ST 1580	`sub.w #N,Ak`	Subtract immediate address word

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
ST 1588	`sub.w #N,Sk`	Subtract scalar/immediate integer word
ST 1600	`mul.h #N,Ak`	Multiply immediate address halfword
ST 1608	`mul.h #N,Sk`	Multiply scalar/immediate integer halfword
ST 1680	`mul.w #N,Ak`	Multiply immediate address word
ST 1688	`mul.w #N,Sk`	Multiply scalar/immediate integer word
ST 1700	`div.h #N,Ak`	Divide immediate address halfword
ST 1708	`div.h #N,Sk`	Divide scalar/scalar integer halfword
ST 1780	`div.w #N,Ak`	Divide immediate address word
ST 1788	`div.w #N,Sk`	Divide scalar/scalar integer word
ST 1800	`ld.w #N,VL`	Load VL with an immediate
ST 1808	`add.s #N,Sk`	Add scalar/immediate single float
ST 1880	`ld.w #N,VS`	Load VS from an immediate
ST 1888	`sub.s #N,Sk`	Subtract scalar/immediate single float
ST 1908	`mul.s #N,Sk`	Multiply scalar/immediate single float
ST 1980	`shf.w #N,Sk`	Shift scalar word/immediate
ST 1988	`div.s #N,Sk`	Divide scalar/scalar single float
ST 1A00	`trap #rm,#b`	Force a trap system exception
ST 1A08	`le.s #N,Sk`	Compare less than or equal single
ST 1A88	`lt.s #N,Sk`	Compare less than single
ST 1B00	`eq.h #N,Ak`	Compare equal halfword
ST 1B08	`eq.h #N,Sk`	Compare equal halfword
ST 1B80	`eq.w #N,Ak`	Compare equal word
ST 1B88	`eq.w #N,Sk`	Compare equal word
ST 1C00	`leu.h #N,Ak`	Compare unsigned less than halfword
ST 1C08	`leu.h #N,Sk`	Compare unsigned less than or equal to halfword
ST 1C80	`leu.w #N,Ak`	Compare unsigned less than word

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
ST 1C88	`leu.w #N,Sk`	Compare unsigned less than or equal to word
ST 1D00	`ltu.h #N,Ak`	Compare unsigned less than halfword
ST 1D08	`ltu.h #N,Sk`	Compare unsigned less than halfword
ST 1D80	`ltu.w #N,Ak`	Compare unsigned less than word
ST 1D88	`ltu.w #N,Sk`	Compare unsigned less than word
ST 1E00	`le.h #N,Ak`	Compare less than or equal halfword
ST 1E08	`le.h #N,Sk`	Compare less than or equal halfword
ST 1E80	`le.w #N,Ak`	Compare less than or equal word
ST 1E88	`le.w #N,Sk`	Compare less than or equal word
ST 1F00	`lt.h #N,Ak`	Compare less than halfword
ST 1F08	`lt.h #N,Sk`	Compare less than halfword
ST 1F80	`lt.w #N,Ak`	Compare less than word
ST 1F88	`lt.w #N,Sk`	Compare less than word
ST 2000	`call` *effa*	Call a subroutine
ST 2100	`calls` *effa*	Call a subroutine
ST 2200	`callq` *effa*	Push the PC and jump
ST 2300	`pfork` *effa*`,Ak`	Post a fork
ST 2400	`ste.b Sk,` *effa*	Store an extended scalar byte
ST 2500	`ste.h Sk,` *effa*	Store an extended scalar halfword
ST 2600	`ste.s Sk,` *effa*	Store an extended scalar single float
ST 2600	`ste.w Sk,` *effa*	Store an extended scalar word
ST 2700	`ste.d Sk,` *effa*	Store an extended scalar double float
ST 2700	`ste.l Sk,` *effa*	Store an extended scalar longword
ST 2800	`ld.b` *effa*`,Ak`	Load address register byte
ST 2900	`ld.h` *effa*`,Ak`	Load address register halfword
ST 2A00	`ld.w` *effa*`,Ak`	Load address register word

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
ST 2B00	`incr.w` *effa*`,Ak`	Increment resource structure data
ST 2C00	`st.b Ak,`*effa*	Store address register byte
ST 2D00	`st.h Ak,`*effa*	Store address register halfword
ST 2E00	`st.w Ak,`*effa*	Store address register word
ST 2F00	`incr.l` *effa*`,Sk`	Increment long resource structure
ST 3000	`ld.b` *effa*`,Sk`	Load scalar byte
ST 3100	`ld.h` *effa*`,Sk`	Load scalar halfword
ST 3200	`ld.s` *effa*`,Sk`	Load scalar single float
ST 3200	`ld.w` *effa*`,Sk`	Load scalar word
ST 3300	`ld.d` *effa*`,Sk`	Load scalar double float
ST 3300	`ld.l` *effa*`,Sk`	Load scalar longword
ST 3400	`st.b Sk,`*effa*	Store scalar byte
ST 3500	`st.h Sk,`*effa*	Store scalar halfword
ST 3600	`st.s Sk,`*effa*	Store scalar single float
ST 3600	`st.w Sk,`*effa*	Store scalar word
ST 3700	`st.d Sk,`*effa*	Store scalar double float
ST 3700	`st.l Sk,`*effa*	Store scalar longword
ST 3800	`ld.b` *effa*`,Vk`	Load vector byte
ST 3900	`ld.h` *effa*`,Vk`	Load vector halfword
ST 3A00	`ld.s` *effa*`,Vk`	Load vector single float
ST 3A00	`ld.w` *effa*`,Vk`	Load vector word
ST 3B00	`ld.d` *effa*`,Vk`	Load vector double float
ST 3B00	`ld.l` *effa*`,Vk`	Load vector longword
ST 3C00	`st.b Vk,`*effa*	Store vector byte
ST 3D00	`st.h Vk,`*effa*	Store vector halfword
ST 3E00	`st.s Vk,`*effa*	Store vector single float

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
ST 3E00	`st.w Vk,`*effa*	Store vector word
ST 3F00	`st.d Vk,`*effa*	Store vector double float
ST 3F00	`st.l Vk,`*effa*	Store vector longword
ST 4000	`cvtw.b Aj,Ak`	Convert word to byte
ST 4040	`cvtw.h Aj,Ak`	Convert word to halfword
ST 4080	`cvtb.w Aj,Ak`	Convert byte to word
ST 40C0	`cvth.w Aj,Ak`	Convert halfword to word
ST 4100	`cvtw.b Sj,Sk`	Convert word to byte
ST 4140	`cvtw.h Sj,Sk`	Convert word to halfword
ST 4180	`cvtb.w Sj,Sk`	Convert byte to word
ST 41C0	`cvth.w Sj,Sk`	Convert halfword to word
ST 4200	`cvtw.s Sj,Sk`	Convert word to single float
ST 4240	`cvts.w Sj,Sk`	Convert single float to word
ST 4280	`cvtd.s Sj,Sk`	Convert double float to single float
ST 42C0	`cvts.d Sj,Sk`	Convert single float to double float
ST 4300	`cvts.l Sj,Sk`	Convert single float to longword
ST 4340	`cvtd.l Sj,Sk`	Convert double float to longword
ST 4380	`cvtl.s Sj,Sk`	Convert longword to single float
ST 43C0	`cvtl.d Sj,Sk`	Convert longword to double float
ST 4400	`ldpa Aj,Ak`	Load a physical byte address into Ak
ST 4440	`shf #n,Ak`	Logical shift short immediate
ST 4480	`ld.h #n,Ak`	Load short immediate into Ak
ST 44C0	`ld.w #n,Ak`	Load short immediate into Ak
ST 4500	`cvtl.w Sj,Sk`	Convert longword to word
ST 4540	`cvtw.l Sj,Sk`	Convert word to longword
ST 4580	`plc.t Sj,Sk`	Count the number of ones in Sj

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
ST 45C0	`tzc Sj,Sk`	Count of trailing zeros in Sj
ST 4600	`eq.h Aj,Ak`	Compare equal halfword
ST 4640	`eq.w Aj,Ak`	Compare equal word
ST 4680	`eq.h #n,Ak`	Compare equal halfword
ST 46C0	`eq.w #n,Ak`	Compare equal word
ST 4700	`eq.b Sj,Sk`	Compare equal byte
ST 4740	`eq.h Sj,Sk`	Compare equal halfword
ST 4780	`eq.w Sj,Sk`	Compare equal word
ST 47C0	`eq.l Sj,Sk`	Compare equal longword
ST 4800	`leu.h Aj,Ak`	Compare unsigned less than or equal to halfword
ST 4840	`leu.w Aj,Ak`	Compare unsigned less than or equal to word
ST 4880	`leu.h #n,Ak`	Compare unsigned less than or equal halfword
ST 48C0	`leu.w #n,Ak`	Compare unsigned less than or equal word
ST 4900	`leu.b Sj,Sk`	Compare less than or equal to byte
ST 4940	`leu.h Sj,Sk`	Compare less than or equal to halfword
ST 4980	`leu.w Sj,Sk`	Compare less than or equal to word
ST 49C0	`leu.l Sj,Sk`	Compare less than or equal to longword
ST 4A00	`ltu.h Aj,Ak`	Compare unsigned less than halfword
ST 4A40	`ltu.w Aj,Ak`	Compare unsigned less than word
ST 4A80	`ltu.h #n,Ak`	Compare unsigned less than halfword
ST 4AC0	`ltu.w #n,Ak`	Compare unsigned less than word
ST 4B00	`ltu.b Sj,Sk`	Compare less than byte
ST 4B40	`ltu.h Sj,Sk`	Compare less than halfword
ST 4B80	`ltu.w Sj,Sk`	Compare less than word
ST 4BC0	`ltu.l Sj,Sk`	Compare less than longword
ST 4C00	`le.h Aj,Ak`	Compare less than or equal signed halfword

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
ST 4C40	`le.w Aj,Ak`	Compare less than or equal signed word
ST 4C80	`le.h #n,Ak`	Compare less than or equal halfword
ST 4CC0	`le.w #n,Ak`	Compare less than or equal word
ST 4D00	`le.b Sj,Sk`	Compare less than or equal byte
ST 4D40	`le.h Sj,Sk`	Compare less than or equal halfword
ST 4D80	`le.w Sj,Sk`	Compare less than or equal word
ST 4DC0	`le.l Sj,Sk`	Compare less than or equal longword
ST 4E00	`lt.h Aj,Ak`	Compare less than signed halfword
ST 4E40	`lt.w Aj,Ak`	Compare less than signed word
ST 4E80	`lt.h #n,Ak`	Compare less than halfword
ST 4EC0	`lt.w #n,Ak`	Compare less than word
ST 4F00	`lt.b Sj,Sk`	Compare less than byte
ST 4F40	`lt.h Sj,Sk`	Compare less than halfword
ST 4F80	`lt.w Sj,Sk`	Compare less than word
ST 4FC0	`lt.l Sj,Sk`	Compare less than longword
ST 5000	`add.w Sj,Ak`	Add scalar to address word
ST 5040	`shf Aj,Ak`	Shift an address
ST 5080	`mov Aj,Ak`	Move address register
ST 50C0	`mov Sj,Ak`	Move 32 bits of Sj into Ak
ST 5100	`mov.s Sj,Sk`	Move scalar register double float
ST 5100	`mov.w Sj,Sk`	Move scalar register word
ST 5140	`shf Sj,Sk`	Shift a scalar
ST 5180	`mov.d Sj,Sk`	Move scalar register single float
ST 5180	`mov.l Sj,Sk`	Move scalar register longword
ST 51C0	`mov Aj,Sk`	Move an address to a scalar
ST 5200	`and Aj,Ak`	AND address register

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
ST 5240	`or Aj,Ak`	OR address register
ST 5280	`xor Aj,Ak`	Exclusive OR address register
ST 52C0	`not Aj,Ak`	Complement address register
ST 5300	`and Sj,Sk`	AND scalar/scalar
ST 5340	`or Sj,Sk`	OR scalar/scalar
ST 5380	`xor Sj,Sk`	Exclusive OR scalar/scalar
ST 53C0	`not Sj,Sk`	Complement scalar/scalar
ST 5400	`le.s Sj,Sk`	Compare less than or equal single float
ST 5440	`le.d Sj,Sk`	Compare less than or equal double float
ST 5480	`lt.s Sj,Sk`	Compare less than single float
ST 54C0	`lt.d Sj,Sk`	Compare less than double float
ST 5500	`add.s Sj,Sk`	Add scalar/scalar single float
ST 5540	`add.d Sj,Sk`	Add scalar/scalar double float
ST 5580	`sub.s Sj,Sk`	Subtract scalar/scalar single float
ST 55C0	`sub.d Sj,Sk`	Subtract scalar/scalar double float
ST 5600	`eq.s Sj,Sk`	Compare equal single float
ST 5640	`eq.d Sj,Sk`	Compare equal double float
ST 5680	`neg.h Aj,Ak`	Negate address register halfword
ST 56C0	`neg.w Aj,Ak`	Negate address register word
ST 5700	`mul.s Sj,Sk`	Multiply scalar/scalar single float
ST 5740	`mul.d Sj,Sk`	Multiply scalar/scalar double float
ST 5780	`div.s Sj,Sk`	Divide scalar/scalar single float
ST 57C0	`div.d Sj,Sk`	Divide scalar/scalar double float
ST 5800	`add.h Aj,Ak`	Add address register halfword
ST 5840	`add.w Aj,Ak`	Add address register word
ST 5880	`add.h #n,Ak`	Add short immediate address halfword

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
ST 58C0	`add.w #n,Ak`	Add short immediate address word
ST 5900	`add.b Sj,Sk`	Add scalar/scalar integer byte
ST 5940	`add.h Sj,Sk`	Add scalar/scalar integer halfword
ST 5980	`add.w Sj,Sk`	Add scalar/scalar integer word
ST 59C0	`add.l Sj,Sk`	Add scalar/scalar integer longword
ST 5A00	`sub.h Aj,Ak`	Subtract address register halfword
ST 5A40	`sub.w Aj,Ak`	Subtract address register word
ST 5A80	`sub.h #n,Ak`	Subtract short immediate address halfword
ST 5AC0	`sub.w #n,Ak`	Subtract short immediate address word
ST 5B00	`sub.b Sj,Sk`	Subtract scalar/scalar integer byte
ST 5B40	`sub.h Sj,Sk`	Subtract scalar/scalar integer halfword
ST 5B80	`sub.w Sj,Sk`	Subtract scalar/scalar integer word
ST 5BC0	`sub.l Sj,Sk`	Subtract scalar/scalar integer longword
ST 5C00	`mul.h Aj,Ak`	Multiply address register halfword
ST 5C40	`mul.w Aj,Ak`	Multiply address register word
ST 5C80	`mul.h #n,Ak`	Multiply short immediate address halfword
ST 5CC0	`mul.w #n,Ak`	Multiply short immediate address word
ST 5D00	`mul.b Sj,Sk`	Multiply scalar/scalar integer byte
ST 5D40	`mul.h Sj,Sk`	Multiply scalar/scalar integer halfword
ST 5D80	`mul.w Sj,Sk`	Multiply scalar/scalar integer word
ST 5DC0	`mul.l Sj,Sk`	Multiply scalar/scalar integer longword
ST 5E00	`div.h Aj,Ak`	Divide address register halfword
ST 5E40	`div.w Aj,Ak`	Divide address register word
ST 5E80	`div.h #n,Ak`	Divide short immediate address halfword
ST 5EC0	`div.w #n,Ak`	Divide short immediate address word
ST 5F00	`div.b Sj,Sk`	Divide scalar/scalar integer byte

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
ST 5F40	`div.h Sj,Sk`	Divide scalar/scalar integer halfword
ST 5F80	`div.w Sj,Sk`	Divide scalar/scalar integer word
ST 5FC0	`div.l Sj,Sk`	Divide scalar/scalar integer longword
ST 6000	`cvtd.s Vj,Vk`	Convert double float to single float
ST 6040	`cvts.d Vj,Vk`	Convert single float to double float
ST 6080	`cvtl.d Vj,Vk`	Convert longword to double float
ST 60C0	`cvtd.l Vj,Vk`	Convert double float to longword
ST 6100	`mov Sj,Sk,VM`	Load VM(Sj) from Sk
ST 6140	`mov Sj,VM,Sk`	Load Sk from VM(Sj)
ST 61C0	`lop Sj,Sk`	Leading one's position in Sj
ST 6200	`tzc Vj,Vk`	Trailing zero count vector
ST 6240	`lop Vj,Vk`	Leading ones position vector
ST 62C0	`not Vj,Vk`	Complement a vector
ST 6300	`shf Sj,Vk`	Shift a vector accumulator
ST 6340	`plc.t Vj,Vk`	Population count of a vector
ST 6380	`cprs.f Vj,Vk`	Compress a vector (!VM)
ST 63C0	`cprs.t Vj,Vk`	Compress a vector (VM)
ST 6400	`eq.s Vj,Vk`	Compare equal single
ST 6440	`eq.d Vj,Vk`	Compare equal double precision
ST 6480	`neg.s Vj,Vk`	Negate vector/vector single float
ST 64C0	`neg.d Vj,Vk`	Negate vector/vector double float
ST 6500	`eq.s Sj,Vk`	Compare equal single
ST 6540	`eq.d Sj,Vk`	Compare equal double precision
ST 6580	`neg.s Sj,Sk`	Negate scalar/scalar single float
ST 65C0	`neg.d Sj,Sk`	Negate scalar/scalar double float
ST 6600	`le.s Vj,Vk`	Compare less than or equal single

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
ST 6640	`le.d Vj,Vk`	Compare less than or equal double float
ST 6680	`lt.s Vj,Vk`	Compare less than single
ST 66C0	`lt.d Vj,Vk`	Compare less than double float
ST 6700	`le.s Sj,Vk`	Compare less than or equal single
ST 6740	`le.d Sj,Vk`	Compare less than or equal double float
ST 6780	`lt.s Sj,Vk`	Compare less than single
ST 67C0	`lt.d Sj,Vk`	Compare less than double float
ST 6800	`eq.b Vj,Vk`	Compare equal byte
ST 6840	`eq.h Vj,Vk`	Compare equal halfword
ST 6880	`eq.w Vj,Vk`	Compare equal word
ST 68C0	`eq.l Vj,Vk`	Compare equal longword
ST 6900	`eq.b Sj,Vk`	Compare equal byte
ST 6940	`eq.h Sj,Vk`	Compare equal halfword
ST 6980	`eq.w Sj,Vk`	Compare equal word
ST 69C0	`eq.l Sj,Vk`	Compare equal longword
ST 6A00	`le.b Vj,Vk`	Compare less than or equal byte
ST 6A40	`le.h Vj,Vk`	Compare less than or equal halfword
ST 6A80	`le.w Vj,Vk`	Compare less than or equal word
ST 6AC0	`le.l Vj,Vk`	Compare less than or equal longword
ST 6B00	`le.b Sj,Vk`	Compare less than or equal byte
ST 6B40	`le.h Sj,Vk`	Compare less than or equal halfword
ST 6B80	`le.w Sj,Vk`	Compare less than or equal word
ST 6BC0	`le.l Sj,Vk`	Compare less than or equal longword
ST 6C00	`lt.b Vj,Vk`	Compare less than byte
ST 6C40	`lt.h Vj,Vk`	Compare less than halfword
ST 6C80	`lt.w Vj,Vk`	Compare less than word

