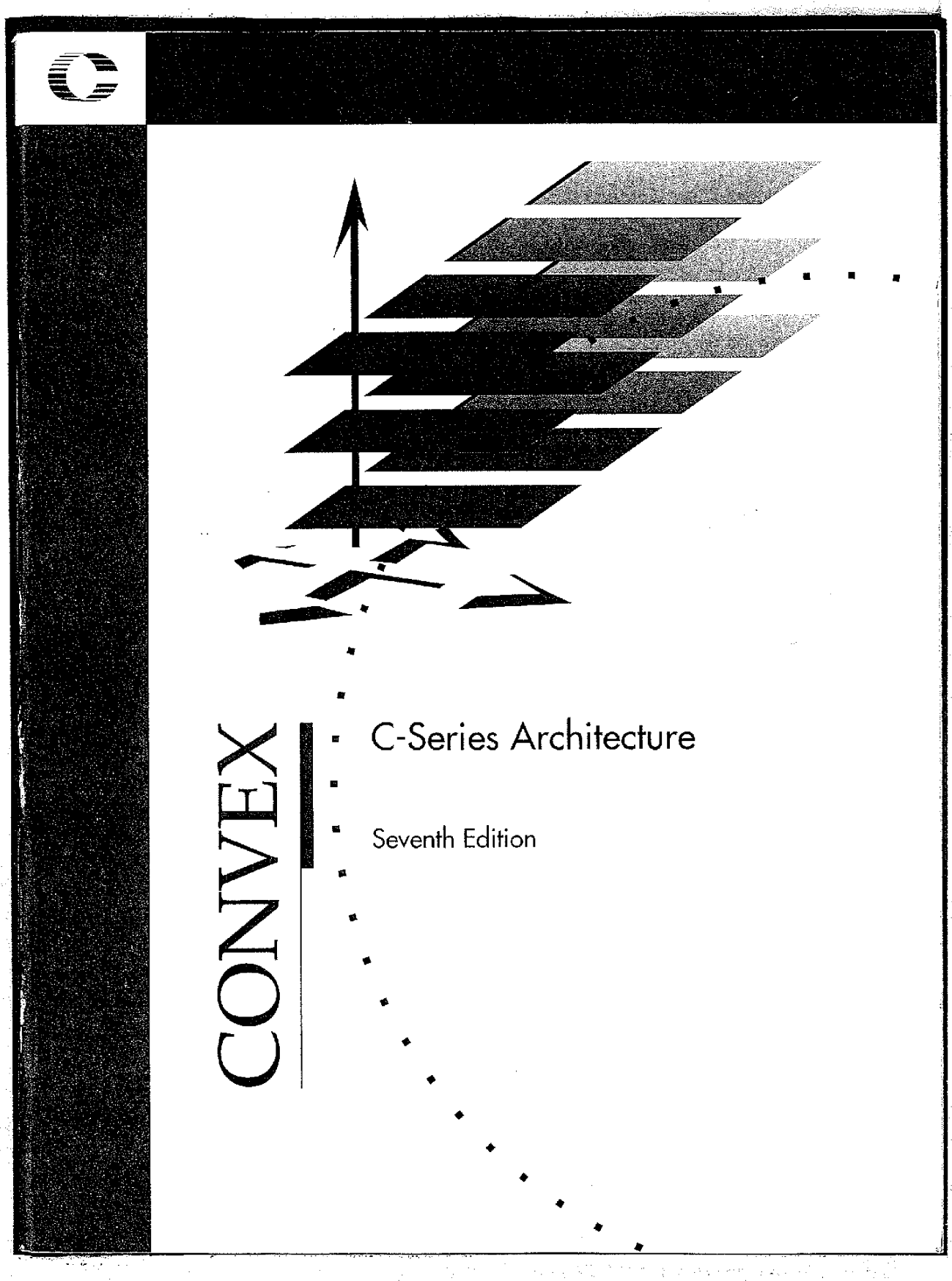


EDTX 2-04-CV-120
51056DOC065300

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

CONVEX
C-Series Acrhitecture

Order No. DSW-300

Seventh Edition
March 1994

CONVEX Press
Richardson, Texas
United States of America

CONVEX
C-Series Architecture

Order No. DSW-300

Printed in the United States of America

Revision information for
CONVEX
C-Series Architecture

Edition	Document No.	Description
Seventh	081-011830-001	Released March 1994. Documents C4600 Series as part of the C-Series architecture. Changes the document name to *C-Series Architecture*.
Sixth	081-011830-000	Released March 1993. Separates the assembly language instruction set to the new *CONVEX Assembly Language Reference Manual (C Series)*, from the *CONVEX Architecture Reference Manual (C100, C200 Series)*, renamed the *CONVEX Assembly Language Reference Manual (C Series)*. Documents C3400 and C3800 Series CPUs as part of the C Series architecture and clarifies the separation of C100 Series CPUs (single-processor operation) from multiprocessing CPU operation.
Fifth	081-000050-203	Released May 1990. Updates the fourth edition of *CONVEX Architecture Reference Manual (C100, C200 Series)*. Documents the addition of the load cache bypass instruction.
Fourth	081-000050-202	Released May 1989. Updates the third edition of *CONVEX Architecture Reference Manual (C100, C200 Series)*. Documents the addition of the memory instruction duals.
Third	081-000050-201	Released October 1988. Updates the version 2.0 of *CONVEX C1/C120/C210 Architecture Reference Manual*. Changes document name to *CONVEX Architecture Reference Manual (C100, C200 Series)*. Documents the C100 Series architecture (C1 and C120 CPUs) and the C200 Series architecture (C201, C202, C210, C220, C230, and C240 CPUs).
Second	081-000050-200	Released February 1988, version 2.0. Updates version 1.2 of *CONVEX Architecture Handbook*. Changes document name to *CONVEX C1/C120/C210 Architecture Reference Manual*. Documents the expansion of the original instruction set, transparent changes to the interrupt structure, and changes to the C210A architecture and generic C1 Series architecture.
First	080-000120-000	Released 1984. Version 1.0 (initial release) of the *CONVEX Architecture Handbook*.

Contents

EDTX 2-04-CV-120
51056DOC065305

4 Memory management 69

5 Multiprocessor management 133

Figures

Tables

Using this manual

Purpose and audience

The *CONVEX C-Series Architecture* describes the architecture of the CONVEX C-Series supercomputers. It is a companion to the *CONVEX Assembly Language Reference Manual (C Series)*.

This document applies to all CONVEX C-Series architecture CPUs, including the C100, C3200, C3400, C3800, and C4600 Series CPUs. It serves as a tool to help engineers and software developers make maximum use of any CONVEX processor's facilities.

Notational conventions

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

Text notation

Text notation conventions set apart special items.

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

  Example:

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


EDTX 2-04-CV-120
51056DOC065319

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

  Example:

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

- Bold uppercase names designate keycap names.

  Example:

  RETURN

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

  Example:

  ESC Q

- If two keycap names are separated by a hyphen, they are 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 bits as fields 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 mandatory 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 previous conventions are used in the example that follows to define the commands in the user interface.

Example:

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

In the example:

- `command` is required and may be abbreviated to `com` (square brackets indicate optional portion).
- If a command option (indicated by a list in braces, separated by a vertical slash) is used, then either `.t` or `.f`, if required.
- If a command option (indicated 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.

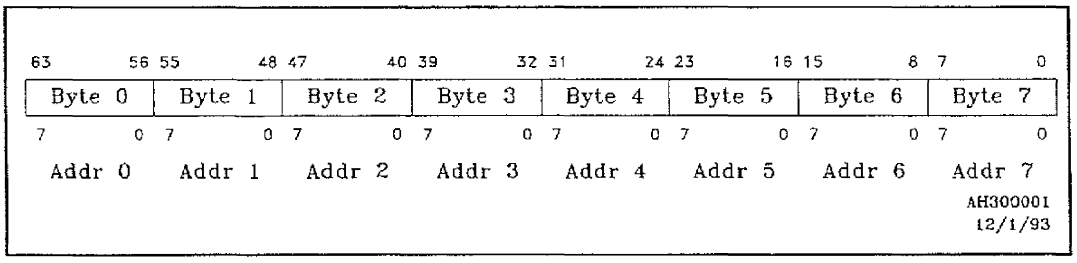
Data notation

The following 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.
  - For C100 Series CPUs, only the standard instruction can be used. In the standard instruction, the first halfword is an op code and the remaining halfwords are operands.
  - For multiprocessing C-Series CPUs (C3200, C3400, C3800, and C4600 Series CPUs), either the standard or the extended instruction can be used. In the extended instruction, 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 (LIFO).
- 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.

Notes, cautions, and warnings

This document presents notes, cautions, and warnings in the following formats.

Note

A Note highlights supplemental information.

Caution

A Caution highlights information necessary to avoid damage to the system.

Warning

A warning highlights information necessary to avoid injury to personnel.

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 Assembly Language Reference Manual (C Series)*, **Order No. DSW-301**—This manual is a reference guide for developing software for CONVEX C-Series processors. It contains the formats for the CONVEX C-Series instruction set.
- *CONVEX Processor Diagnostics Manual (C Series)*, **Order No. DSW-302**—This manual documents the service processor unit (SPU)-based processor diagnostics for CONVEX supercomputers.
- *CONVEX System Manager's Guide*, **Order No. DSW-004**—This manual is written for 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.

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 "DSW-" or "DHW-."

The order number for the *CONVEX C Series Architecture* is DHW-300.

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.
- From all locations in the United States, CONVEX employees call (800)545-4839.

EDTX 2-04-CV-120
51056DOC065326

- From locations in Canada, customers and CONVEX employees call (800)345-2384.
- From all other locations, contact the nearest CONVEX office.

Introduction 1

This document is a reference for the CONVEX C-Series architecture. As new model numbers are added to any series, the material in this document may apply in whole or in part. These lists should not be considered exclusive.

The C100 Series includes the C1 and C120 CP Us.

The implementation of the C-Series architecture on the C200 Series and C3200 Series CPUs is identical, therefore references in this book are written for the C3200 Series CPUs. The C3200 Series includes the C210, C220, C230, C240, C3210, C3220, C3230, C3240 CPUs.

The C3400 Series includes the C3410ES, C3420ES, C3410, C3420, C3430, C3440, C3460, and C3480 CPUs.

The C3800 Series includes the C3810, C3820, C3830, C3840, C3460, and C3880 CPUs.

The C4600 Series includes the C4610, C4620, C4630, and C4640 CPUs

All C100 Series CPUs are single processors. Multiprocessing C-Series CPUs include the C3200, C3400, C3800, and C4600 Series CPUs.

EDTX 2-04-CV-120
51056DOC065329

The CONVEX C-Series architecture

The architecture presented in this manual defines the specifications of the central processing unit (CPU) of the CONVEX supercomputers.

The term *architecture* is defined as the attributes of a system as seen by the programmer (the conceptual structure and functional behavior), as distinct from the organization of the data flow and controls, the logical design, and the physical implementation. Within this context, an architectural specification defines the following attributes, as perceived by the programmer and the hardware designer:

- Data representations
- Register sets
  - General registers
  - Communication registers
- Instruction set
- Memory management
  - Physical address space
  - Virtual address space
  - Process structure and control
  - Virtual memory management
  - Virtual memory protection
  - Caches
- Multiprocessor management
  - Automatic self-allocating processors
  - Parallel processing mechanisms
  - Forking and spawning mechanism
  - Memory protection mechanisms
- Exception and interrupt mechanisms

The instruction set is described in the *CONVEX Assembly Language Reference Manual (C Series).*

The CONVEX C-Series architecture incorporates the following features:

- An integrated vector processor incorporated within the system for high-speed operation
- A full range of fixed and floating-point data types

- A total capacity of four Gbytes of virtual memory - two Gbytes available to support large user programs and data, and two Gbytes to support the operating system
- Large, high-speed register sets (address, scalar, and vector) that support high-performance operation for address calculations in parallel with scalar and vector calculations
- Communication registers and multiprocessing structures (in the multiprocessing C-Series CPUs)
- Multilevel protection systems that support and separate users, thereby enhancing system reliability and increasing the performance of operating system functions

All CONVEX C-Series CPUs share a common architecture, in most respects. However, some software, such as the ConvexOS operating system, use features whose implementation varies among different CONVEX CPUs. Although not immediately visible to the user, the fine detail and construction of these features are visible to the ConvexOS software.

Chapter summaries

This section contains brief summaries of the chapters that follow. In addition, a brief summary of the instruction set is found in the *CONVEX Assembly Language Instruction Set (C Series)*.

- Data representations and operations
- General registers
- Memory management
- Multiprocessor management
- Exceptions and interrupts
- Implementation-specific features

Data representations

There are three binary numeric data representations:

- Signed fixed point integer
- Unsigned fixed point integer
- Floating point

The CONVEX processors support four fixed-point integer precisions. Signed fixed-point numbers are interpreted as two's complement representations. Integer quantities exist in four lengths:

- **Byte**—8 bits
- **Halfword**—16 bits
- **Word**—32 bits
- **Longword**—64 bits

The CONVEX CPUs support both native and IEEE standard floating point number representations in two formats: single-precision word (32 bits) and double-precision longword (64 bits). Both formats are interpreted as binary, normalized fractions with an implicit value of "1" in the most-significant bit of the fraction. The exponent is a biased power of two, scale factor.

An address or logical value is treated as an unsigned 32-bit integer usually contained in the address registers. For numeric purposes, an address register may be treated as a signed or unsigned 32-bit integer.

Virtual addresses are byte-granular. Instruction operands in memory may begin on any byte boundary that allows all byte locations within a given data type to be used, even though the operands may be unrelated.

Operations are performed in integer and floating-point. Floating-point operations are performed in native and IEEE modes. The differences are delineated for add, subtract, multiply, divide, square root, compare, and conversion operations.

Register sets

There are three general register sets and several status registers. The three register sets are partitioned according to the type of operand to be manipulated:

- Address registers
- Scalar registers
- Vector register

There are four general status registers and three privileged flags. The four status registers are:

- Program counter (PC)
- Processor status word (PSW)
- Scalar stride zero (SS0 - C4600 Series CPUs only)
- Scalar stride one (SS1 - C4600 Series CPUs only))

The three privileged flags are:

- Interrupt on (ION)
- Realtime interrupt on (RT_ION - C3400 Series CPUs only)
- Vector valid flag (VV)

Memory management

The memory management unit (MMU) supports the operating system in providing a versatile and reliable virtual memory programming environment. The CONVEX C-Series architecture provides 4 Gbytes of virtual memory in its virtual address space partitioned into eight 512-Mbyte segments. Four segments are allocated to the operating system and four segments to the user. The maximum size of a user program (instructions and data) is limited to 2 Gbytes. The operating system data structures and instructions necessary to manage the user program occupy the remaining 2 Gbytes of virtual storage.

Because the address space of the CONVEX system architecture is virtual, an address may be a valid logical address, but the referenced data may or may not be in physical memory. Memory is managed as pages on a fixed-size basis.

Since the operating system is embedded within the user-virtual address space, it must be protected from the user. The memory protection system protects the user's programs from other users' programs, while supporting time-sharing and operating system structures.

This system is based on hierarchical structures called *rings* and

- supports embedding the operating system in the user's virtual address space,
- contains certain access violations to the user's process,
- permits implementing the operating system efficiently, and
- enhances operating system call processing by reducing the time for context switching.

Multiprocessor management

Multiprocessing is the creation and scheduling of individual processes on any subcomplex. The multiprocessor management hardware incorporated in each C-Series architecture CPU provides the operating system and user a simple and flexible set of instructions for dynamic allocation, deallocation, and communication. Each CPU in a C-Series architecture complex operates independently as a 64-bit supercomputer. The multiprocessor management hardware binds these CPUs into a tightly coupled set with shared memory. This implements a multi-instruction multi-data (MIMD) architecture that provides a parallel execution environment for user applications.

Exceptions and interrupts

Exceptions occur when a currently executing program encounters event such as arithmetic inconsistencies, address translation faults, or some asynchronous event (such as an interrupt). When an exception occurs, control is transferred to a predetermined address whose value is a function of the exception.

Interrupts are the result of events that occur asynchronously and belong to the system, not to the executing process. When an interrupt occurs, the processor jumps to a particular interrupt handler determined by the interrupt source.

All I/O data references by the CPU are memory mapped. There are no explicit I/O instructions. The I/O registers and memory status bits are referenced through the appropriate logical-to-physical address mapping.

Implementation-specific features

Some CONVEX CPUs implement some CPU functions through registers located in the I/O address space. However, the CPU uses only a fraction of I/O address space for physical implementation of registers. Registers in I/O address space are addressed in much the same way as elements of main memory. This allows access to a number of subsystems required for proper operation of the various machines. The I/O address space is implementation specific, resulting in significant differences between the single processor and multiprocessor implementations.

The C3400, C3800, C4600 Series CPUs do not have I/O address space.

EDTX 2-04-CV-120
51056DOC065336

Instruction set

The CONVEX C-Series architecture includes an instruction set that provides minimum functionality per instruction.

The instruction set is projected orthogonally, that is, each instruction op code is defined so that it is a constant hexadecimal address distance from another op code. Orthogonally specifying the instruction set simplifies instruction decoding by hardware.

Even though the fundamental addressable unit is the byte, instructions are addressed on a halfword (even byte) boundary. An instruction may be one, two, three, or four halfwords in length, equivalent to 16, 32, 48, or 64 bits, respectively.

A standard instruction is one to three halfwords in length. An extended instruction is two to four halfwords in length, since the extended instructions contain a halfword prefix of either `7EF0` or `7EF8`, prior to the op code itself. See the *CONVEX Assembly Language Reference Manual (C Series)* for details about the instruction set.

EDTX 2-04-CV-120
51056DOC065337

Data representations 2

CONVEX C-Series CPUs support three data representations:

- Signed numeric fixed-point integer
- Unsigned numeric fixed-point integer
- Numeric floating-point integer

An address or logical value is treated as unsigned. The C-Series architecture supports the IEEE and native floating-point data representations with a 64-bit, double-precision format and a 32-bit, single-precision format. However, the *complete* IEEE floating-point specification is *not* supported in the C-Series architecture. Specifically, the C-Series architecture uses the same algorithms to compute both IEEE and native floating-point values.

Instructions that manipulate the data representations found in this chapter are discussed in the *CONVEX Assembly Language Reference Manual (C Series).*

EDTX 2-04-CV-120
51056DOC065339

Basic data representations

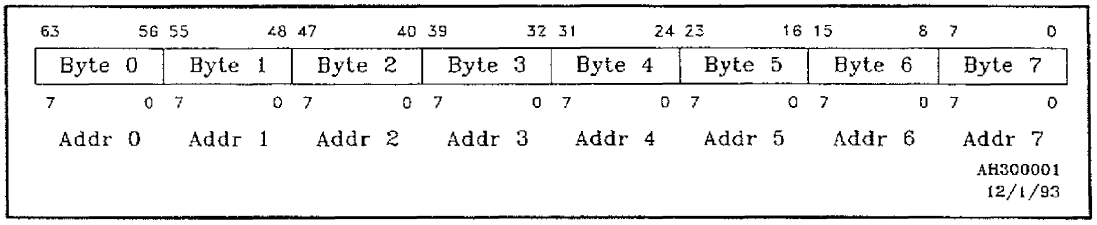
The C-Series architecture has four basic addressable data representations. Each data representation must start on an addressable byte boundary:

- **Byte**—8 contiguous bits
- **Halfword**—16 contiguous bits
- **Word**—32 contiguous bits
- **Longword**—64 contiguous bits

Bit numbering is left to right, *n*-1 through 0, where *n* is the number of bits in the data type. 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 left to right, 0 through 7. The most-significant bit is associated with the leftmost byte. Figure 2 shows the ordering of bits and bytes within a 64-bit longword.

Figure 2 Memory longword structure



Data representation memory alignment

The C-Series virtual address space is *byte granular*, meaning that operands can begin on any byte boundary, unless otherwise noted in a particular instruction definition. Overall system performance may degrade when operands do not begin on appropriate boundaries.

Data representations should be aligned on a boundary address as specified in the following alignment rules, to ensure maximum execution speed:

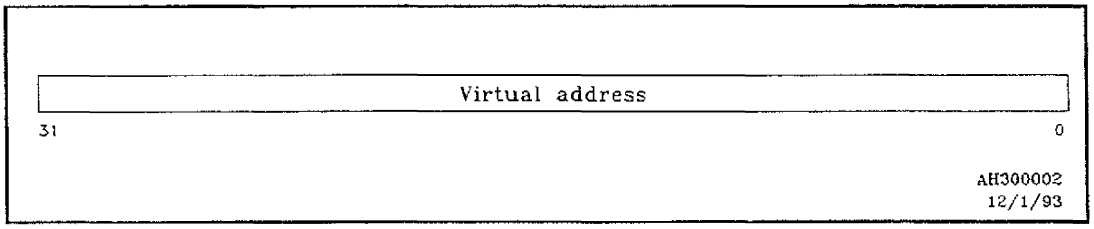
- **Byte (8 bits)**—No preference
- **Halfword (16 bits)**—Least-significant address bit = 0
- **Word (32 bits)**—Least-significant two address bits = 0
- **Longword (64 bits)**—Least-significant three address bits = 0

Virtual addresses

Many virtual addresses reside either in instructions or in memory as indirect addresses. They are always unsigned, 32-bit integers.

Figure 3 shows the virtual address format.

Figure 3 Virtual address format



Mixed mode arithmetic

Unless otherwise specified, mixed mode arithmetic on data representations or manipulations on operands in registers must follow the provided conventions. Results that can be reproduced from one implementation to another cannot be guaranteed if defined conventions are circumvented.

Note

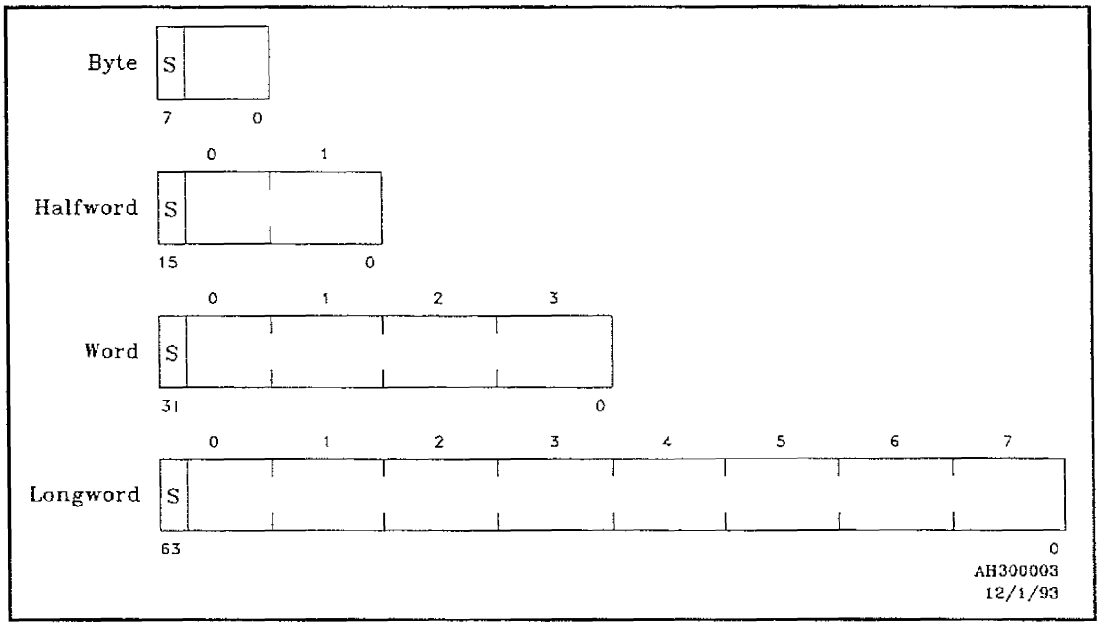
Any attempt to circumvent these conventions through knowledge of an internal representation can produce inaccurate results and is not recommended.

Signed fixed-point integer representations

The C-Series architecture defines four signed fixed-point integer representations: 8, 16, 32, and 64 bits.

The formats of these four fixed-point data types are shown in Figure 4.

Figure 4 Signed fixed-point integer representations



In Figure 4, S is the sign bit. A binary 0 denotes positive. A binary 1 denotes negative. Signed fixed-point numbers use the two's complement numbering system.

If $0 \le i \le n\text{-}2$, where n is the number of bits in the data item, then bit i has weight 2^i.

The most-significant bit, the sign bit, has a weight equal to $-1 \times 2^{n-1}$, where n is the number of bits in the data item.