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
ST 6CC0	`lt.l Vj,Vk`	Compare less than longword
ST 6D00	`lt.b Sj,Vk`	Compare less than byte
ST 6D40	`lt.h Sj,Vk`	Compare less than halfword
ST 6D80	`lt.w Sj,Vk`	Compare less than word
ST 6DC0	`lt.l Sj,Vk`	Compare less than longword
ST 6E00	`neg.b Vj,Vk`	Negate vector/vector integer byte
ST 6E40	`neg.h Vj,Vk`	Negate vector/vector integer halfword
ST 6E80	`neg.w Vj,Vk`	Negate vector/vector integer word
ST 6EC0	`neg.l Vj,Vk`	Negate vector/vector integer longword
ST 6F00	`neg.b Sj,Sk`	Negate scalar/scalar integer byte
ST 6F40	`neg.h Sj,Sk`	Negate scalar/scalar integer halfword
ST 6F80	`neg.w Sj,Sk`	Negate scalar/scalar integer word
ST 6FC0	`neg.l Sj,Sk`	Negate scalar/scalar integer longword
ST 7000	`nop`	No operation (branch never)
ST 7100	`br`	Branch always
ST 7200	`bri.f`	Branch on ION false
ST 7300	`bri.t`	Branch on ION true
ST 7400	`bra.f`	Branch on address carry false
ST 7500	`bra.t`	Branch on address carry true
ST 7600	`brs.f`	Branch on scalar carry false
ST 7700	`brs.t`	Branch on scalar carry true
ST 7800	`ldvi.b Vj,Vk`	Index load vector byte
ST 7840	`ldvi.h Vj,Vk`	Index load vector halfword
ST 7880	`ldvi.s Vj,Vk`	Index load vector single float
ST 7880	`ldvi.w Vj,Vk`	Index load vector word
ST 78C0	`ldvi.d Vj,Vk`	Index load vector double float

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
ST 78C0	`ldvi.l Vj,Vk`	Index load vector longword
ST 7900	`cvtw.s Vj,Vk`	Convert word to single float
ST 7940	`cvts.w Vj,Vk`	Convert single float to word
ST 7980	`cvtw.l Vj,Vk`	Convert word to longword
ST 79C0	`cvtl.w Vj,Vk`	Convert longword to word
ST 7A00	`stvi.b Vk,Vj`	Index store vector byte
ST 7A40	`stvi.h Vk,Vj`	Index store vector halfword
ST 7A80	`stvi.s Vk,Vj`	Index store vector single
ST 7A80	`stvi.w Vk,Vj`	Index store vector word
ST 7AC0	`stvi.d Vk,Vj`	Index store vector double
ST 7AC0	`stvi.l Vk,Vj`	Index store vector longword
ST 7B00	`stvi.b Sk,Vj`	Scalar index store vector byte
ST 7B40	`stvi.h Sk,Vj`	Scalar index store vector halfword
ST 7B80	`stvi.s Sk,Vj`	Scalar index store vector single float
ST 7B80	`stvi.w Sk,Vj`	Scalar index store vector word
ST 7BC0	`stvi.d Sk,Vj`	Scalar index store vector double float
ST 7BC0	`stvi.l Sk,Vj`	Scalar index store vector longword
ST 7C00	`ldsdr Ak`	Load process SDRs
ST 7C08	`ldkdr Ak`	Load all eight SDRs
ST 7C10	`ln.s Sk`	Natural logarithm of a single precision number
ST 7C18	`ln.d Sk`	Natural logarithm of a double precision number
ST 7C20	`patu`	Purge the entire ATU
ST 7C28	`pate Ak`	Purge ATU entry
ST 7C30	`pich`	Purge the Icache
ST 7C38	`plch`	Purge the Lcache
ST 7C40	`mov PSW,Ak`	Store the PSW into an address register

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
ST 7C48	`mov Ak,PSW`	Load an address register into the PSW
ST 7C50	`mov PC,Ak`	Load the next PC address
ST 7C58	`idle Sk`	Idle the CPU
ST 7C60	`mov ITR,Sk`	Move the ITC
ST 7C68	`mov Sk,ITR`	Load NITC
ST 7C78	`mov Sk,ITSR`	Load ITSR with a scalar
ST 7C80	`rtnq`	Pop the PC and jump
ST 7C88	`cfork`	Clear a fork
ST 7C90	`rtn`	Return from subroutine call
ST 7C98	`wfork`	Wait for a fork
ST 7CA0	`join`	Join all threads
ST 7CA8	`rtnc`	Return from a context block
ST 7CB0	`exp.s Sk`	Exponent of a single float
ST 7CB8	`exp.d Sk`	Exponent of a double float
ST 7CC0	`sin.s Sk`	Sine of a single precision number
ST 7CC8	`sin.d Sk`	Sine of a double precision number
ST 7CE0	`cos.s Sk`	Cosine of a single-precision number
ST 7CE8	`cos.d Sk`	Cosine of a double-precision number
ST 7D00	`psh.w Ak`	Push an address register
ST 7D10	`pop.w Ak`	Pop word into address register
ST 7D20	`psh.w Sk`	Push Sk<31..0> onto the stack
ST 7D28	`psh.l Sk`	Push Sk<63..0> onto the stack
ST 7D30	`pop.w Sk`	Pop Sk <31..0> from the stack
ST 7D38	`pop.l Sk`	Pop Sk <63..0> from the stack
ST 7D40	`eni`	Enable interrupts; set ION to 1
ST 7D48	`dsi`	Disable interrupts; reset ION to 0

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
ST 7D50	`bkpt`	Breakpoint
ST 7D58	`msync`	Synchronize stores/loads to memory
ST 7D60	`mski Sk`	Mask out interrupt
ST 7D68	`xmti Sk`	Transmit interrupt
ST 7D70	`mov Sk,VV`	Move scalar to vector valid flag
ST 7D78	`tstvv`	Test value of vector valid flag
ST 7D80	`mov VS,Ak`	Move VS to Ak
ST 7D88	`mov Ak,VS`	Move Ak to VS
ST 7D90	`mov VL,Ak`	Move VL to Ak
ST 7D98	`mov Ak,VL`	Move Ak to VL
ST 7DA0	`mov.w VS,Sk`	Move VS to Sk
ST 7DA8	`mov.w Sk,VS`	Move Sk to VS
ST 7DB0	`mov.w VL,Sk`	Move VL to Sk
ST 7DB8	`mov.w Sk,VL`	Move Sk to VL
ST 7DC0	`diag Ak`	Execute nonstandard microcode sequence
ST 7DC8	`pbkpt`	Force process breakpoint exception
ST 7DD0	`sqrt.s Sk`	Square root of a single float
ST 7DD8	`sqrt.d Sk`	Square root of a double float
ST 7DE0	`casr`	Compare and swap
ST 7DF0	`atan.s Sk`	Arc-tangent of a single float
ST 7DF8	`atan.d Sk`	Arc-tangent of a double float
ST 7E00	`sum.b Vk`	Sum a vector of bytes
ST 7E08	`sum.h Vk`	Sum a vector of halfwords
ST 7E10	`sum.w Vk`	Sum a vector of words
ST 7E18	`sum.l Vk`	Sum a vector of longwords
ST 7E20	`all Sk`	AND reduce a vector

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
ST 7E20	`all Vk`	AND reduce a vector
ST 7E28	`any Sk`	OR reduce a vector
ST 7E28	`any Vk`	OR reduce a vector
ST 7E30	`parity Vk`	Exclusive OR reduce a vector
ST 7E40	`max.b Vk`	Maximum of a vector of bytes
ST 7E48	`max.h Vk`	Maximum of a vector of halfwords
ST 7E50	`max.w Vk`	Maximum of a vector of words
ST 7E58	`max.l Vk`	Maximum of a vector of longwords
ST 7E60	`min.b Vk`	Minimum of a vector of bytes
ST 7E68	`min.h Vk`	Minimum of a vector of halfwords
ST 7E70	`min.w Vk`	Minimum of a vector of words
ST 7E78	`min.l Vk`	Minimum of a vector of longwords
ST 7E80	`sum.s Vk`	Sum a vector of single float
ST 7E88	`sum.d Vk`	Sum a vector of double float
ST 7E90	`prod.s Vk`	Multiply reduce a vector of single float
ST 7E98	`prod.d Vk`	Multiply reduce a vector of double float
ST 7EA0	`max.s Vk`	Maximum of a vector of single float
ST 7EA8	`max.d Vk`	Maximum of a vector of double float
ST 7EB0	`min.s Vk`	Minimum of a vector of single float
ST 7EB8	`min.d Vk`	Minimum of a vector of double float
ST 7EC0	`prod.b Vk`	Multiply reduce a vector of bytes
ST 7EC8	`prod.h Vk`	Multiply reduce a vector of halfwords
ST 7ED0	`prod.w Vk`	Multiply reduce a vector of words
ST 7ED8	`prod.l Vk`	Multiply reduce a vector of longwords
ST 7E0E	`plc.f VM,Sk`	Load the number of zeros in VM into Sk
ST 7E08	`plc.t VM,Sk`	Load the number of ones in VM into Sk

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
ST 8000	`mov Vi,Sj,Sk`	Move a vector element to a scalar
ST 8200	`mov Si,Sj,Vk`	Move a scalar to a vector element
ST 8400	`merg.t Vi,Vj,Vk`	Merge vector/vector
ST 8600	`mask.t Vi,Vj,Vk`	Mask vector/vector
ST 8800	`merg.f Vi,Sj,Vk`	Merge vector/scalar (!VM)
ST 8A00	`mask.f Vi,Sj,Vk`	Mask vector/scalar (!VM)
ST 8C00	`merg.t Vi,Sj,Vk`	Merge vector/scalar (VM)
ST 8E00	`mask.t Vi,Sj,Vk`	Mask vector/scalar (VM)
ST 9000	`mul.s Vi,Vj,Vk`	Multiply vector/vector single float
ST 9200	`mul.d Vi,Vj,Vk`	Multiply vector/vector double float
ST 9400	`div.s Vi,Vj,Vk`	Divide vector/vector single float
ST 9600	`div.d Vi,Vj,Vk`	Divide vector/vector double float
ST 9800	`mul.s Vi,Sj,Vk`	Multiply vector/scalar single float
ST 9A00	`mul.d Vi,Sj,Vk`	Multiply vector/scalar double float
ST 9C00	`div.s Vi,Sj,Vk`	Divide vector/scalar single float
ST 9E00	`div.d Vi,Sj,Vk`	Divide vector/scalar double float
ST A000	`and Vi,Vj,Vk`	AND two vectors
ST A200	`or Vi,Vj,Vk`	OR two vectors
ST A400	`xor Vi,Vj,Vk`	Exclusive OR two vectors
ST A600	`shf Vi,Vj,Vk`	Shift vector/vector
ST A800	`and Vi,Sj,Vk`	AND vector/scalar
ST AA00	`or Vi,Sj,Vk`	OR vector/scalar
ST AC00	`xor Vi,Sj,Vk`	Exclusive OR vector/scalar
ST AE00	`shf Vi,Sj,Vk`	Shift a vector accumulator
ST B000	`add.s Vi,Vj,Vk`	Add vector/vector single float
ST B200	`add.d Vi,Vj,Vk`	Add vector/vector double float

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
ST B400	`sub.s Vi,Vj,Vk`	Subtract vector/vector single float
ST B600	`sub.d Vi,Vj,Vk`	Subtract vector/vector double float
ST B800	`add.s Vi,Sj,Vk`	Add vector/scalar single float
ST BA00	`add.d Vi,Sj,Vk`	Add vector/scalar double float
ST BC00	`sub.s Vi,Sj,Vk`	Subtract vector/scalar single float
ST BE00	`sub.d Vi,Sj,Vk`	Subtract vector/scalar double float
ST C000	`add.b Vi,Vj,Vk`	Add vector/vector integer byte
ST C200	`add.h Vi,Vj,Vk`	Add vector/vector integer halfword
ST C400	`add.w Vi,Vj,Vk`	Add vector/vector integer word
ST C600	`add.l Vi,Vj,Vk`	Add vector/vector integer longword
ST C800	`add.b Vi,Sj,Vk`	Add vector/scalar integer byte
ST CA00	`add.h Vi Sj,Vk`	Add vector/scalar integer halfword
ST CC00	`add.w Vi,Sj,Vk`	Add vector/scalar integer word
ST CE00	`add.l Vi,Sj,Vk`	Add vector/scalar integer longword
ST D000	`sub.b Vi,Vj,Vk`	Subtract vector/vector integer byte
ST D200	`sub.h Vi,Vj,Vk`	Subtract vector/vector integer halfword
ST D400	`sub.w Vi,Vj,Vk`	Subtract vector/vector integer word
ST D600	`sub.l Vi,Vj,Vk`	Subtract vector/vector integer longword
ST D800	`sub.b Vi,Sj,Vk`	Subtract vector/scalar integer byte
ST DA00	`sub.h Vi,Sj,Vk`	Subtract vector/scalar integer halfword
ST DC00	`sub.w Vi,Sj,Vk`	Subtract vector/scalar integer word
ST DE00	`sub.l Vi,Sj,Vk`	Subtract vector/scalar integer longword
ST E000	`mul.b Vi,Vj,Vk`	Multiply vector/vector integer byte
ST E200	`mul.h Vi,Vj,Vk`	Multiply vector/vector integer halfword
ST E400	`mul.w Vi,Vj,Vk`	Multiply vector/vector integer word
ST E600	`mul.l Vi,Vj,Vk`	Multiply vector/vector integer longword

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
ST E800	`mul.b Vi,Sj,Vk`	Multiply vector/scalar integer byte
ST EA00	`mul.h Vi,Sj,Vk`	Multiply vector/scalar integer halfword
ST EC00	`mul.w Vi,Sj,Vk`	Multiply vector/scalar integer word
ST EE00	`mul.l Vi,Sj,Vk`	Multiply vector/scalar integer longword
ST F000	`div.b Vi,Vj,Vk`	Divide vector/vector integer byte
ST F200	`div.h Vi,Vj,Vk`	Divide vector/vector integer halfword
ST F400	`div.w Vi,Vj,Vk`	Divide vector/vector integer word
ST F600	`div.l Vi,Vj,Vk`	Divide vector/vector integer longword
ST F800	`div.b Vi,Sj,Vk`	Divide vector/scalar integer byte
ST FA00	`div.h Vi,Sj,Vk`	Divide vector/scalar integer halfword
ST FC00	`div.w Vi,Sj,Vk`	Divide vector/scalar integer word
ST FE00	`div.l Vi,Sj,Vk`	Divide vector/scalar integer longword
E0 0100	`tst` *Ceffa*	Test communication register lock bit
E0 0200	`lck` *Ceffa*	Lock communication register
E0 0300	`ulk` *Ceffa*	Unlock communication register
E0 0400	`ldea` *effa*`,Sk`	Load effective address/scalar
E0 0500	`spawn` *effa*`,Ak`	Spawn a fork
E0 0600	`ldcmr` *effa*`,Ak`	Load communication registers
E0 0700	`stcmr Ak,` *effa*	Store communication registers
E0 0800	`popr Ak,` *effa*	Pop resource/address register
E0 0900	`pshr Ak,` *effa*	Push address register/resource
E0 0A00	`rcvr.w` *effa*`,Ak`	Receive address register/resource
E0 0C00	`sndr.w Ak,` *effa*	Send address register/resource
E0 0D00	`sndr.l Sk,` *effa*	Send scalar register/resource
E0 0E00	`rcvr.l` *effa*`,Sk`	Receive scalar register/resource
E0 2000	`putr.w Ak,` *effa*	Put address/resource

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
E0 2100	`putr.l Sk,`*effa*	Put scalar/resource
E0 2200	`getr.w`*effa*`,Ak`	Get resource/address
E0 2300	`getr.l`*effa*`,Sk`	Get resource/scalar
E0 2400	`ste.b.f Sk,`*effa*	Store extended scalar byte (!VM)
E0 2500	`ste.h.f Sk,`*effa*	Store extended scalar halfword (!VM)
E0 2600	`ste.s.f Sk,`*effa*	Store extended scalar single (!VM)
E0 2600	`ste.w.f Sk,`*effa*	Store extended scalar word (!VM)
E0 2700	`ste.d.f Sk,`*effa*	Store extended scalar double (!VM)
E0 2700	`ste.l.f Sk,`*effa*	Store extended scalar longword (!VM)
E0 2800	`matr.w Ak,`*effa*	Match address/resource
E0 2900	`mat.w Ak,`*Ceffa*	Match address/communication
E0 2A00	`get.w`*Ceffa*`,Ak`	Get communication/address
E0 2B00	`rcv.w`*Ceffa*`,Ak`	Receive communication/address
E0 2D00	`inc.w`*Ceffa*`,Ak`	Increment communication/address
E0 2E00	`put.w Ak,`*Ceffa*	Put address/communication
E0 2F00	`snd.w Ak,`*Ceffa*	Send address/communication
E0 3000	`matr.l Sk,`*effa*	Match scalar/resource
E0 3100	`mat.l Sk,`*Ceffa*	Match scalar/communication
E0 3200	`get.l`*Ceffa*`,Sk`	Get communication/scalar
E0 3300	`rcv.l`*Ceffa*`,Sk`	Receive communication/scalar
E0 3500	`inc.l`*Ceffa*`,Sk`	Increment communication/scalar
E0 3600	`put.l Sk,`*Ceffa*	Put scalar/communication
E0 3700	`snd.l Sk,`*Ceffa*	Send scalar/communication
E0 3800	`ld.b.f`*effa*`,Vk`	Load vector byte (!VM)
E0 3900	`ld.h.f`*effa*`,Vk`	Load vector halfword (!VM)
E0 3A00	`ld.s.f`*effa*`,Vk`	Load vector single float (!VM)

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
E0 3A00	`ld.w.f` *effa*`,Vk`	Load vector word (!VM)
E0 3B00	`ld.d.f` *effa*`,Vk`	Load vector double float (!VM)
E0 3B00	`ld.l.f` *effa*`,Vk`	Load vector longword (!VM)
E0 3C00	`st.b.f Vk,` *effa*	Store vector byte (!VM)
E0 3D00	`st.h.f Vk,` *effa*	Store vector halfword (!VM)
E0 3E00	`st.s.f Vk,` *effa*	Store vector single float (!VM)
E0 3E00	`st.w.f Vk,` *effa*	Store vector word (!VM)
E0 3F00	`st.d.f Vk,` *effa*	Store vector double float (!VM)
E0 3F00	`st.l.f Vk,` *effa*	Store vector longword (!VM)
E0 4000	`cvtw.b Vj,Vk`	Convert word to byte
E0 4040	`cvtw.h Vj,Vk`	Convert word to halfword
E0 4080	`cvtb.w Vj,Vk`	Convert byte to word
E0 40C0	`cvth.w Vj,Vk`	Convert halfword to word
E0 4100	`cvtw.b.f Vj,Vk`	Convert word to byte (!VM)
E0 4140	`cvtw.h.f Vj,Vk`	Convert word to halfword (!VM)
E0 4180	`cvtb.w.f Vj,Vk`	Convert byte to word (!VM)
E0 41C0	`cvth.w.f Vj,Vk`	Convert halfword to word (!VM)
E0 4200	`cvts.l Vj,Vk`	Convert single float to longword
E0 4240	`cvtd.w Vj,Vk`	Convert double to word float
E0 4280	`cvtl.s Vj,Vk`	Convert longword to single float
E0 42C0	`cvtw.d Vj,Vk`	Convert word to double float
E0 4300	`cvts.l.f Vj,Vk`	Convert single to longword (!VM)
E0 4340	`cvtd.w.f Vj,Vk`	Convert double to word (!VM)
E0 4380	`cvtl.s.f Vj,Vk`	Convert longword to single (!VM)
E0 43C0	`cvtw.d.f Vj,Vk`	Convert word to double (!VM)
E0 4400	`enal Sj,Sk`	Enable local CPU interrupt