Both of the previous statements can be combined and represented as the following expression for signed fixed-point integers:

$$-1 \times 2^{n-1} \times b_{n-1} + \sum_{i=0}^{n-2} 2^i \times b_i$$

where

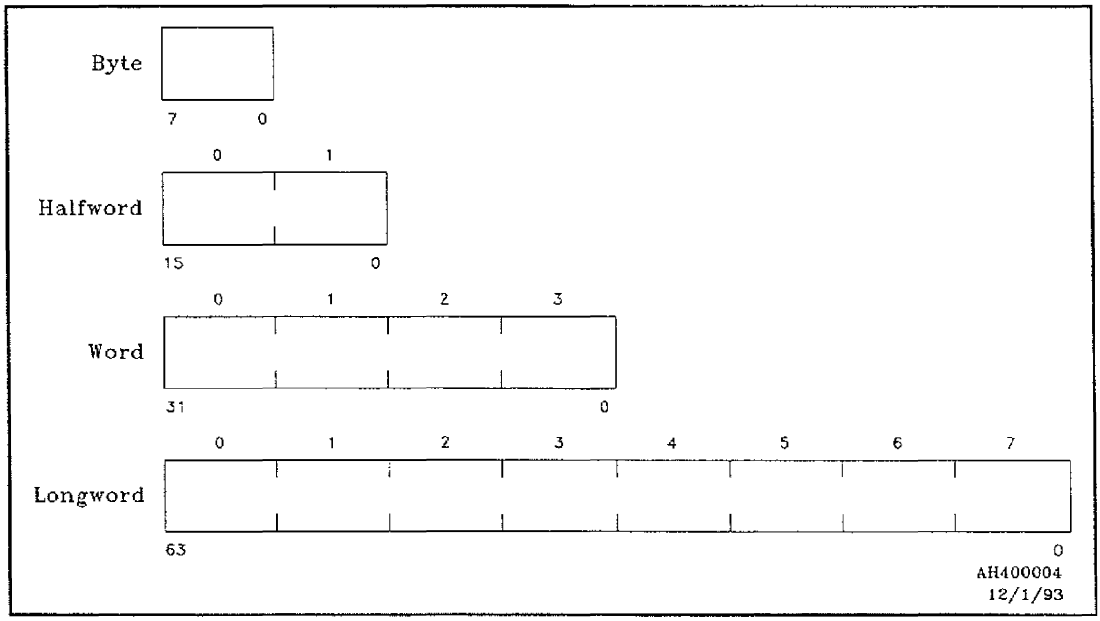
- n is the number of bits in the data item, and
- $b_i = 0$ if bit i is clear, and $b_i = 1$ if bit i is set.

Unsigned fixed-point integer representations

The C-Series architecture defines four unsigned fixed-point integer representations: 8, 16, 32, and 64 bits.

The formats of these four fixed-point data types are shown in Figure 5.

Figure 5 Unsigned fixed-point integer representations



If $0 \le i < (n-1)$ where n is the number of bits in the data item, then bit i has weight 2^i.

An unsigned fixed-point integer is represented as

$$\sum_{i=0}^{n-1} 2^i \times b_i$$

where

- n is the number of bits in the data type, and
- $b_i = 0$ if bit i is clear, and $b_i = 1$ if bit i is set.

Floating-point representations

The C-Series architecture supports native and IEEE-standard floating-point number representations in two formats:

- A single-precision word (32 bits)
- A double-precision longword (64 bits)

Both formats have biased binary exponents and normalized binary fractions. The fractions have an implicit 1 bit in the most-significant bit position.

The C-Series architecture *does not* support the *complete* IEEE floating-point specification. Specifically, it does not support the following:

- Gradual underflow
- IEEE rounding algorithms
- Directed rounding

The C-Series architecture uses the same algorithms to compute IEEE and native floating-point values. However, some floating-point exception conditions are treated differently:

- Not a number (NaN)
- Infinity
- Overflow
- Underflow

These algorithms are presented in this chapter following the native floating-point format and IEEE floating-point format discussions.

Native floating-point implementation

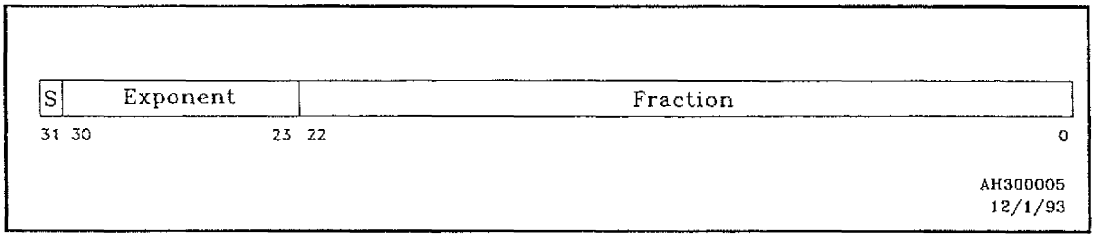
The C-Series native floating-point formats define the following operands as valid input:

- **Normalized**—The exponent is not all zeros.
- **Reserved**—The exponent is all zeros, the fraction can be anything, the sign is 1.

Native single-precision floating-point format

The format of the single-precision (32-bit) floating-point number is shown in Figure 6.

Figure 6 Native single-precision floating-point format



S	The sign bit. A binary 0 denotes positive, a binary 1 denotes negative. This form is termed the sign-magnitude representation.
Exponent	A binary-biased exponent. The algebraic value of the exponent is determined by subtracting 128 from the unsigned binary value of bits <30..23>.
Fraction	A fractional value. An implicit 1 bit is to the left of bit <22>. The binary point is to the left of the implicit 1 bit.

The input operands of a native single-precision (32-bit) floating-point number are shown in Table 1.

Table 1 Native single-precision input operands

Sign (S)	Exponent (e)	Fraction (f)	Value	Name
1	0	NA	None	Reserved operand
0	0	NA	0	Floating-point zero
0	0	0	0	True zero
1/0	1 ... 255	NA	$(-1)^S (2^{e-128}) (2^{-1} + \text{fraction})$	Normalized number

The dynamic range of a native single-precision (32-bit) floating-point number is shown in Table 2.

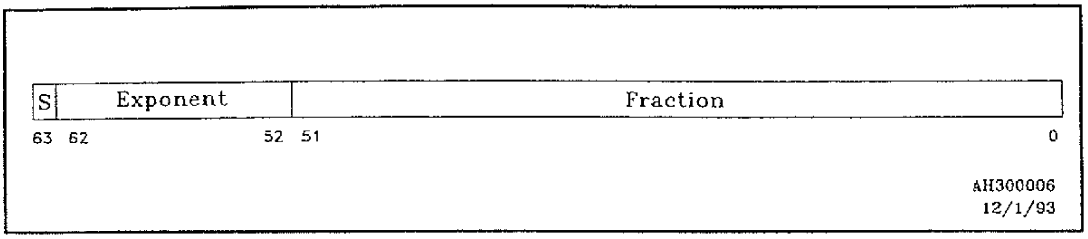
Table 2 Native single-precision dynamic range

Value	Hexadecimal	Approximate value
Largest positive	7FFF FFFF	$+1.7014117 \times 10^{+38}$
Smallest positive	0080 0000	$+2.9387359 \times 10^{-39}$
Zero	0000 0000	0
Smallest negative	8080 0000	$-2.9387359 \times 10^{-39}$
Largest negative	FFFF FFFF	$-1.7014117 \times 10^{+38}$

Native double-precision floating-point format

The format of the double-precision (64-bit) floating-point number is shown in Figure 7.

Figure 7 Native double-precision floating-point format



S	The sign bit. A binary 0 denotes positive, a binary 1 denotes negative. This form is termed the sign-magnitude representation.
Exponent	An 11-bit binary-biased exponent. The decimal value of the exponent is determined by subtracting 1,024 from the unsigned binary value of bits <62..52>.
Fraction	A fractional value. An implicit 1 bit is to the left of bit <51>. The binary point is to the left of the implicit 1 bit.

The input operands of a native double-precision (64-bit) floating-point number are shown in Table 3.

Table 3 Native double-precision input operands

Sign (S)	Exponent (e)	Fraction (f)	Value	Name
1	0	NA	None	Reserved operand
0	0	NA	0	Floating-point zero
0	0	0	0	True zero
1/0	1... 2047	NA	$(-1)^S (2^{e-1024}) (2^{-1} + \text{fraction})$	Normalized number

The input operands of a native double-precision (64-bit) floating-point number are shown in Table 4.

Table 4 Native double-precision dynamic range

Value	Hexadecimal	Approximate value
Largest positive	7FFF FFFF FFFF FFFF	$+8.988465674311579 \times 10^{+307}$
Smallest positive	0010 0000 0000 0000	$+5.562684646268003 \times 10^{-309}$
Zero	0000 0000 0000 0000	0
Smallest negative	8010 0000 0000 0000	$-5.562684646268003 \times 10^{-309}$
Largest negative	FFFF FFFF FFFF FFFF	$-8.988465674311579 \times 10^{+307}$

Native reserved operands

There are certain reserved or special operands within the native floating-point format. In particular, these operands initiate an exception when used as input to a floating-point operation. A native floating-point number (single or double) that has a sign bit of 1 and an exponent of 0 is defined as a *reserved operand*. The value of the fraction bits is unimportant. A reserved operand exception is detected if a reserved operand is encountered during a native floating-point numeric operation (for example, add, subtract, compare, or max).

A reserved operand is the result of a floating-point overflow. A reserved operand is also generated from illegal operations (divide-by-zero, for example). In cases where the input operand or operands are representable numbers, but where a reserved operand is returned as the result, no reserved operand exception is generated. However, a reserved operand exception is generated if the result is then used as an input operand to a subsequent operation.

Native floating-point zero

A native *zero* is a floating-point number with an exponent of 0 and a sign of 0. The value of the fraction is unimportant.

True zero is a native floating-point zero with a fraction of all zeros.

True zero is always returned when the result of an operation is zero. If two floating-point zeros with different fractions are compared for floating-point equality, the result is true.

Native rounding

All floating-point operations may be thought of as calculating the infinitely precise result based on the operands and the operation (add or subtract, for example). The value returned is

the representable result (normalized number or true zero) that is closest to the infinitely precise result. If the infinitely precise result is *exactly* halfway between two possible representations, the one that has a least-significant bit of zero is returned. This method is sometimes called *rounding to nearest* or *unbiased rounding to even*, denoted as R^*.

For all operations except divide and square root, this rounding is implemented by first calculating three additional result bits that are less significant than the LSB of the mantissa. The three bits are called the *guard*, *round*, and *sticky* bit, from MSB to LSB respectively. The sticky bit indicates whether any binary ones were shifted right and out of the round bit during any alignment operation. If the guard bit is set and either the round bit, the sticky bit, or the LSB of the result is set, a one is added to the LSB of the result.

For divide and square root operations, only the guard and round bits are calculated. Rounding is performed by adding one to the LSB of the result if the guard bit is set and either the round or the LSB of the result is set.

The 4600 Series CPUs support an alternate rounding mode for division and square root. This mode is based on the fact that the infinitely precise result of a divide or square root can never be exactly half way between two possible representable values. Therefore, if the guard bit of a divide or square root intermediate result is set, then either the round bit or the uncalculated sticky bit must also be set. Rounding is then performed by adding one to the LSB of the result, if the guard bit alone is set. This rounding mode is scan selectable at boot.

Native operations

The following subsections detail the results returned and the exceptions generated (if any) for native-mode floating-point operations. They contain details that are specific to the current implementation but are not part of the architecture. In particular, when the reserved operand is returned, it can be in one of two specific forms in the current implementation. These two forms are the `RSV0` and `RSV1` and are described in Table 5.

When a reserved operand is returned as a result, the C-Series architecture specifies that any legal form of the reserved operand may be returned. The two specific forms of the reserved operands that are currently returned are implementation-specific, and may be changed in the future. Table 5 lists the abbreviations used during each IEEE arithmetic operation.

In the native-mode definitions (specifically with respect to operands and results), the descriptions imply the positive or negative form of the value. When the symbols used for these definitions are preceded by + or -, the specific value is positive or negative. For example, NORM represents a positive or negative normalized number, while -NORM represents a negative normalized number only.

Native compare operations

Only a comparison status is returned for compare operations. Input operands versus exceptions generated are identical to add or subtract operations, except that UN and OV exceptions are not possible.

Table 5 Native floating-point nomenclature

Nomenclature type	Symbol	Description
Input operands	NORM	A normalized number.
	ZERO	Any form of zero. True zero where all bits are 0 or a *dirty zero* where the sign and exponent bits are 0, but one or more mantissa bits are 1.
	RSV	Any form of the reserved operand.
	INT	A nonzero two's-complement integer.
	INT0	Integer zero.
Operation results	NORM	A normalized number.
	0	True zero, or integer zero.
	RSV0	A form of the reserved operand where the sign is a 1, all exponent bits are 0, and all mantissa bits are 0.
	RSV1	A form of the reserved operand where the sign is a 1, all exponent bits are 0, and all mantissa bits are 0, except the LSB, which is a 1.
	INT	A nonzero representable integer.
	TRN	The least significant bits of an integer whose value contains more bits of precision than can be stored in the result.
Result conditions	(NM)	If result is a normalized number.
	(OV)	If overflow results.
	(UN)	If underflow results.
	(IN)	If result is a representable integer.
	(IO)	If integer overflow results.
Exceptions	RO	Illegal input operand.
	SQRN	Square root of a negative number.
	FDZ	Floating-point divide-by-zero.
	SIV	Integer overflow.
	OV	Floating-point overflow.
	UN	Floating-point underflow.
	FIN	Floating-point intrinsic error.
Exception states	0	Exception did not occur.
	1	Exception did occur.

Native add or subtract

Table 6 lists the exceptions encountered for each respective operand combination used in a native add or subtract operation.

Table 6 Native operation results—add or subtract

Operand A	Operand B	Result	RO	SQRN	FDZ	SIV	OV	UN
ZERO	ZERO	0	0	0	0	0	0	0
ZERO	NORM	NORM	0	0	0	0	0	0
ZERO	RSV	RSV1	1	0	0	0	0	0
NORM	ZERO	NORM	0	0	0	0	0	0
NORM	NORM	(NM) NORM	0	0	0	0	0	0
NORM	NORM	(OV) RSV0	0	0	0	0	1	0
NORM	NORM	(UN) 0	0	0	0	0	0	1
NORM	RSV	RSV1	1	0	0	0	0	0
RSV	ZERO	RSV1	1	0	0	0	0	0
RSV	NORM	RSV1	1	0	0	0	0	0
RSV	RSV	RSV1	1	0	0	0	0	0

Native multiply operations

Table 7 lists the exceptions encountered for each respective operand combination used in a native multiply operation.

Table 7 Native operation results—multiply

Operand A	Operand B	Result	RO	SQRN	DZ	SIV	OV	UN
ZERO	ZERO	0	0	0	0	0	0	0
ZERO	NORM	0	0	0	0	0	0	0
ZERO	RSV	RSV1	1	0	0	0	0	0
NORM	ZERO	0	0	0	0	0	0	0
NORM	NORM	(NM) NORM	0	0	0	0	0	0
NORM	NORM	(OV) RSV0	0	0	0	0	1	0
NORM	NORM	(UF) 0	0	0	0	0	0	1
NORM	RSV	RSV1	1	0	0	0	0	0
RSV	ZERO	RSV1	1	0	0	0	0	0
RSV	NORM	RSV1	1	0	0	0	0	0
RSV	RSV	RSV1	1	0	0	0	0	0

Native divide operations

Table 8 lists the exceptions encountered for each respective operand combination used in a native divide operation.

Table 8 Native operation results—divide

Numerator	Denominator	Result	RO	SQRN	DZ	SIV	OV	UN
ZERO	ZERO	RSV0	0	0	1	0	0	0
ZERO	NORM	0	0	0	0	0	0	0
ZERO	RSV	RSV1	1	0	0	0	0	0
NORM	ZERO	RSV0	0	0	1	0	0	0
NORM	NORM	(NM) NORM	0	0	0	0	0	0
NORM	NORM	(OV) RSV0	0	0	0	0	1	0
NORM	NORM	(UF) 0	0	0	0	0	0	1
NORM	RSV	RSV1	1	0	0	0	0	0
RSV	ZERO	RSV1	1	0	1	0	0	0
RSV	NORM	RSV1	1	0	0	0	0	0
RSV	RSV	RSV1	1	0	0	0	0	0

Native square root operations

lists the exceptions encountered for each operand type used in a native square root operation.

Note

The square root operation is not part of the C100 Series. The square root operation is performed in hardware on all multiprocessing CPUs.

Table 9 Native operation results—square root

Operand	Result	RO	SQRN	FDZ	SIV	OV	UN
ZERO	0	0	0	0	0	0	0
+NORM	NORM	0	0	0	0	0	0
-NORM	NORM[1]	0	1	0	0	0	0
RSV	RSV1	1	0	0	0	0	0

[1]Result returned is the square root of the absolute value of NORM when NORM is negative.

Native min/max operations

Table 10 lists the exceptions encountered for each operand type used in native min/max operations.

Table 10 Native operation results—min/max operations

Operand A	Operand B	Result	RO	SQRN	FDZ	SIV	OV	UN
ZERO	ZERO	0	0	0	0	0	0	0
ZERO	NORM	NORM or 0	0	0	0	0	0	0
ZERO	RSV	RSV1	1	0	0	0	0	0
NORM	ZERO	NORM or 0	0	0	0	0	0	0
NORM	NORM	NORM	0	0	0	0	0	0
NORM	RSV	RSV1	1	0	0	0	0	0
RSV	ZERO	RSV1	1	0	0	0	0	0
RSV	NORM	RSV1	1	0	0	0	0	0
RSV	RSV	RSV1	1	0	0	0	0	0

Native conversion operations

Table 11, Table 12, and Table 13 list the exceptions encountered for each operand type used in each type of native conversion operation.

Table 11 lists the exceptions encountered for each operand type used in a native float-to-fixed conversion.

Table 11 Native operation results—float-to-fixed conversions

Operand	Result	RO	SQRN	FDZ	SIV	OV	UN
ZERO	0	0	0	0	0	0	0
NORM	(IN) INT	0	0	0	0	0	0
NORM	(IO) TRN	0	0	0	1	0	0
RSV	None	1	0	0	1	0	0

If integer overflow occurs, the result returned is the least-significant bits of the exact result. The number of least-significant bits returned is dependent on the size of the result. For example, when converting to a word integer, the 32 least-significant bits of the exact result are returned. When the

input operand is RSV, the result of the operation is implementation-dependent.

Note

Some early C1 and C200 implementations do not write to the result register of a scalar conversion if an exception is encountered during the execution of the conversion.

Table 12 lists the exceptions encountered for each operand type used in a native fixed-to-float conversion.

Table 12 Native operation results—fixed-to-float conversions

Operand	Result	RO	SQRN	FDZ	SIV	OV	UN
INT	NORM	0	0	0	0	0	0
INT0	0	0	0	0	0	0	0

Table 13 lists the exceptions encountered for each operand type used in a native float-to-float conversion.

Table 13 Native operation results—float-to-float conversions

Operand	Result	RO	SQRN	FDZ	SIV	OV	UN
ZERO	0	0	0	0	0	0	0
NORM	(NM) NORM	0	0	0	0	0	0
NORM	(OV) RSV0	0	0	0	0	1	0
NORM	(UN) 0	0	0	0	0	0	1
RSV	RSV1	1	0	0	0	0	0

Overflow and underflow are only possible when converting double-precision to single-precision values. When the operand is RSV, the result returned is RSV-translated to the format of the output operand.

IEEE floating-point implementation

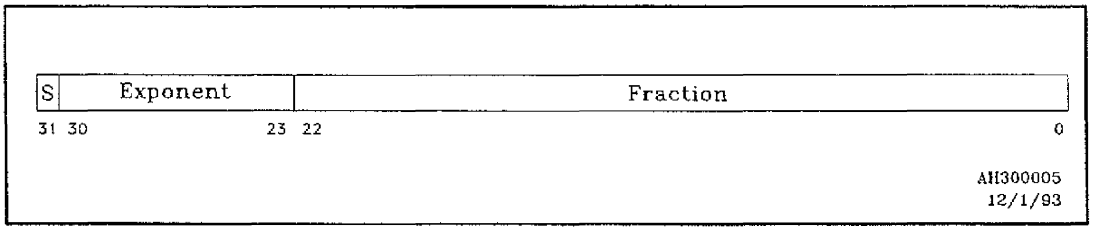
The C-Series implementation of the IEEE floating-point standard defines the following operands as valid input:

- **Normalized**—The exponent is not all zeros or all ones.
- **Denormalized**—The exponent is all zeros, the fraction is nonzero, and the sign is 1 or 0. C-Series architecture always treats this number as *true zero*.
- **NaN**—The exponent is all ones, the fraction is nonzero, and the sign is 1 or 0.
- **Infinity**—The exponent is all ones, the fraction is zero, and the sign is 1 or 0.
- **True zero**—The exponent is all zeros, the fraction is all zeros, and the sign is 1 or 0.

IEEE single-precision floating-point format

The format of the single-precision (32-bit) floating-point number is shown in Figure 8.

Figure 8 IEEE single-precision floating-point format



S	The sign bit. A binary 0 denotes positive, a binary 1 denotes negative. Numbers in this form are termed sign magnitude.
Exponent	A binary biased exponent. The decimal value of the exponent is determined by subtracting 127 from the unsigned binary value of bits <30..23> and using the result as a power of 2.
Fraction	A fractional value. An implicit 1 bit is to the left of bit <22>. The binary point is to right of the implicit 1 bit.

The input operands found in a IEEE single-precision (32-bit) floating-point number are shown in Table 14.

Table 14 IEEE single-precision input operands

Sign (S)	Exponent (e)	Fraction (f)	Value	Name
1/0	255	Not 0	None	NaN (not a number)
1/0	255	0	$(-1)^S \times \infty$	Infinity
1/0	1... 254	NA	$(-1)^S (2^{e-127}) (2^0 + \text{fraction})$	Normalized number
1/0	0	Not 0	$(-1)^S (2^{e-126}) (0 + \text{fraction})$	Denormalized number[1]
1/0	0	0	0	Floating-point zero

[1]The C-Series architecture always treats this number as *true zero*.

The dynamic range of an IEEE single-precision (32-bit) floating-point number is shown in Table 15.

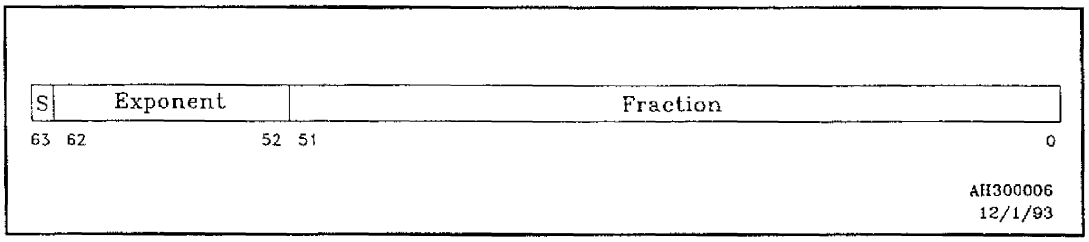
Table 15 IEEE single-precision dynamic range

Value	Hexadecimal	Approximate value
Largest positive	7F7F FFFF	$+3.4028235 \times 10^{+38}$
Smallest positive	0080 0000	$+1.1754944 \times 10^{-38}$
Zero	0000 0000	0
Smallest negative	8080 0000	$-1.1754944 \times 10^{-38}$
Largest negative	FF7F FFFF	$-3.4028235 \times 10^{+38}$

IEEE double-precision floating-point format

The format of the double-precision (64-bit) floating-point number is shown in Figure 9.

Figure 9 IEEE double-precision floating-point format



S	The sign bit. A binary 0 denotes positive, a binary 1 denotes negative. Numbers in this form are termed *sign magnitude*.
Exponent	An 11-bit, binary-biased exponent. The decimal value of the exponent is determined by subtracting 1,023 from the unsigned binary value of bits <62..52> and using the result as a power of 2.
Fraction	A fractional value. An implicit 1 bit is to the left of bit <51>. The binary point is to the right of the implicit 1 bit.

The input operands found in an IEEE double-precision (64-bit) floating-point number are shown in Table 16.

Table 16 IEEE double-precision input operands

Sign (S)	Exponent (e)	Fraction (f)	Value	Name
1/0	2047	Not 0	None	NaN (not a number)
1/0	2047	0	$(-1)^S \times \infty$	Infinity
1/0	1... 2046	NA	$(-1)^S (2^{e-1023}) (2^0 + \text{fraction})$	Normalized number
1/0	0	Not 0	$(-1)^S (2^{e-1022}) (0 + \text{fraction})$	Denormalized number[1]
1/0	0	0	0	Floating-point zero

[1] The C-Series architecture always treats this number as *true zero*.

The dynamic range of a IEEE double-precision (64-bit) floating-point number is shown in Table 17.

Table 17 IEEE double-precision dynamic range

Value	Hexadecimal	Approximate value
Largest positive	7FEF FFFF FFFF FFFF	$+1.797693134862316 \times 10^{+308}$
Smallest positive	0010 0000 0000 0000	$+2.225073858507201 \times 10^{-308}$
Zero	0000 0000 0000 0000	0
Smallest negative	8010 0000 0000 0000	$-2.225073858507201 \times 10^{-308}$
Largest negative	FFEF FFFF FFFF FFFF	$-1.797693134862316 \times 10^{+308}$

IEEE special operands

There are certain special operands within the IEEE floating-point format. In particular, these operands indicate values that cannot be accurately represented within the format or initiate exception processing if that value is used as an operand to a subsequent arithmetic computation.

A number that has an exponent of all ones and a fraction of all zeros is called *infinity*. This value is generally produced when the result of a computation is too large to be represented within the format (larger than *largest*). The sign of this number is generally maintained as the correct sign for the operation of the result. If a large positive number is multiplied by a large negative number and the result is out of the range of resolution, infinity is returned and the sign bit is set (negative, since the true answer is negative).

A number that has an exponent of all ones and a fraction that is not all zeros is called not a number (NaN). This value is generally produced when no computation was possible, such as an attempt to divide-by-zero, or if one of the operands of the operation was NaN.

While infinity is produced by certain operations, it is treated as NaN when it is used as an input to an operation. Thus, if NaN or infinity is used as an operand, the reserved operand exception is generated.

IEEE floating-point zero

An IEEE *zero* is a floating-point number with an exponent of 0 and a sign of either 1 or 0. If the fraction is all zeros, this value is said to be *true zero*. Otherwise, it is a *denormalized* number. In the C-Series implementation of IEEE floating-point format, it is always treated as true zero. When true zero or a denormalized

number is used as an operand of a computation, any nonzero fraction bits are forced to zero and the hidden bit is not inserted.

True zero is always returned when the result of an operation is zero. In addition, when exponent underflow occurs, true zero is returned, and the UN bit in the PSW is set. The sign of any true zero returned is implementation-specific. However, for all current implementations, the following rules apply:

1. For IEEE add and subtract operations, the sign bit of any zero result is always a zero.
2. For multiply and divide operations, the sign bit of any zero result is the exclusive OR of the sign bits of the two operands.

IEEE rounding

Rounding in IEEE mode is identical to rounding in Native mode. Refer to the "Native rounding" section on page 21.

IEEE operations

The following subsections detail the results returned and the exceptions generated (if any) for IEEE-mode floating-point operations. This subsection contains details that are specific to the current implementation but are not part of the architecture.

When NaN is returned as a result, the C-Series architecture only specifies that any legal form of NaN may be returned. The specific form of NaN currently returned is implementation-specific.

When zero is returned as a result, the architecture only requires that it be a true zero (that is, the sign of the true zero may be either 1 or 0). Thus, any reference to the sign of a zero result described in this subsection is implementation-specific and subject to change in other implementations of this architecture. Table 18 lists the abbreviations used during each IEEE arithmetic operation.

In the IEEE-mode definitions, specifically with respect to operands and results, the descriptions imply the positive or negative form of the value. When the symbols used for these definitions are preceded by + or -, the specific value is positive or negative. For example, NORM represents a positive or negative normalized number, while -NORM represents a negative normalized number only.

IEEE compare operations

No result other than comparison status is returned for IEEE compare operations. Any exceptions generated as a result of an input operand combination are identical to the exceptions generated for the IEEE add or subtract operations except that UN and OV exceptions are not possible.

IEEE add or subtract operations

Table 19 lists the exceptions encountered for each respective operand combination used in an IEEE add or subtract operation.

IEEE multiply operations

Table 20 lists the exceptions encountered for each respective operand combination used in an IEEE multiply operation.

IEEE divide operations

Table 21 lists the exceptions encountered for each respective operand combination used in an IEEE divide operation.

IEEE min/max operations

Table 22 lists the exceptions encountered for each respective operand combination used in an IEEE min/max operation.

Table 18 IEEE floating-point nomenclature

Nomenclature type	Symbol	Description
Input operands	INT	A nonzero two's-complement integer.
	INT0	Integer zero.
	DEN	A denormalized number.
	INF	Infinity.
	NaN	Not a number.
	NORM	A normalized number.
	ZERO	True zero, sign bit is 1 or 0.
Operation results	INFs	Infinity where the sign bit is the sign of the numerical result.
	NaN1	A particular form of NaN where the sign bit is a 1, the exponent bits are all ones, and the fraction bits are all zeroes except for the least-significant bit, which is a 1.
	0	True zero with a sign bit of 0, or an integer zero.
	0s	True zero where the sign bit is the exclusive or of the sign bits of the input operands.
	NORM	A normalized number.
	INT	A nonzero representable integer.
	TRN	The least-significant bits of an integer whose exact value contains more bits than can be stored in the result.
Result conditions	(NM)	If result is a normalized number.
	(OV)	If overflow results.
	(UN)	If underflow results.
	(IN)	If result is a representable integer.
	(IO)	If integer overflow results.
Exceptions	RO	Illegal input operand.
	SQRN	Square root of a negative number.
	FDZ	Floating-point divide-by-zero.
	SIV	Integer overflow.
	OV	Floating-point overflow.
	UN	Floating-point underflow.
Exception states	0	Exception did not occur.
	1	Exception did occur.

Table 19 IEEE operation results—add or subtract

Operand A	Operand B	Result	RO	SQRN	FDZ	SIV	OV	UN
ZERO	ZERO	+0	0	0	0	0	0	0
ZERO	DEN	+0	0	0	0	0	0	0
ZERO	NORM	NORM	0	0	0	0	0	0
ZERO	INF	NaN1	1	0	0	0	0	0
ZERO	NaN	NaN1	1	0	0	0	0	0
DEN	DEN	+0	0	0	0	0	0	0
DEN	INF	NaN1	1	0	0	0	0	0
DEN	NaN	NaN1	1	0	0	0	0	0
DEN	ZERO	+0	0	0	0	0	0	0
DEN	NORM	NORM	0	0	0	0	0	0
NORM	ZERO	NORM	0	0	0	0	0	0
NORM	DEN	NORM	0	0	0	0	0	0
NORM	INF	NaN1	1	0	0	0	0	0
NORM	NaN	NaN1	1	0	0	0	0	0
NORM	NORM	(OV) INFs	0	0	0	0	1	0
NORM	NORM	(NM) NORM	0	0	0	0	0	0
NORM	NORM	(UN) +0	0	0	0	0	0	1
INF	ZERO	NaN1	1	0	0	0	0	0
INF	DEN	NaN1	1	0	0	0	0	0
INF	NORM	NaN1	1	0	0	0	0	0
INF	INF	NaN1	1	0	0	0	0	0
INF	NaN	NaN1	1	0	0	0	0	0
NaN	NaN	NaN1	1	0	0	0	0	0
NaN	ZERO	NaN1	1	0	0	0	0	0
NaN	DEN	NaN1	1	0	0	0	0	0
NaN	NORM	NaN1	1	0	0	0	0	0
NaN	INF	NaN1	1	0	0	0	0	0

Table 20 IEEE operation results—multiply

Operand A	Operand B	Result	RO	SQRN	FDZ	SIV	OV	UN
ZERO	ZERO	0s	0	0	0	0	0	0
ZERO	DEN	0s	0	0	0	0	0	0
ZERO	NORM	0s	0	0	0	0	0	0
ZERO	INF	NaN1	1	0	0	0	0	0
ZERO	NaN	NaN1	1	0	0	0	0	0
DEN	ZERO	0s	0	0	0	0	0	0
DEN	DEN	0s	0	0	0	0	0	0
DEN	NORM	0s	0	0	0	0	0	0
DEN	INF	NaN1	1	0	0	0	0	0
DEN	NaN	NaN1	1	0	0	0	0	0
NORM	ZERO	0s	0	0	0	0	0	0
NORM	DEN	0s	0	0	0	0	0	0
NORM	NORM	(OV) INFs	0	0	0	0	1	0
NORM	NORM	(NM) NORM	0	0	0	0	0	0
NORM	NORM	(UN) 0s	0	0	0	0	0	1
NORM	INF	NaN1	1	0	0	0	0	0
NORM	NaN	NaN1	1	0	0	0	0	0
INF	ZERO	NaN1	1	0	0	0	0	0
INF	DEN	NaN1	1	0	0	0	0	0
INF	NORM	NaN1	1	0	0	0	0	0
INF	INF	NaN1	1	0	0	0	0	0
INF	NaN	NaN1	1	0	0	0	0	0
NaN	ZERO	NaN1	1	0	0	0	0	0
NaN	DEN	NaN1	1	0	0	0	0	0
NaN	NORM	NaN1	1	0	0	0	0	0
NaN	INF	NaN1	1	0	0	0	0	0
NaN	NaN	NaN1	1	0	0	0	0	0

Table 21 IEEE operation results—divide

Operand A	Denominator	Result	RO	SQRN	FDZ	SIV	OV	UN
ZERO	ZERO	NaN1	0	0	1	0	0	0
ZERO	DEN	NaN1	0	0	1	0	0	0
ZERO	NORM	0s	0	0	0	0	0	0
ZERO	INF	NaN1	1	0	0	0	0	0
ZERO	NaN	NaN1	1	0	0	0	0	0
DEN	ZERO	NaN1	0	0	1	0	0	0
DEN	DEN	NaN1	0	0	1	0	0	0
DEN	NORM	0s	0	0	0	0	0	0
DEN	INF	NaN1	1	0	0	0	0	0
DEN	NaN	NaN1	1	0	0	0	0	0
NORM	ZERO	INFs	0	0	1	0	0	0
NORM	DEN	INFs	0	0	1	0	0	0
NORM	NORM	(OV) INFs	0	0	0	0	1	0
NORM	NORM	(NM) NORM	0	0	0	0	0	0
NORM	NORM	(UN) 0s	0	0	0	0	0	1
NORM	INF	NaN1	1	0	0	0	0	0
NORM	NaN	NaN1	1	0	0	0	0	0
INF	ZERO	NaN1	1	0	1	0	0	0
INF	DEN	NaN1	1	0	1	0	0	0
INF	NORM	NaN1	1	0	0	0	0	0
INF	INF	NaN1	1	0	0	0	0	0
INF	NaN	NaN1	1	0	0	0	0	0
NaN	ZERO	NaN1	1	0	1	0	0	0
NaN	DEN	NaN1	1	0	1	0	0	0
NaN	NORM	NaN1	1	0	0	0	0	0
NaN	INF	NaN1	1	0	0	0	0	0
NaN	NaN	NaN1	1	0	0	0	0	0

Table 22 IEEE operation results—min/max

Operand A	Operand B	Result	RO	SQRN	FDZ	SIV	OV	UN
ZERO	ZERO	+0	0	0	0	0	0	0
ZERO	DEN	+0	0	0	0	0	0	0
ZERO	NORM	NORM or +0	0	0	0	0	0	0
ZERO	INF	NaN1	1	0	0	0	0	0
ZERO	NaN	NaN1	1	0	0	0	0	0
DEN	ZERO	+0	0	0	0	0	0	0
DEN	DEN	+0	0	0	0	0	0	0
DEN	NORM	NORM or +0	0	0	0	0	0	0
DEN	INF	NaN1	1	0	0	0	0	0
DEN	NaN	NaN1	1	0	0	0	0	0
NORM	ZERO	NORM or +0	0	0	0	0	0	0
NORM	DEN	NORM or +0	0	0	0	0	0	0
NORM	NORM	NORM	0	0	0	0	0	0
NORM	INF	NaN1	1	0	0	0	0	0
NORM	NaN	NaN1	1	0	0	0	0	0
INF	ZERO	NaN1	1	0	0	0	0	0
INF	DEN	NaN1	1	0	0	0	0	0
INF	NORM	NaN1	1	0	0	0	0	0
INF	INF	NaN1	1	0	0	0	0	0
INF	NaN	NaN1	1	0	0	0	0	0
NaN	ZERO	NaN1	1	0	0	0	0	0
NaN	DEN	NaN1	1	0	0	0	0	0
NaN	NORM	NaN1	1	0	0	0	0	0
NaN	INF	NaN1	1	0	0	0	0	0
NaN	NaN	NaN1	1	0	0	0	0	0

IEEE square root operations

Table 23 lists the exception encountered for each operand type used in an IEEE square root operation.

Table 23 IEEE operation results—square root

Operand	Result	RO	SQRN	FDZ	SIV	OV	UN
+ZERO	0	0	0	0	0	0	0
-ZERO	0	0	0	0	0	0	0
+DEN	0	0	0	0	0	0	0
-DEN	0	0	0	0	0	0	0
+NORM	NORM	0	0	0	0	0	0
-NORM	NORM[1]	0	1	0	0	0	0
INF	NaN1	1	0	0	0	0	0
NaN	NaN1	1	0	0	0	0	0

[1]The result returned is the square root of the absolute value of NORM when NORM is negative.

Note

The square root operation is not part of the C100 Series. The square root operation is performed in hardware on all multiprocessing C-Series CPUs.

IEEE conversion operations

Table 24, Table 25, and Table 26 list the exceptions encountered for each operand type used in IEEE conversion operations.

Table 24 lists the exceptions encountered for each operand type used in an IEEE float-to-fixed conversion.

Table 24 IEEE operation results—float-to-fixed conversions

Operand	Result	RO	SQRN	FDZ	SIV	OV	UN
ZERO	0	0	0	0	0	0	0
DEN	0	0	0	0	0	0	0
NORM	(IN) INT	0	0	0	0	0	0
NORM	(IO) TRN	0	0	0	1	0	0
INF	None	1	0	0	1	0	0
NaN	None	1	0	0	1	0	0

If integer overflow occurs, the result returned is the least-significant bits of the exact result. The number of least-significant bits returned is dependent upon the size of the result. For example, when converting to a word integer, the 32 least-significant bits of the exact result are returned. When the input operand is NaN or infinity, the result of the operation is implementation dependent.

Note

Some early C1 and C200 (C3200) implementations do not write to the result register of a scalar conversion if an exception is encountered during the conversion.

Table 25 lists the exceptions encountered for each operand type used in an IEEE fixed-to-float conversion.

Table 25 IEEE operation results—fixed-to-float conversions

Operand	Result	RO	SQRN	FDZ	SIV	OV	UN
INT	NORM	0	0	0	0	0	0
INT0	0	0	0	0	0	0	0

Table 26 lists the exceptions encountered for each operand type used in an IEEE float-to-float conversion.

Table 26 IEEE operation results—float-to-float conversions

Operand	Result	RO	SQRN	FDZ	SIV	OV	UN
ZERO	0	0	0	0	0	0	0
DEN	0	0	0	0	0	0	0
NORM	(NM) NORM	0	0	0	0	0	0
NORM	(OV) INFs	0	0	0	0	1	0
NORM	(UN) 0s	0	0	0	0	0	1
INF	NaN1	1	0	0	0	0	0
NaN	NaN1	1	0	0	0	0	0

Overflow and underflow are possible only when converting double-precision values to single-precision values. When the operand is infinity or NaN, the result returned is NaN-translated to the format of the output operand.

Native and IEEE floating-point algorithms

This section details the floating-point algorithms used by the C-Series instruction set for both IEEE and native mode arithmetic. The C-Series architecture does not support the complete IEEE floating-point specification, only the IEEE floating-point format is used. These algorithms involve rounding, sequencing of operations, and other considerations. The following exceptions are defined in the algorithms:

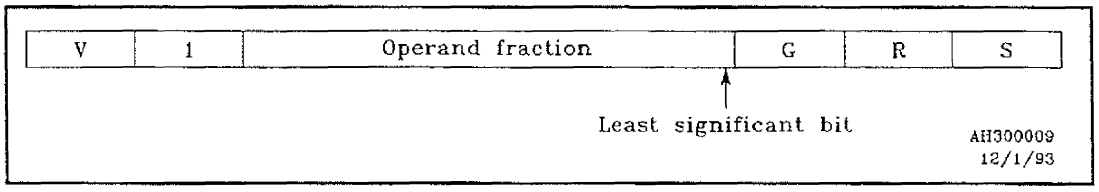
- **Input exception**—In native mode, an input-reserved operand. In IEEE mode, an NaN or infinity value.
- **Output exception**—An output-reserved operand in native mode or a NaN or infinity value in IEEE mode.

Add or subtract

To add or subtract:

1. The fractions of the floating-point operands are expanded internally as follows:
   - A 1 is appended to the higher bit position of the fraction.
   - Two guard bits are appended to the right of the least-significant fraction bit. These bits are referred to as *G* and *R*, respectively.
   - A sticky bit is appended to the right of the two guard bits. The sticky, or *S*, bit is the OR of all bits to the right of the R bit.
   - An additional bit is appended to the higher fraction, the *V* bit, for overflow. The internal floating-point format is illustrated in Figure 10, where the initial values of the V, G, R, and S bits are all 0.

Figure 10 Internal floating-point format



2. The exponents of the two fractions are compared. The fraction of the smaller exponent is shifted right by an amount equal to the absolute difference of the exponents. All right-shifted bits are shifted through the G, R, and S bits.

3. Any binary ones shifted past the two guard bits are remembered in S.
4. If any of the input operands are zero, all fraction bits are set to zero.
5. If any of the input operands is an input exception,
   - No shifting occurs and there is an output exception. The output of the add or sub is an output exception.
   - Otherwise, the two fractions are algebraically added or subtracted according to the sign and op code.
6. If the result is zero, the exponent is set to zero and not normalized. Otherwise, the result is normalized
   - If V becomes 1, the intermediate result is right-shifted by one bit position, an OR operation is performed on R and S, and the result is placed in S.
   - If a generated subtract was performed, the intermediate result is left-shifted until a normalized intermediate result is obtained. Zero or S may be shifted into R from the right. G is loaded with R; S is always unchanged.
7. The intermediate result is rounded as shown in Table 27.

Table 27 Intermediate result rounding - add, subtract, multiply

G	R	S	ROUNDING PERFORMED (TO LSB)
0	0	0	Add 0
0	0	1	Add 0
0	1	0	Add 0
0	1	1	Add 0
1	0	0	Add LSB of fraction (round to nearest even)
1	0	1	Add 1
1	1	0	Add 1
1	1	1	Add 1

8. The rounded intermediate result is normalized again, and the exponent is adjusted, if necessary, to yield the final result.

Multiply

Multiplying two normalized floating-point numbers produces an intermediate result that is either normalized or at most requires one left shift. To multiply,

1. If either of the two operands is an input exception, the result is an output exception.
2. If either of the two operands is zero, the result is a true zero.
3. Otherwise, the exponents are added, keeping an extra bit of precision to account for a normalization shift that could correct an exponent overflow.
4. The two fractions are multiplied right to left.
5. The G, R, and S bits are maintained during intermediate calculations.
6. The result is post-normalized, if required.
7. The intermediate result is rounded, as outlined in Table 27.
8. The rounded intermediate result is normalized again and the exponent adjusted, if necessary, to yield the final result.

Divide

Dividing two normalized floating-point numbers produces an intermediate result that is normalized. To divide,

1. If either of the two operands is an input exception, the result is an output exception.
2. If the divisor is zero, the result is an output exception. Also, PSW (FDZ), the floating divide-by-zero bit, is set to 1.
3. The exponents are subtracted, producing the result's exponent.
4. The numerator mantissa is divided by the denominator mantissa. An $(n+2)$-bit quotient is generated where n is the length of the mantissas of the operands. The two additional quotient bits represent the G and R bits. The state of the S bit is implementation specific. The S bit may always be assumed to be 0, or may represent the OR of some portion of, if not the entire remainder.
5. The intermediate result is rounded, as outlined in Table 28.
6. The rounded intermediate result is normalized again and the exponent adjusted, if necessary, to yield the final result.

Table 28 Intermediate result rounding - divide

		Rounding performed	
G	R	C Series compatible	Industry compatible
0	0	Add 0	Add 0
0	1	Add 0	Add 0
1	0	Add LSB	Add 1
1	1	Add 1	Add 1

Conversions

The following rules apply when converting an arithmetic value from one data type to another.

1. When converting from floating-point to fixed-point, always round toward zero (truncate).
2. When converting from floating-point to fixed-point, properly normalize the integer. If this results in more mantissa bits than are available, round the mantissa to its appropriate size.
3. Rounding from floating-point to fixed-point can be achieved by adding 0.5 to the floating-point operand, then executing the floating-point to fixed-point instruction. The sign used on the 0.5 value should be the same as the sign on the operand. Thus:
   - RND (3.4) equals TRUNCATE (3.9) = 3
   - RND (3.5) equals TRUNCATE (4.0) = 4

Register sets

3

The CONVEX C-Series architecture allows for asynchronous and overlapped fetch and execute functions by partitioning both addresses and operands into three general register sets.

- Address registers
- Scalar registers
- Vector registers

This partitioning of the general registers enables address, scalar, and vector calculations to be performed in parallel.

The architecture has four special purpose registers.

- Program counter (PC)
- Processor status word (PSW)
- Scalar stride zero (SS0 - C4600 Series CPUs only)
- Scalar stride one (SS1 - C4600 Series CPUs only)

For the multiprocessing C-Series CPUs (CPUs other than the C100 Series), the C-Series architecture has an additional register set, used for CPU communications in the multiprocessing environment. These registers are presented in the "Communication registers" section in Chapter 5.

The CONVEX C-Series architecture also uses three privileged flags:

- Interrupt on (ION)
- Realtime interrupt on (RT_ION - C3400 Series CPUs only)
- Vector valid (VV)

EDTX 2-04-CV-120
51056DOC065377

All address, scalar, and vector registers support multiple data lengths, which occupy the following bit positions:

- **Byte**—bits <7..0>
- **Halfword**—bits <15..0>
- **Word**—bits <31..0>
- **Longword**—bits <63..0>
- **Single-precision**—bits <31..0>
- **Double-precision**—bits <63..0>

When an operand with precision less than the destination register is loaded, *the remaining unused bits of the destination register are left unchanged*. For example, when a 16-bit integer is loaded into a 32-bit address register, the 16 high order bits of the register (bits <31..16>) are undisturbed. Each data type is accessed in a specific way:

- A byte is loaded into or read from bits <7..0> of a register.
- A halfword is loaded into or read from bits <15..0> of a register.
- A word (integer or single-precision) is loaded into or read from bits <31..0> of a register.
- A longword (integer or double-precision) is loaded into or read from bits <63..0> of a register.

Address registers

All CPUs, except the C4600 Series, have eight 32-bit address (A) registers, A0 through A7. Although the registers in the following list have specific, predefined functions, all except register A0 can also be used as general purpose address registers.

- A0 is the stack pointer (SP).
- A3, A4, and A5 are implicitly used by some trap handlers (for example, page fault, system exceptions, and so forth).
- A5 is implicitly used by some instructions.
- A6 is the argument pointer (AP).
- A7 is the frame pointer (FP).

Register A0 is used in two additional ways. When register A0 is specified in an addressing operation, zero is used in place of the true value contained in register A0. When register A0 is used as a source or destination for an arithmetic operation, the true value is used.

The following can be loaded into address registers:

- Signed or unsigned fixed-point integers
- Operands used as addresses or index values
- Operands that are manipulated in parallel with a computation performed in scalar or vector registers

Longword operands *cannot* be loaded into an address register, since address registers are only 32 bits in length.

C4600 Series
The C4600 Series CPUs contain thirty-two 32-bit address registers, A0 to A31. Address registers A0, A6 and A7 retain their use as Stack Pointer, Argument Pointer, and Frame Pointer respectively. New instructions for C4600 Series CPUs perform all byte, halfword and word operations on A0 to A31. The *C-Series Assembly Language Reference* describes these instructions.

Scalar registers

All CPUs, except the C4600 Series, contain eight 64-bit scalar (S) registers, S0 through S7. The S registers can contain logical, fixed-point integer, or floating-point operands.

A signed or unsigned scalar fixed-point integer value can be loaded into either an address or scalar register. Generally, operands used only for numeric processing are loaded into the scalar registers.

C4600 Series
C4600 series CPUs contain 28 64-bit scalar registers, S0 to S27. C4600 series-specific instructions perform all integer and floating-point operations on scalar registers S0 to S27.

Vector registers

There are five types of registers in the vector register set:

- Vector accumulators (V)
- Vector length (VL)
- Vector stride (VS)
- Vector merge (VM)
- Vector first (VF, C4600 only)

Vector accumulators

All CPUs, except C4600 Series, contain eight vector accumulators (V), V0 through V7. Each vector accumulator may contain up to 128 64-bit register *operands* or *elements*. These operands can be integer, logical, or floating-point values. When an operand less than 64 bits is loaded into a 64-bit element, *the unused bits are unchanged*.

Individual elements within a vector accumulator are referenced by appending the element number to the designated vector accumulator. The first element of V1 is referenced as V1<0> (origin 0 indexing). The 22nd element of V1 is referenced as V1<21>.

C4600 Series

The C4600 Series CPU's vector register set consists of 16 vector accumulators, V0 to V15. Each vector accumulator may contain up to 128 64-bit registers. C4600 series-specific instructions perform vector and vector/scalar operations using V0 to V15 and S0 to S27.