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
E0 4440	`shf.w Sj,Sk`	Shift a scalar word
E0 4480	`enag Sj,Sk`	Enable all global CPU interrupts
E0 4500	`cvtw.d Sj,Sk`	Convert word to double float
E0 4540	`cvtd.w Sj,Sk`	Convert double float to word
E0 4580	`mov BE(Sj),Sk`	Move BE(scalar) to a scalar
E0 45C0	`mov Sk,BE(Sj)`	Move scalar to BE(scalar)
E0 4700	`frint.s Sj,Sk`	Integerize float single scalar
E0 4740	`frint.d Sj,Sk`	Integerize float double scalar
E0 5880	`frint.s Vj,Vk`	Integerize float single vector
E0 58C0	`frint.d Vj,Vk`	Integerize float double vector
E0 5980	`frint.s.f Vj,Vk`	Integerize single vector (!VM)
E0 59C0	`frint.d.f Vj,Vk`	Integerize double vector (!VM)
E0 5D00	`sqrt.s Vj,Vk`	Square root single vector/vector
E0 5D40	`sqrt.d Vj,Vk`	Square root double vector/vector
E0 5F00	`sqrt.s.f Vj,Vk`	Square root single (!VM)
E0 5F40	`sqrt.d.f Vj,Vk`	Square root double (!VM)
E0 6000	`cvtd.s.f Vj,Vk`	Convert double to single (!VM)
E0 6040	`cvts.d.f Vj,Vk`	Convert single to double (!VM)
E0 6080	`cvtl.d.f Vj,Vk`	Convert longword to double (!VM)
E0 60C0	`cvtd.l.f Vj,Vk`	Convert double to longword (!VM)
E0 6200	`tzc.f Vj,Vk`	Trailing zero count vector (!VM)
E0 6240	`lop.f Vj,Vk`	Leading ones position vector (!VM)
E0 6280	`xpnd.f Vj,Vk`	Expand a vector (!VM)
E0 62C0	`not.f Vj,Vk`	Complement a vector (!VM)
E0 6300	`shf.f Sj,Vk`	Shift vector/scalar (!VM)
E0 6340	`plc.t.f Vj,Vk`	Population count of vector (!VM)

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
E0 6400	`eq.s.f Vj,Vk`	Compare equal single (!VM)
E0 6440	`eq.d.f Vj,Vk`	Compare equal double (!VM)
E0 6480	`neg.s.f Vj,Vk`	Negate single (!VM)
E0 64C0	`neg.d.f Vj,Vk`	Negate double (!VM)
E0 6500	`eq.s.f Sj,Vk`	Compare equal single (!VM)
E0 6540	`eq.d.f Sj,Vk`	Compare equal double (!VM)
E0 6600	`le.s.f Vj,Vk`	Compare less than or equal single (!VM)
E0 6640	`le.d.f Vj,Vk`	Compare less than or equal double (!VM)
E0 6680	`lt.s.f Vj,Vk`	Compare less than single (!VM)
E0 66C0	`lt.d.f Vj,Vk`	Compare less than double (!VM)
E0 6700	`le.s.f Sj,Vk`	Compare less than or equal single (!VM)
E0 6740	`le.d.f Sj,Vk`	Compare less than or equal double (!VM)
E0 6780	`lt.s.f Sj,Vk`	Compare less than single (!VM)
E0 67C0	`lt.d.f Sj,Vk`	Compare less than double (!VM)
E0 6800	`eq.b.f Vj,Vk`	Compare equal byte (!VM)
E0 6840	`eq.h.f Vj,Vk`	Compare equal halfword (!VM)
E0 6880	`eq.w.f Vj,Vk`	Compare equal word (!VM)
E0 68C0	`eq.l.f Vj,Vk`	Compare equal long (!VM)
E0 6900	`eq.b.f Sj,Vk`	Compare equal byte (!VM)
E0 6940	`eq.h.f Sj,Vk`	Compare equal halfword (!VM)
E0 6980	`eq.w.f Sj,Vk`	Compare equal word (!VM)
E0 69C0	`eq.l.f Sj,Vk`	Compare equal long (!VM)
E0 6A00	`le.b.f Vj,Vk`	Compare less than or equal byte (!VM)
E0 6A40	`le.h.f Vj,Vk`	Compare less than or equal half (!VM)
E0 6A80	`le.w.f Vj,Vk`	Compare less than or equal word (!VM)
E0 6AC0	`le.l.f Vj,Vk`	Compare less than or equal long (!VM)

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
E0 6B00	`le.b.f Sj,Vk`	Compare less than or equal byte (!VM)
E0 6B40	`le.h.f Sj,Vk`	Compare less than or equal half (!VM)
E0 6B80	`le.w.f Sj,Vk`	Compare less than or equal word (!VM)
E0 6BC0	`le.l.f Sj,Vk`	Compare less than or equal long (!VM)
E0 6C00	`lt.b.f Vj,Vk`	Compare less than byte (!VM)
E0 6C40	`lt.h.f Vj,Vk`	Compare less than halfword (!VM)
E0 6C80	`lt.w.f Vj,Vk`	Compare less than word (!VM)
E0 6CC0	`lt.l.f Vj,Vk`	Compare less than long (!VM)
E0 6D00	`lt.b.f Sj,Vk`	Compare less than byte (!VM)
E0 6D40	`lt.h.f Sj,Vk`	Compare less than halfword (!VM)
E0 6D80	`lt.w.f Sj,Vk`	Compare less than word (!VM)
E0 6DC0	`lt.l.f Sj,Vk`	Compare less than long (!VM)
E0 6E00	`neg.b.f Vj,Vk`	Negate byte vector (!VM)
E0 6E40	`neg.h.f Vj,Vk`	Negate halfword (!VM)
E0 6E80	`neg.w.f Vj,Vk`	Negate word (!VM)
E0 6EC0	`neg.l.f Vj,Vk`	Negate longword (!VM)
E0 7800	`ldvi.b.f Vj,Vk`	Index load vector byte (!VM)
E0 7840	`ldvi.h.f Vj,Vk`	Index load vector halfword (!VM)
E0 7880	`ldvi.s.f Vj,Vk`	Index load vector single (!VM)
E0 7880	`ldvi.w.f Vj,Vk`	Index load vector word (!VM)
E0 78C0	`ldvi.d.f Vj,Vk`	Index load vector double (!VM)
E0 78C0	`ldvi.l.f Vj,Vk`	Index load vector longword (!VM)
E0 7900	`cvtw.s.f Vj,Vk`	Convert word to single (!VM)
E0 7940	`cvts.w.f Vj,Vk`	Convert single to word (!VM)
E0 7980	`cvtw.l.f Vj,Vk`	Convert word to longword (!VM)
E0 79C0	`cvtl.w.f Vj,V`	Convert longword to word (!VM)

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
E0 7A00	`stvi.b.f Vk,Vj`	Index store vector byte (!VM)
E0 7A40	`stvi.h.f Vk,Vj`	Index store vector halfword (!VM)
E0 7A80	`stvi.s.f Vk,Vj`	Index store vector single (!VM)
E0 7A80	`stvi.w.f Vk,Vj`	Index store vector word (!VM)
E0 7AC0	`stvi.d.f Vk,Vj`	Index store vector double (!VM)
E0 7AC0	`stvi.l.f Vk,Vj`	Index store vector longword (!VM)
E0 7B00	`stvi.b.f Sk,Vj`	Scalar index store vector byte (!VM)
E0 7B40	`stvi.h.f Sk,Vj`	Scalar index store vector half (!VM)
E0 7B80	`stvi.s.f Sk,Vj`	Scalar index store vector single (!VM)
E0 7B80	`stvi.w.f Sk,Vj`	Scalar index store vector word (!VM)
E0 7BC0	`stvi.d.f Sk,Vj`	Scalar index store vector double (!VM)
E0 7BC0	`stvi.l.f Sk,Vj`	Scalar index store vector long (!VM)
E0 7C00	`mov Sk,CIR`	Move scalar to CIR
E0 7C08	`mov CIR,Sk`	Move CIR a scalar
E0 7C10	`mov TOC,Sk`	Move TOC to a scalar
E0 7C18	`mov CPUID,Sk`	Move CPU identification to scalar
E0 7C20	`mov Sk,TTR`	Move scalar to TTR
E0 7C28	`mov TTR,Sk`	Move TTR /scalar
E0 7C30	`mov Sk,TOC`	Move a scalar to TOC
E0 7C38	`ctrsg`	Move scalar to CPU timer
E0 7C40	`mov Sk,VMU`	Load VM<127..64> from Sk
E0 7C48	`mov VMU,Sk`	Load Sk from VM<127..64>
E0 7C50	`mov Sk,VML`	Load VM<63..0> from Sk
E0 7C58	`mov VML,Sk`	Load Sk from VM<63..0>
E0 7C60	`mov Sk,ICR`	Move Scalar to ICR
E0 7C68	`mov ICR,Sk`	Move ICR to scalar

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
E0 7C70	`mov Sk,TCPU`	Move Scalar to TCPU register
E0 7C78	`mov TCPU,Sk`	Move the TCPU Register to Scalar
E0 7C80	`eni_idle Sk`	Enable interrupts and idle the CPU
E0 7C88	`eni_rtn`	Return from subroutine
E0 7C90	`pref`	Purge reference bits
E0 7CB0	`mov Sk,TID`	Load TID from scalar
E0 7CB8	`mov TID,Sk`	Load scalar with TID
E0 7E00	`sum.b.f Vk`	Sum a vector of bytes (!VM)
E0 7E08	`sum.h.f Vk`	Sum a vector of halfwords (!VM)
E0 7E10	`sum.w.f Vk`	Sum a vector of words (!VM)
E0 7E18	`sum.l.f Vk`	Sum a vector of longwords (!VM)
E0 7E20	`all.f Sk`	AND reduce a vector (!VM)
E0 7E20	`all.f Vk`	AND reduce a vector (!VM)
E0 7E28	`any.f Sk`	OR reduce a vector (!VM)
E0 7E28	`any.f Vk`	OR reduce a vector (!VM)
E0 7E30	`parity.f Vk`	Exclusive OR reduce vector (!VM)
E0 7E40	`max.b.f Vk`	Maximum of vector of bytes (!VM)
E0 7E48	`max.h.f Vk`	Maximum of vector of halfwords (!VM)
E0 7E50	`max.w.f Vk`	Maximum of vector of words (!VM)
E0 7E58	`max.l.f Vk`	Maximum of vector of longwords (!VM)
E0 7E60	`min.b.f Vk`	Minimum of vector of bytes (!VM)
E0 7E68	`min.h.f Vk`	Minimum of vector of halfwords (!VM)
E0 7E70	`min.w.f Vk`	Minimum of vector of words (!VM)
E0 7E78	`min.l.f Vk`	Minimum of vector of longwords (!VM)
E0 7E80	`sum.s.f Vk`	Sum a vector of single (!VM)
E0 7E88	`sum.d.f Vk`	Sum a vector of double (!VM)

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
E0 7E90	`prod.s.f Vk`	Multiply reduce single vector (!VM)
E0 7E98	`prod.d.f Vk`	Multiply reduce double vector (!VM)
E0 7EA0	`max.s.f Vk`	Maximum of vector of singles (!VM)
E0 7EA8	`max.d.f Vk`	Maximum of vector of doubles (!VM)
E0 7EB0	`min.s.f Vk`	Minimum of vector of singles (!VM)
E0 7EB8	`min.d.f Vk`	Minimum of vector of doubles (!VM)
E0 7EC0	`prod.b.f Vk`	Multiply reduce byte vector (!VM)
E0 7EC8	`prod.h.f Vk`	Multiply reduce halfword vector (!VM)
E0 7ED0	`prod.w.f Vk`	Multiply reduce word vector (!VM)
E0 7ED8	`prod.l.f Vk`	Multiply reduce longword vector (!VM)
E0 8000	`sub.s Si,Vj,Vk`	Subtract scalar/vector single float
E0 8200	`sub.d Si,Vj,Vk`	Subtract scalar/vector double float
E0 8400	`div.s Si,Vj,Vk`	Divide scalar/vector single float
E0 8600	`div.d Si,Vj,Vk`	Divide scalar/vector double float
E0 8800	`sub.s.f Si,Vj,Vk`	Subtract scalar/vector single (!VM)
E0 8A00	`sub.d.f Si,Vj,Vk`	Subtract scalar/vector double (!VM)
E0 8C00	`div.s.f Si,Vj,Vk`	Divide scalar/vector single (!VM)
E0 8E00	`div.d.f Si,Vj,Vk`	Divide scalar/vector double (!VM)
E0 9000	`mul.s.f Vi,Vj,Vk`	Multiply single vectors (!VM)
E0 9200	`mul.d.f Vi,Vj,Vk`	Multiply double vectors (!VM)
E0 9400	`div.s.f Vi,Vj,Vk`	Divide single vectors (!VM)
E0 9600	`div.d.f Vi,Vj,Vk`	Divide double vectors (!VM)
E0 9800	`mul.s.f Vi,Sj,Vk`	Multiply vector/scalar single (!VM)
E0 9A00	`mul.d.f Vi,Sj,Vk`	Multiply vector/scalar double (!VM)
E0 9C00	`div.s.t Vi,Sj,Vk`	Divide vector/scalar single (VM)
E0 9E00	`div.d.t Vi,Sj,Vk`	Divide vector/scalar double (VM)

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
E0 A000	`and.f Vi,Vj,Vk`	AND two vectors (!VM)
E0 A200	`or.f Vi,Vj,Vk`	OR two vectors (!VM)
E0 A400	`xor.f Vi,Vj,Vk`	Exclusive OR two vectors (!VM)
E0 A600	`shf.f Vi,Vj,Vk`	Shift vector/vector (!VM)
E0 A800	`and.f Vi,Sj,Vk`	AND vector/scalar (!VM)
E0 AA00	`or.f Vi,Sj,Vk`	OR vector/scalar (!VM)
E0 AC00	`xor.f Vi,Sj,Vk`	Exclusive OR vector/scalar (!VM)
E0 AE00	`shf.f Vi,Sj,Vk`	Shift vector/scalar (!VM)
E0 B000	`add.s.f Vi,Vj,Vk`	Add vector/vector single (!VM)
E0 B200	`add.d.f Vi,Vj,Vk`	Add vector/vector double (!VM)
E0 B400	`sub.s.f Vi,Vj,Vk`	Subtract single vectors (!VM)
E0 B600	`sub.d.f Vi,Vj,Vk`	Subtract double vectors (!VM)
E0 B800	`add.s.f Vi,Sj,Vk`	Add vector/scalar single (!VM)
E0 BA00	`add.d.f Vi,Sj,Vk`	Add vector/scalar double (!VM)
E0 BC00	`sub.s.f Vi,Sj,Vk`	Subtract vector/scalar single (!VM)
E0 BE00	`sub.d.f Vi,Sj,Vk`	Subtract vector/scalar double (!VM)
E0 C000	`add.b.f Vi,Vj,Vk`	Add vector/vector byte (!VM)
E0 C200	`add.h.f Vi,Vj,Vk`	Add vector/vector halfword (!VM)
E0 C400	`add.w.f Vi,Vj,Vk`	Add vector/vector word (!VM)
E0 C600	`add.l.f Vi,Vj,Vk`	Add vector/vector longword (!VM)
E0 C800	`add.b.f Vi,Sj,Vk`	Add vector/scalar byte (!VM)
E0 CA00	`add.h.f Vi Sj,Vk`	Add vector/scalar halfword (!VM)
E0 CC00	`add.w.f Vi,Sj,Vk`	Add vector/scalar word (!VM)
E0 CE00	`add.l.f Vi,Sj,Vk`	Add vector/scalar longword (!VM)
E0 D000	`sub.b.f Vi,Vj,Vk`	Subtract byte vectors (!VM)
E0 D200	`sub.h.f Vi,Vj,Vk`	Subtract halfword vectors (!VM)

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
E0 D400	`sub.w.f Vi,Vj,Vk`	Subtract word vectors (!VM)
E0 D600	`sub.l.f Vi,Vj,Vk`	Subtract longword vectors (!VM)
E0 D800	`sub.b.f Vi,Sj,Vk`	Subtract vector/scalar byte (!VM)
E0 DA00	`sub.h.f Vi,Sj,Vk`	Subtract vector/scalar halfword (!VM)
E0 DC00	`sub.w.f Vi,Sj,Vk`	Subtract vector/scalar word (!VM)
E0 DE00	`sub.l.f Vi,Sj,Vk`	Subtract vector/scalar longword (!VM)
E0 E000	`mul.b.f Vi,Vj,Vk`	Multiply byte vectors (!VM)
E0 E200	`mul.h.f Vi,Vj,Vk`	Multiply halfword vectors (!VM)
E0 E400	`mul.w.f Vi,Vj,Vk`	Multiply word vectors (!VM)
E0 E600	`mul.l.f Vi,Vj,Vk`	Multiply longword vectors (!VM)
E0 E800	`mul.b.f Vi,Sj,Vk`	Multiply vector/scalar byte (!VM)
E0 EA00	`mul.h.f Vi,Sj,Vk`	Multiply vector/scalar halfword (!VM)
E0 EC00	`mul.w.f Vi,Sj,Vk`	Multiply vector/scalar word (!VM)
E0 EE00	`mul.l.f Vi,Sj,Vk`	Multiply vector/scalar longword (!VM)
E0 F000	`div.b.f Vi,Vj,Vk`	Divide byte vectors (!VM)
E0 F200	`div.h.f Vi,Vj,Vk`	Divide halfword vectors (!VM)
E0 F400	`div.w.f Vi,Vj,Vk`	Divide word vectors (!VM)
E0 F600	`div.l.f Vi,Vj,Vk`	Divide longword vectors (!VM)
E0 F800	`div.b.f Vi,Sj,Vk`	Divide vector/scalar byte (!VM)
E0 FA00	`div.h.f Vi,Sj,Vk`	Divide vector/scalar halfword (!VM)
E0 FC00	`div.w.f Vi,Sj,Vk`	Divide vector/scalar word (!VM)
E0 FE00	`div.l.f Vi,Sj,Vk`	Divide vector/scalar longword (!VM)
E1 2400	`ste.b.t Sk,`*effa*	Store extended scalar byte (VM)
E1 2500	`ste.h.t Sk,`*effa*	Store extended scalar halfword (VM)
E1 2600	`ste.w.t Sk,`*effa*	Store extended scalar word (VM)
E1 2700	`ste.d.t Sk,`*effa*	Store extended scalar double (VM)