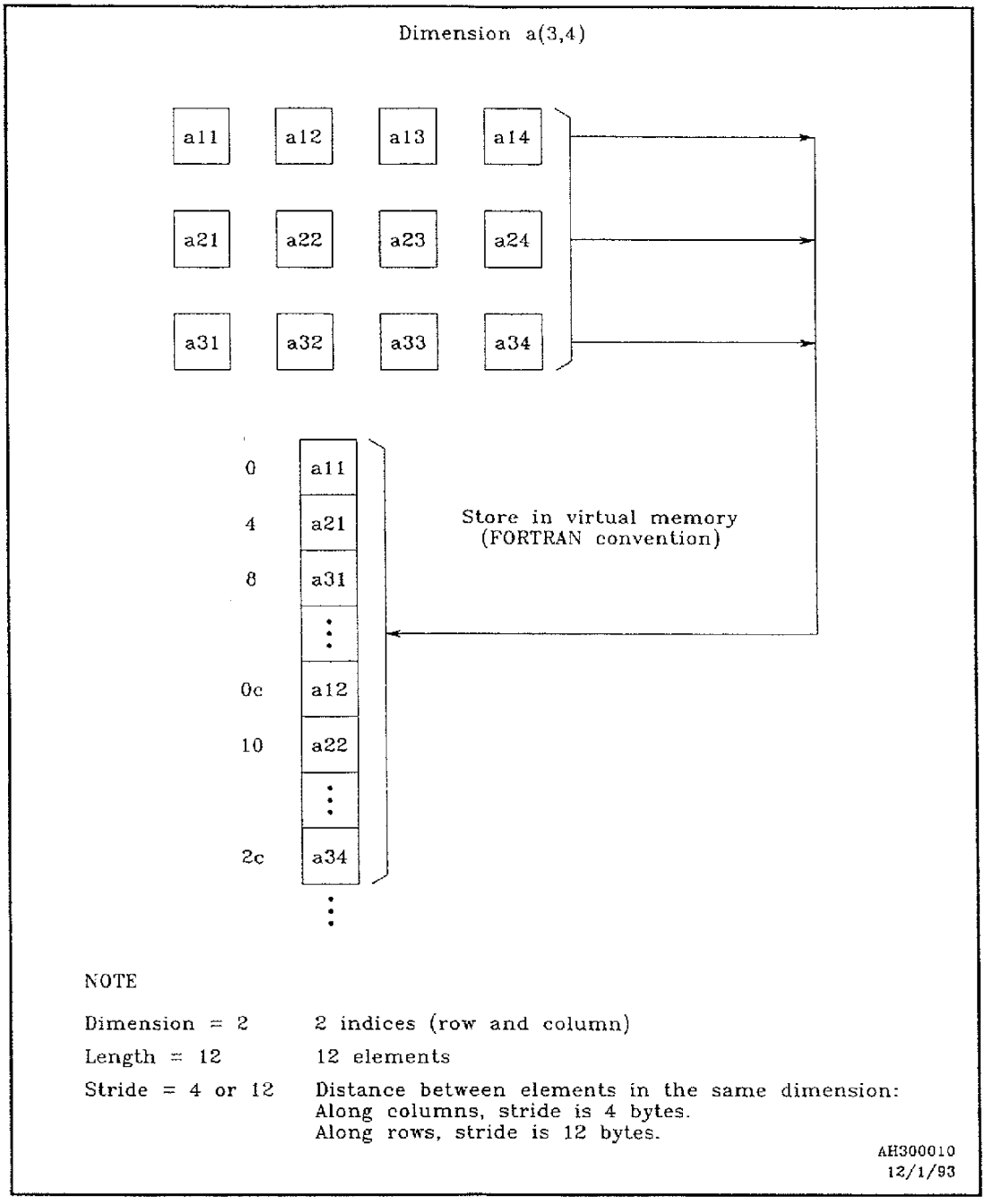
Array (vector) terminology

An array (vector) is a data structure composed of elements. Arrays have four general characteristics:

- **Data type**—This is the way in which bits are grouped and interpreted. The data type identifies the size of the operand and the significance of the bits in the operand.
- **Dimension**—This is the number of indices necessary to reference a particular element. For example, an array with three rows and four columns is a two-dimensional array.
- **Length**—This is the total number of elements in the array and is limited by the compiler and virtual address space. For example, an array with three rows and four columns has a length of twelve.
- **Stride**—This is the distance in bytes between adjacent array elements along the same dimension. For example, a one-dimensional word vector has a stride of four bytes.

Figure 11 illustrates an example of the vector terminology used in vector processing when manipulating a 3×4 array of words called a (3,4).

Figure 11 Vector terminology



NOTE

Dimension = 2	2 indices (row and column)
Length = 12	12 elements
Stride = 4 or 12	Distance between elements in the same dimension: Along columns, stride is 4 bytes. Along rows, stride is 12 bytes.

AH300010
12/1/93

Vector length register

A vector accumulator can contain a maximum of 128 elements of the same data representation and precision. The vector length register (VL) is used to specify the exact number of elements stored in a vector accumulator.

VL may contain any value from 0 to 128. An attempt to load VL with a negative value results in setting VL to 0. When VL is 0, no vector operation is performed.

An attempt to load VL with a value greater than 128 results in setting VL to 128. This allows arrays of up to 128 elements to be handled directly with vector instructions.

Even though the VL register has a maximum value of 128, a vector in memory can be any arbitrary length up to the user virtual address space limit of 2 Gbytes. Arrays longer than 128 elements or variable arrays that could exceed 128, are handled in software by coding a loop around a group of vector instructions that handles up to 128 elements at a time. This is called *strip mining* and is generated automatically by CONVEX vectoring compilers.

Vector stride register

The 32-bit vector stride register (VS) specifies the distance in bytes between adjacent array elements as they are accessed in memory. If VS contains a positive value, adjacent vector register elements are loaded and stored from memory by adding sequential multiples of VS to the initial address of the array base. If VS contains a negative value, adjacent vector register elements are loaded and stored from memory. This is done by subtracting sequential multiples of the absolute value of VS from the initial address of the array base. In the latter case, logically adjacent elements reside in decreasing locations in virtual memory.

Unpredictable results may occur on store operations if the absolute value of VS is nonzero but smaller than the width of the operands. If VS is 0, the referenced operand is correctly used repetitively as a source or destination.

Vector merge register

The vector merge register (VM) holds the status of element-by-element array comparisons and controls array manipulations such as `compress`, `expand`, `merge` and `operate-under-mask`. The VM register is 128 bits in length, with one bit position for each element in a V register. In a vector compare operation, a bit is set if the result of the corresponding compare is true. Otherwise, each respective bit is cleared. Typical uses of the VM register (as supported by the CONVEX instruction set) are

- Vector clipping
- Population count (the number of successful compares)
- Sparse vector manipulation
- Array compression, expansion, and merging
- The number and location of zero or threshold crossings
- Support operations that are performed under mask (under mask operations are not available on C100 Series CPU)

Vector first register - C4600

The vector register set of the C4600 Series CPUs contains an additional vector register called the vector first register (VF).

VF specifies the first element of vector register Vi, Vj or Vk accessed by a vector instruction, provided that the MSB of the corresponding 5-bit register select field of the instruction is set. VF *cannot* be applied to operations on VM.

VF is seven bits in length and may contain a value between 0 and 127. If the value of VF plus the value of VL is greater than 128, the effective value of VL for vector instructions that use VF is 128 minus VF. This effective VL value determines the number of results written to a vector register or VM, or the number of elements stored to memory.

If the value of VF plus Sj is greater than 127 in the `mov Vi,Sj,Sk` and `mov Si,Sj,Vk` instructions, then the selected element of the vector register is equal to (VF plus Sj) mod 128. Therefore, the vector register wraps *for these two instructions only.*

If Vi or Vj of an instruction specifies the same register as Vk of the instruction, and VF is applied to Vk, and VL is greater than VF, then elements of the shared register *may* be written (as Vk) before they are read (as Vi or Vj, depending of the hardware implementation). In this case, the result in Vk is architecturally undefined. The instruction `merg.x Vi,Vj,Vk` has the same behavior if Vi or Vj are the same as Vk.

Special purpose registers

The C-Series architecture uses up to four 32-bit special purpose registers.

- Program counter (PC)
- Processor status word (PSW)
- Scalar stride registers (SS0 and SS1 - C4600 only)

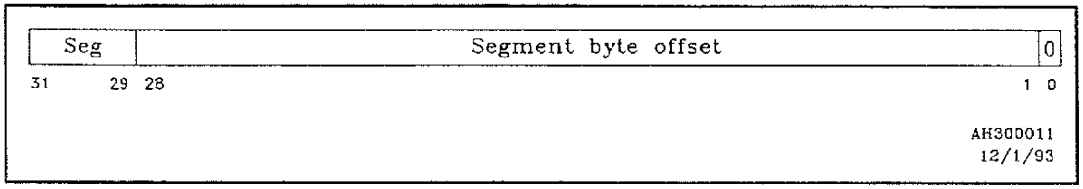
Program counter

The program counter (PC) contains the address pointing to the next executable instruction in a process. It is not part of the address register set. This separation permits address generation without regard to the true state of the PC.

C Series processors are highly pipelined. While they support PC-relative branching, there is no general support for PC-relative addressing.

The structure of the PC is shown in Figure 12.

Figure 12 Program counter format



When the PC increments to reference the next instruction, the specific bits incremented are a function of PC<31>.

If PC<31> is set, then PC<30..1> are incremented.

If PC<31> is clear, then PC<28..1> are incremented.

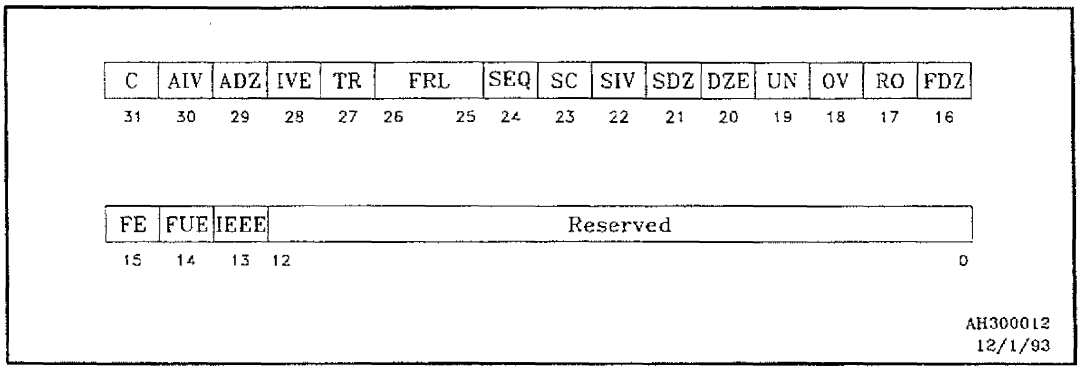
PC<0> is treated as zero.

Processor status word

The processor status word (PSW) is a user-accessible, 32-bit status register that indicates the processor state. This register contains flags that enable or disable exception processing and indicate the results of numerical operations. The PSW contains no privileged mode bits.

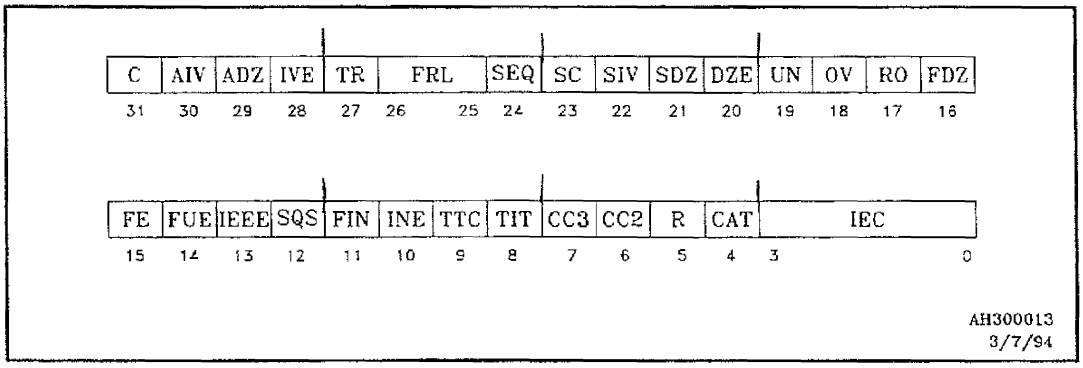
The structure of the PSW for C100 Series CPUs is shown in Figure 13.

Figure 13 Processor status word—C100 Series CPUs



The structure of the PSW for the C3200, C3400, C3800, C4600 Series CPUs is shown in Figure 14.

Figure 14 Processor status word—C3200, C3400, C3800, C4600 Series CPUs



Universal PSW bit definitions

Each bit of the PSW defined in the following subsections applies to all implementations of the C-Series architecture.

Bit <31>—Carry (C)

This bit, also known as address carry, is set to the carry-out value for specified operations involving the address (A) registers, including arithmetic operations, compare operations, and communication register instructions (using the A registers). For compare operations, if the comparison is false, the carry bit is cleared; if it is true, the carry bit is set.

In the 4600 Series CPUs, bit <31> is also called CC0. This synonym is used for convenience only, and the bit continues to function as C.

Bit <30>—Address overflow (AIV)

This bit indicates a fixed-point integer overflow occurred during specified operations on the address (A) registers. If AIV is clear, no overflow has occurred since this bit was last cleared. If AIV is set, at least one overflow has occurred since this bit was last cleared.

Bit <29>—Address divide-by-zero (ADZ)

This bit indicates an address divide-by-zero occurred during an operation using the address (A) registers. If ADZ is clear, no integer division with a zero divisor occurred since this bit was last cleared. If ADZ is set, at least one integer division with a zero divisor has occurred since this bit was last cleared.

Bit <28>—Integer overflow trap enable (IVE)

If this bit is set, and either SIV (bit <22>) or AIV (bit <30>) is set, an integer trap occurs. If IVE is clear, no trap occurs.

Bit <27>—Trace (TR)

If this bit is set, an instruction trace trap occurs after the processor executes one instruction. The process context is saved, which includes the contents of the program counter (PC). When execution returns from the trace trap handler, the process context is restored, and the instruction referenced by the PC is executed before a trace trap occurs again. For the trace mode to function properly, you must also set SEQ (bit <24>).

Bit <26..25>—Frame length (FRL)
These bits indicate the type of frame created by the last `call` instruction, trap, or fault:

11	Short frame
10	Long frame
01	Extended frame
00	Context return block

These bits in the PSW on the top of the stack are used by the `rtn` (return) instruction to unwind the stack after a subroutine call or exception. When PSW (FRL) indicates a context return block, the current ring must be ring 0, and the `rtnc` (return from a context block) instruction must be used. Frame lengths and return blocks are discussed in the "Resource structures" section in Chapter 4.

Bit <24> Sequential (SEQ)
This bit controls pipelining within the processor. If this bit is clear, the processor operates with maximum pipelining and overlap. If this bit is set, the processor executes all instructions sequentially; that is, the execution of the next instruction is initiated only after the previous instruction has been executed.

Bit <23>—Scalar carry (SC)
This bit is set to the carry-out value for operations involving the scalar (S) registers, including arithmetic operations, compare operations and communication register instructions (using the S registers). For compare operations' if the comparison is false, the scalar carry bit is cleared; if it is true, the scalar carry bit is set.

In the C4600 CPUs, bit <23> is also called CC1. This synonym is used for convenience only, and the bit continues to function as SC.

Bit <22>—Integer overflow (SIV)
This bit indicates a fixed-point integer overflow occurred during specified operations on a scalar (S) or vector (V) register. If SIV is clear, no overflow occurred since this bit was last cleared. If SIV is set, at least one overflow occurred since this bit was last cleared.

Bit <21>—Integer divide-by-zero (SDZ)
This bit indicates an integer divide-by-zero occurred during an operations using a scalar (S) or vector (V) register. If SDZ is clear, no integer division with a zero divisor occurred since this bit was last cleared. If SDZ is set, at least one integer division with a zero divisor occurred since this bit was last cleared.

Bit <20>—Divide-by-zero trap enable (DZE)
If this bit is set, and either SDZ (bit <21>) or ADZ (bit <29>) is set, a trap occurs. If DZE is clear, no trap occurs.

Bit <19>—Floating-point underflow (UN)
This bit indicates a floating-point underflow occurred during specified operations on a scalar (S) or vector (V) register. If UN is clear, no floating-point underflow occurred since this bit was last cleared. If UN is set, at least one floating-point underflow occurred since this bit was last cleared.

Bit <18>—Floating-point overflow (OV)
This bit indicates a floating-point overflow occurred during specified operations on a scalar (S) or vector (V) register. If OV is clear, no floating-point overflow occurred since this bit was last cleared. If OV is set, at least one floating-point overflow occurred since this bit was last cleared.

Bit <17>—Reserved operand (RO)
This bit indicates a floating-point operation on a reserved operand (Native mode), infinity, or NaN (IEEE mode) was detected during an operation on a scalar (S) or vector (V) register. If RO is clear, a reserved operand was not detected since this bit was last cleared. If RO is set, at least one floating-point operation on a reserved operand occurred since this bit was last set.

Bit <16>—Floating-point divide-by-zero (FDZ)
This bit indicates a floating-point divide-by-zero occurred during a divide operation on a scalar (S) or vector (V) register. If FDZ is clear, no floating-point division with a zero divisor occurred since this bit was last cleared. If FDZ is set, at least one floating-point division with a zero divisor occurred since this bit was last cleared.

Bit <15>—Floating-point trap enable (FE)
If this bit is set, and either OV, RO, or FDZ are set, a floating-point trap occurs. If FE is clear, no trap occurs.

Bit <14>—Floating-point underflow trap enable (FUE)
If this bit is set and UN is set, a floating-point underflow trap occurs. If FUE is clear, a floating-point underflow trap does not occur. In both cases, if a floating-point underflow is detected, *true zero* is the result.

Bit <13>—IEEE floating-point format (IEEE)
This bit enables and disables IEEE floating-point operations. If IEEE is set, IEEE floating-point operations are enabled. If IEEE is clear, native floating-point operations are enabled. This PSW bit allows an upgraded C100 Series CPU to process IEEE-format arithmetic.

Reserved (RES)

Bits <12..0> (for C100 Series CPUs)

Bits <7..5> (for C200/C3200 CPUs)

Bits <7..4> (for C3400/C3800 Series CPUs)

Bits <5..4> (for C4600 Series CPUs)

Extended PSW bit definitions

In addition to the previously defined universal PSW bits, the following sections define PSW bits exclusively for multiprocessing C-Series CPUs.

Bit <12>—Sequential store enable (SQS)
If this bit is clear, stores to memory may occur in non-sequential order. If this bit is set, all stores to memory occur in instruction execution order.

This bit is ignored by the C4600 Series architecture, because all stores are sequential by default.

Bit <11>—Intrinsic error (FIN)
This bit indicates an intrinsic instruction detected an error. If this bit is set, the IEC bits (PSW<3..0>) contain a code that specifies the type of error.

Bit <10>—Intrinsic error trap enable (INE)
If this bit is set, and FIN (bit<11>) is set, a floating-point exception trap occurs. If this bit is clear, no trap occurs.

Bit <9>—Trace thread concurrency trap (TTC)
This bit causes a trace trap any time a thread is created or terminated. If this bit is set, an instruction trace trap occurs prior to a CPU entering the hardware idle state and after leaving the hardware idle state. The `wfork`, `idle`, and `join` instructions can cause the CPU to enter the idle state. Acceptance of a posted fork causes the hardware to leave the idle state. Refer to the "Instruction trace trap" section in Chapter 6 for more information.

Bit <8>—Thread initialization trap (TIT)
This bit causes a trace trap any time a CPU picks up a fork. If this bit is set when a CPU picks up a fork, a trace trap will be taken to allow a handler to initialize the user-indicated code. A code of `0x800` (class 8, no qualifier) is placed in register A5 to distinguish this trap from the other trace traps. This trap is based on the PSW in the fork block in the communication registers. This is a user trap, that is, it occurs in the ring where it was executed. The CPU does not have to be in sequential mode for TIT traps to function correctly.

Bit <7>—Condition code 3 (CC3)
This bit can be used as the target of C4600-specific compare instructions, and as the branch condition of C4600-specific branch instructions.

Bit <6> - Condition Code 2 (CC2)
This bit can be used as the target of C4600-specific compare instructions, and as the branch condition of C4600-specific branch instructions.

Bit <4>—Communication address trap (CAT)
On C200/C3200 Series CPUs only, this bit is set whenever the CPU detects an invalid communication register address, which causes a system exception (ring violation) to occur. This bit is not used on C3400/C3800/C4600 Series CPUs.

This bit remains set until the trap is recognized, in order to allow a trap to be *remembered* in the event of ring crossing (`sysc`, interrupt, and so forth). This ensures that the trap is attributed to the correct ring of execution. The hardware clears this bit in the extended frame passed to the system exception handler when the trap is processed.

Bit <3..0>—Intrinsic error code (IEC)
When FIN (bit<11>) is set, IEC (bits <3..0>) contains a code that specifies the type of error encountered by the intrinsic instruction. Each intrinsic instruction that encounters an error first clears these bits, if they were set from a previous error that occurred with INE (bit <10>) clear. The new code is entered in these bits, and FIN is set. If INE is set, an arithmetic trap occurs. If INE is clear, no trap occurs. If INE is clear, only the last intrinsic trap is meaningful. Other intrinsic traps may have occurred, but were disregarded.

Intrinsic traps are processed by the same trap handler as the other PSW arithmetic traps, RO, FDZ, and UN. For arithmetic traps that can be enabled, the enable bit must be examined to determine the type of the current trap. Specifically, if some types of traps are enabled (that is, FUE (bit<14>) or FE (bit <15>) is set) and intrinsic traps are not (that is, INE (bit<10>) is clear), the enable bit must be examined.

The valid meaning of the IEC bits when there is no trap is:

`0000` A square root operation with a negative operand (vector or scalar) was attempted.

`0001` An overflow occurred when an exponential operation (`exp.s`, `exp.d`) was attempted.

`0010` An argument to a logarithmic operation (`ln.s`, `ln.d`) was less than or equal to zero.

`0110` The absolute value of an argument to a sine operation (`sin.s`, `sin.d`) was too large.

`0111` The absolute value of an argument to a cosine operation (`cos.s`, `cos.d`) was too large.

Scalar stride registers - C4600

The C4600 Series CPUs contain two 32-bit Scalar Stride Registers, SS0 and SS1. The `ld0` and `ld1` instructions use these registers to permit explicit cache prefetching under software control. An `ld0` loads the data addressed by `<effa>` into the specified destination register, and accelerates the data addressed by `<effa>+SS0` to the data cache. An ld1 performs the same function using SS1. This mechanism greatly improves the data cache hit rate for non-vectorizable routines operating on large data sets.

Privileged flags

The interrupts on (ION), realtime interrupts on (RT_ON), and vector valid (VV) flags are privileged binary flags that control certain operations.

Interrupts on

The interrupts on (ION) flag enables and disables external interrupts. The instructions `bri.f`, `bri.t`, `jmp.f`, and `jmp.t` test the state of the ION flag. The privileged instructions `eni` and `dsi` enable and disable the interrupts by setting and clearing ION (respectively). See the "Interrupt system" section in Chapter 6 for more detail.

Realtime interrupts on

The realtime interrupts on (RT_ION) flag is a privileged binary flag that enables and disables realtime external interrupts on the C3400 Series CPUs in realtime mode. It is used in the same way that ION is used. Details of C3400 Series realtime support are discussed in the "Interrupt system" section in Chapter 6.

Vector valid

The vector valid (VV) flag is used by the operating system for saving and restoring the vector accumulators in a demand mode.

The `mov Sk,VV` instruction loads the VV flag from Sk. This instruction is privileged in the C100 Series CPUs. On multiprocessing CPUs, it may be executed in any ring, but performs no operation when performed from rings 1 through 4.

The `tstvv` instruction loads the value of VV into PSW (SC).

Memory management 4

The CONVEX C-Series architecture allows 4 Gbytes of virtual (logical) address space. This virtual memory is partitioned into eight 512-Mbyte segments. Four segments are allocated to the user and four to the operating system (OS). This division means that a user program (instructions and data) can occupy up to 2 Gbytes of virtual storage.

The memory management system controls an extremely flexible and reliable virtual memory programming environment. Although the address space of the C-Series architecture is byte-addressable, memory is managed on a fixed-size page basis. Even though an address may be a valid virtual address, the referenced data may or may not be in physical memory.

The CONVEX C-Series architecture implements a process the same way that the UNIX operating system defines a process in that a CONVEX process has a protected address space, context, and a state. A CONVEX process exists as a single thread in the C100 Series architecture. However, in the multiprocessing architectures, a CONVEX process exits as one or more threads, unlike a UNIX operating system process. Therefore, it cannot be considered an atomic structure. A process is controlled by maintaining a process stack. These stack entries are called return blocks and contain vital information for controlling the execution of a process.

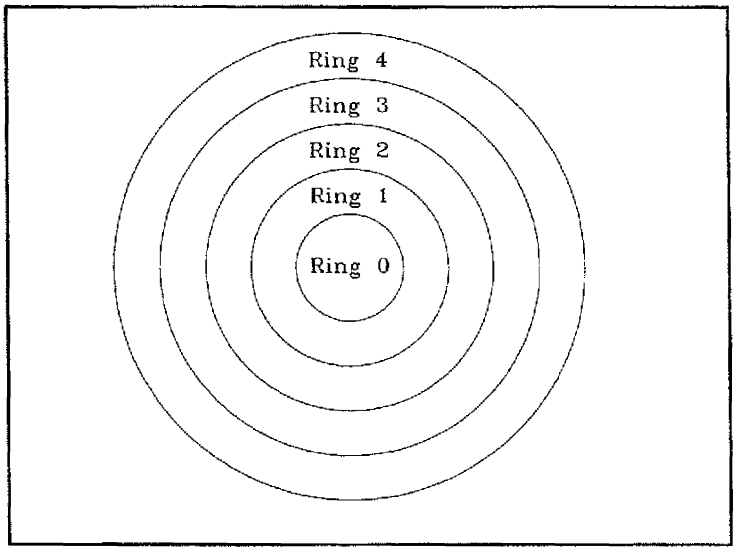
Physical address space

C-Series CPUs have implementation-specific physical address and I/O space. Chapter 7, "Implementation-specific features," describes the details of each implementation's physical and I/O address space.

Virtual address space

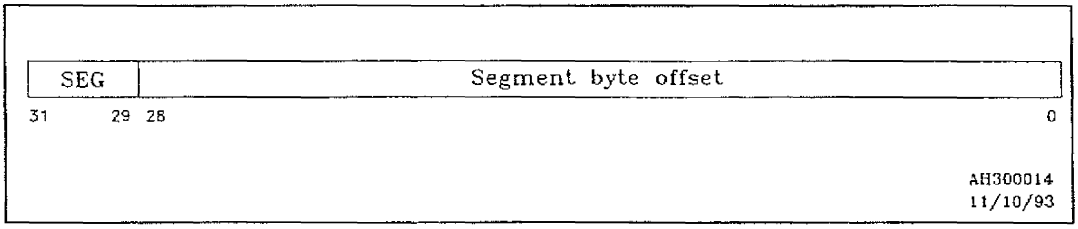
The virtual address space for a C-Series CPU is 4 Gbytes. This address space is logically divided into eight 512-Mbyte segments (see Table 29). These eight segments are distributed through five partitions called *rings of execution* or just *rings*. These five rings are illustrated in Figure 15.

Figure 15 Ring structure of the virtual address space



Virtual addresses are generated by using the program counter (PC) or any of six different operand addressing modes. The format of a virtual address is shown in Figure 16.

Figure 16 Virtual address format



The SEG field (bits <31..29>) of the virtual address defines the segment (0 through 7), and is assigned as follows:

- Segment 0 is always assigned to ring 0, which contains the operating system kernel. A set of instructions, called *privileged instructions*, can only be executed in Ring 0.
- Segment 1 is always assigned to ring 1.
- Segment 2 is always assigned to ring 2.
- Segment 3 is always assigned to ring 3.
- Segments 4, 5, 6, and 7 are always assigned to ring 4.

By allocating virtual memory segments to a ring structure, the architecture provides the memory protection system with a simple means of preventing, detecting, and handling memory protection violations. The operating system kernel and data structures are located in the innermost ring (ring 0), other kernel data structures are located in rings 1, 2, and 3, and all user processes are located in the outermost ring (ring 4). The privilege-level of a ring is inversely related to the ring number. Ring 0 has the highest privilege level. Therefore, the operating system (in ring 0) has all the privileges necessary to perform its functions.

Each CONVEX process has a protected address space with a corresponding privilege-level, achieved by segmenting virtual memory with a ring structure. By segmenting memory, the architecture supports individual address partitions for user code, static data, dynamic data (stacks), and memory protection.

Four segments are allocated for user processes (2 Gbytes), and four to the operating system (2 Gbytes). This allocation scheme permits a user process to locate instruction code in one segment, static data in a second segment, and dynamic data (stacks) in a third segment. Table 29 shows the structure of the C-Series architecture virtual address space.

Table 29 C-Series architecture virtual address space

Ring	Virtual address	Virtual address space (segment)	Owner	Ring execution priority
0	0000 0000 1FFF FFFF	0	SYSTEM	HIGHEST (0)
1	2000 0000 3FFF FFFF	1		
2	4000 0000 5FFF FFFF	2		
3	6000 0000 7FFF FFFF	3		
4	8000 0000 9FFF FFFF	4	USER	
	A000 0000 BFFF FFFF	5		
	C000 0000 DFFF FFFF	6		
	E000 0000 FFFF FFFF	7		LOWEST (4)

Data referenced by a byte-virtual address can begin on any arbitrary byte boundary. A 64-bit operand can begin on any one of eight byte boundaries. The byte address generated by an instruction references the first byte (byte 0) of an operand. However, where storage allocation is not controlled by the system, the best CPU performance is obtained if certain memory alignment rules are followed. The recommended boundaries for aligning each respective data representation in memory are:

- **Byte**—Not applicable.
- **Halfword**—Least-significant address bit is 0.
- **Word**—Least-significant two address bits are 00.
- **Longword**—Least-significant three address bits are 000.

Addressing modes

C-Series instructions that reference main memory must generate one or more main memory addresses. Many of these instructions explicitly generate an address called the *effective address*, or *effa*. Several different addressing modes can generate an *effa*. These modes use different combinations of address registers, the immediate field of the instruction, and the contents of memory to form the effa. These addressing modes are listed in Table 30.

Table 30 C-Series addressing modes

Addressing mode	Assembler form	Address
Absolute	`address`	address
Deferred	`(Aj)`	Aj
Indexed	`offset(Aj)`	Aj + offset
Indexed Deferred (C4600 only)	`(Ai,Aj)`	Ai + Aj
Indirect Absolute	`@addr`	Mem(address)
Indirect Deferred	`@(Aj)`	Mem(Aj)
Indirect Indexed	`@offset(Aj)`	Mem(Aj + offset)
Indirect Indexed Deferred (C4600 only)	`@(Ai,Aj)`	Mem(Ai + Aj)

In Table 30, *address* and *offset* refer to the 16-bit or 32-bit immediate encoded in the instruction, `Ai` and `Aj` refer to the contents of a register, and Mem(x) refers to the contents of memory location x.

The Indexed Deferred and Indirect Indexed Deferred modes are only supported on the C4600 Series CPUs.

See the CONVEX Assembly Reference Manual for additional information, including a description of how these modes are encoded in an instruction.

Process structures

A *process* is a collection of one or more threads and associated data within a virtual address space defined by a *context*. A process includes the current values of the PC, PSW, A, S, V, and communication registers, and variables for the threads at any given time during execution.

A *thread* is a single stream of execution. The C100 Series CPUs support only one thread per process, while the multiprocessing C-Series CPUs support up to 32 multiple threads of execution per process (2^5 in TID <4..0>). The maximum number of threads that can execute *simultaneously* in a process is a function of the number of CPUs in a complex.

A process consists of a protected address space, a state, a hardware context, and a software context.

The *state* of a process is the condition of a process at any given instant. The state of a process changes in response to system events. The process states are defined as:

- **Executing**—A process that is actually using a CPU at a given instant.
- **Sleeping**—A process that is idle and not executing on a CPU.
- **Blocked**—A process that cannot continue execution and is waiting for an external system event to occur before the process can continue execution.
- **Ready**—A process that is temporarily stopped in order for some other process to continue execution.

The *context* of a process consists of a hardware context and a software context. The hardware context consists of the contents of all or part of a CPU's general and status register sets. The software context consists of all or part of the program's variables and other data structures within the program and in the operating system on behalf of the program.

A C-Series process is constructed with two general partitions. One partition is the user program which resides in ring 4. Ring 4 is comprised of segments 4 through 7, and spans 2 Gbytes.

The other partition is the operating system kernel, its data structures, and other shared resources that are shared by all user processes. This kernel part of the operating system includes the page tables used for address translation, buffers for disk or terminal records, and the various control blocks created by the operating system for the user.

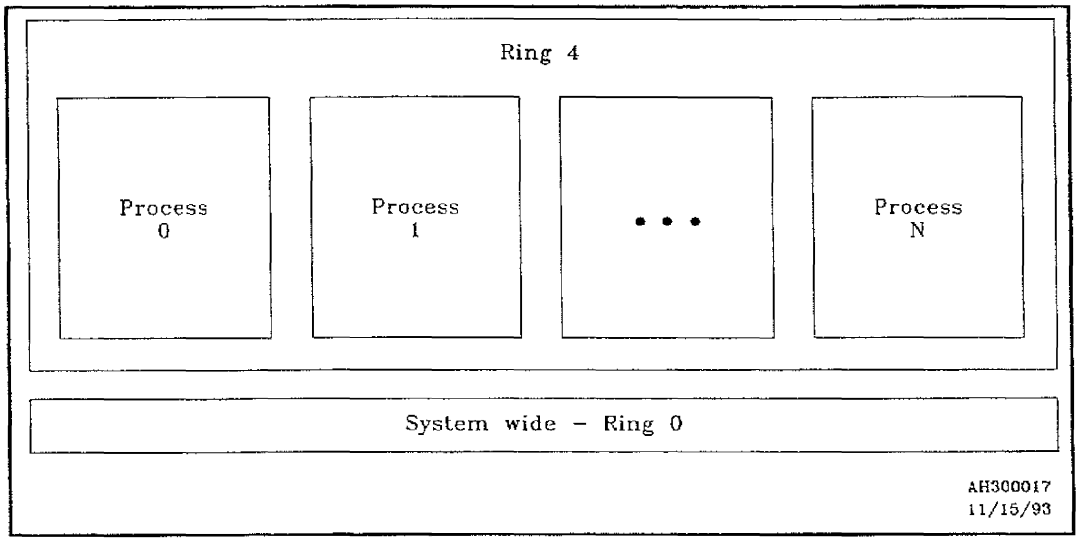
The shared part of the operating system resides in ring 0. The operating system's kernel data structures reside in rings 0, 1, 2,

and 3. These data structures vary with the operating system implementation and are not part of the C-Series architecture.

Since ring 0 is system-wide and not process-wide, every process shares the same ring 0. Interrupt processing is an example of a system-wide service that is performed in ring 0.

The partitioned structure of a CONVEX process is shown in Figure 17.

Figure 17 Process, system, and ring structures



Process control

The process control mechanism uses stacks, stack frames, and process return blocks to manage process activity. These data structures contain the software and hardware context information used for controlling the execution of a process.

Stacks and stack frames

Stacks are generally used as *dynamic storage* allocated and deallocated during the execution of a user program. Stacks contain the hardware and software context. This process state information is managed in units called *stack frames*. A stack frame typically consists of an area that contains the register contents from the previous execution context, an area that contains storage for temporary variables local to this context, and values necessary to manage the current stack frame, as well as a link to the previous frame.

A *stack* is an array organized as a last-in-first-out (LIFO) buffer. It is sometimes called a push-down stack. The C-Series architecture implements a stack as an array of 32-bit words, although longword operands can be used in stack operations. This means that all instruction set primitives that manipulate the stack increment or decrement the stack pointer (SP) by four or eight.

The architecture defines three registers to maintain a stack:

- The stack pointer (SP, A0) is discussed in the "Stack operations" section on page 77
- The argument pointer (AP, A6) references the first argument contained in a stack frame that is pushed on the stack when a subroutine is entered.
- The frame pointer (FP, A7) provides dynamic linkage between frames contained on a stack.

One, two, or all three of these registers are affected, depending on the type of operation performed on the stack. Generally, subroutine entry and exit use all three registers.

The following subsections describe some of the types of operations performed on a stack, stack frames, process return blocks, and stack structure for subroutine entry and exit.

Refer to the *CONVEX Assembly Language Reference Manual (C Series)* for specific details about instructions used in stack operations and management.

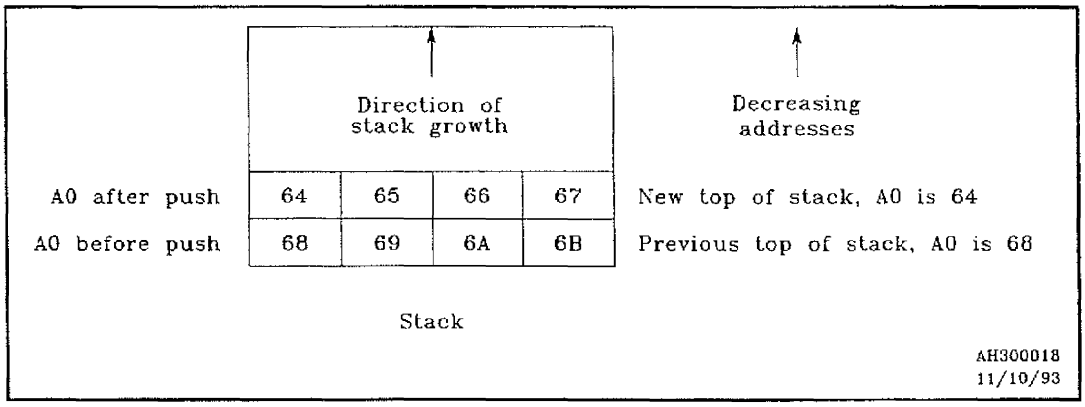
Stack operations

The C-Series architecture supports two primitive operations on a stack. A *push* operation, using a `psh` instruction, stores an operand on the stack and decrements the SP (A0) by 4 or 8. Pushing a word will decrement the current value in SP by 4. The word is then stored in the location referenced by the new value of the SP. Address register SP (A0) points to the top element of the stack (the last location used).

A *pop* operation, using a `pop` instruction, removes an operand from the stack and increments the SP (A0) by 4 or 8. Popping a word from the stack will fetch the top element from memory and increment the stack pointer by 4.

The example in Figure 18 shows the top of the stack is initially at byte 68 (hex).

Figure 18 `Push` and `pop` stack operations



Pushing a word onto the stack requires that the stack pointer (A0) first be decremented by 4 (68 - 4 = 64). The word to be pushed is then stored into bytes 64, 65, 66, and 67.

Popping a word from the top of the stack fetches bytes 64, 65, 66, and 67. Then it increments the stack pointer (A0) by 4 (64 + 4 = 68).

Only 32-bit and 64-bit quantities are supported in the C-Series instruction sets for push and pop operations on a stack. The stack should be initialized to begin on an integral 4-byte address boundary. Overt modification of the SP (by instructions that manipulate A0) by quantities other than multiples of four is not recommended. Even though the processor will continue to function, performance will be degraded.

No explicit stack overflow or stack underflow detection is performed by the hardware. Stack overflow and underflow may be detected by surrounding the allocated stack with inaccessible pages. Software-reserved bits in the protection fields of the no-access PTEs may be used to differentiate this type of access violation from other possible causes. Consequently, the protection trap handler can determine the reason for invocation.

Process return blocks

The hardware context of a process is managed with a data structure called *return blocks*. Depending upon the return block type, the information contained in a stack frame may include the contents of all or part of a CPU's general and status register sets (hardware context), and the contents of all or part of the program's variables (software context).

The C-Series architecture defines four types of return blocks:

- **Short**—A short return block is formed as a result of executing a `calls` instruction. The return address, PSW (FRL<1..0> = 11), A7, and A6 are saved.
- **Long**—A long return block is formed as a result of executing a `call` instruction. The return address, PSW (FRL<1..0> = 10), registers A1, A2,..., A7, and scalar registers S1, S2,..., S7 are saved.
- **Extended**—An extended return block is formed as a result of a system call, trap, or breakpoint. The return block contains the return address, PSW (FRL<1..0> = 01), all A registers, all S registers, plus some additional registers on multiprocessing C-Series CPUs. The saved SP (A0) references the value of A0 *prior* to the saving of the extended return block. The frame pointer (A7) that is saved in the extended return block references the value of A7 *prior* to the extended return block being saved.
- **Context**—A context return block may be formed as a result of a system exception. The context return block contains an extended return block plus internal CPU state. This internal CPU state information is unique to each CPU implementation. A context return block is pushed on the Ring 0 process stack.

The respective lengths of the short and long return blocks are identical for all CPU implementations in the C Series.

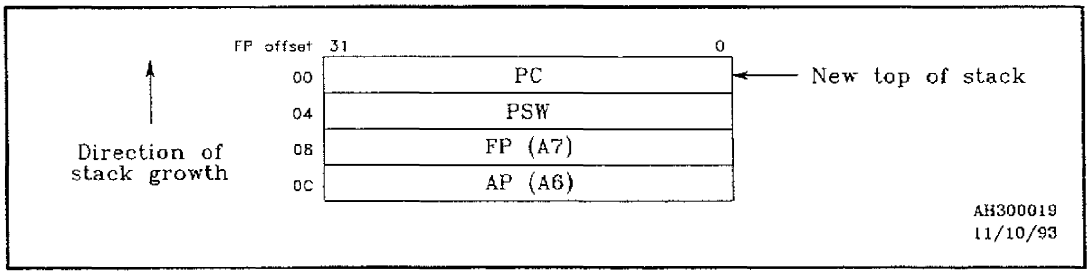
The length of an extended return block and the length of the context return block are both CPU-specific. That is, the C100 Series, the C3200/C3400/C3800 Series, and the C4600 Series CPUs have different extended return block lengths.

Short return block

A *short return block* is formed as a result of executing a `calls` instruction. The return address (PC), PSW (FRL=11), frame pointer (A7), and argument pointer (A6) are saved on the current stack.

After the short return block is pushed on the stack, the frame pointer (FP) is set equal to the stack pointer (SP). The format of a short return block is shown in Figure 19.

Figure 19 Short return block

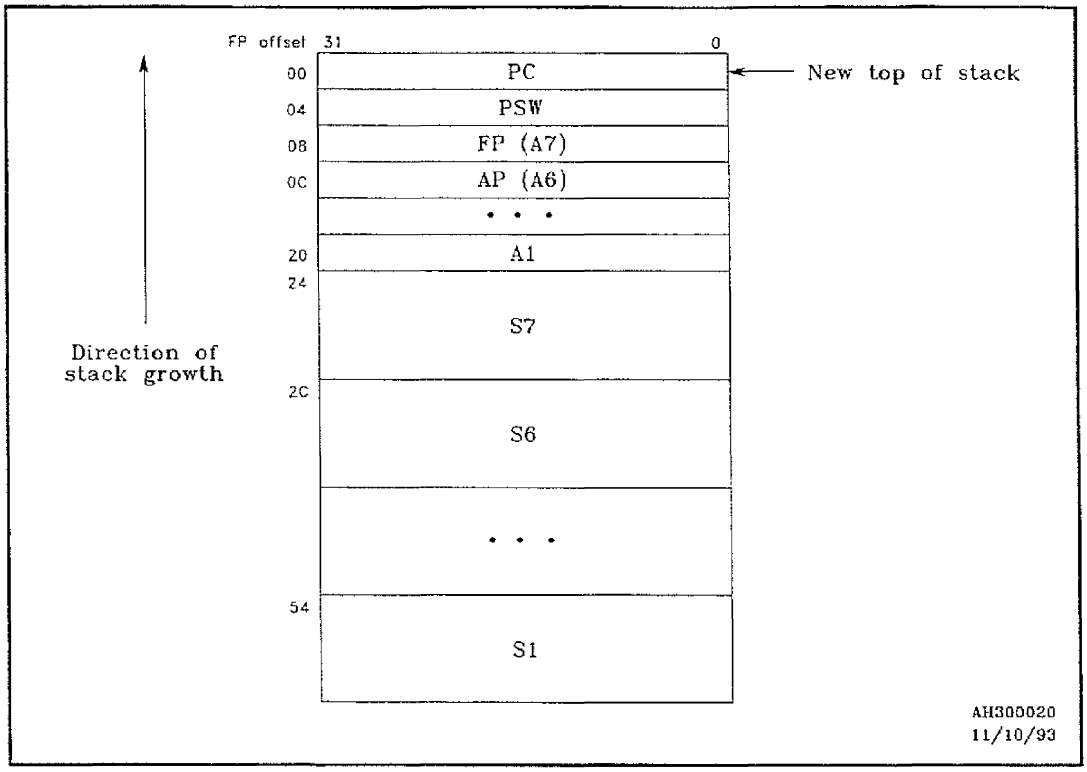


Long return block

A *long return block* is formed as a result of executing a `call` instruction. The return address (PC), PSW (FRL=10), address registers A1 through A7, and scalar registers S1 through S7 are saved. A0 and S0 are not saved.

After the long return block is pushed on the stack, the frame pointer (FP) is set equal to the stack pointer (SP). The format of a long return block is shown in Figure 20.

Figure 20 Long return block

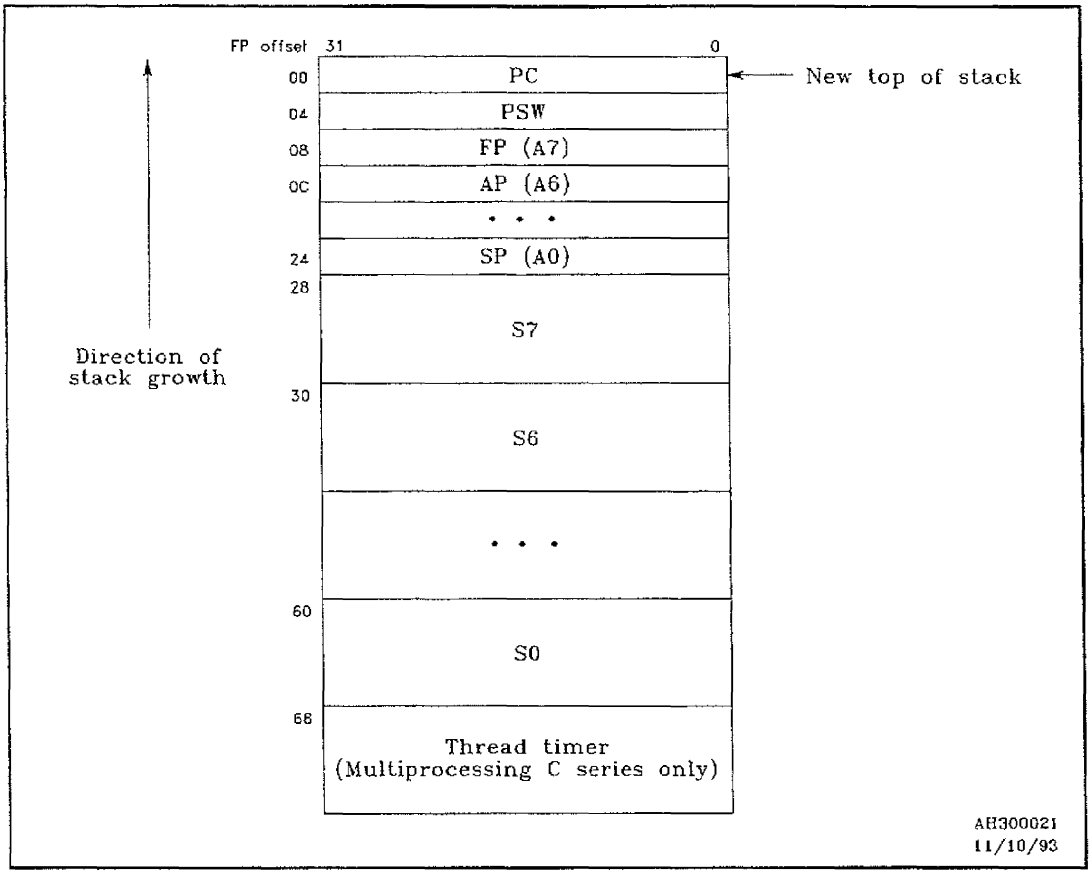


Extended return block - C100 and C3200/C3400/C3800
An *extended return block* is formed as a result of a system call (`sysc` instruction), an exception, or a breakpoint.

The extended return block contains the return address (PC), PSW (FRL=01), all the A registers, all the S registers and the thread timer. The stack pointer (A0) that is saved in the extended return block references is the value of A0 *prior* to the extended return block being saved.

After the extended return block is pushed on the stack, the frame pointer (A7) is set equal to the stack pointer (SP). The format of an extended return block is shown in Figure 21.

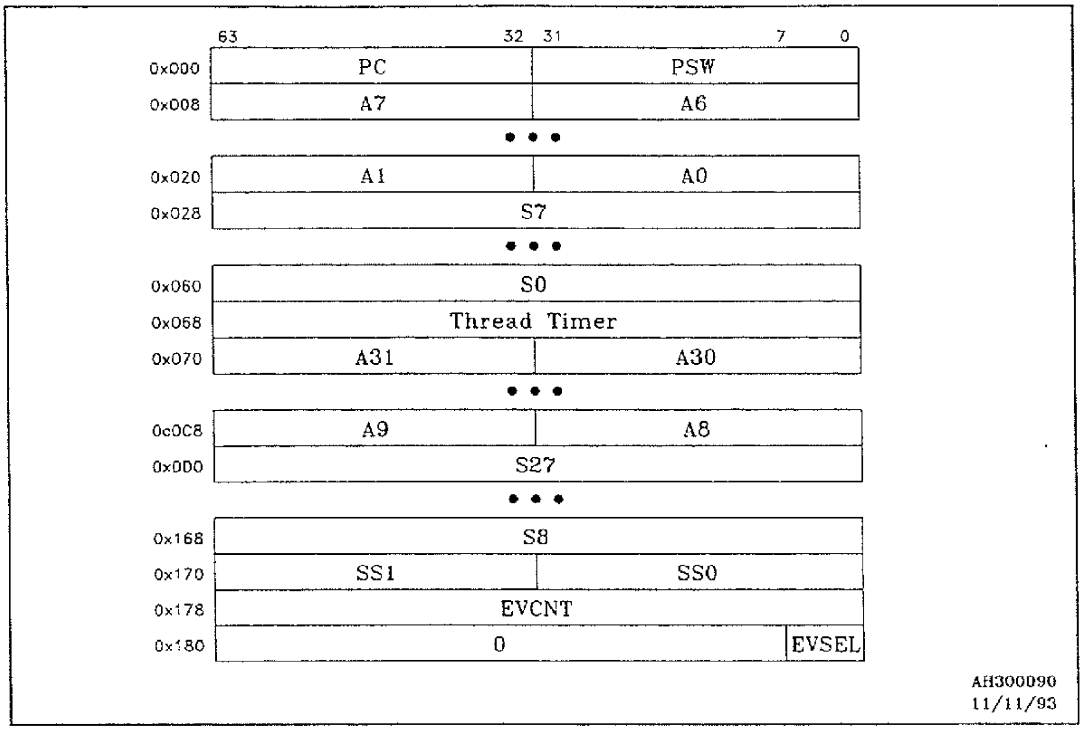
Figure 21 C100 and C3200/C3400/C3800 Extended return block



Extended return block - C4600

The C4600 CPU's extended return block contain the 32 address registers, the 28 scalar registers, the PC, PSW, SS1, SS0, thread timer, EVCNT, and EVSEL. The structure of an extended return block is shown in Figure 22. Note that the header of the C4600 series extended return block is identical to a C3200/C3400/C3800 extended return block.