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
E1 2700	`ste.l.t Sk,`*effa*	Store extended scalar longword (VM)
E1 3800	`ld.b.t` *effa*`,Vk`	Load vector byte (VM)
E1 3900	`ld.h.t` *effa*`,Vk`	Load vector halfword (VM)
E1 3A00	`ld.s.t` *effa*`,Vk`	Load vector single float (VM)
E1 3A00	`ld.w.t` *effa*`,Vk`	Load vector word (VM)
E1 3B00	`ld.d.t` *effa*`,Vk`	Load vector double float (VM)
E1 3B00	`ld.l.t` *effa*`,Vk`	Load vector longword (VM)
E1 3C00	`st.b.t Vk,`*effa*	Store vector byte (VM)
E1 3D00	`st.h.t Vk,`*effa*	Store vector halfword (VM)
E1 3E00	`st.s.t Vk,`*effa*	Store vector single float (VM)
E1 3E00	`st.w.t Vk,`*effa*	Store vector word (VM)
E1 3F00	`st.d.t Vk,`*effa*	Store vector double float (VM)
E1 3F00	`st.l.t Vk,`*effa*	Store vector longword (VM)
E1 4100	`cvtw.b.t Vj,Vk`	Convert word to byte (VM)
E1 4140	`cvtw.h.t Vj,Vk`	Convert word to halfword (VM)
E1 4180	`cvtb.w.t Vj,Vk`	Convert byte to word (VM)
E1 41C0	`cvth.w.t Vj,Vk`	Convert halfword to word (VM)
E1 4300	`cvts.l.t Vj,Vk`	Convert single to longword (VM)
E1 4340	`cvtd.w.t Vj,Vk`	Convert double to word (VM)
E1 4380	`cvtl.s.t Vj,Vk`	Convert longword to single (VM)
E1 43C0	`cvtw.d.t Vj,Vk`	Convert word to double (VM)
E1 5980	`frint.s.t Vj,Vk`	Integerize single vector (VM)
E1 59C0	`frint.d.t Vj,Vk`	Integerize double vector (VM)
E1 5F00	`sqrt.s.t Vj,Vk`	Square root single (VM)
E1 5F40	`sqrt.d.t Vj,Vk`	Square root double (VM)
E1 6000	`cvtd.s.t Vj,Vk`	Convert double to single (VM)

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
E1 6040	`cvts.d.t Vj,Vk`	Convert single to double (VM)
E1 6080	`cvtl.d.t Vj,Vk`	Convert longword to double (VM)
E1 60C0	`cvtd.l.t Vj,Vk`	Convert double to longword (VM)
E1 6200	`tzc.t Vj,Vk`	Trailing zero count vector (VM)
E1 6240	`lop.t Vj,Vk`	Leading ones position vector (VM)
E1 6280	`xpnd.t Vj,Vk`	Expand a vector (VM)
E1 62C0	`not.t Vj,Vk`	Complement a vector (VM)
E1 6300	`shf.t Sj,Vk`	Shift vector/scalar (VM)
E1 6340	`plc.t.t Vj,Vk`	Population count of vector (VM)
E1 6400	`eq.s.t Vj,Vk`	Compare equal single (VM)
E1 6440	`eq.d.t Vj,Vk`	Compare equal double (VM)
E1 6480	`neg.s.t Vj,Vk`	Negate single (VM)
E1 64C0	`neg.d.t Vj,Vk`	Negate double (VM)
E1 6500	`eq.s.t Sj,Vk`	Compare equal single (VM)
E1 6540	`eq.d.t Sj,Vk`	Compare equal double (VM)
E1 6600	`le.s.t Vj,Vk`	Compare less than or equal single (VM)
E1 6640	`le.d.t Vj,Vk`	Compare less than or equal double (VM)
E1 6680	`lt.s.t Vj,Vk`	Compare less than single (VM)
E1 66C0	`lt.d.t Vj,Vk`	Compare less than double (VM)
E1 6700	`le.s.t Sj,Vk`	Compare less than or equal single (VM)
E1 6740	`le.d.t Sj,Vk`	Compare less than or equal double (VM)
E1 6780	`lt.s.t Sj,Vk`	Compare less than single (VM)
E1 67C0	`lt.d.t Sj,Vk`	Compare less than double (VM)
E1 6800	`eq.b.t Vj,Vk`	Compare equal byte (VM)
E1 6840	`eq.h.t Vj,Vk`	Compare equal halfword (VM)
E1 6880	`eq.w.t Vj,Vk`	Compare equal word (VM)

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
E1 68C0	`eq.l.t Vj,Vk`	Compare equal long (VM)
E1 6900	`eq.b.t Sj,Vk`	Compare equal byte (VM)
E1 6940	`eq.h.t Sj,Vk`	Compare equal halfword (VM)
E1 6980	`eq.w.t Sj,Vk`	Compare equal word (VM)
E1 69C0	`eq.l.t Sj,Vk`	Compare equal long (VM)
E1 6A00	`le.b.t Vj,Vk`	Compare less than or equal byte (VM)
E1 6A40	`le.h.t Vj,Vk`	Compare less than or equal half (VM)
E1 6A80	`le.w.t Vj,Vk`	Compare less than or equal word (VM)
E1 6AC0	`le.l.t Vj,Vk`	Compare less than or equal long (VM)
E1 6B00	`le.b.t Sj,Vk`	Compare less than or equal byte (VM)
E1 6B40	`le.h.t Sj,Vk`	Compare less than or equal half (VM)
E1 6B80	`le.w.t Sj,Vk`	Compare less than or equal word (VM)
E1 6BC0	`le.l.t Sj,Vk`	Compare less than or equal long (VM)
E1 6C00	`lt.b.t Vj,Vk`	Compare less than byte (VM)
E1 6C40	`lt.h.t Vj,Vk`	Compare less than halfword (VM)
E1 6C80	`lt.w.t Vj,Vk`	Compare less than word (VM)
E1 6CC0	`lt.l.t Vj,Vk`	Compare less than long (VM)
E1 6D00	`lt.b.t Sj,Vk`	Compare less than byte (VM)
E1 6D40	`lt.h.t Sj,Vk`	Compare less than halfword (VM)
E1 6D80	`lt.w.t Sj,Vk`	Compare less than word (VM)
E1 6DC0	`lt.l.t Sj,Vk`	Compare less than long (VM)
E1 6E00	`neg.b.t Vj,Vk`	Negate byte vector (VM)
E1 6E40	`neg.h.t Vj,Vk`	Negate halfword (VM)
E1 6E80	`neg.w.t Vj,Vk`	Negate word (VM)
E1 6EC0	`neg.l.t Vj,Vk`	Negate longword (VM)
E1 7800	`ldvi.b.t Vj,Vk`	Index load vector byte (VM)

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
E1 7840	`ldvi.h.t Vj,Vk`	Index load vector halfword (VM)
E1 7880	`ldvi.s.t Vj,Vk`	Index load vector single (VM)
E1 7880	`ldvi.w.t Vj,Vk`	Index load vector word (VM)
E1 78C0	`ldvi.d.t Vj,Vk`	Index load vector double (VM)
E1 78C0	`ldvi.l.t Vj,Vk`	Index load vector longword (VM)
E1 7900	`cvtw.s.t Vj,Vk`	Convert word to single (VM)
E1 7940	`cvts.w.t Vj,Vk`	Convert single to word (VM)
E1 7980	`cvtw.l.t Vj,Vk`	Convert word to longword (VM)
E1 79C0	`cvtl.w.t Vj,Vk`	Convert longword to word (VM)
E1 7A00	`stvi.b.t Vk,Vj`	Index store vector byte (VM)
E1 7A40	`stvi.h.t Vk,Vj`	Index store vector halfword (VM)
E1 7A80	`stvi.s.t Vk,Vj`	Index store vector single (VM)
E1 7A80	`stvi.w.t Vk,Vj`	Index store vector word (VM)
E1 7AC0	`stvi.d.t Vk,Vj`	Index store vector double (VM)
E1 7AC0	`stvi.l.t Vk,Vj`	Index store vector longword (VM)
E1 7B00	`stvi.b.t Sk,Vj`	Scalar index store vector byte (VM)
E1 7B40	`stvi.h.t Sk,Vj`	Scalar index store vector half (VM)
E1 7B80	`stvi.s.t Sk,Vj`	Scalar index store vector single (VM)
E1 7B80	`stvi.w.t Sk,Vj`	Scalar index store vector word (VM)
E1 7BC0	`stvi.d.t Sk,Vj`	Scalar index store vector double (VM)
E1 7BC0	`stvi.l.t Sk,Vj`	Scalar index store vector long (VM)
E1 7E00	`sum.b.t Vk`	Sum a vector of bytes (VM)
E1 7E08	`sum.h.t Vk`	Sum a vector of halfwords (VM)
E1 7E10	`sum.w.t Vk`	Sum a vector of words (VM)
E1 7E18	`sum.l.t Vk`	Sum a vector of longwords (VM)
E1 7E20	`all.t Sk`	AND reduce a vector (VM)

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
E1 7E20	`all.t Vk`	AND reduce a vector (VM)
E1 7E28	`any.t Sk`	OR reduce a vector (VM)
E1 7E28	`any.t Vk`	OR reduce a vector (VM)
E1 7E30	`parity.t Vk`	Exclusive OR reduce vector (VM)
E1 7E40	`max.b.t Vk`	Maximum of vector of bytes (VM)
E1 7E48	`max.h.t Vk`	Maximum of vector of halfwords (VM)
E1 7E50	`max.w.t Vk`	Maximum of vector of words (VM)
E1 7E58	`max.l.t Vk`	Maximum of vector of longwords (VM)
E1 7E60	`min.b.t Vk`	Minimum of vector of bytes (VM)
E1 7E68	`min.h.t Vk`	Minimum of vector of halfwords (VM)
E1 7E70	`min.w.t Vk`	Minimum of vector of words (VM)
E1 7E78	`min.l.t Vk`	Minimum of vector of longwords (VM)
E1 7E80	`sum.s.t Vk`	Sum a vector of single (VM)
E1 7E88	`sum.d.t Vk`	Sum a vector of double (VM)
E1 7E90	`prod.s.t Vk`	Multiply reduce single vector (VM)
E1 7E98	`prod.d.t Vk`	Multiply reduce double vector (VM)
E1 7EA0	`max.s.t Vk`	Maximum of vector of singles (VM)
E1 7EA8	`max.d.t Vk`	Maximum of vector of doubles (VM)
E1 7EB0	`min.s.t Vk`	Minimum of vector of singles (VM)
E1 7EB8	`min.d.t Vk`	Minimum of vector of doubles (VM)
E1 7EC0	`prod.b.t Vk`	Multiply reduce byte vector (VM)
E1 7EC8	`prod.h.t Vk`	Multiply reduce halfword vector (VM)
E1 7ED0	`prod.w.t Vk`	Multiply reduce word vector (VM)
E1 7ED8	`prod.l.t Vk`	Multiply reduce longword vector (VM)
E1 8800	`sub.s.t Si,Vj,Vk`	Subtract scalar/vector single (VM)
E1 8A00	`sub.d.t Si,Vj,Vk`	Subtract scalar/vector double (VM)

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
E1 8C00	`div.s.t Si,Vj,Vk`	Divide scalar/vector single (VM)
E1 8E00	`div.d.t Si,Vj,Vk`	Divide scalar/vector double (VM)
E1 9000	`mul.s.t Vi,Vj,Vk`	Multiply single vectors (VM)
E1 9200	`mul.d.t Vi,Vj,Vk`	Multiply double vectors (VM)
E1 9400	`div.s.t Vi,Vj,Vk`	Divide single vectors (VM)
E1 9600	`div.d.t Vi,Vj,Vk`	Divide double vectors (VM)
E1 9800	`mul.s.t Vi,Sj,Vk`	Multiply vector/scalar single (VM)
E1 9A00	`mul.d.t Vi,Sj,Vk`	Multiply vector/scalar double (VM)
E1 9C00	`div.s.f Vi,Sj,Vk`	Divide vector/scalar single (!VM)
E1 9E00	`div.d.f Vi,Sj,Vk`	Divide vector/scalar double (!VM)
E1 A000	`and.t Vi,Vj,Vk`	AND two vectors (VM)
E1 A200	`or.t Vi,Vj,Vk`	OR two vectors (VM)
E1 A400	`xor.t Vi,Vj,Vk`	Exclusive OR two vectors (VM)
E1 A600	`shf.t Vi,Vj,Vk`	Shift vector/vector (VM)
E1 A800	`and.t Vi,Sj,Vk`	AND vector/scalar (VM)
E1 AA00	`or.t Vi,Sj,Vk`	OR vector/scalar (VM)
E1 AC00	`xor.t Vi,Sj,Vk`	Exclusive OR vector/scalar (VM)
E1 AE00	`shf.t Vi,Sj,Vk`	Shift vector/scalar (VM)
E1 B000	`add.s.t Vi,Vj,Vk`	Add vector/vector single (VM)
E1 B200	`add.d.t Vi,Vj,Vk`	Add vector/vector double (VM)
E1 B400	`sub.s.t Vi,Vj,Vk`	Subtract single vectors (VM)
E1 B600	`sub.d.t Vi,Vj,Vk`	Subtract double vectors (VM)
E1 B800	`add.s.t Vi,Sj,Vk`	Add vector/scalar single (VM)
E1 BA00	`add.d.t Vi,Sj,Vk`	Add vector/scalar double (VM)
E1 BC00	`sub.s.t Vi,Sj,Vk`	Subtract vector/scalar single (VM)
E1 BE00	`sub.d.t Vi,Sj,Vk`	Subtract vector/scalar double (VM)

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
E1 C000	`add.b.t Vi,Vj,Vk`	Add vector/vector byte (VM)
E1 C200	`add.h.t Vi,Vj,Vk`	Add vector/vector halfword (VM)
E1 C400	`add.w.t Vi,Vj,Vk`	Add vector/vector word (VM)
E1 C600	`add.l.t Vi,Vj,Vk`	Add vector/vector longword (VM)
E1 C800	`add.b.t Vi,Sj,Vk`	Add vector/scalar byte (VM)
E1 CA00	`add.h.t Vi Sj,Vk`	Add vector/scalar halfword (VM)
E1 CC00	`add.w.t Vi,Sj,Vk`	Add vector/scalar word (VM)
E1 CE00	`add.l.t Vi,Sj,Vk`	Add vector/scalar longword (VM)
E1 D000	`sub.b.t Vi,Vj,Vk`	Subtract byte vectors (VM)
E1 D200	`sub.h.t Vi,Vj,Vk`	Subtract halfword vectors (VM)
E1 D400	`sub.w.t Vi,Vj,Vk`	Subtract word vectors (VM)
E1 D600	`sub.l.t Vi,Vj,Vk`	Subtract longword vectors (VM)
E1 D800	`sub.b.t Vi,Sj,Vk`	Subtract vector/scalar byte (VM)
E1 DA00	`sub.h.t Vi,Sj,Vk`	Subtract vector/scalar halfword (VM)
E1 DC00	`sub.w.t Vi,Sj,Vk`	Subtract vector/scalar word (VM)
E1 DE00	`sub.l.t Vi,Sj,Vk`	Subtract vector/scalar longword (VM)
E1 E000	`mul.b.t Vi,Vj,Vk`	Multiply byte vectors (VM)
E1 E200	`mul.h.t Vi,Vj,Vk`	Multiply halfword vectors (VM)
E1 E400	`mul.w.t Vi,Vj,Vk`	Multiply word vectors (VM)
E1 E600	`mul.l.t Vi,Vj,Vk`	Multiply longword vectors (VM)
E1 E800	`mul.b.t Vi,Sj,Vk`	Multiply vector/scalar byte (VM)
E1 EA00	`mul.h.t Vi,Sj,Vk`	Multiply vector/scalar halfword (VM)
E1 EC00	`mul.w.t Vi,Sj,Vk`	Multiply vector/scalar word (VM)
E1 EE00	`mul.l.t Vi,Sj,Vk`	Multiply vector/scalar longword (VM)
E1 F000	`div.b.t Vi,Vj,Vk`	Divide byte vectors (VM)
E1 F200	`div.h.t Vi,Vj,Vk`	Divide halfword vectors (VM)

Table 17 (continued)
Instructions sorted by op code

Op code (Hex)	Instruction mnemonic	Instruction description
E1 F400	`div.w.t Vi,Vj,Vk`	Divide word vectors (VM)
E1 F600	`div.l.t Vi,Vj,Vk`	Divide longword vectors (VM)
E1 F800	`div.b.t Vi,Sj,Vk`	Divide vector/scalar byte (VM)
E1 FA00	`div.h.t Vi,Sj,Vk`	Divide vector/scalar halfword (VM)
E1 FC00	`div.w.t Vi,Sj,Vk`	Divide vector/scalar word (VM)
E1 FE00	`div.l.t Vi,Sj,Vk`	Divide vector/scalar longword (VM)

Instructions sorted by mnemonic

B

Table 18
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`add.b Vi,Sj,Vk`	ST C800	Add vector/scalar integer byte
`add.b Vi,Vj,Vk`	ST C000	Add vector/vector integer byte
`add.b.f Vi,Sj,Vk`	E0 C800	Add vector/scalar byte (!VM)
`add.b.f Vi,Vj,Vk`	E0 C800	Add vector/vector byte (!VM)
`add.b.t Vi,Sj,Vk`	E1 C800	Add vector/scalar byte (VM)
`add.b.t Vi,Vj,Vk`	E1 C000	Add vector/vector byte (VM)
`add.d Sj,Sk`	ST 5540	Add scalar/scalar double float
`add.d Vi,Sj,Vk`	ST BA00	Add vector/scalar double float
`add.d Vi,Vj,Vk`	ST B200	Add vector/vector double float
`add.d.f Vi,Sj,Vk`	E0 BA00	Add vector/scalar double (!VM)
`add.d.f Vi,Vj,Vk`	E0 B200	Add vector/vector double (!VM)
`add.d.t Vi,Sj,Vk`	E1 BA00	Add vector/scalar double (VM)
`add.d.t Vi,Vj,Vk`	E1 B200	Add vector/vector double (VM)
`add.h #N,Ak`	ST 1400	Add immediate address halfword
`add.h #n,Ak`	ST 5880	Add short immediate address halfword
`add.h #N,Sk`	ST 1408	Add scalar/immediate integer halfword
`add.h Aj,Ak`	ST 5800	Add address register halfword
`add.h Sj,Sk`	ST 5940	Add scalar/scalar integer halfword
`add.h Vi Sj,Vk`	ST CA00	Add vector/scalar integer halfword

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`add.h Vi,Vj,Vk`	ST C200	Add vector/vector integer halfword
`add.h.f Vi Sj,Vk`	E0 CA00	Add vector/scalar halfword (!VM)
`add.h.f Vi,Vj,Vk`	E0 C200	Add vector/vector halfword (!VM)
`add.h.t Vi Sj,Vk`	E1 CA00	Add vector/scalar halfword (VM)
`add.h.t Vi,Vj,Vk`	E1 C200	Add vector/vector halfword (VM)
`add.l Sj,Sk`	ST 59C0	Add scalar/scalar integer longword
`add.l Vi,Sj,Vk`	ST CE00	Add vector/scalar integer longword
`add.l Vi,Vj,Vk`	ST C600	Add vector/vector integer longword
`add.l.f Vi,Sj,Vk`	E0 CE00	Add vector/scalar longword (!VM)
`add.l.f Vi,Vj,Vk`	E0 C600	Add vector/vector longword (!VM)
`add.l.t Vi,Sj,Vk`	E1 CE00	Add vector/scalar longword (VM)
`add.l.t Vi,Vj,Vk`	E1 C600	Add vector/vector longword (VM)
`add.s #N,Sk`	ST 1808	Add scalar/immediate single float
`add.s Sj,Sk`	ST 5500	Add scalar/scalar single float
`add.s Vi,Sj,Vk`	ST B800	Add vector/scalar single float
`add.s Vi,Vj,Vk`	ST B000	Add vector/vector single float
`add.s.f Vi,Sj,Vk`	E0 B800	Add vector/scalar single (!VM)
`add.s.f Vi,Vj,Vk`	E0 B000	Add vector/vector single (!VM)
`add.s.t Vi,Sj,Vk`	E1 B800	Add vector/scalar single (VM)
`add.s.t Vi,Vj,Vk`	E1 B000	Add vector/vector single (VM)
`add.w #N,Ak`	ST 1480	Add immediate address word
`add.w #n,Ak`	ST 58C0	Add short immediate address word
`add.w #N,Sk`	ST 1488	Add scalar/immediate integer word
`add.w Aj,Ak`	ST 5840	Add address register word
`add.w Sj,Ak`	ST 5000	Add scalar to address word
`add.w Sj,Sk`	ST 5980	Add scalar/scalar integer word

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`add.w Vi,Sj,Vk`	ST CC00	Add vector/scalar integer word
`add.w Vi,Vj,Vk`	ST C400	Add vector/vector integer word
`add.w.f Vi,Sj,Vk`	E0 CC00	Add vector/scalar word (!VM)
`add.w.f Vi,Vj,Vk`	E0 C400	Add vector/vector word (!VM)
`add.w.t Vi,Sj,Vk`	E1 CC00	Add vector/scalar word (VM)
`add.w.t Vi,Vj,Vk`	E1 C400	Add vector/vector word (VM)
`all Sk`	ST 7E20	AND reduce a vector
`all Vk`	ST 7E20	AND reduce a vector
`all.f S.`	E0 7E20	AND reduce a vector (!VM)
`all.f V.`	E0 7E20	AND reduce a vector (!VM)
`all.t S.`	E1 7E20	AND reduce a vector (VM)
`all.t V.`	E1 7E20	AND reduce a vector (VM)
`and #N,Ak`	ST 1200	AND immediate to address register
`and #N,Sk`	ST 1208	AND scalar/immediate
`and Aj,Ak`	ST 5200	AND address register
`and Sj,Sk`	ST 5300	AND scalar/scalar
`and Vi,Sj,Vk`	ST A800	AND vector/scalar
`and Vi,Vj,Vk`	ST A000	AND two vectors
`and.f Vi,Sj,Vk`	E0 A800	AND vector/scalar (!VM)
`and.f Vi,Vj,Vk`	E0 A000	AND two vectors (!VM)
`and.t Vi,Sj,Vk`	E1 A800	AND vector/scalar (VM)
`and.t Vi,Vj,Vk`	E1 A000	AND two vectors (VM)
`any S.`	ST 7E28	OR reduce a vector
`any V.`	ST 7E28	OR reduce a vector
`any.f Sk`	E0 7E28	OR reduce a vector (!VM)
`any.f Vk`	E0 7E28	OR reduce a vector (!VM)

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`any.t Sk`	E1 7E28	OR reduce a vector (VM)
`any.t Vk`	E1 7E28	OR reduce a vector (VM)
`atan.d Sk`	ST 7DF8	Arc-tangent of a double float
`atan.s Sk`	ST 7DF0	Arc-tangent of a single float
`bkpt`	ST 7D50	Breakpoint
`br`	ST 7100	Branch always
`bra.f`	ST 7400	Branch on address carry false
`bra.t`	ST 7500	Branch on address carry true
`bri.f`	ST 7200	Branch on ION false
`bri.t`	ST 7300	Branch on ION true
`brs.f`	ST 7600	Branch on scalar carry false
`brs.t`	ST 7700	Branch on scalar carry true
`call` *effa*	ST 2000	Call a subroutine
`callq` *effa*	ST 2200	Push the PC and jump
`calls` *effa*	ST 2100	Call a subroutine
`casr`	ST 7DE0	Compare and swap
`cfork`	ST 7C88	Clear a fork
`cos.d Sk`	ST 7CE8	Cosine of a double-precision number
`cos.s Sk`	ST 7CE0	Cosine of a single-precision number
`cprs.f Vj,Vk`	ST 6380	Compress a vector (!VM)
`cprs.t Vj,Vk`	ST 63C0	Compress a vector (VM)
`ctrsg`	E0 7C38	Move scalar to CPU timer
`cvtb.w Aj,Ak`	ST 4080	Convert byte to word
`cvtb.w Sj,Sk`	ST 4180	Convert byte to word
`cvtb.w Vj,Vk`	E0 4080	Convert byte to word
`cvtb.w.f Vj,Vk`	E0 4180	Convert byte to word (!VM)

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`cvtb.w.t Vj,Vk`	E1 4180	Convert byte to word (VM)
`cvtd.l Sj,Sk`	ST 4340	Convert double float to longword
`cvtd.l Vj,Vk`	ST 60C0	Convert double float to longword
`cvtd.l.f Vj,Vk`	E0 60C0	Convert double to longword (!VM)
`cvtd.l.t Vj,Vk`	E1 60C0	Convert double to longword (VM)
`cvtd.s Sj,Sk`	ST 4280	Convert double float to single float
`cvtd.s Vj,Vk`	ST 6000	Convert double float to single float
`cvtd.s.f Vj,Vk`	E0 6000	Convert double to single (!VM)
`cvtd.s.t Vj,Vk`	E1 6000	Convert double to single (VM)
`cvtd.w Sj,Sk`	E0 4540	Convert double float to word
`cvtd.w Vj,Vk`	E0 4240	Convert double to word float
`cvtd.w.f Vj,Vk`	E0 4340	Convert double to word (!VM)
`cvtd.w.t Vj,Vk`	E1 4340	Convert double to word (VM)
`cvth.w Aj,Ak`	ST 40C0	Convert halfword to word
`cvth.w Sj,Sk`	ST 41C0	Convert halfword to word
`cvth.w Vj,Vk`	E0 40C0	Convert halfword to word
`cvth.w.f Vj,Vk`	E0 41C0	Convert halfword to word (!VM)
`cvth.w.t Vj,Vk`	E1 41C0	Convert halfword to word (VM)
`cvtl.d Sj,Sk`	ST 43C0	Convert longword to double float
`cvtl.d Vj,Vk`	ST 6080	Convert longword to double float
`cvtl.d.f Vj,Vk`	E0 6080	Convert longword to double (!VM)
`cvtl.d.t Vj,Vk`	E1 6080	Convert longword to double (VM)
`cvtl.s Sj,Sk`	ST 4380	Convert longword to single float
`cvtl.s Vj,Vk`	E0 4280	Convert longword to single float
`cvtl.s.f Vj,Vk`	E0 4380	Convert longword to single (!VM)
`cvtl.s.t Vj,Vk`	E1 4380	Convert longword to single (VM)

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`cvtl.w Sj,Sk`	ST 4500	Convert longword to word
`cvtl.w Vj,Vk`	ST 79C0	Convert longword to word
`cvtl.w.f Vj,Vk`	E0 79C0	Convert longword to word (!VM)
`cvtl.w.t Vj,Vk`	E1 79C0	Convert longword to word (VM)
`cvts.d Sj,Sk`	ST 42C0	Convert single float to double float
`cvts.d Vj,Vk`	ST 6040	Convert single float to double float
`cvts.d.f Vj,Vk`	E0 6040	Convert single to double (!VM)
`cvts.d.t Vj,Vk`	E1 6040	Convert single to double (VM)
`cvts.l Sj,Sk`	ST 4300	Convert single float to longword
`cvts.l Vj,Vk`	E0 4200	Convert single float to longword
`cvts.l.f Vj,Vk`	E0 4300	Convert single to longword (!VM)
`cvts.l.t Vj,Vk`	E1 4300	Convert single to longword (VM)
`cvts.w Sj,Sk`	ST 4240	Convert single float to word
`cvts.w Vj,Vk`	ST 7940	Convert single float to word
`cvts.w.f Vj,Vk`	E0 7940	Convert single to word (!VM)
`cvts.w.t Vj,Vk`	E1 7940	Convert single to word (VM)
`cvtw.b Aj,Ak`	ST 4000	Convert word to byte
`cvtw.b Sj,Sk`	ST 4100	Convert word to byte
`cvtw.b Vj,Vk`	E0 4000	Convert word to byte
`cvtw.b.f Vj,Vk`	E0 4100	Convert word to byte (!VM)
`cvtw.b.t Vj,Vk`	E1 4100	Convert word to byte (VM)
`cvtw.d Sj,Sk`	E0 4500	Convert word to double float
`cvtw.d Vj,Vk`	E0 42C0	Convert word to double float
`cvtw.d.f Vj,Vk`	E0 43C0	Convert word to double (!VM)
`cvtw.d.t Vj,Vk`	E1 43C0	Convert word to double (VM)
`cvtw.h Aj,Ak`	ST 4040	Convert word to halfword

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`cvtw.h Sj,Sk`	ST 4140	Convert word to halfword
`cvtw.h Vj,Vk`	E0 4040	Convert word to halfword
`cvtw.h.f Vj,Vk`	E0 4140	Convert word to halfword (!VM)
`cvtw.h.t Vj,Vk`	E1 4140	Convert word to halfword (VM)
`cvtw.l Sj,Sk`	ST 4540	Convert word to longword
`cvtw.l Vj,Vk`	ST 7980	Convert word to longword
`cvtw.l.f Vj,Vk`	E0 7980	Convert word to longword (!VM)
`cvtw.l.t Vj,Vk`	E1 7980	Convert word to longword (VM)
`cvtw.s Sj,Sk`	ST 4200	Convert word to single float
`cvtw.s Vj,Vk`	ST 7900	Convert word to single float
`cvtw.s.f Vj,Vk`	E0 7900	Convert word to single (!VM)
`cvtw.s.t Vj,Vk`	E1 7900	Convert word to single (VM)
`diag Ak`	ST 7DC0	Execute nonstandard microcode sequence
`div.b Sj,Sk`	ST 5F00	Divide scalar/scalar integer byte
`div.b Vi,Sj,Vk`	ST F800	Divide vector/scalar integer byte
`div.b Vi,Vj,Vk`	ST F000	Divide vector/vector integer byte
`div.b.f Vi,Sj,Vk`	E0 F800	Divide vector/scalar byte (!VM)
`div.b.f Vi,Vj,Vk`	E0 F000	Divide byte vectors (!VM)
`div.b.t Vi,Sj,Vk`	E1 F800	Divide vector/scalar byte (VM)
`div.b.t Vi,Vj,Vk`	E1 F000	Divide byte vectors (VM)
`div.d Si,Vj,Vk`	E0 8600	Divide scalar/vector double float
`div.d Sj,Sk`	ST 57C0	Divide scalar/scalar double float
`div.d Vi,Sj,Vk`	ST 9E00	Divide vector/scalar double float
`div.d Vi,Vj,Vk`	ST 9600	Divide vector/vector double float
`div.d.f Si,Vj,Vk`	E0 8E00	Divide scalar/vector double (!VM)
`div.d.f Vi,Sj,Vk`	E1 9E00	Divide vector/scalar double (!VM)

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`div.d.f Vi,Vj,Vk`	E0 9600	Divide double vectors (!VM)
`div.d.t Si,Vj,Vk`	E1 8E00	Divide scalar/vector double (VM)
`div.d.t Vi,Sj,Vk`	E0 9E00	Divide vector/scalar double (VM)
`div.d.t Vi,Vj,Vk`	E1 9600	Divide double vectors (VM)
`div.h #N,Ak`	ST 1700	Divide immediate address halfword
`div.h #n,Ak`	ST 5E80	Divide short immediate address halfword
`div.h #N,Sk`	ST 1708	Divide scalar/scalar integer halfword
`div.h Aj,Ak`	ST 5E00	Divide address register halfword
`div.h Sj,Sk`	ST 5F40	Divide scalar/scalar integer halfword
`div.h Vi,Sj,Vk`	ST FA00	Divide vector/scalar integer halfword
`div.h Vi,Vj,Vk`	ST F200	Divide vector/vector integer halfword
`div.h.f Vi,Sj,Vk`	E0 FA00	Divide vector/scalar halfword (!VM)
`div.h.f Vi,Vj,Vk`	E0 F200	Divide halfword vectors (!VM)
`div.h.t Vi,Sj,Vk`	E1 FA00	Divide vector/scalar halfword (VM)
`div.h.t Vi,Vj,Vk`	E1 F200	Divide halfword vectors (VM)
`div.l Sj,Sk`	ST 5FC0	Divide scalar/scalar integer longword
`div.l Vi,Sj,Vk`	ST FE00	Divide vector/scalar integer longword
`div.l Vi,Vj,Vk`	ST F600	Divide vector/vector integer longword
`div.l.f Vi,Sj,Vk`	E0 FE00	Divide vector/scalar longword (!VM)
`div.l.f Vi,Vj,Vk`	E0 F600	Divide longword vectors (!VM)
`div.l.t Vi,Sj,Vk`	E1 FE00	Divide vector/scalar longword (VM)
`div.l.t Vi,Vj,Vk`	E1 F600	Divide longword vectors (VM)
`div.s #N,Sk`	ST 1988	Divide scalar/scalar single float
`div.s Si,Vj,Vk`	E0 8400	Divide scalar/vector single float
`div.s Sj,Sk`	ST 5780	Divide scalar/scalar single float
`div.s Vi,Sj,Vk`	ST 9C00	Divide vector/scalar single float

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`div.s Vi,Vj,Vk`	ST 9400	Divide vector/vector single float
`div.s.f Si,Vj,Vk`	E0 8C00	Divide scalar/vector single (!VM)
`div.s.f Vi,Sj,Vk`	E1 9C00	Divide vector/scalar single (!VM)
`div.s.f Vi,Vj,Vk`	E0 9400	Divide single vectors (!VM)
`div.s.t Si,Vj,Vk`	E1 8C00	Divide scalar/vector single (VM)
`div.s.t Vi,Sj,Vk`	E1 9C00	Divide vector/scalar single (VM)
`div.s.t Vi,Vj,Vk`	E1 9400	Divide single vectors (VM)
`div.w #N,Ak`	ST 1780	Divide immediate address word
`div.w #n,Ak`	ST 5EC0	Divide short immediate address word
`div.w #N,Sk`	ST 1788	Divide scalar/scalar integer word
`div.w Aj,Ak`	ST 5E40	Divide address register word
`div.w Sj,Sk`	ST 5F80	Divide scalar/scalar integer word
`div.w Vi,Sj,Vk`	ST FC00	Divide vector/scalar integer word
`div.w Vi,Vj,Vk`	ST F400	Divide vector/vector integer word
`div.w.f Vi,Sj,Vk`	E0 FC00	Divide vector/scalar word (!VM)
`div.w.f Vi,Vj,Vk`	E0 F400	Divide word vectors (!VM)
`div.w.t Vi,Sj,Vk`	E1 FC00	Divide vector/scalar word (VM)
`div.w.t Vi,Vj,Vk`	E1 F400	Divide word vectors (VM)
`dsi`	ST 7D48	Disable interrupts; reset ION to 0
`enag Sj,Sk`	E0 4480	Enable all global CPU interrupts
`enal Sj,Sk`	E0 4400	Enable local CPU interrupt
`eni`	ST 7D40	Enable interrupts; set ION to 1
`eni_idle Sk`	E0 7C80	Enable interrupts and idle the CPU
`eni_rtn`	E0 7C88	Return from subroutine
`eq.b Sj,Sk`	ST 4700	Compare equal byte
`eq.b Sj,Vk`	ST 6900	Compare equal byte

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`eq.b Vj,Vk`	ST 6800	Compare equal byte
`eq.b.f Sj,Vk`	E0 6900	Compare equal byte (!VM)
`eq.b.f Vj,Vk`	E0 6800	Compare equal byte (!VM)
`eq.b.t Sj,Vk`	E1 6900	Compare equal byte (VM)
`eq.b.t Vj,Vk`	E1 6800	Compare equal byte (VM)
`eq.d Sj,Sk`	ST 5640	Compare equal double float
`eq.d Sj,Vk`	ST 6540	Compare equal double precision
`eq.d Vj,Vk`	ST 6440	Compare equal double precision
`eq.d.f Sj,Vk`	E0 6540	Compare equal double (!VM)
`eq.d.f Vj,Vk`	E0 6440	Compare equal double (!VM)
`eq.d.t Sj,Vk`	E1 6540	Compare equal double (VM)
`eq.d.t Vj,Vk`	E1 6440	Compare equal double (VM)
`eq.h #N,Ak`	ST 1B00	Compare equal halfword
`eq.h #n,Ak`	ST 4680	Compare equal halfword
`eq.h #N,Sk`	ST 1B08	Compare equal halfword
`eq.h Aj,Ak`	ST 4600	Compare equal halfword
`eq.h Sj,Sk`	ST 4740	Compare equal halfword
`eq.h Sj,Vk`	ST 6940	Compare equal halfword
`eq.h Vj,Vk`	ST 6840	Compare equal halfword
`eq.h.f Sj,Vk`	E0 6940	Compare equal halfword (!VM)
`eq.h.f Vj,Vk`	E0 6840	Compare equal halfword (!VM)
`eq.h.t Sj,Vk`	E1 6940	Compare equal halfword (VM)
`eq.h.t Vj,Vk`	E1 6840	Compare equal halfword (VM)
`eq.l Sj,Sk`	ST 47C0	Compare equal longword
`eq.l Sj,Vk`	ST 69C0	Compare equal longword
`eq.l Vj,Vk`	ST 68C0	Compare equal longword