Figure 22 C4600 Extended return block



Context block

A *context block* may be formed as a result of a system exception. The context block is an extended return block with internal machine state pushed prior to the extended return block. This internal state, or context portion of the return block, is implementation-dependent. A context block is always pushed on a ring 0 process stack. The only field that distinguishes a context return block from an extended return block is the frame length bits in the PSW, FRL=00.

Return from a return block

The following instructions are used to return using each of these return blocks:

- The `rtn` instruction is used to return using the short, long and extended return blocks
- The `rtnc` instruction is used to return using a context return block

Stack frame structures

The structure of a stack frame for subroutine entry and exit is described in Figure 23.

The return block shown as part of the stack frame structure is one of the standard return blocks described in this section.

Figure 23 Stack frame structure for subroutine entry

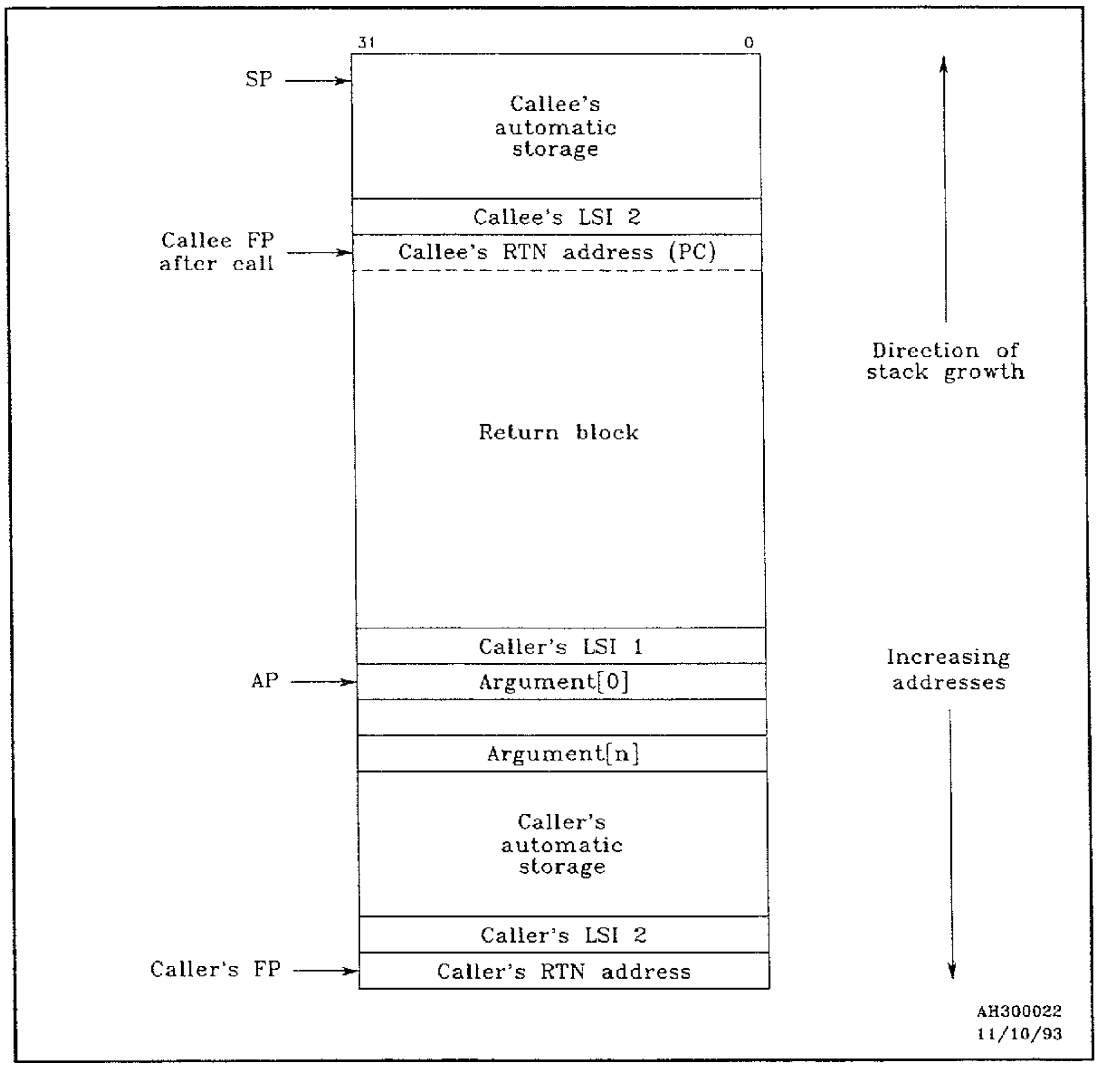
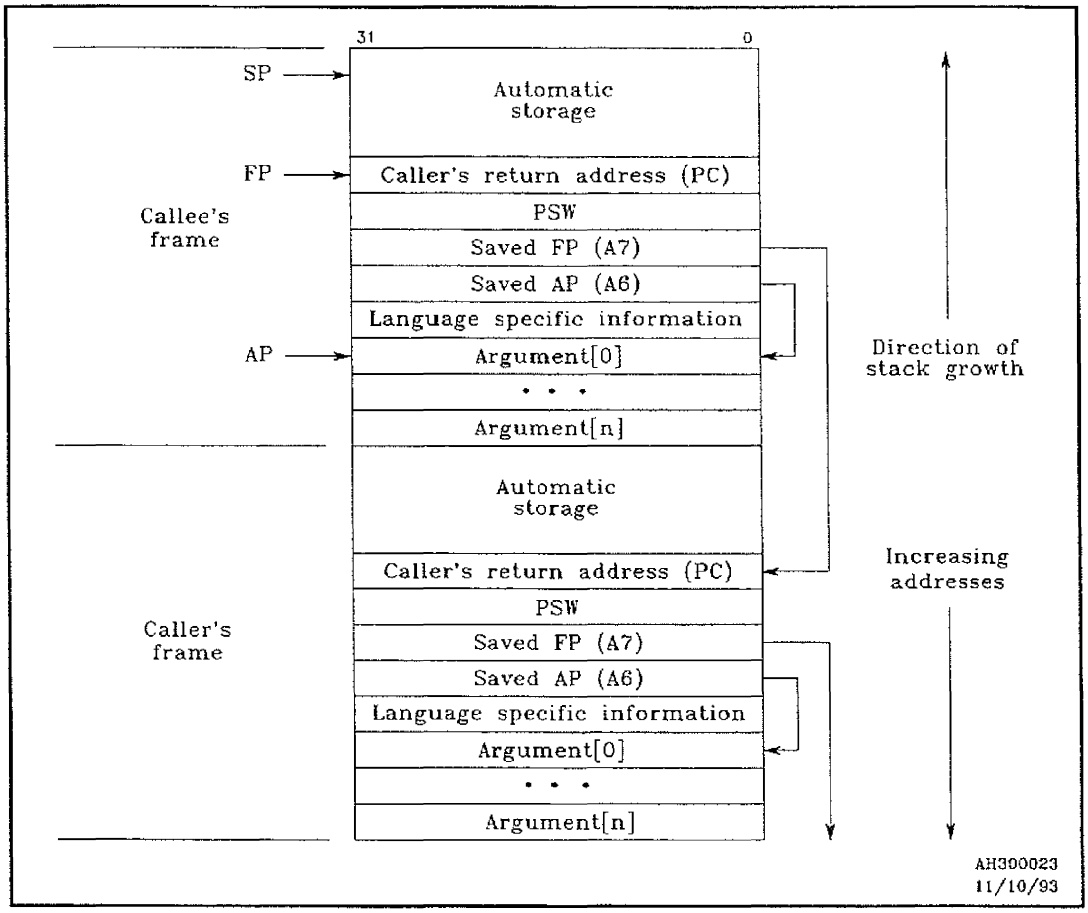


Figure 24 illustrates two short stack frames, generated by a `calls` instruction, and how they are linked together.

Figure 24 Stack structure after a short call



Stack switching

There is one stack per ring, with the exception of ring 0 (the highest priority ring) which can have more. Since each ring has at least one stack, the stack allocated to ring 4 is logically different from the stack for ring 3, for ring 2, and so on. A ring 0 stack is allocated for each thread that enters the kernel to handle subroutine calls, exceptions, and interrupts.

The ring 0 stack must always be aligned on a 4-byte word boundary. A machine exception occurs if it is not properly aligned. Ring 0 also has several other stacks used by the system for interrupt and exception processing.

A *system call* performs the following procedure for switching stacks:

1. After a successful system call, a new stack frame is created in the target ring, and an extended subroutine return block is pushed onto the target stack (the called routine's stack).
2. The stack pointer of the new stack is:
   a. **For C100 Series CPUs**—initially loaded from byte address `0000 0048` of page 0 of the called ring.
   b. **For multiprocessing C-Series CPUs**—initially loaded from the system resource structure. A pointer to this structure located at byte address `0000 0048` of page 0 of ring 0.
3. After the extended return block is pushed, the SP (A0) is copied into the FP (A7).
4. The PC is loaded with the value from the gate referenced by `sysc`. Refer to the `sysc` instruction description in the *CONVEX Assembly Language Reference Manual (C Series)*. Refer to the discussion on gate processing in the "Inter-ring procedure call and return" section on page 121.

The stack pointer value saved in the extended return block represents the value of the caller's stack pointer at the time of the system call. The stack pointer value is saved in order to make a proper return from a multiplexed stack resource structure. It is the link back to the outer ring's stack and is contained within the extended return block pushed on the inner ring's stack.

Arguments for the system call are maintained in a programmer-defined area, such as an argument packet or on the stack. Additional details for an inward system call are covered in the description of the `sysc` instruction and in the section on inner-ring procedure calls and returns.

The converse of a system call is a *system return*, which is implemented with a `rtn` instruction. Unlike a system call, no gate processing is necessary.

An inner ring (the kernel) can unconditionally access an outer ring (a process), so memory protection is not required. A system return is similar to a normal return with the following differences:

- The PC ring field can change
- All returns must be the same ring or outward (away from ring 0)

- The return block popped off the stack must be an extended or context type
- After the return block is popped from the stack:
  - **For C100 Series CPUs**—The updated SP of the inner ring is restored to byte address `0000 0048` of page 0 of the ring containing the `rtn` instruction.
  - **For multiprocessing C-Series CPUs**—The updated SP of the inner ring is restored to the system resource structure pointed to by the system resource structure pointer located at byte address `0000 0048` of page 0 of ring 0.

This guarantees that the stack will be initialized to the proper values (with subsequent system calls to the same ring).

Resource structures

Resource structures are pre-determined memory locations used to store specific registers, flags and lock bits.

Shared resource structures

In multiprocessing C-Series CPUs, the communication registers can be viewed as a form of fully semaphored memory, available in considerably smaller quantities than virtual memory. Communication registers and related operations are described in the "Communication registers" section in Chapter 5. One of the primary functions of communication registers is providing software a means to relocate frequently accessed data from virtual memory to a location with internal locks.

Memory duals of the communication instructions perform primitive functions using virtual memory that are analogous to functions that manipulate communication registers. Software can use the memory duals of the communication instructions to create data structures in memory, then relocate the critical data structures to a communication register set.

These memory duals operate on a data structure called a *shared resource structure*. A shared resource structure is a simple shared-access memory structure used by the multiprocessing C-Series CPUs. The shared resource structure defines a two or three 32-bit word memory format that includes a data word or longword, and synchronization bytes to synchronize access to the structure.

For operations on data words, the first word of the structure is the synchronization word, and the second is a data word. For operations on longword data, the first word of the structure is the synchronization word, the second is the most-significant word of the data longword, and the third word is the least-significant word of the data longword. For both structure types, the synchronization word contains a *lock* byte and a *valid* byte.

The lock byte is the first-level of semaphoring, and is set to `0xFF` while the shared resource structure is in transition. A structure is in transition when data is being written to the structure or read from the structure. Since the shared resource structure is semaphored, the lock byte must be successfully "test-and-set" as the first-level access of semaphoring. Test-and-set is an indivisible operation provided by the memory system.

An *atomic* operation is an indivisible operation. That is, once the operation begins, no other operation or event, such as interrupts, may intervene until the operation is complete.

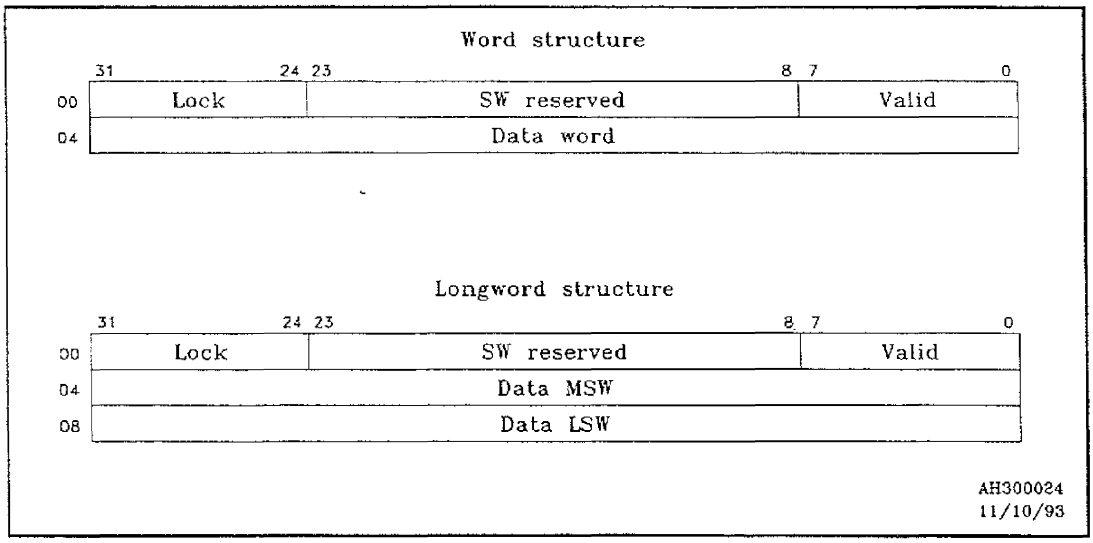
The shared resource structure is specifically used by some instructions when atomically incrementing and decrementing memory, or when pushing and popping a shared resource.

The next synchronization byte is the valid byte, which is set if valid data follows the synchronization (lock) byte. The *valid* byte is used by the `sndr`, `rcvr`, and `incr` instructions to indicate that valid data is present in the shared resource structure. The increment instructions, `incr.w` and `incr.l`, operate on a resource structure by incrementing or decrementing the data field by the contents of an address register.

These two synchronization bytes model the semaphoring inherent in the communication registers. The lock byte models the inherent indivisible access to the communication registers provided by their primitive functions. That is, the memory system doesn't provide primitive operations like *send* and *receive*. The valid byte models the communication lock bit, showing whether valid data is in the register, that is, the structure is "valid."

The format of the word and longword shared resource structures is shown in Figure 25.

Figure 25 Word and longword shared resource structures



As an example of how this structure works, consider the `rcvr.w` *effa*`,Ak` instruction. This instruction is the memory dual of the `rcv.w` *Ceffa*`,Ak` instruction in the "Communication registers" section in Chapter 5.

First, a test-and-set is performed on the lock byte. If this succeeds (the lock byte was initially `00`), the valid byte is read. Next, the data word is read into address register Ak. If the valid byte is `0xFF`, then valid data exists and a success status of 1 is returned in carry (C). Otherwise, a failure status of 0 is returned.

If the test-and-set of the lock byte succeeds, the data is always read into register Ak. This occurs, regardless of the state of the valid byte, because the `rcv.w` communication register instruction always reads the contents of the communication register into register Ak. A single-level of synchronization is required for the communication registers as previously mentioned.

Stack resource structures

An extension of the shared resource structure, called a *stack resource structure,* is provided to allow stack operations, such as push and pop, on a stack of word resource structures. The resource instructions, `pshr` and `popr`, perform these stack operations. These instructions make the word resource structure operate as a stack, with the header located at the base of the stack. As shown in Figure 26, this stack grows "upwards," that is, the addresses of stack entries increase with stack growth, which is the reverse of the process stack.

These instructions use the second word of the word resource structure as a stack *index* to a contiguous array of elements immediately following the resource structure in the stack. Since these instructions ignore the valid byte (bits <7..0>) in the word resource structure, this byte is reserved for future use by hardware.

Instead of the valid byte, the word resource structure contains a depth word (index) that shows the number of elements in the structure.

The `pshr Ak, <effa>` instruction pushes data onto this structure by successfully test-and-setting the lock byte, then adding 4 to the index, and writing the pushed value to the base address + 4 + new index count. The value of 4 is added to the index to increment past the index word.

The `popr <effa>, Ak` instruction pops data from this structure by successfully test-and-setting the lock byte, reading data from base address + 4 + index, and then decrementing the index by 4.

Figure 26 shows the stack resource structure as used by the `pshr` and `popr` resource instructions.

Figure 26 Stack resource structure

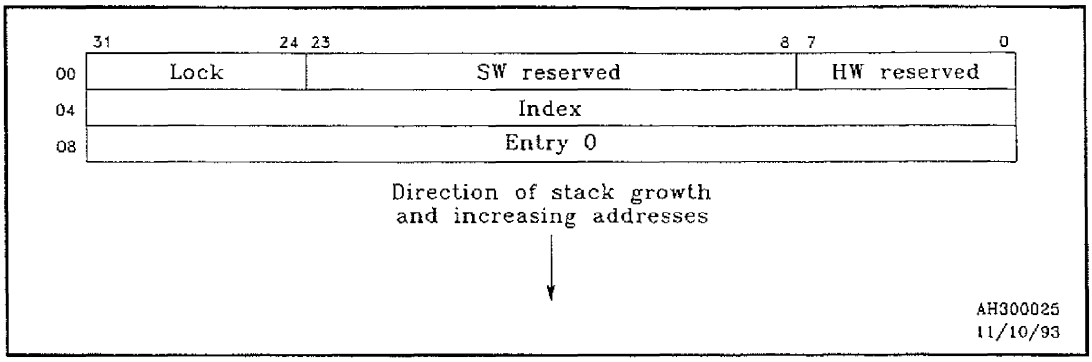
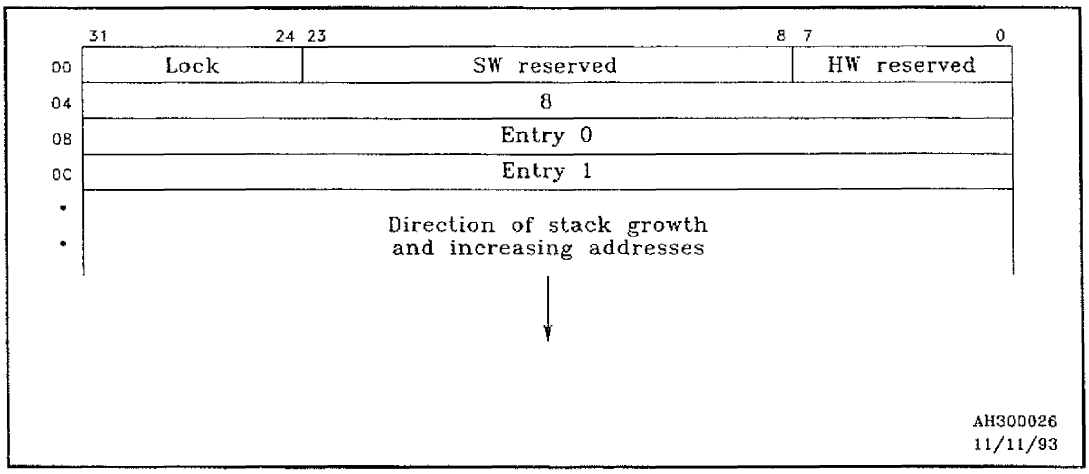


Figure 27 shows an example of a stack resource structure containing two pushed entries.

Figure 27 Stack resource structure with two pushed entries



The address of the top of the stack is 4 + 8 = C. The address of the top of the resource stack is always located by taking the address of the stack resource structure index plus the contents of the stack resource structure index.

Executing a `popr` instruction would return the value "Entry 1" in the specified register and decrement the index by 4 making the top of the stack now 4 + 4 = 8. A `pshr` adds 4 to the index and then writes the entry to the structure at its new top. The `popr` instruction returns an underflow status and the value returned in register Ak is invalid if the index in the resource structure is zero.

System resource structures

When a process enters ring 0 (called *ring crossing*), the state of that process is pushed onto a stack in ring 0. This state is either an extended frame for interrupts, system calls and exceptions, or a context frame for page faults.

The C100 Series CPUs have a stack pointer in page 0 that points to this stack. It is loaded from page 0 during a ring crossing. If a fault occurs in ring 0, a context frame is pushed on a stack that is always available with a separate context stack pointer.

However, since more than one thread in a process could be crossing rings or faulting at the same time, the multiprocessing C-Series CPUs define a *system resource structure* to manage allocation of available stacks in ring 0. The system resource structure is a stack of pointers to available stacks that are allocated to threads. Accesses to the system resource structure are synchronized by placing part of this structure in a communication register with the other part contained in memory. The communication register lock bit is used as the semaphore in order to control contention between multiple threads. Refer to the "Communication registers" section in Chapter 5, "Multiprocessor management," for more information concerning communication register operations.

The system resource structure for ring 0 is managed differently than a process stack resource structure in ring 4. Whenever a thread either crosses a ring or faults (in any ring), the virtual address of the communication register contained in address `0000 0048` of page 0 (of the ring being entered) is read. Ring 0 is for faults, interrupts, and system exceptions.

This communication register contains the base address and stack index to a list of available stack pointers located in memory.

These stack pointers point to system stacks used for cross ring calls and returns, and for saving and restoring context blocks.

Figure 28 shows the base address and index of the system resource structure contained in this communication register.

Figure 28 System resource structure

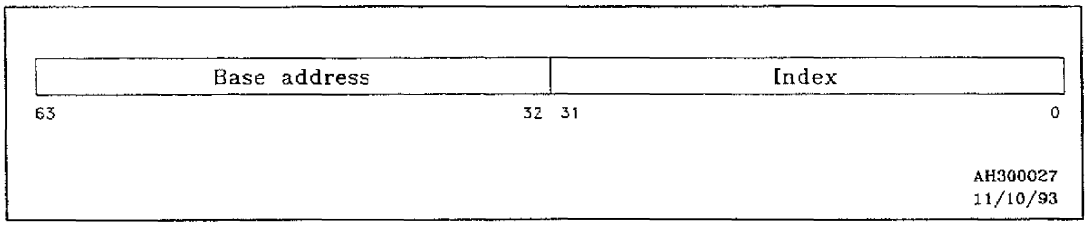
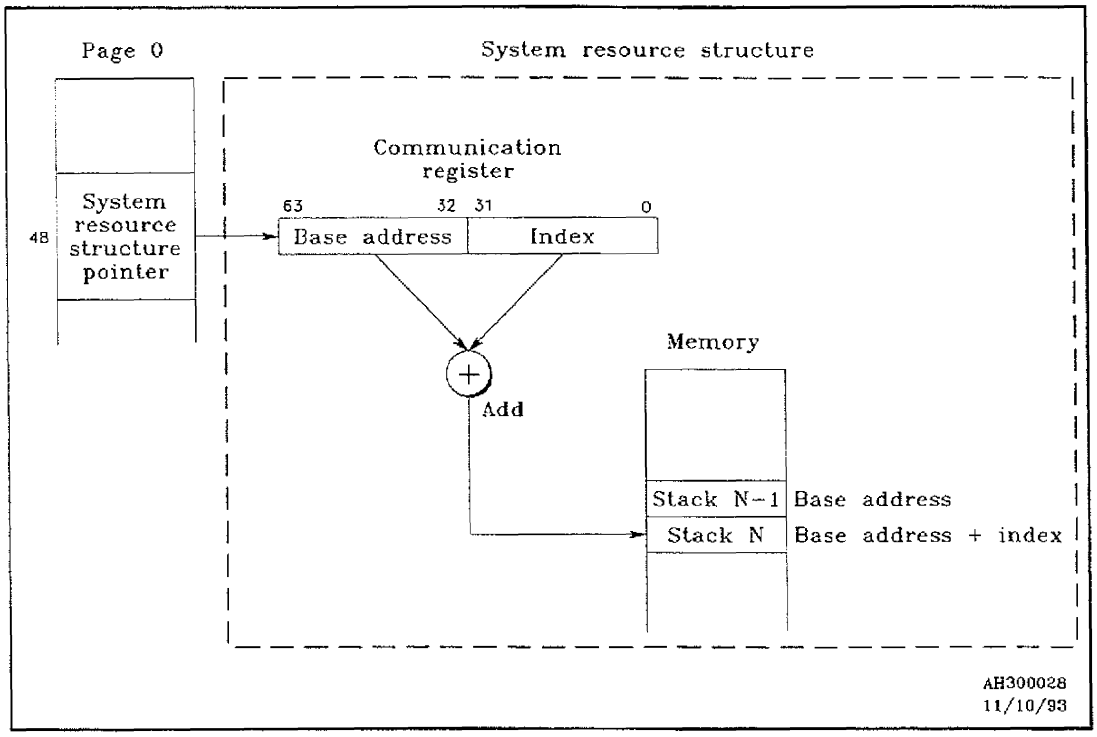


Figure 29 shows how a stack is allocated and deallocated from the system resource structure.