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`eq.l.f Sj,Vk`	E0 69C0	Compare equal long (!VM)
`eq.l.f Vj,Vk`	E0 68C0	Compare equal long (!VM)
`eq.l.t Sj,Vk`	E1 69C0	Compare equal long (VM)
`eq.l.t Vj,Vk`	E1 68C0	Compare equal long (VM)
`eq.s Sj,Sk`	ST 5600	Compare equal single float
`eq.s Sj,Vk`	ST 6500	Compare equal single
`eq.s Vj,Vk`	ST 6400	Compare equal single
`eq.s.f Sj,Vk`	E0 6500	Compare equal single (!VM)
`eq.s.f Vj,Vk`	E0 6400	Compare equal single (!VM)
`eq.s.t Sj,Vk`	E1 6500	Compare equal single (VM)
`eq.s.t Vj,Vk`	E1 6400	Compare equal single (VM)
`eq.w #N,Ak`	ST 1B80	Compare equal word
`eq.w #n,Ak`	ST 46C0	Compare equal word
`eq.w #N,Sk`	ST 1B88	Compare equal word
`eq.w Aj,Ak`	ST 4640	Compare equal word
`eq.w Sj,Sk`	ST 4780	Compare equal word
`eq.w Sj,Vk`	ST 6980	Compare equal word
`eq.w Vj,Vk`	ST 6880	Compare equal word
`eq.w.f Sj,Vk`	E0 6980	Compare equal word (!VM)
`eq.w.f Vj,Vk`	E0 6880	Compare equal word (!VM)
`eq.w.t Sj,Vk`	E1 6980	Compare equal word (VM)
`eq.w.t Vj,Vk`	E1 6880	Compare equal word (VM)
`exit`	ST 0000	Error exit instruction
`exp.d Sk`	ST 7CB8	Exponent of a double float
`exp.s Sk`	ST 7CB0	Exponent of a single float
`frint.d Sj,Sk`	E0 4740	Integerize float double scalar

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`frint.d Vj,Vk`	E0 58C0	Integerize float double vector
`frint.d.f Vj,Vk`	E0 59C0	Integerize double vector (!VM)
`frint.d.t Vj,Vk`	E1 59C0	Integerize double vector (VM)
`frint.s Sj,Sk`	E0 4700	Integerize float single scalar
`frint.s Vj,Vk`	E0 5880	Integerize float single vector
`frint.s.f Vj,Vk`	E0 5980	Integerize single vector (!VM)
`frint.s.t Vj,Vk`	E1 5980	Integerize single vector (VM)
`get.l` *Ceffa*`,Sk`	E0 3200	Get communication/scalar
`get.w` *Ceffa*`,Ak`	E0 2A00	Get communication/address
`getr.l` *effa*`,Sk`	E0 2300	Get resource/scalar
`getr.w` *effa*`,Ak`	E0 2200	Get resource/address
`halt #N,Ak`	ST 1000	Halt the CPU
`idle Sk`	ST 7C58	Idle the CPU
`inc.l` *Ceffa*`,Sk`	E0 3500	Increment communication/scalar
`inc.w` *Ceffa*`,Ak`	E0 2D00	Increment communication/address
`incr.l` *effa*`,Sk`	ST 2F00	Increment long resource structure
`incr.w` *effa*`,Ak`	ST 2B00	Increment resource structure data
`jmp` *effa*	ST 0100	Jump always
`jmpa.f` *effa*	ST 0400	Jump on address carry false
`jmpa.t` *effa*	ST 0500	Jump on address carry true
`jmpi.f` *effa*	ST 0200	Jump on ION false
`jmpi.t` *effa*	ST 0300	Jump on ION true
`jmps.f` *effa*	ST 0600	Jump on scalar carry false
`jmps.t` *effa*	ST 0700	Jump on scalar carry true
`join`	ST 7CA0	Join all threads
`lck` *Ceffa*	E0 0200	Lock communication register

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`ld.b` *effa*`,Ak`	ST 2800	Load address register byte
`ld.b` *effa*`,Sk`	ST 3000	Load scalar byte
`ld.b` *effa*`,Vk`	ST 3800	Load vector byte
`ld.b.f` *effa*`,Vk`	E0 3800	Load vector byte (!VM)
`ld.b.t` *effa*`,Vk`	E1 3800	Load vector byte (VM)
`ld.d #N,Sk`	ST 1008	Load double float immediate upper 32 bits
`ld.d` *effa*`,Sk`	ST 3300	Load scalar double float
`ld.d` *effa*`,Vk`	ST 3B00	Load vector double float
`ld.d.f` *effa*`,Vk`	E0 3B00	Load vector double float (!VM)
`ld.d.t` *effa*`,Vk`	E1 3B00	Load vector double float (VM)
`ld.dl #N,Sk`	ST 1188	Load 64-bit floating immediate
`ld.du #N,Sk`	ST 1088	Load 64-bit floating immediate
`ld.h #N,Ak`	ST 1100	Load halfword immediate into Ak
`ld.h #n,Ak`	ST 4480	Load short immediate into Ak
`ld.h` *effa*`,Ak`	ST 2900	Load address register halfword
`ld.h` *effa*`,Sk`	ST 3100	Load scalar halfword
`ld.h` *effa*`,Vk`	ST 3900	Load vector halfword
`ld.h.f` *effa*`,Vk`	E0 3900	Load vector halfword (!VM)
`ld.h.t` *effa*`,Vk`	E1 3900	Load vector halfword (VM)
`ld.l #N,Sk`	ST 1108	Load 32-bit immediate sign-extended to 64 bits
`ld.l` *effa*`,Sk`	ST 3300	Load scalar longword
`ld.l` *effa*`,Vk`	ST 3B00	Load vector longword
`ld.l` *effa*`,VLS`	ST 0A00	Load VS and VL from memory
`ld.l.f` *effa*`,Vk`	E0 3B00	Load vector longword (!VM)
`ld.l.t` *effa*`,Vk`	E1 3B00	Load vector longword (VM)
`ld.ll #N,Sk`	ST 1188	Load 64-bit integer immediate

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`ld.lu #N,Sk`	ST 1088	Load 64-bit integer immediate
`ld.s #N,Sk`	ST 1188	Load a single float immediate
`ld.s` *effa*`,Sk`	ST 3200	Load scalar single float
`ld.s` *effa*`,Vk`	ST 3A00	Load vector single float
`ld.s.f` *effa*`,Vk`	E0 3A00	Load vector single float (!VM)
`ld.s.t` *effa*`,Vk`	E1 3A00	Load vector single float (VM)
`ld.u #N,Sk`	ST 1088	Load immediate
`ld.w #N,Ak`	ST 1180	Load immediate into Ak
`ld.w #n,Ak`	ST 44C0	Load short immediate into Ak
`ld.w #N,Sk`	ST 1188	Load a 32-bit immediate
`ld.w #N,VL`	ST 1800	Load VL with an immediate
`ld.w #N,VS`	ST 1880	Load VS from an immediate
`ld.w` *effa*`,Ak`	ST 2A00	Load address register word
`ld.w` *effa*`,Sk`	ST 3200	Load scalar word
`ld.w` *effa*`,Vk`	ST 3A00	Load vector word
`ld.w.f` *effa*`,Vk`	E0 3A00	Load vector word (!VM)
`ld.w.t` *effa*`,Vk`	E1 3A00	Load vector word (VM)
`ld.x` *effa*`,VM`	ST 0B00	Load VM from memory
`ldcmr` *effa*`,Ak`	E0 0600	Load communication registers
`ldea` *effa*`,Ak`	ST 0900	Load effective address
`ldea` *effa*`,Sk`	E0 0400	Load effective address/scalar
`ldkdr Ak`	ST 7C08	Load all eight SDRs
`ldpa Aj,Ak`	ST 4400	Load a physical byte address into Ak
`ldsdr Ak`	ST 7C00	Load process SDRs
`ldvi.b Vj,Vk`	ST 7800	Index load vector byte
`ldvi.b.f Vj,Vk`	E0 7800	Index load vector byte (!VM)

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`ldvi.b.t Vj,Vk`	E1 7800	Index load vector byte (VM)
`ldvi.d Vj,Vk`	ST 78C0	Index load vector double float
`ldvi.d.f Vj,Vk`	E0 78C0	Index load vector double (!VM)
`ldvi.d.t Vj,Vk`	E1 78C0	Index load vector double (VM)
`ldvi.h Vj,Vk`	ST 7840	Index load vector halfword
`ldvi.h.f Vj,Vk`	E0 7840	Index load vector halfword (!VM)
`ldvi.h.t Vj,Vk`	E1 7840	Index load vector halfword (VM)
`ldvi.l Vj,Vk`	ST 78C0	Index load vector longword
`ldvi.l.f Vj,Vk`	E0 78C0	Index load vector longword (!VM)
`ldvi.l.t Vj,Vk`	E1 78C0	Index load vector longword (VM)
`ldvi.s Vj,Vk`	ST 7880	Index load vector single float
`ldvi.s.f Vj,Vk`	E0 7880	Index load vector single (!VM)
`ldvi.s.t Vj,Vk`	E1 7880	Index load vector single (VM)
`ldvi.w Vj,Vk`	ST 7880	Index load vector word
`ldvi.w.f Vj,Vk`	E0 7880	Index load vector word (!VM)
`ldvi.w.t Vj,Vk`	E1 7880	Index load vector word (VM)
`le.b Sj,Sk`	ST 4D00	Compare less than or equal byte
`le.b Sj,Vk`	ST 6B00	Compare less than or equal byte
`le.b Vj,Vk`	ST 6A00	Compare less than or equal byte
`le.b.f Sj,Vk`	E0 6B00	Compare less than or equal byte (!VM)
`le.b.f Vj,Vk`	E0 6A00	Compare less than or equal byte (!VM)
`le.b.t Sj,Vk`	E1 6B00	Compare less than or equal byte (VM)
`le.b.t Vj,Vk`	E1 6A00	Compare less than or equal byte (VM)
`le.d Sj,Sk`	ST 5440	Compare less than or equal double float
`le.d Sj,Vk`	ST 6740	Compare less than or equal double float
`le.d Vj,Vk`	ST 6640	Compare less than or equal double float

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`le.d.f Sj,Vk`	E0 6740	Compare less than or equal double (!VM)
`le.d.f Vj,Vk`	E0 6640	Compare less than or equal double (!VM)
`le.d.t Sj,Vk`	E1 6740	Compare less than or equal double (VM)
`le.d.t Vj,Vk`	E1 6640	Compare less than or equal double (VM)
`le.h #N,Ak`	ST 1E00	Compare less than or equal halfword
`le.h #n,Ak`	ST 4C80	Compare less than or equal halfword
`le.h #N,Sk`	ST 1E08	Compare less than or equal halfword
`le.h Aj,Ak`	ST 4C00	Compare less than or equal signed halfword
`le.h Sj,Sk`	ST 4D40	Compare less than or equal halfword
`le.h Sj,Vk`	ST 6B40	Compare less than or equal halfword
`le.h Vj,Vk`	ST 6A40	Compare less than or equal halfword
`le.h.f Sj,Vk`	E0 6B40	Compare less than or equal half (!VM)
`le.h.f Vj,Vk`	E0 6A40	Compare less than or equal half (!VM)
`le.h.t Sj,Vk`	E1 6B40	Compare less than or equal half (VM)
`le.h.t Vj,Vk`	E1 6A40	Compare less than or equal half (VM)
`le.l Sj,Sk`	ST 4DC0	Compare less than or equal longword
`le.l Sj,Vk`	ST 6BC0	Compare less than or equal longword
`le.l Vj,Vk`	ST 6AC0	Compare less than or equal longword
`le.l.f Sj,Vk`	E0 6BC0	Compare less than or equal long (!VM)
`le.l.f Vj,Vk`	E0 6AC0	Compare less than or equal long (!VM)
`le.l.t Sj,Vk`	E1 6BC0	Compare less than or equal long (VM)
`le.l.t Vj,Vk`	E1 6AC0	Compare less than or equal long (VM)
`le.s #N,Sk`	ST 1A08	Compare less than or equal single
`le.s Sj,Sk`	ST 5400	Compare less than or equal single float
`le.s Sj,Vk`	ST 6700	Compare less than or equal single
`le.s Vj,Vk`	ST 6600	Compare less than or equal single

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`le.s.f Sj,Vk`	E0 6700	Compare less than or equal single (!VM)
`le.s.f Vj,Vk`	E0 6600	Compare less than or equal single (!VM)
`le.s.t Sj,Vk`	E1 6700	Compare less than or equal single (VM)
`le.s.t Vj,Vk`	E1 6600	Compare less than or equal single (VM)
`le.w #N,Ak`	ST 1E80	Compare less than or equal word
`le.w #n,Ak`	ST 4CC0	Compare less than or equal word
`le.w #N,Sk`	ST 1E88	Compare less than or equal word
`le.w Aj,Ak`	ST 4C40	Compare less than or equal signed word
`le.w Sj,Sk`	ST 4D80	Compare less than or equal word
`le.w Sj,Vk`	ST 6B80	Compare less than or equal word
`le.w Vj,Vk`	ST 6A80	Compare less than or equal word
`le.w.f Sj,Vk`	E0 6B80	Compare less than or equal word (!VM)
`le.w.f Vj,Vk`	E0 6A80	Compare less than or equal word (!VM)
`le.w.t Sj,Vk`	E1 6B80	Compare less than or equal word (VM)
`le.w.t Vj,Vk`	E1 6A80	Compare less than or equal word (VM)
`leu.b Sj,Sk`	ST 4900	Compare less than or equal to byte
`leu.h #N,Ak`	ST 1C00	Compare unsigned less than halfword
`leu.h #n,Ak`	ST 4880	Compare unsigned less than or equal halfword
`leu.h #N,Sk`	ST 1C08	Compare unsigned less than or equal to halfword
`leu.h Aj,Ak`	ST 4800	Compare unsigned less than or equal to halfword
`leu.h Sj,Sk`	ST 4940	Compare less than or equal to halfword
`leu.l Sj,Sk`	ST 49C0	Compare less than or equal to longword
`leu.w #n,Ak`	ST 48C0	Compare unsigned less than or equal word
`leu.w #N,Ak`	ST 1C80	Compare unsigned less than word
`leu.w #N,Sk`	ST 1C88	Compare unsigned less than or equal to word

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`leu.w Aj,Ak`	ST 4840	Compare unsigned less than or equal to word
`leu.w Sj,Sk`	ST 4980	Compare less than or equal to word
`ln.d Sk`	ST 7C18	Natural logarithm of a double precision number
`ln.s Sk`	ST 7C10	Natural logarithm of a single precision number
`lop Sj,Sk`	ST 61C0	Leading one's position in Sj
`lop Vj,Vk`	ST 6240	Leading ones position vector
`lop.f Vj,Vk`	E0 6240	Leading ones position vector (!VM)
`lop.t Vj,Vk`	E1 6240	Leading ones position vector (VM)
`lt.b Sj,Sk`	ST 4F00	Compare less than byte
`lt.b Sj,Vk`	ST 6D00	Compare less than byte
`lt.b Vj,Vk`	ST 6C00	Compare less than byte
`lt.b.f Sj,Vk`	E0 6D00	Compare less than byte (!VM)
`lt.b.f Vj,Vk`	E0 6C00	Compare less than byte (!VM)
`lt.b.t Sj,Vk`	E1 6D00	Compare less than byte (VM)
`lt.b.t Vj,Vk`	E1 6C00	Compare less than byte (VM)
`lt.d Sj,Sk`	ST 54C0	Compare less than double float
`lt.d Sj,Vk`	ST 67C0	Compare less than double float
`lt.d Vj,Vk`	ST 66C0	Compare less than double float
`lt.d.f Sj,Vk`	E0 67C0	Compare less than double (!VM)
`lt.d.f Vj,Vk`	E0 66C0	Compare less than double (!VM)
`lt.d.t Sj,Vk`	E1 67C0	Compare less than double (VM)
`lt.d.t Vj,Vk`	E1 66C0	Compare less than double (VM)
`lt.h #N,Ak`	ST 1F00	Compare less than halfword
`lt.h #n,Ak`	ST 4E80	Compare less than halfword
`lt.h #N,Sk`	ST 1F08	Compare less than halfword
`lt.h Aj,Ak`	ST 4E00	Compare less than signed halfword

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`lt.h Sj,Sk`	ST 4F40	Compare less than halfword
`lt.h Sj,Vk`	ST 6D40	Compare less than halfword
`lt.h Vj,Vk`	ST 6C40	Compare less than halfword
`lt.h.f Sj,Vk`	E0 6D40	Compare less than halfword (!VM)
`lt.h.f Vj,Vk`	E0 6C40	Compare less than halfword (!VM)
`lt.h.t Sj,Vk`	E1 6D40	Compare less than halfword (VM)
`lt.h.t Vj,Vk`	E1 6C40	Compare less than halfword (VM)
`lt.l Sj,Sk`	ST 4FC0	Compare less than longword
`lt.l Sj,Vk`	ST 6DC0	Compare less than longword
`lt.l Vj,Vk`	ST 6CC0	Compare less than longword
`lt.l.f Sj,Vk`	E0 6DC0	Compare less than long (!VM)
`lt.l.f Vj,Vk`	E0 6CC0	Compare less than long (!VM)
`lt.l.t Sj,Vk`	E1 6DC0	Compare less than long (VM)
`lt.l.t Vj,Vk`	E1 6CC0	Compare less than long (VM)
`lt.s #N,Sk`	ST 1A88	Compare less than single
`lt.s Sj,Sk`	ST 5480	Compare less than single float
`lt.s Sj,Vk`	ST 6780	Compare less than single
`lt.s Vj,Vk`	ST 6680	Compare less than single
`lt.s.f Sj,Vk`	E0 6780	Compare less than single (!VM)
`lt.s.f Vj,Vk`	E0 6680	Compare less than single (!VM)
`lt.s.t Sj,Vk`	E1 6780	Compare less than single (VM)
`lt.s.t Vj,Vk`	E1 6680	Compare less than single (VM)
`lt.w #N,Ak`	ST 1F80	Compare less than word
`lt.w #n,Ak`	ST 4EC0	Compare less than word
`lt.w #N,Sk`	ST 1F88	Compare less than word
`lt.w Aj,Ak`	ST 4E40	Compare less than signed word

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`lt.w Sj,Sk`	ST 4F80	Compare less than word
`lt.w Sj,Vk`	ST 6D80	Compare less than word
`lt.w Vj,Vk`	ST 6C80	Compare less than word
`lt.w.f Sj,Vk`	E0 6D80	Compare less than word (!VM)
`lt.w.f Vj,Vk`	E0 6C80	Compare less than word (!VM)
`lt.w.t Sj,Vk`	E1 6D80	Compare less than word (VM)
`lt.w.t Vj,Vk`	E1 6C80	Compare less than word (VM)
`ltu.b Sj,Sk`	ST 4B00	Compare less than byte
`ltu.h #N,Ak`	ST 1D00	Compare unsigned less than halfword
`ltu.h #n,Ak`	ST 4A80	Compare unsigned less than halfword
`ltu.h #N,Sk`	ST 1D08	Compare unsigned less than halfword
`ltu.h Aj,Ak`	ST 4A00	Compare unsigned less than halfword
`ltu.h Sj,Sk`	ST 4B40	Compare less than halfword
`ltu.l Sj,Sk`	ST 4BC0	Compare less than longword
`ltu.w #N,Ak`	ST 1D80	Compare unsigned less than word
`ltu.w #n,Ak`	ST 4AC0	Compare unsigned less than word
`ltu.w #N,Sk`	ST 1D88	Compare unsigned less than word
`ltu.w Aj,Ak`	ST 4A40	Compare unsigned less than word
`ltu.w Sj,Sk`	ST 4B80	Compare less than word
`mask.f Vi,Sj,Vk`	ST 8A00	Mask vector/scalar (!VM)
`mask.t Vi,Sj,Vk`	ST 8E00	Mask vector/scalar (VM)
`mask.t Vi,Vj,Vk`	ST 8600	Mask vector/vector
`mat.l Sk,`*Ceffa*	E0 3100	Match scalar/communication
`mat.w Ak,`*Ceffa*	E0 2900	Match address/communication
`matr.l Sk,`*effa*	E0 3000	Match scalar/resource
`matr.w Ak,`*effa*	E0 2800	Match address/resource

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`max.b Vk`	ST 7E40	Maximum of a vector of bytes
`max.b.f Vk`	E0 7E40	Maximum of vector of bytes (!VM)
`max.b.t Vk`	E1 7E40	Maximum of vector of bytes (VM)
`max.d Vk`	ST 7EA8	Maximum of a vector of double float
`max.d.f Vk`	E0 7EA8	Maximum of vector of doubles (!VM)
`max.d.t Vk`	E1 7EA8	Maximum of vector of doubles (VM)
`max.h Vk`	ST 7E48	Maximum of a vector of halfwords
`max.h.f Vk`	E0 7E48	Maximum of vector of halfwords (!VM)
`max.h.t Vk`	E1 7E48	Maximum of vector of halfwords (VM)
`max.l Vk`	ST 7E58	Maximum of a vector of longwords
`max.l.f Vk`	E0 7E58	Maximum of vector of longwords (!VM)
`max.l.t Vk`	E1 7E58	Maximum of vector of longwords (VM)
`max.s Vk`	ST 7EA0	Maximum of a vector of single float
`max.s.f Vk`	E0 7EA0	Maximum of vector of singles (!VM)
`max.s.t Vk`	E1 7EA0	Maximum of vector of singles (VM)
`max.w Vk`	ST 7E50	Maximum of a vector of words
`max.w.f Vk`	E0 7E50	Maximum of vector of words (!VM)
`max.w.t Vk`	E1 7E50	Maximum of vector of words (VM)
`merg.f Vi,Sj,Vk`	ST 8800	Merge vector/scalar (!VM)
`merg.t Vi,Sj,Vk`	ST 8C00	Merge vector/scalar
`merg.t Vi,Vj,Vk`	ST 8400	Merge vector/vector
`min.b Vk`	ST 7E60	Minimum of a vector of bytes
`min.b.f Vk`	E0 7E60	Minimum of vector of bytes (!VM)
`min.b.t Vk`	E1 7E60	Minimum of vector of bytes (VM)
`min.d Vk`	ST 7EB8	Minimum of a vector of double float
`min.d.f Vk`	E0 7EB8	Minimum of vector of doubles (!VM)

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`min.d.t Vk`	E1 7EB8	Minimum of vector of doubles (VM)
`min.h Vk`	ST 7E68	Minimum of a vector of halfwords
`min.h.f Vk`	E0 7E68	Minimum of vector of halfwords (!VM)
`min.h.t Vk`	E1 7E68	Minimum of vector of halfwords (VM)
`min.l Vk`	ST 7E78	Minimum of a vector of longwords
`min.l.f Vk`	E0 7E78	Minimum of vector of longwords (!VM)
`min.l.t Vk`	E1 7E78	Minimum of vector of longwords (VM)
`min.s Vk`	ST 7EB0	Minimum of a vector of single float
`min.s.f Vk`	E0 7EB0	Minimum of vector of singles (!VM)
`min.s.t Vk`	E1 7EB0	Minimum of vector of singles (VM)
`min.w Vk`	ST 7E70	Minimum of a vector of words
`min.w.f Vk`	E0 7E70	Minimum of vector of words (!VM)
`min.w.t Vk`	E1 7E70	Minimum of vector of words (VM)
`mov Aj,Ak`	ST 5080	Move address register
`mov Aj,Sk`	ST 51C0	Move an address to a scalar
`mov Ak,PSW`	ST 7C48	Load an address register into the PSW
`mov Ak,VL`	ST 7D98	Move Ak to VL
`mov Ak,VS`	ST 7D88	Move Ak to VS
`mov BE(Sj),Sk`	E0 4580	Move BE(scalar) to a scalar
`mov CIR,Sk`	E0 7C08	Move CIR to a scalar
`mov CPUID,Sk`	E0 7C18	Move CPU identification to scalar
`mov ICR,Sk`	E0 7C68	Move ICR to a scalar
`mov ITR,Sk`	ST 7C60	Move the ITC
`mov PC,Ak`	ST 7C50	Load the next PC address
`mov PSW,Ak`	ST 7C40	Store the PSW into an address register
`mov Si,Sj,Vk`	ST 8200	Move a scalar to a vector element

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`mov Sj,Ak`	ST 50C0	Move 32 bits of Sj into Ak
`mov Sk,BE(Sj)`	E0 45C0	Move scalar to BE(scalar)
`mov Sj,Sk,VM`	ST 6100	Load VM(Sj) from Sk
`mov Sj,VM,Sk`	ST 6140	Load Sk from VM(Sj)
`mov Sk,CIR`	E0 7C00	Move scalar to CIR
`mov Sk,ICR`	E0 7C60	Move Scalar to ICR
`mov Sk,ITR`	ST 7C68	Load NITC
`mov Sk,ITSR`	ST 7C78	Load ITSR with a scalar
`mov Sk,TCPU`	E0 7C70	Move Scalar to TCPU register
`mov Sk,TID`	E0 7CB0	Load TID from scalar
`mov Sk,TOC`	E0 7C30	Move a scalar to TOC
`mov Sk,TTR`	E0 7C20	Move scalar to TTR
`mov Sk,VML`	E0 7C50	Load VM<63..0> from Sk
`mov Sk,VMU`	E0 7C40	Load VM<127..64> from Sk
`mov Sk,VV`	ST 7D70	Move scalar to vector valid flag
`mov TCPU,Sk`	E0 7C78	Move the TCPU Register to Scalar
`mov TID,Sk`	E0 7CB8	Load scalar with TID
`mov TOC,Sk`	E0 7C10	Move TOC to a scalar
`mov TTR,Sk`	E0 7C28	Move TTR /scalar
`mov Vi,Sj,Sk`	ST 8000	Move a vector element to a scalar
`mov VL,Ak`	ST 7D90	Move VL to Ak
`mov VML,Sk`	E0 7C58	Load Sk from VM<63..0>
`mov VMU,Sk`	E0 7C48	Load Sk from VM<127..64>
`mov VS,Ak`	ST 7D80	Move VS to Ak
`mov.d Sj,Sk`	ST 5180	Move scalar register single float
`mov.l Sj,Sk`	ST 5180	Move scalar register longword

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`mov.s Sj,Sk`	ST 5100	Move scalar register double float
`mov.w Sj,Sk`	ST 5100	Move scalar register word
`mov.w Sk,VL`	ST 7DB8	Move Sk to VL
`mov.w Sk,VS`	ST 7DA8	Move Sk to VS
`mov.w VL,Sk`	ST 7DB0	Move VL to Sk
`mov.w VS,Sk`	ST 7DA0	Move VS to Sk
`mski Sk`	ST 7D60	Mask out interrupt
`msync`	ST 7D58	Synchronize stores/loads to memory
`mul.b Sj,Sk`	ST 5D00	Multiply scalar/scalar integer byte
`mul.b Vi,Sj,Vk`	ST E800	Multiply vector/scalar integer byte
`mul.b Vi,Vj,Vk`	ST E000	Multiply vector/vector integer byte
`mul.b.f Vi,Sj,Vk`	E0 E800	Multiply vector/scalar byte (!VM)
`mul.b.f Vi,Vj,Vk`	E0 E000	Multiply byte vectors (!VM)
`mul.b.t Vi,Sj,Vk`	E1 E800	Multiply vector/scalar byte (VM)
`mul.b.t Vi,Vj,Vk`	E1 E000	Multiply byte vectors (VM)
`mul.d Sj,Sk`	ST 5740	Multiply scalar/scalar double float
`mul.d Vi,Sj,Vk`	ST 9A00	Multiply vector/scalar double float
`mul.d Vi,Vj,Vk`	ST 9200	Multiply vector/vector double float
`mul.d.f Vi,Sj,Vk`	E0 9A00	Multiply vector/scalar double (!VM)
`mul.d.f Vi,Vj,Vk`	E0 9200	Multiply double vectors (!VM)
`mul.d.t Vi,Sj,Vk`	E1 9A00	Multiply vector/scalar double (VM)
`mul.d.t Vi,Vj,Vk`	E1 9200	Multiply double vectors (VM)
`mul.h #N,Ak`	ST 1600	Multiply immediate address halfword
`mul.h #n,Ak`	ST 5C80	Multiply short immediate address halfword
`mul.h #N,Sk`	ST 1608	Multiply scalar/immediate integer halfword
`mul.h Aj,Ak`	ST 5C00	Multiply address register halfword

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`mul.h Sj,Sk`	ST 5D40	Multiply scalar/scalar integer halfword
`mul.h Vi,Sj,Vk`	ST EA00	Multiply vector/scalar integer halfword
`mul.h Vi,Vj,Vk`	ST E200	Multiply vector/vector integer halfword
`mul.h.f Vi,Sj,Vk`	E0 EA00	Multiply vector/scalar halfword (!VM)
`mul.h.f Vi,Vj,Vk`	E0 E200	Multiply halfword vectors (!VM)
`mul.h.t Vi,Sj,Vk`	E1 EA00	Multiply vector/scalar halfword (VM)
`mul.h.t Vi,Vj,Vk`	E1 E200	Multiply halfword vectors (VM)
`mul.l Sj,Sk`	ST 5DC0	Multiply scalar/scalar integer longword
`mul.l Vi,Sj,Vk`	ST EE00	Multiply vector/scalar integer longword
`mul.l Vi,Vj,Vk`	ST E600	Multiply vector/vector integer longword
`mul.l.f Vi,Sj,Vk`	E0 EE00	Multiply vector/scalar longword (!VM)
`mul.l.f Vi,Vj,Vk`	E0 E600	Multiply longword vectors (!VM)
`mul.l.t Vi,Sj,Vk`	E1 EE00	Multiply vector/scalar longword (VM)
`mul.l.t Vi,Vj,Vk`	E1 E600	Multiply longword vectors (VM)
`mul.s #N,Sk`	ST 1908	Multiply scalar/immediate single float
`mul.s Sj,Sk`	ST 5700	Multiply scalar/scalar single float
`mul.s Vi,Sj,Vk`	ST 9800	Multiply vector/scalar single float
`mul.s Vi,Vj,Vk`	ST 9000	Multiply vector/vector single float
`mul.s.f Vi,Sj,Vk`	E0 9800	Multiply vector/scalar single (!VM)
`mul.s.f Vi,Vj,Vk`	E0 9000	Multiply single vectors (!VM)
`mul.s.t Vi,Sj,Vk`	E1 9800	Multiply vector/scalar single (VM)
`mul.s.t Vi,Vj,Vk`	E1 9000	Multiply single vectors (VM)
`mul.w #N,Ak`	ST 1680	Multiply immediate address word
`mul.w #n,Ak`	ST 5CC0	Multiply short immediate address word
`mul.w #N,Sk`	ST 1688	Multiply scalar/immediate integer word
`mul.w Aj,Ak`	ST 5C40	Multiply address register word

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`mul.w Sj,Sk`	ST 5D80	Multiply scalar/scalar integer word
`mul.w Vi,Sj,Vk`	ST EC00	Multiply vector/scalar integer word
`mul.w Vi,Vj,Vk`	ST E400	Multiply vector/vector integer word
`mul.w.f Vi,Sj,Vk`	E0 EC00	Multiply vector/scalar word (!VM)
`mul.w.f Vi,Vj,Vk`	E0 E400	Multiply word vectors (!VM)
`mul.w.t Vi,Sj,Vk`	E1 EC00	Multiply vector/scalar word (VM)
`mul.w.t Vi,Vj,Vk`	E1 E400	Multiply word vectors (VM)
`neg.b Sj,Sk`	ST 6F00	Negate scalar/scalar integer byte
`neg.b Vj,Vk`	ST 6E00	Negate vector/vector integer byte
`neg.b.f Vj,Vk`	E0 6E00	Negate byte vector (!VM)
`neg.b.t Vj,Vk`	E1 6E00	Negate byte vector (VM)
`neg.d Sj,Sk`	ST 65C0	Negate scalar/scalar double float
`neg.d Vj,Vk`	ST 64C0	Negate vector/vector double float
`neg.d.f Vj,Vk`	E0 64C0	Negate double (!VM)
`neg.d.t Vj,Vk`	E1 64C0	Negate double (VM)
`neg.h Aj,Ak`	ST 5680	Negate address register halfword
`neg.h Sj,Sk`	ST 6F40	Negate scalar/scalar integer halfword
`neg.h Vj,Vk`	ST 6E40	Negate vector/vector integer halfword
`neg.h.f Vj,Vk`	E0 6E40	Negate halfword (!VM)
`neg.h.t Vj,Vk`	E1 6E40	Negate halfword (VM)
`neg.l Sj,Sk`	ST 6FC0	Negate scalar/scalar integer longword
`neg.l Vj,Vk`	ST 6EC0	Negate vector/vector integer longword
`neg.l.f Vj,Vk`	E0 6EC0	Negate longword (!VM)
`neg.l.t Vj,Vk`	E1 6EC0	Negate longword (VM)
`neg.s Sj,Sk`	ST 6580	Negate scalar/scalar single float
`neg.s Vj,Vk`	ST 6480	Negate vector/vector single float

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`neg.s.f Vj,Vk`	E0 6480	Negate single (!VM)
`neg.s.t Vj,Vk`	E1 6480	Negate single (VM)
`neg.w Aj,Ak`	ST 56C0	Negate address register word
`neg.w Sj,Sk`	ST 6F80	Negate scalar/scalar integer word
`neg.w Vj,Vk`	ST 6E80	Negate vector/vector integer word
`neg.w.f Vj,Vk`	E0 6E80	Negate word (!VM)
`neg.w.t Vj,Vk`	E1 6E80	Negate word (VM)
`nop`	ST 7000	No operation (branch never)
`not Aj,Ak`	ST 52C0	Complement address register
`not Sj,Sk`	ST 53C0	Complement scalar/scalar
`not Vj,Vk`	ST 62C0	Complement a vector
`not.f Vj,Vk`	E0 62C0	Complement a vector (!VM)
`not.t Vj,Vk`	E1 62C0	Complement a vector (VM)
`or #N,Ak`	ST 1280	OR immediate to address register
`or #N,Sk`	ST 1288	OR scalar/immediate
`or Aj,Ak`	ST 5240	OR address register
`or Sj,Sk`	ST 5340	OR scalar/scalar
`or Vi,Sj,Vk`	ST AA00	OR vector/scalar
`or Vi,Vj,Vk`	ST A200	OR two vectors
`or.f Vi,Sj,Vk`	E0 AA00	OR vector/scalar (!VM)
`or.f Vi,Vj,Vk`	E0 A200	OR two vectors (!VM)
`or.t Vi,Sj,Vk`	E1 AA00	OR vector/scalar (VM)
`or.t Vi,Vj,Vk`	E1 A200	OR two vectors (VM)
`parity Vk`	ST 7E30	Exclusive OR reduce a vector
`parity.f Vk`	E0 7E30	Exclusive OR reduce vector (!VM)
`parity.t Vk`	E1 7E30	Exclusive OR reduce vector (VM)

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`pate Ak`	ST 7C28	Purge ATU entry
`patu`	ST 7C20	Purge the entire ATU
`pbkpt`	ST 7DC8	Force process breakpoint exception
`pfork` *effa*`,Ak`	ST 2300	Post a fork
`pich`	ST 7C30	Purge the Icache
`plc.f VM,Sk`	ST 7EE0	Load the number of zero's in VM into Sk
`plc.t Sj,Sk`	ST 4580	Count the number of ones in Sj
`plc.t Vj,Vk`	ST 6340	Population count of a vector
`plc.t VM,Sk`	ST 7EE8	Load the number of ones in VM into Sk
`plc.t.f Vj,Vk`	E0 6340	Population count of vector (!VM)
`plc.t.t Vj,Vk`	E1 6340	Population count of vector (VM)
`plch`	ST 7C38	Purge the Lcache
`pop.l Sk`	ST 7D38	Pop Sk <63..0> from the stack
`pop.w Ak`	ST 7D10	Pop word into address register
`pop.w Sk`	ST 7D30	Pop Sk <31..0> from the stack
`popr Ak,` *effa*	E0 0800	Pop resource/address register
`pref`	E0 7C90	Purge reference bits
`prod.b Vk`	ST 7EC0	Multiply reduce a vector of bytes
`prod.b.f Vk`	E0 7EC0	Multiply reduce byte vector (!VM)
`prod.b.t Vk`	E1 7EC0	Multiply reduce byte vector (VM)
`prod.d Vk`	ST 7E98	Multiply reduce a vector of double float
`prod.d.f Vk`	E0 7E98	Multiply reduce double vector (!VM)
`prod.d.t Vk`	E1 7E98	Multiply reduce double vector (VM)
`prod.h Vk`	ST 7EC8	Multiply reduce a vector of halfwords
`prod.h.f Vk`	E0 7EC8	Multiply reduce halfword vector (!VM)
`prod.h.t Vk`	E1 7EC8	Multiply reduce halfword vector (VM)

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`prod.l Vk`	ST 7ED8	Multiply reduce a vector of longwords
`prod.l.f Vk`	E0 7ED8	Multiply reduce longword vector (!VM)
`prod.l.t Vk`	E1 7ED8	Multiply reduce longword vector (VM)
`prod.s Vk`	ST 7E90	Multiply reduce a vector of single float
`prod.s.f Vk`	E0 7E90	Multiply reduce single vector (!VM)
`prod.s.t Vk`	E1 7E90	Multiply reduce single vector (VM)
`prod.w Vk`	ST 7ED0	Multiply reduce a vector of words
`prod.w.f Vk`	E0 7ED0	Multiply reduce word vector (!VM)
`prod.w.t Vk`	E1 7ED0	Multiply reduce word vector (VM)
`psh.l Sk`	ST 7D28	Push Sk<63..0> onto the stack
`psh.w Ak`	ST 7D00	Push an address register
`psh.w Sk`	ST 7D20	Push Sk<31..0> onto the stack
`pshea` *effa*	ST 0D00	Push effective address
`pshr Ak,` *effa*	E0 0900	Push address register/resource
`put.l Sk,` *Ceffa*	E0 3600	Put scalar/communication
`put.w Ak,` *Ceffa*	E0 2E00	Put address/communication
`putr.l Sk,` *effa*	E0 2100	Put scalar/resource
`putr.w Ak,` *effa*	E0 2000	Put address/resource
`rcv.l` *Ceffa*`, Sk`	E0 3300	Receive communication/scalar
`rcv.w` *Ceffa*`, Ak`	E0 2B00	Receive communication/address
`rcvr.l` *effa*`, Sk`	E0 0E00	Receive scalar register/resource
`rcvr.w` *effa*`, Ak`	E0 0A00	Receive address register/resource
`rtn`	ST 7C90	Return from subroutine call
`rtnc`	ST 7CA8	Return from a context block
`rtnq`	ST 7C80	Pop the PC and jump
`shf #N, Ak`	ST 1380	Logical shift immediate