Figure 29 Accessing the system resource structure for multiprocessing C-Series CPUs



After the address of the communication register is read, the communication register shown in Figure 28 and Figure 29 must be successfully received. If this register cannot be received immediately, the receive operation retries until the register is successfully received.

The 64-bit data portion of this register is subdivided into two words. The most-significant word is the base address which contains the virtual memory address of a list of stack pointers for available stacks. The least-significant word is the stack index which is the byte offset from the virtual address of the next available stack pointer plus 4.

When a stack is allocated from the list, the stack pointer is fetched by decrementing the stack index by 4 and the contents (base address + decremented index) are read. The decremented value of the index is sent back to the communication register, making the system resource structure available for access by other threads.

When the thread eventually exits ring 0 via the `rtn` or `rtnc` instructions, the stack is returned to the structure in the following sequence:

1. Address `0000 0048` in page 0 is read to fetch the communication register address that must subsequently be received.
2. The stack pointer to be returned is written to memory at address (base address + index).
3. The index is incremented by 4 to reflect the stack pointer being "pushed" onto the structure.
4. The incremented value is sent back to the communication registers, making the structure available for access by other threads.

Virtual memory management

The address space of the C-Series architecture is implemented as a virtual address space. Since the virtual address space normally spans a larger range of memory addresses than the physical address space, a virtual address may not be associated with a valid physical address at any given time. Therefore, the referenced data may or may not be in physical memory.

Virtual-to-physical address translation is performed by the Address Translation Unit (ATU). The ATU accelerates the translation of virtual addresses to physical memory by an internal address cache. The ATU is described in "Virtual-to-physical address translation" section on page 110.

The C-Series architecture manages this type of memory structure by implementing the following memory management mechanisms and structures:

- **Segment**—A virtual-address contiguous 512-Mbyte block of memory.
- **Segment descriptor register (SDR)**—A 32-bit register containing information necessary to begin translating a virtual address offset to an address in physical memory.
- **Ring of execution**—Corresponds to a memory segment or range of segments with respect to the virtual address space of a process.
- **Page**—A contiguous 4-Kbyte block of memory that is both virtual-address and physical-address contiguous.
- **Page frame**—A page stored in physical memory.
- **Page frame base**—The beginning address (zero based) of a page in memory.
- **Page table**—A table that contains 4-byte entries called page table entries (PTEs). It begins on an integral page boundary and is contained in one page frame or less.
- **Page table entry (PTE)**—One of several 32-bit entries containing information necessary to translate a virtual address to a physical address. Other status bits within a PTE determine if a page is resident in physical memory and determine the validity of the memory reference from a protection viewpoint. A PTE is aligned on an integral word boundary. Refer to "Page table entries" section on page 101.
- **Referenced bit**—A bit associated with a page frame that indicates a valid read or write has occurred. Referenced bits are discussed in the "Referenced and modified bits" section in Chapter 7.

- **Modified bit**—A bit associated with a page frame that indicates a valid write has occurred. Modified bits are discussed in the "Referenced and modified bits" section in Chapter 7.
- **Address translation unit (ATU)**—A programmer- invisible address cache that is maintained in hardware. The ATU contains the most recently used virtual-to-physical address translations.

Although the entire virtual address space is always available to a user process, much less physical memory may be installed in the processor, and even less memory may be available to a given process. Consequently, the CPU contains an address translation mechanism that dynamically maps the virtual memory pages of a process onto physical memory page frames during process execution. This mechanism uses a hierarchical tree of lookup tables to perform the required address translation.

Multiprocessing extensions

The following subsections present the extensions added to the C-Series architecture that deal with virtual memory management. All of these architectural extensions are required to support the ability to *multiprocess*, or run a process on two or more CPUs simultaneously.

The extensions for multiprocessing CPU complexes add and define the following memory management attributes:

- **CPU**—One central processing unit, consisting of a scalar and vector subsystem.
- **Complex**—The entire set of one or more CPUs in a configuration.
- **Subcomplex**—Any subset of a complex.
- **Process**—A collection of instruction streams within a single virtual address space, that is, sharing the same SDRs.
- **Thread**—Any single instruction stream executing within a process.

As previously defined, a *process* is a collection of one or more *threads*. A supercomputer with more than one CPU in a complex could have a process executing on the entire complex, with one thread executing on each CPU.

Each CPU contains two registers which help define the memory management scheme for threads:

- **Communication index register**

  The communications index register (CIR) defines which set of the communication registers is being used by the process executing on a CPU. Each CPU has one CIR. The working relationship of processes, CIRs, and the communication registers is described in the "Communication registers" section in Chapter 5.

  The CIR defines which segment descriptor registers (SDRs) are in use by a processor, since the SDRs reside in the communication registers.

  The C3200 Series complexes implement a CIR as a 3-bit register field, allowing eight different index values to be represented. A 3-bit CIR does not limit a C3200 Series complex to only eight processes, but allows a maximum of eight processes to be loaded in the communication registers at one time.

  The C3400/C3800/C4600 Series complexes implement a CIR as a 5-bit register field, allowing 32 different index values to be represented. A 5-bit CIR does not limit a C3400/C3800/C4600 Series complex to only 32 processes, but allows a maximum of 32 processes to be loaded in the communication registers simultaneously.

- **Thread identifier register**

  The C3200/C3400/C3800 Series complexes use a 5-bit thread identifier register (TID) register to subdivide a process into disjointed threads. Up to 32 threads may exist in the same process, that is, have the same CIR. The TID makes the threads unique. The TID is primarily used for implementing unshared memory in the multiprocessing implementation.

  The C4600 Series complexes use a 5-bit TID register, but only three bits (<bits2..0>) are supported. Bits <4..3> are ignored. Therefore, only eight threads can exist in the C4600 Series complexes.

  The manner in which a processor *becomes* a particular TID is described in the "Multithreaded execution" section in Chapter 5.

Shared and unshared memory

As stated earlier, a process is a collection of up to 32 threads. A process can view each page of virtual memory as either *shared* or *unshared*.

Shared memory means that more than one thread may use the same virtual address to access the same physical location in memory. Proper synchronization must be maintained by software in this case.

Unshared memory means that each thread uses the same virtual address to access different physical locations in memory; for example, the stack. A process that may or may not execute in parallel (depending on whether idle CPUs are available) needs to be able to use the stack without additional software overhead. This ensures that one CPU is not popping something that another CPU pushed, and so forth.

The CIR identifies a process and its respective virtual-to-physical memory address translation. The multiprocessing CPU implementation expands the address translation mechanism by adding a level to the multilevel tree of lookup tables in order to accommodate multiple threads within a process. The PTE has 32 thread-level entries. In order for each thread within a process to have a unique address space, a thread-level PTE is indexed by the TID. The TID modifies the virtual-to-physical address translation for the thread to allow each thread of a process to have private (unshared) physical memory for unshared variables. Since threads are created and terminated by the hardware, independent of the operating system, the hardware must perform the equivalent of dynamic memory allocation. At the time a process is loaded for execution, the operating system defines a "pool" of available thread identifiers within the communication registers of the process. The operating system also marks the address translation descriptors (in memory) for all unshared pages per thread of execution, and allocates a page of memory for each TID in the "pool" for each unshared page. The hardware then allocates TIDs and corresponding unshared memory from this pool as CPUs enter and exit the process. Refer to Chapter 5, "Multiprocessor management," for more information regarding thread identification.

Segment descriptor register

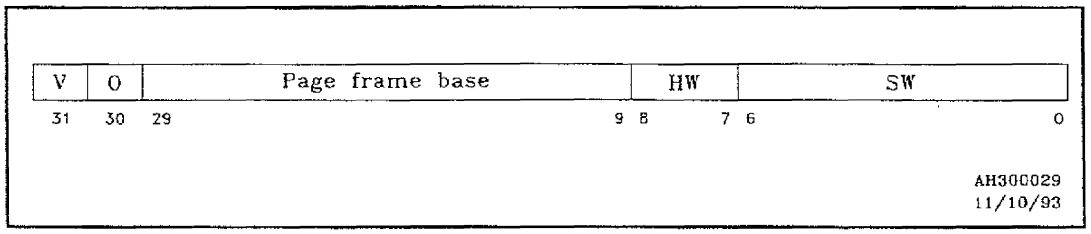
The first level of virtual-to-physical address translation involves a set of eight segment descriptor registers (SDR), one for each segment of physical memory numbered 0-7 (see Table 29).

Each SDR has a page frame base that points to the beginning of a first-level page table. The SDR also contains validity protection information. When a process is loaded for execution, the appropriate information is loaded into a CPU's SDRs. Although functionally identical, the format for an SDR is different between the C100 Series implementation and the multiprocessing implementations.

SDR format - C100

Figure 30 shows the format of a C100 Series SDR.

Figure 30 SDR Format—C100 Series CPUs



Bit <31>—Valid segment (V)
If this bit is clear, this segment is not valid. A segment is invalid when no translated virtual addresses can be associated with any pages in this segment. If an invalid segment is referenced, a system exception is generated and an error code is loaded into address register A5 after a context block is saved.

Bit <30>—Hardware reserved (0)
Must be zero.

Bits <29..9>—Page frame base (PFB)
The PFB is the high order 21 bits <29..9> of a 30-bit physical address, which points to the beginning of a first-level page table in physical memory. Bits <8..2> of this physical address come from bits <28..22> of the virtual address to be translated. Bits <1..0> of the physical address are zero. The first-level page table for a process requires 512 bytes. A single 4-Kbyte page frame contains all the first-level page tables for the eight segments of a single process.

Bits <8..7>—Hardware reserved (HW)
These bits have no meaning at this time. System software must not use these bits.

Bits <6..0>—Software reserved (SW)
The page frame bits in the SDR permit the operating system to indicate alternate structures for the 4-Kbyte page containing the first-level page table.

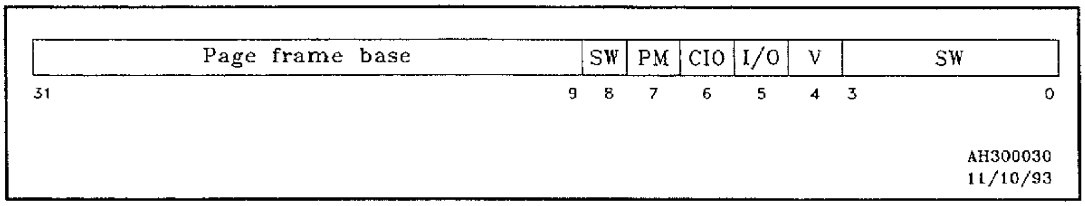
One possible structure is to only allocate 512 bytes rather than 4 Kbytes to the first-level lookup.

Another possible structure is to configure the 512-byte page as one of the eight possible 512-byte partitions in a 4-Kbyte page. This permits multiple first-level lookups to be physically contained in one page frame.

SDR format - C3200/C3400/C3800

Figure 31 shows the format of the C3200/C3400/C3800 Series CPU SDR.

Figure 31 SDR format—C3200/C3400/C3800 Series CPUs



Bits <31..9>—Page frame base
If a valid reference to a segment occurs, then bits <31..9> become the most-significant 23 bits of a 32-bit physical address, which points to the beginning of a first-level page table in physical memory. Bits <8..2> of this physical address come from bits <28..22> of the virtual address to be translated. Bits <1..0> of the physical address are zero. The first-level page table for a process requires 512 bytes. A single 4-Kbyte page frame contains all the first-level page tables for the eight segments of a single process.

Bit <8>—Software reserved (SW)
This bit has no meaning at this time. System software may use these bits in the future.

Bit <7>—Process monitor (PM)
Process monitoring is enabled when this bit is set to 1, and disabled when it is cleared to 0. This function is not generally available because special test equipment is required.

Bit <6>—Channel I/O (CIO)
When this bit is set, it indicates that the target segment is valid for I/O references from a channel processor.

Bit <5>—Input/Output (I/O)
When this bit is set, it indicates that the target segment is to be interpreted as part of I/O address space and not physical memory. This bit is implemented on the C3200 Series CPUs only.

Bit <4>—Valid segment (V)
When this bit is cleared, this segment is not valid. A segment is invalid if no translated virtual addresses can be associated with any pages in the segment.

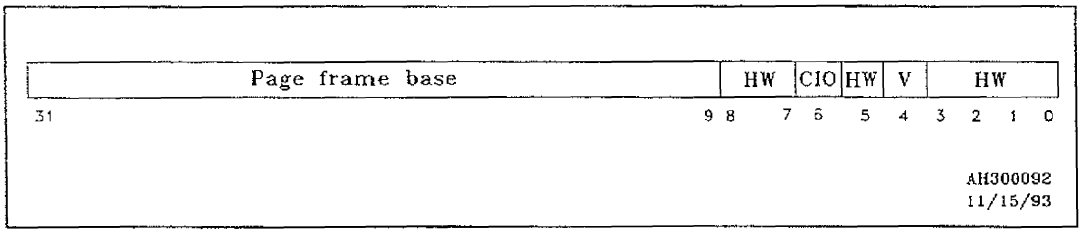
If an invalid page is referenced, a system exception is signaled, a context block is saved, and an error code is loaded into address register A5.

Bits <3..0>—Software reserved (SW)
These bits have no meaning at this time. System software may use these bits.

SDR format - C4600

Figure 32 shows the format of the C4600 Series CPU SDR.

Figure 32 SDR format—C4600 series CPUs



Bits <31..9>—Page Frame Base
If a valid reference to a segment occurs, these bits become the most-significant 23 bits of the physical address, which points to the beginning of a first-level page table in physical memory.

Bits <8..7>—Hardware reserved

Bit <6>—Channel I/O (CIO)
When this bit is set it indicates that the target segment is valid for I/O references from a channel processor.

Bits <5>—Hardware reserved

Bit <4>—Valid segment (V)
When this bit is cleared, this segment is not valid. A segment is invalid if no translated virtual addresses can be associated with any pages in the segment.

Bits <3..0>—Hardware reserved

Page table entries

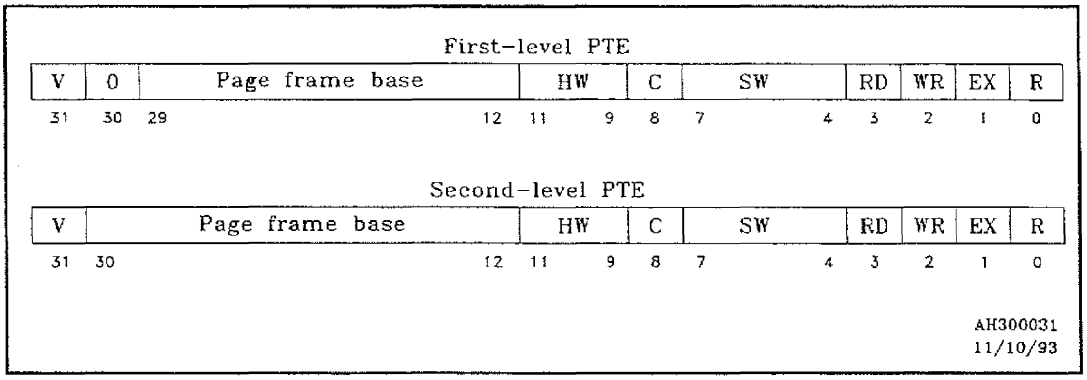
The second and third levels of virtual-to-physical address translation are accomplished using first- and second-level page table entries (PTEs). These are similar in function to segment descriptors. Like the SDRs, the format of the PTE for each architecture is slightly different.

A PTE is a 32-bit word aligned on an integral 32-bit boundary. The least-significant two bits of the byte address are 00. There are 128 PTEs in a first-level page table and 1,024 PTEs in a second-level page table. A PTE determines the validity of a memory reference and the physical memory location of a valid reference. A valid reference meets two requirements. First, the PTE must be valid. Second, the type of access being made (read, write, or execute) must be allowed by the appropriate protection bit of the PTE.

PTE format - C100

Figure 33 shows the formats of resident first- and second-level PTEs for C100 Series CPUs.

Figure 33 Resident PTE format—C100 Series CPUs



The following subsections describe the meaning of the bits in Figure 33.

Bit <31>—Valid PTE (V)

This bit indicates the validity of the PTE. If this bit is clear, any reference to this PTE while attempting to reference the associated page frame is an invalid reference. Conversely, if this bit is set, any reference to this PTE is a valid reference.

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

Bit <30..12>—Page frame base (PFB)
If a valid reference to a resident page occurs, then bits <30..12> become the most-significant 19 bits of a 31-bit physical address. The page frame base is modulo 4 Kbytes. For a first-level PTE, bit <30> is always 0. For a second-level PTE, bit <30> may be 0 or 1.

Bit <11..9>—Hardware reserved (HW)
These bits are reserved for potential use by hardware and are not interpreted. It is recommended that software not use these bits.

Bit <8>—Encached (C)
When this bit is clear, the data associated with the reference is encached in the CPUs data cache. When this bit is set, the referenced data is *not* encached.

Bit <7..4>—Software reserved (SW)
These bits are reserved for potential software use.

Bit <3>—Read access (RD)
This bit indicates the validity of a read access to the referenced page frame. A 0 indicates that no read access is permitted. A 1 indicates that a read access is permitted to the referenced page frame. If a read access is attempted and this bit is cleared, a system exception is signaled and an error code is loaded into address register A5.

Bit <2>—Write access (WR)
This bit indicates the validity of a write access to the referenced page frame. A 0 indicates that no write access is permitted. A 1 indicates that a write access is permitted to the referenced page frame. If a write access is attempted and this bit is cleared, a system exception is signaled, and an error code is loaded into address register A5.

Bit <1>—Execute access (EX)
This bit indicates the validity of an execute access (branch or jump to instruction) to the referenced page frame. A 0 indicates that no execute access is permitted. A 1 indicates that an execute access is permitted to the referenced page frame. If an execute access is attempted and this bit is cleared, a system exception is signaled, and an error code is loaded into address register A5.

Bit <0>—Resident page (R)
A 1 indicates the presence of the referenced page frame in the physical address space of the process. A 0 indicates the absence of the referenced page frame in physical memory. In this case, a page fault occurs and causes a system exception. If the referenced page frame is present, bits <30..12> are used as the

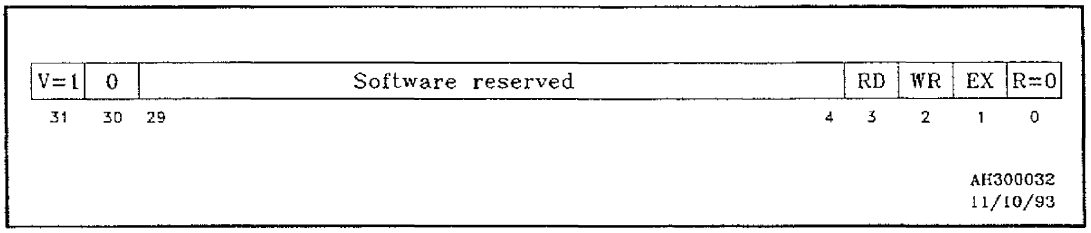
page frame address of the referenced page frame. Bit <0> is interpreted for valid PTEs only.

Note

Segment out-of-bounds errors may be detected by clearing all unused PTE valid bits to zero. Thus, during virtual-to-physical address translation for invalid pages, an out-of-bounds reference causes a system exception.

Figure 34 shows the format of a nonresident PTE for C100 Series CPUs.

Figure 34 Nonresident PTE format—C100 Series CPUs



Note

Bit <30> must be cleared. If bit <30> is set, an I/O reference can occur. The read, write, and execute bits are then interpreted to determine if the reference is valid.

PTE format - C3200/C3400/C3800

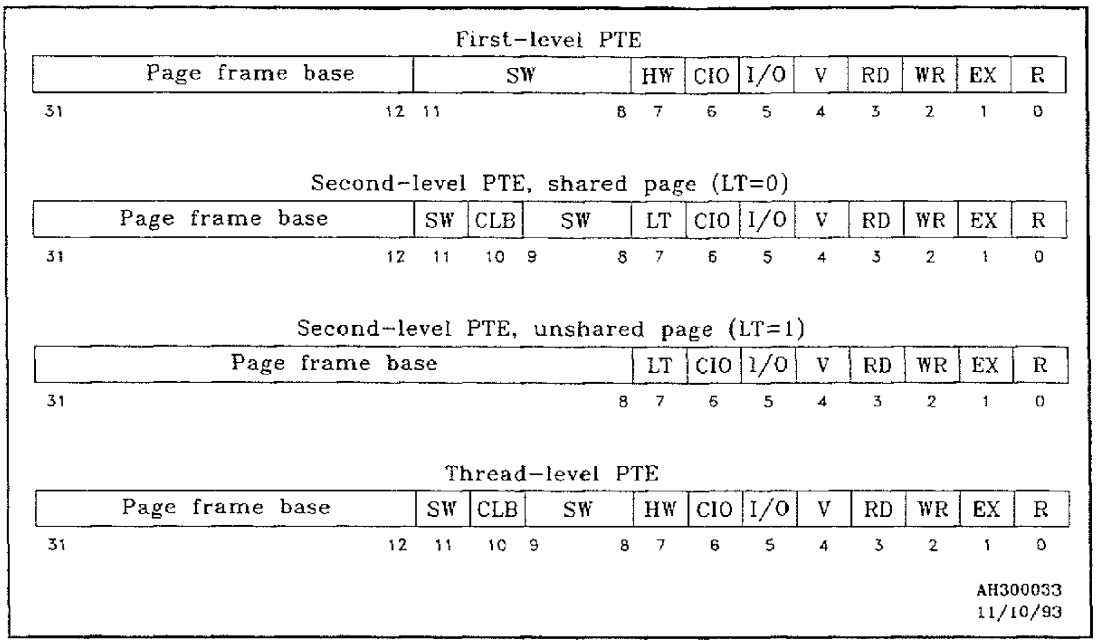
The C3200/C3400/C3800 Series CPU implementation expands the address translation mechanism by adding a level to the multilevel tree of lookup tables. The additional level of lookup tables is needed in order to accommodate multiple treads within a process. There are 32 thread-level PTE entries.

In order for each thread within a process to have a unique address space, a thread-level PTE is indexed by the thread identification register (TID).

Figure 35 shows the format of a resident PTE for the C3200/C3400/C3800 Series CPUs. Notice the thread-level PTE. If the level 3 (LT) bit (bit <7>) of the second-level PTE is set, the page frame base in the second-level entry is expanded to bits <31..8> and is used as the base address of a TID-indexed table of thread-level entries. These thread-level entries provide the final level of translation to the data page. If the LT bit is cleared, the pages do not require any thread-level translation.

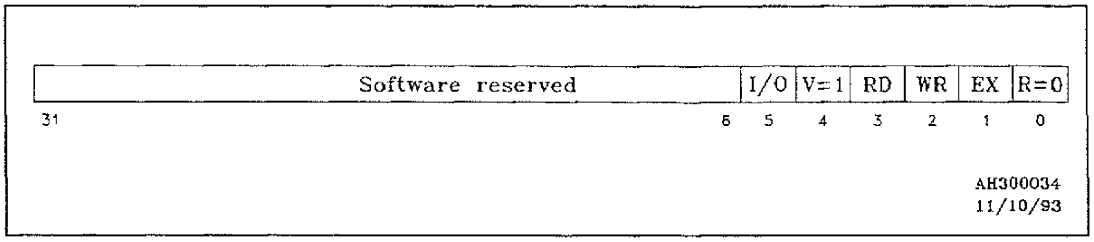
The LT bit determines whether data pages are shared or unshared between threads. When the LT bit is set, the data pages for different threads in a process are unshared. When the LT bit is clear, the data pages are shared between all threads in a process.

Figure 35 Resident PTE format—C3200/C3400/C3800 Series CPUs



The format of nonresident PTE for the C3200/C3400/C3800 Series CPUs is shown in Figure 36.

Figure 36 Nonresident PTE format—C3200/C3400/C3800 Series CPUs



In Figure 35 and Figure 36, all software-reserved fields are not interpreted by hardware when a PTE is valid (valid bit is set) and nonresident (resident bit is cleared). The read, write, and execute bits are interpreted by the hardware whenever a reference is made to a PTE with the valid bit set.

Note

A segment out-of-bounds error is detected when an invalid PTE is accessed. A reference to an invalid PTE results in a system exception.

The following subsections describe the meaning of the bits in Figure 35 and Figure 36.

Bits<31..12> or bits <31..8>—Page frame base (PFB)
If a valid reference to a resident shared page occurs, then bits <31..12> become the most-significant 20 bits of a 32-bit physical byte address. The page frame base is modulo 4,096 bytes. For unshared pages, bits <31..8> become the most-significant 24 bits of a 32-bit physical address of the thread-level PTE.