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`shf #n,Ak`	ST 4440	Logical shift short immediate
`shf #N,Sk`	ST 1388	Shift scalar/immediate
`shf Aj,Ak`	ST 5040	Shift an address
`shf Sj,Sk`	ST 5140	Shift a scalar
`shf Sj,Vk`	ST 6300	Shift a vector accumulator
`shf Vi,Sj,Vk`	ST AE00	Shift a vector accumulator
`shf Vi,Vj,Vk`	ST A600	Shift vector/vector
`shf.f Sj,Vk`	E0 6300	Shift vector/scalar (!VM)
`shf.f Vi,Sj,Vk`	E0 AE00	Shift vector/scalar (!VM)
`shf.f Vi,Vj,Vk`	E0 A600	Shift vector/vector (!VM)
`shf.t Sj,Vk`	E1 6300	Shift vector/scalar (VM)
`shf.t Vi,Sj,Vk`	E1 AE00	Shift vector/scalar (VM)
`shf.t Vi,Vj,Vk`	E1 A600	Shift vector/vector (VM)
`shf.w #N,Sk`	ST 1980	Shift scalar word/immediate
`shf.w Sj,Sk`	E0 4440	Shift a scalar word
`sin.d Sk`	ST 7CC8	Sine of a double precision number
`sin.s Sk`	ST 7CC0	Sine of a single precision number
`snd.l Sk,`*Ceffa*	E0 3700	Send scalar/communication
`snd.w Ak,`*Ceffa*	E0 2F00	Send address/communication
`sndr.l Sk,`*effa*	E0 0D00	Send scalar register/resource
`sndr.w Ak,`*effa*	E0 0C00	Send address register/resource
`spawn`*effa*`,Ak`	E0 0500	Spawn a fork
`sqrt.d Sk`	ST 7DD8	Square root of a double float
`sqrt.d Vj,Vk`	E0 5D40	Square root double vector/vector
`sqrt.d.f Vj,Vk`	E0 5F40	Square root double (!VM)
`sqrt.d.t Vj,Vk`	E1 5F40	Square root double (VM)

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`sqrt.s Sk`	ST 7DD0	Square root of a single float
`sqrt.s Vj,Vk`	E0 5D00	Square root single vector/vector
`sqrt.s.f Vj,Vk`	E0 5F00	Square root single (!VM)
`sqrt.s.t Vj,Vk`	E1 5F00	Square root single (VM)
`st.b Ak,`*effa*	ST 2C00	Store address register byte
`st.b Sk,`*effa*	ST 3400	Store scalar byte
`st.b Vk,`*effa*	ST 3C00	Store vector byte
`st.b.f Vk,`*effa*	E0 3C00	Store vector byte (!VM)
`st.b.t Vk,`*effa*	E1 3C00	Store vector byte (VM)
`st.d Sk,`*effa*	ST 3700	Store scalar double float
`st.d Vk,`*effa*	ST 3F00	Store vector double float
`st.d.f Vk,`*effa*	E0 3F00	Store vector double float (!VM)
`st.d.t Vk,`*effa*	E1 3F00	Store vector double float (VM)
`st.h Ak,`*effa*	ST 2D00	Store address register halfword
`st.h Sk,`*effa*	ST 3500	Store scalar halfword
`st.h Vk,`*effa*	ST 3D00	Store vector halfword
`st.h.f Vk,`*effa*	E0 3D00	Store vector halfword (!VM)
`st.h.t Vk,`*effa*	E1 3D00	Store vector halfword (VM)
`st.l Sk,`*effa*	ST 3700	Store scalar longword
`st.l Vk,`*effa*	ST 3F00	Store vector longword
`st.l VLS,`*effa*	ST 0E00	Store VS and VL to memory
`st.l.f Vk,`*effa*	E0 3F00	Store vector longword (!VM)
`st.l.t Vk,`*effa*	E1 3F00	Store vector longword (VM)
`st.s Sk,`*effa*	ST 3600	Store scalar single float
`st.s Vk,`*effa*	ST 3E00	Store vector single float
`st.s.f Vk,`*effa*	E0 3E00	Store vector single float (!VM)

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`st.s.t Vk,` *effa*	E1 3E00	Store vector single float (VM)
`st.w Ak,` *effa*	ST 2E00	Store address register word
`st.w Sk,` *effa*	ST 3600	Store scalar word
`st.w Vk,` *effa*	ST 3E00	Store vector word
`st.w.f Vk,` *effa*	E0 3E00	Store vector word (!VM)
`st.w.t Vk,` *effa*	E1 3E00	Store vector word (VM)
`st.x VM,` *effa*	ST 0F00	Store VM into memory
`stcmr Ak,` *effa*	E0 0700	Store communication registers
`ste.b Sk,` *effa*	ST 2400	Store an extended scalar byte
`ste.b.f Sk,` *effa*	E0 2400	Store extended scalar byte (!VM)
`ste.b.t Sk,` *effa*	E1 2400	Store extended scalar byte (VM)
`ste.d Sk,` *effa*	ST 2700	Store an extended scalar double float
`ste.d.f Sk,` *effa*	E0 2700	Store extended scalar double (!VM)
`ste.d.t Sk,` *effa*	E1 2700	Store extended scalar double (VM)
`ste.h Sk,` *effa*	ST 2500	Store an extended scalar halfword
`ste.h.f Sk,` *effa*	E0 2500	Store extended scalar halfword (!VM)
`ste.h.t Sk,` *effa*	E1 2500	Store extended scalar halfword (VM)
`ste.l Sk,` *effa*	ST 2700	Store an extended scalar longword
`ste.l.f Sk,` *effa*	E0 2700	Store extended scalar longword (!VM)
`ste.l.t Sk,` *effa*	E1 2700	Store extended scalar longword (VM)
`ste.s Sk,` *effa*	ST 2600	Store an extended scalar single float
`ste.s.f Sk,` *effa*	E0 2600	Store extended scalar single (!VM)
`ste.s.t Sk,` *effa*	E1 2600	Store extended scalar single (VM)
`ste.w Sk,` *effa*	ST 2600	Store an extended scalar word
`ste.w.f Sk,` *effa*	E0 2600	Store extended scalar word (!VM)
`ste.w.t Sk,` *effa*	E1 2600	Store extended scalar word (VM)

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`stvi.b Sk,Vj`	ST 7B00	Scalar index store vector byte
`stvi.b Vk,Vj`	ST 7A00	Index store vector byte
`stvi.b.f Sk,Vj`	E0 7B00	Scalar index store vector byte (!VM)
`stvi.b.f Vk,Vj`	E0 7A00	Index store vector byte (!VM)
`stvi.b.t Sk,Vj`	E1 7B00	Scalar index store vector byte (VM)
`stvi.b.t Vk,Vj`	E1 7A00	Index store vector byte (VM)
`stvi.d Sk,Vj`	ST 7BC0	Scalar index store vector double float
`stvi.d Vk,Vj`	ST 7AC0	Index store vector double
`stvi.d.f Sk,Vj`	E0 7BC0	Scalar index store vector double (!VM)
`stvi.d.f Vk,Vj`	E0 7AC0	Index store vector double (!VM)
`stvi.d.t Sk,Vj`	E1 7BC0	Scalar index store vector double (VM)
`stvi.d.t Vk,Vj`	E1 7AC0	Index store vector double (VM)
`stvi.h Sk,Vj`	ST 7B40	Scalar index store vector halfword
`stvi.h Vk,Vj`	ST 7A40	Index store vector halfword
`stvi.h.f Sk,Vj`	E0 7B40	Scalar index store vector half (!VM)
`stvi.h.f Vk,Vj`	E0 7A40	Index store vector halfword (!VM)
`stvi.h.t Sk,Vj`	E1 7B40	Scalar index store vector half (VM)
`stvi.h.t Vk,Vj`	E1 7A40	Index store vector halfword (VM)
`stvi.l Sk,Vj`	ST 7BC0	Scalar index store vector longword
`stvi.l Vk,Vj`	ST 7AC0	Index store vector longword
`stvi.l.f Sk,Vj`	E0 7BC0	Scalar index store vector long (!VM)
`stvi.l.f Vk,Vj`	E0 7AC0	Index store vector longword (!VM)
`stvi.l.t Sk,Vj`	E1 7BC0	Scalar index store vector long (VM)
`stvi.l.t Vk,Vj`	E1 7AC0	Index store vector longword (VM)
`stvi.s Sk,Vj`	ST 7B80	Scalar index store vector single float
`stvi.s Vk,Vj`	ST 7A80	Index store vector single

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`stvi.s.f Sk,Vj`	E0 7B80	Scalar index store vector single (!VM)
`stvi.s.f Vk,Vj`	E0 7A80	Index store vector single (!VM)
`stvi.s.t Sk,Vj`	E1 7B80	Scalar index store vector single (VM)
`stvi.s.t Vk,Vj`	E1 7A80	Index store vector single (VM)
`stvi.w Sk,Vj`	ST 7B80	Scalar index store vector word
`stvi.w Vk,Vj`	ST 7A80	Index store vector word
`stvi.w.f Sk,Vj`	E0 7B80	Scalar index store vector word (!VM)
`stvi.w.f Vk,Vj`	E0 7A80	Index store vector word (!VM)
`stvi.w.t Sk,Vj`	E1 7B80	Scalar index store vector word (VM)
`stvi.w.t Vk,Vj`	E1 7A80	Index store vector word (VM)
`sub.b Sj,Sk`	ST 5B00	Subtract scalar/scalar integer byte
`sub.b Vi,Sj,Vk`	ST D800	Subtract vector/scalar integer byte
`sub.b Vi,Vj,Vk`	ST D000	Subtract vector/vector integer byte
`sub.b.f Vi,Sj,Vk`	E0 D800	Subtract vector/scalar byte (!VM)
`sub.b.f Vi,Vj,Vk`	E0 D000	Subtract byte vectors (!VM)
`sub.b.t Vi,Sj,Vk`	E1 D800	Subtract vector/scalar byte (VM)
`sub.b.t Vi,Vj,Vk`	E1 D000	Subtract byte vectors (VM)
`sub.d Si,Vj,Vk`	E0 8200	Subtract scalar/vector double float
`sub.d Sj,Sk`	ST 55C0	Subtract scalar/scalar double float
`sub.d Vi,Sj,Vk`	ST BE00	Subtract vector/scalar double float
`sub.d Vi,Vj,Vk`	ST B600	Subtract vector/vector double float
`sub.d.f Si,Vj,Vk`	E0 8A00	Subtract scalar/vector double (!VM)
`sub.d.f Vi,Sj,Vk`	E0 BE00	Subtract vector/scalar double (!VM)
`sub.d.f Vi,Vj,Vk`	E0 B600	Subtract double vectors (!VM)
`sub.d.t Si,Vj,Vk`	E1 8A00	Subtract scalar/vector double (VM)
`sub.d.t Vi,Sj,Vk`	E1 BE00	Subtract vector/scalar double (VM)

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`sub.d.t Vi,Vj,Vk`	E1 B600	Subtract double vectors (VM)
`sub.h #N,Ak`	ST 1500	Subtract immediate address halfword
`sub.h #n,Ak`	ST 5A80	Subtract short immediate address halfword
`sub.h #N,Sk`	ST 1508	Subtract scalar/immediate integer halfword
`sub.h Aj,Ak`	ST 5A00	Subtract address register halfword
`sub.h Sj,Sk`	ST 5B40	Subtract scalar/scalar integer halfword
`sub.h Vi,Sj,Vk`	ST DA00	Subtract vector/scalar integer halfword
`sub.h Vi,Vj,Vk`	ST D200	Subtract vector/vector integer halfword
`sub.h.f Vi,Sj,Vk`	E0 DA00	Subtract vector/scalar halfword (!VM)
`sub.h.f Vi,Vj,Vk`	E0 D200	Subtract halfword vectors (!VM)
`sub.h.t Vi,Sj,Vk`	E1 DA00	Subtract vector/scalar halfword (VM)
`sub.h.t Vi,Vj,Vk`	E1 D200	Subtract halfword vectors (VM)
`sub.l Sj,Sk`	ST 5BC0	Subtract scalar/scalar integer longword
`sub.l Vi,Sj,Vk`	ST DE00	Subtract vector/scalar integer longword
`sub.l Vi,Vj,Vk`	ST D600	Subtract vector/vector integer longword
`sub.l.f Vi,Sj,Vk`	E0 DE00	Subtract vector/scalar longword (!VM)
`sub.l.f Vi,Vj,Vk`	E0 D600	Subtract longword vectors (!VM)
`sub.l.t Vi,Sj,Vk`	E1 DE00	Subtract vector/scalar longword (VM)
`sub.l.t Vi,Vj,Vk`	E1 D600	Subtract longword vectors (VM)
`sub.s #N,Sk`	ST 1888	Subtract scalar/immediate single float
`sub.s Si,Vj,Vk`	E0 8000	Subtract scalar/vector single float
`sub.s Sj,Sk`	ST 5580	Subtract scalar/scalar single float
`sub.s Vi,Sj,Vk`	ST BC00	Subtract vector/scalar single float
`sub.s Vi,Vj,Vk`	ST B400	Subtract vector/vector single float
`sub.s.f Si,Vj,Vk`	E0 8800	Subtract scalar/vector single (!VM)
`sub.s.f Vi,Sj,Vk`	E0 BC00	Subtract vector/scalar single (!VM)

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`sub.s.f Vi,Vj,Vk`	E0 B400	Subtract single vectors (!VM)
`sub.s.t Si,Vj,Vk`	E1 8800	Subtract scalar/vector single (VM)
`sub.s.t Vi,Sj,Vk`	E1 BC00	Subtract vector/scalar single (VM)
`sub.s.t Vi,Vj,Vk`	E1 B400	Subtract single vectors (VM)
`sub.w #N,Ak`	ST 1580	Subtract immediate address word
`sub.w #n,Ak`	ST 5AC0	Subtract short immediate address word
`sub.w #N,Sk`	ST 1588	Subtract scalar/immediate integer word
`sub.w Aj,Ak`	ST 5A40	Subtract address register word
`sub.w Sj,Sk`	ST 5B80	Subtract scalar/scalar integer word
`sub.w Vi,Sj,Vk`	ST DC00	Subtract vector/scalar integer word
`sub.w Vi,Vj,Vk`	ST D400	Subtract vector/vector integer word
`sub.w.f Vi,Sj,Vk`	E0 DC00	Subtract vector/scalar word (!VM)
`sub.w.f Vi,Vj,Vk`	E0 D400	Subtract word vectors (!VM)
`sub.w.t Vi,Sj,Vk`	E1 DC00	Subtract vector/scalar word (VM)
`sub.w.t Vi,Vj,Vk`	E1 D400	Subtract word vectors (VM)
`sum.b Vk`	ST 7E00	Sum a vector of bytes
`sum.b.f Vk`	E0 7E00	Sum a vector of bytes (!VM)
`sum.b.t Vk`	E1 7E00	Sum a vector of bytes (VM)
`sum.d Vk`	ST 7E88	Sum a vector of double float
`sum.d.f Vk`	E0 7E88	Sum a vector of double (!VM)
`sum.d.t Vk`	E1 7E88	Sum a vector of double (VM)
`sum.h Vk`	ST 7E08	Sum a vector of halfwords
`sum.h.f Vk`	E0 7E08	Sum a vector of halfwords (!VM)
`sum.h.t Vk`	E1 7E08	Sum a vector of halfwords (VM)
`sum.l Vk`	ST 7E18	Sum a vector of longwords
`sum.l.f Vk`	E0 7E18	Sum a vector of longwords (!VM)

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`sum.l.t Vk`	E1 7E18	Sum a vector of longwords (VM)
`sum.s Vk`	ST 7E80	Sum a vector of single float
`sum.s.f Vk`	E0 7E80	Sum a vector of single (!VM)
`sum.s.t Vk`	E1 7E80	Sum a vector of single (VM)
`sum.w Vk`	ST 7E10	Sum a vector of words
`sum.w.f Vk`	E0 7E10	Sum a vector of words (!VM)
`sum.w.t Vk`	E1 7E10	Sum a vector of words (VM)
`sysc #r,#g`	ST 1080	Perform a system call
`tac` *effa*	ST 0800	Test and clear a byte in memory
`tas` *effa*	ST 0C00	Test and set a memory byte
`trap #rm,#b`	ST 1A00	Force a trap system exception
`tst` *Ceffa*	E0 0100	Test communication register lock bit
`tstvv`	ST 7D78	Test value of vector valid flag
`tzc Sj,Sk`	ST 45C0	Count of trailing zeros in Sj
`tzc Vj,Vk`	ST 6200	Trailing zero count vector
`tzc.f Vj,Vk`	E0 6200	Trailing zero count vector (!VM)
`tzc.t Vj,Vk`	E1 6200	Trailing zero count vector (VM)
`ulk` *Ceffa*	E0 0300	Unlock communication register
`wfork`	ST 7C98	Wait for a fork
`xmti Sk`	ST 7D68	Transmit interrupt
`xor #N,Ak`	ST 1300	Exclusive OR immediate to address register
`xor #N,Sk`	ST 1308	Exclusive OR scalar/immediate
`xor Aj,Ak`	ST 5280	Exclusive OR address register
`xor Sj,Sk`	ST 5380	Exclusive OR scalar/scalar
`xor Vi,Sj,Vk`	ST AC00	Exclusive OR vector/scalar
`xor Vi,Vj,Vk`	ST A400	Exclusive OR two vectors

Table 18 (continued)
Instructions sorted by mnemonic

Instruction mnemonic	Op code (Hex)	Instruction description
`xor.f Vi,Sj,Vk`	E0 AC00	Exclusive OR vector/scalar (!VM)
`xor.f Vi,Vj,Vk`	E0 A400	Exclusive OR two vectors (!VM)
`xor.t Vi,Sj,Vk`	E1 AC00	Exclusive OR vector/scalar (VM)
`xor.t Vi,Vj,Vk`	E1 A400	Exclusive OR two vectors (VM)
`xpnd.f Vj,Vk`	E0 6280	Expand a vector (!VM)
`xpnd.t Vj,Vk`	E1 6280	Expand a vector (VM)

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

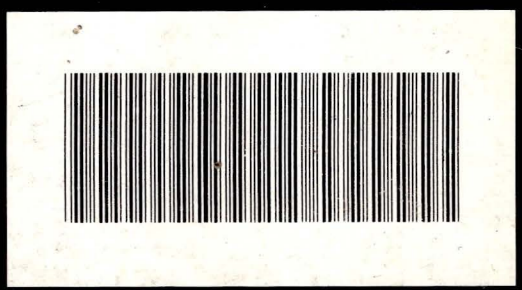
T

U

V

W

X



Order Number
DHW-301

Document Number
081-011930-000