Bits<11..8>—PTE dependent
For unshared data pages, these bits are part of the page frame base of the thread level PTE. The PTE-dependent bits are defined as follows:

- **Bit<11>—Software reserved (SW)**
  This bit is reserved for potential use for software.
- **Bit<10>—Cache load bypass (CLB), or software reserved bit (SW)**
  For level 2 PTEs of shared pages and thread-level PTEs, this bit determines whether the data cache is bypassed (when set to 1), or not (set to 0) on load operations from the target page frame, forcing the load operation to reference physical memory. The CLB bit does not keep the data from being encached on stores.

  For all other page table entries, this bit is reserved for potential use by software.

  In some situations the operating system views the cache consistency rules to be too stringent. That is, the overhead for full synchronization is too expensive, or in some cases impossible to enforce. Therefore, the multiprocessing implementations use the CLB bit as part of the bottom-level PTE (PTE2 or PTET) to force the referenced page frame to appear not encacheable.
- **Bits<9..8>—Software reserved bits (SW)**
  These bits are reserved for potential use for software.

Bit<7>—Level 3 (LT)/thread-level, or hardware reserved (HW)
For level 2 PTEs, this bit determines whether the data page is shared (zero) or unshared (one). For first and thread-level entries, these bits are reserved for potential use by hardware.

Bit<6>—Channel I/O (CIO)
This bit indicates that the target page frame is valid for I/O references from a channel processor.

Bit<5>—Input/Output (I/O)
For C3200 Series CPUs only, this bit indicates that the target page frame is to be interpreted as part of I/O address space and not physical memory.

Bit<4>—Valid PTE (V)
This bit indicates whether or not the PTE is valid. When this bit is set, the PTE is valid. When this bit is cleared, the PTE is invalid. Accessing an invalid PTE results in a *segment out-of-bounds error* and a system exception.

Bit<3>—Read access (RD)
This bit indicates the validity of a read access to the referenced page frame. When this bit is cleared, it indicates that no read access is permitted. When this bit is set, it indicates that a read access is permitted to the referenced page frame. If a read access is attempted, and bit <3> is 0, a system exception is signaled, and an error code is loaded into address register A5.

Bit<2>—Write access (WR)
This bit indicates the validity of a write access to the referenced page frame. When this bit is cleared, it indicates that no write access is permitted. When this bit is set, it indicates that a write access is permitted to the referenced page frame. If a write access is attempted and bit <2> is 0, a system exception is signaled and an error code is loaded into address register A5.

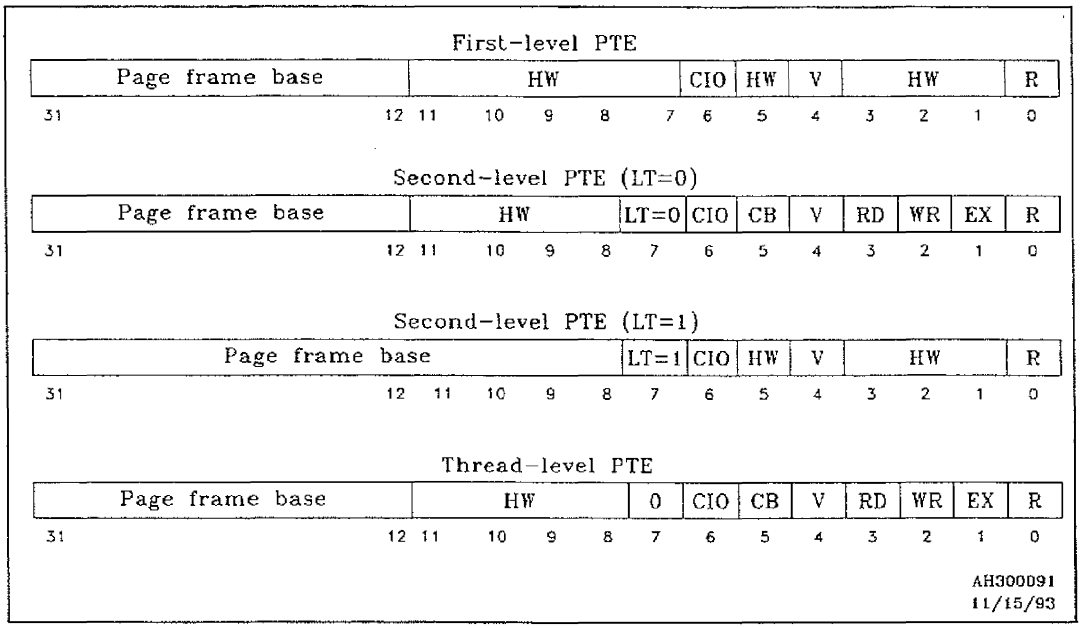
Bit<1>—Execute access (EX)
This bit indicates the validity of an execute access to the referenced page frame by branching or jumping to an instruction, or sequentially executing instructions (that is, "falling" across a page boundary). When this bit is cleared, it indicates that no execute access is permitted. When this bit is set, it indicates that an execute access is permitted to the referenced page frame. If an execute access is attempted, and bit <1> is 0, a system exception is signaled and an error code is loaded into address register A5.

Bit<0>—Resident page (R)
When this bit is set, it indicates the presence of the referenced page frame in the physical address space of the process. When this bit is cleared, it indicates the absence of the referenced page frame in physical memory. In this case, a page fault occurs and causes a system exception. If the referenced page frame is present, bits <31..12> are used as the page frame address of the referenced page frame. Bit <0> is interpreted for valid PTEs only.

PTE format - C4600

The C4600 series CPUs use the same two/three-level page table structure as the other multiprocessing (C3200/C3400/C3800 Series) CPUs. However, the format for the PTE uses different bit assignments, mainly additional HW reserved bits. Figure 37 shows the format of a resident PTE for the C4600 Series CPUs.

Figure 37 Resident PTE format—C4600 Series CPUs



The following subsections describe the meaning of the bits in Figure 37. All bits labeled "HW" are hardware reserved bits and are reserved for future use. The OS should avoid using these bits.

Bits<31..12> or bits <31..8>—Page frame base (PFB)
If a valid reference to a resident shared page occurs, bits <31..12> become the most-significant 20 bits of a 32-bit physical byte address. The page frame base is modulo 4,096 bytes. For unshared pages, bits <31..8> become the most-significant 24 bits of a 32-bit physical address of the thread-level PTE.

Bit<7>—Level 3 (LT)/thread-level
For second-level PTEs, this bit determines whether the data page is shared (zero) or unshared (one).

Bit<6>—Channel I/O (CIO)
This bit indicates that the target page frame is valid for I/O references from a channel processor.

Bit 5—Cache Bypass
For the second-level PTE, this bit indicates that the page frame base points to a page that must not be encached in the processor's data cache. When this bit is set, data loaded from or stored into this page is not accelerated into the CPU's data cache.

Bit<4>—Valid PTE (V)
This bit indicates the validity of the PTE. A 0 indicates an invalid PTE. When this bit is set, it indicates a valid PTE. A segment out-of-bounds error is generated when an invalid PTE is accessed. A reference to an invalid PTE results in a system exception.

Bit<3>—Read access (RD)
This bit indicates the validity of a read access to the referenced page frame. When this bit is cleared, it indicates that no read access is permitted. When this bit is set, it indicates that a read access is permitted to the referenced page frame. If a read access is attempted and bit <3> is 0, a system exception is generated, and an error code is loaded into address register A5.

Bit<2>—Write access (WR)
This bit indicates the validity of a write access to the referenced page frame. When this bit is cleared, it indicates that no write access is permitted. When this bit is set, it indicates that a write access is permitted to the referenced page frame. If a write access is attempted, and bit <2> is 0, a system exception is generated, and an error code is loaded into address register A5.

Bit<1>—Execute access (EX)
This bit indicates the validity of an execute access to the referenced page frame by branching or jumping to an instruction, or sequentially executing instructions (that is, "falling" across a page boundary). When this bit is cleared, it indicates that no execute access is permitted. When this bit is set, it indicates that an execute access is permitted to the referenced page frame. If an execute access is attempted and bit <1> is 0, a system exception is generated and an error code is loaded into address register A5.

Bit<0>—Resident page (R)
When this bit is set, it indicates the presence of the referenced page frame in the physical address space of the process. When this bit is cleared, it indicates the absence of the referenced page frame in physical memory. In this case, a page fault occurs and causes a system exception. If the referenced page frame is present, bits <31..12> are used as the page frame address of the referenced page frame. Bit <0> is interpreted for valid PTEs only.

Thread-level PTE Operation

The C3200/C3400/C3800/C4600 Series implementation supports hardware-implemented unshared memory by adding another level of page table entry (PTE) in the address translation process. The second-level PTE contains the base address of a third-level table of PTEs called the thread-level PTE. If the level 3 (LT) bit in the second-level PTE is set, the table of thread-level PTEs pointed to by the second-level PTE is traversed using the processor's TID to find the page frame base. The format of a valid thread-level PTE for the C3200/C3400/C3800 Series CPUs is shown in Figure 35. The format of a valid thread-level PTE for the C4600 Series CPU is shown in Figure 37.

The indexing into this table is based on TID. The extra translation is performed by the processor's PTE miss resolution mechanism. When the second-level PTE is returned from memory, the state of the LT bit is examined. If the LT bit is not set, the referenced page is shared memory, so the second-level PTE is encached in the ATU cache as the final-level PTE, since that particular second-level PTE includes the page frame base.

If LT is set, the page is unshared and the processor uses the page frame bits of the second-level PTE to request the thread-level PTE for the processor's current TID. The thread-level PTE is then encached in the ATU cache as the final-level PTE. In this manner, two processors executing as different TIDs in the same CIR (process) can use the same virtual address and still have unique physical memory.

Virtual-to-physical address translation

The virtual-to-physical address translation is performed by an address translation unit (ATU) implemented in hardware. The ATU accelerates the translation of virtual-to-physical addresses by way of maintaining an internal address cache of recently translated virtual addresses. Once a translation occurs, the translated addresses are placed in the address cache for the following reasons.

- The steps necessary to translate the virtual addresses require machine cycles that would otherwise be used to execute instructions.
- Programs exhibit temporal and spatial locality of reference. It is probable that once a virtual-to-physical translation is accomplished and encached, subsequent virtual addresses will reference the same page associated with the initial translation.

The ATU accelerates address translations by associating a virtual address with an ATU entry that contains the bottom-level PTE (PTE2 or PTET) associated with the virtual address. The high order bits of the virtual address are used to select the correct ATU entry. The ATU entry provides a convenient place to store these privileges. Some characteristics of the ATU are relevant to the system programmer because:

- The size and structure of the ATU are implementation-dependent.
- Individual entries within the ATU are not explicitly addressable.
- Modification of a PTE in memory does not necessarily have an immediate effect on ATU entries.
- There are several privileged instructions that permit a level of control over ATU address translation in a manner that is ATU implementation-independent. These instructions purge the entire ATU or selective entries. Purging an ATU simply involves marking all entries as invalid so that no encached translations exist. Purging the entire ATU is necessary for process multiplexing. Purging selective ATU entries is used when selective PTE modifications occur, for example, when an address translation fault finds the page frame in physical memory, but not in the physical space of the process.

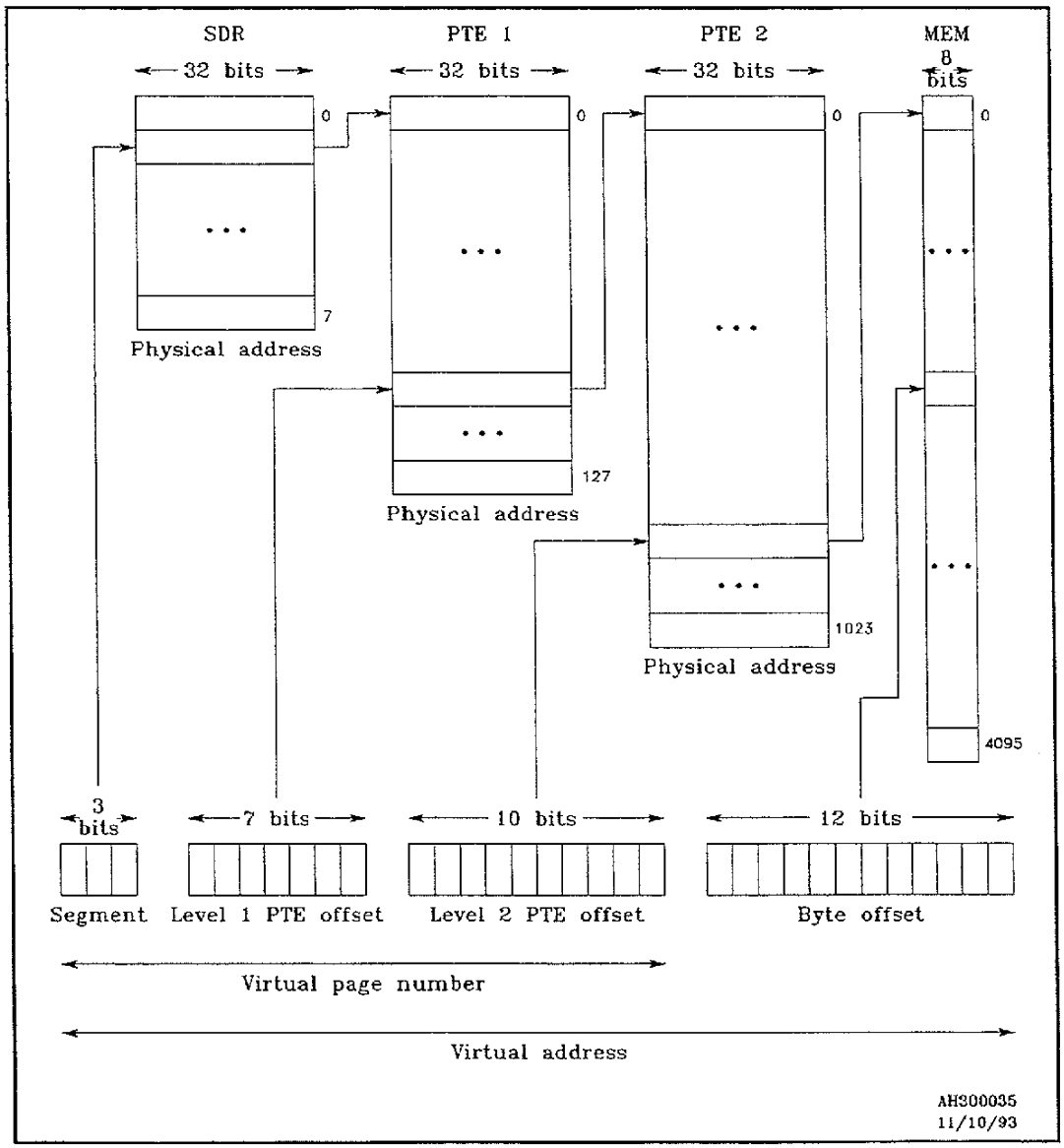
Virtual addresses can be identical in more than one process and do not translate to the same physical address. As a result of this

virtual address space structure, the ATU must be purged when a new user process is scheduled, has its context loaded, and begins execution.

C100

The virtual-to-physical address translation process interprets the structure of a 32-bit virtual address as shown in Figure 38.

Figure 38 Virtual-to-physical address translation—C100 Series CPUs



The following attributes of this address translation process should be noted:

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

C3200/C3400/C3800/C4600

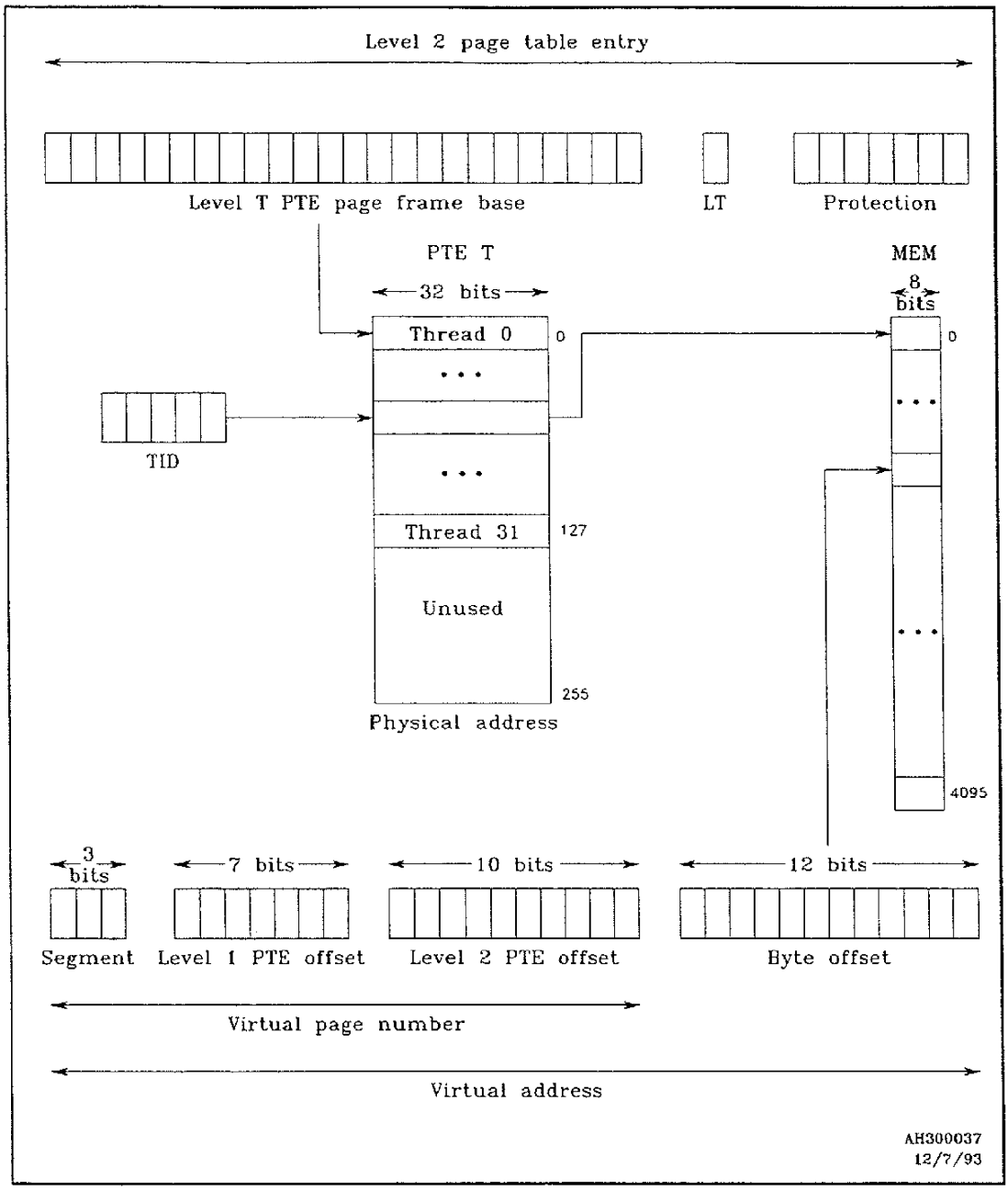
This section describes the virtual-to-physical translation process for multiprocessing C-Series CPUs. It also describes the additional properties of the ATU in multiprocessing C-Series CPUs, as well as additional system programming concerns.

This translation process applies to all multiprocessing C-Series implementations (C3200/C3400/C3800/C4600), even though the PTE bit positions are different for some of the implementations.

The address translation process is expanded to accommodate multiple threads in a process, including unshared and shared memory. For virtual-to-physical address translation purposes, the multiprocessing implementation treats the translation of a shared byte address in the same manner as the C100 Series implementation performs address translation.

Figure 39 shows the translation process from a level-2 PTE to a physical address for *unshared* pages.

Figure 39 Address translation step for unshared pages—multiprocessing C-Series CPUs



The following attributes of this address translation should be noted in addition to the notes presented in the previous section describing the translation process for the C100 Series CPUs:

- The page table referenced by the second or thread-level index may not be resident in physical memory. A page fault can occur when referencing a second or thread-level page table page.
- The access bits in the first-level PTE are never interpreted, that is, no protection access checks are performed when a first-level PTE references a second-level PTE. This is also true for second-level entries used to reference thread-level entries.
- The page frame base from the second-level entry accesses a table of 64 thread-level entries. The initial multiprocessing implementation supports only 32 threads (5-bit TID); half of the level T table is unused.

The ATU accelerates address translations by associating an ATU entry with a <virtual address, CIR, TID> tag. The ATU entry contains the following information.

- All significant bits of the PTE (PTE2 or PTET). This includes the page frame base plus all access control bits.
- The higher order bits of the virtual address encached. Since the page-offset field of the virtual address is not translated, this entry contains, at most, the most-significant 20 bits of the translated virtual address.
- The CIR and TID used to access the PTE.

The characteristics of the multiprocessing C-Series ATU that are relevant to the system programmer, in addition to the characteristics previously described for the C100 Series, are:

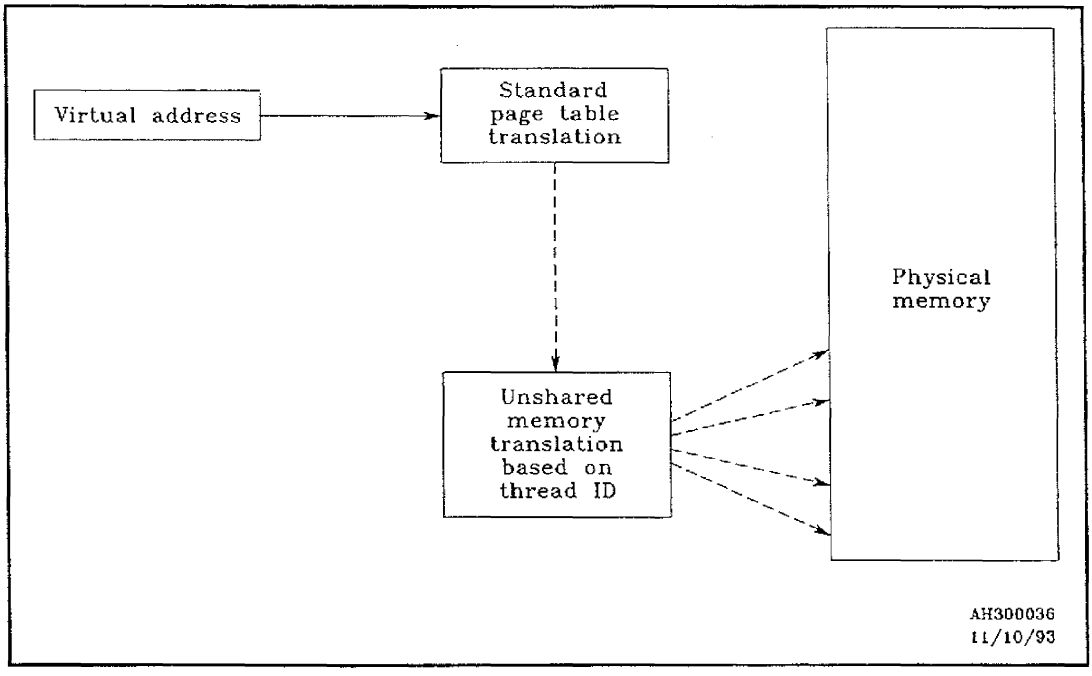
- Each CPU within the complex has an independent ATU cache.
- Several privileged mode instructions exist to permit a level of control over ATU address translation in a manner that is ATU implementation independent. These instructions purge the entire ATU in every CPU within the complex (`patu`), or a selective ATU entry (`pate Ak`). Purging the entire ATU in every CPU is necessary for process multiplexing between CPUs. Purging selective ATU entries within every CPU is necessary when selective PTE modifications occur within a single process. This purging will ensure all threads within the process that are executing concurrently on other CPUs acquire the new translation as soon as possible.

Unshared address translation allows the same virtual address to reference different physical memory locations. This capability permits parallel invocations of re-entrant code to retain separate values for local variables and their own stack. Implementation of unshared address translation for private data pages of threads is built into the virtual-to-physical address translation function. Virtual memory addresses that are thread-private are marked as such in the corresponding PTE.

During address translation, unshared memory addresses require an additional page table level to resolve the physical address, based not only on the virtual address but also on the thread ID (TID) of the thread that initiated the memory access. Page frames used to hold thread-private data pages are allocated dynamically by the hardware ASAP mechanism when a new thread is created. These pages come from a pool of free pages allocated when the system is booted. Memory management of unshared data pages is managed in hardware.

Figure 40 shows the relationship of the address translation of shared memory with the additional level of translation needed for unshared memory.

Figure 40 Virtual-to-physical address translation for unshared pages—multiprocessing C-Series CPUs



Translation of a virtual address to a physical address is usually a one-to-one function. However, implementing unshared memory for threads requires a one-to-many function that translates a virtual address to a physical address. A virtual address designated as unshared memory uses the requesting CPU's thread ID (TID) value when translated to a final physical address.

Referenced and modified bits

Each page frame is associated with a pair of status bits called the referenced and modified (R&M) bits. Their purpose is to dynamically track references to physical memory.

The referenced bit indicates whether a successful memory reference has been made to a page frame since the last time the bit was cleared.

The modified bit indicates whether a successful write was made. A write affects both the referenced and modified bits.

Implementation of the referenced and modified bits is machine-specific. Refer to the "Referenced and modified bits" section in Chapter 7, "Implementation-specific features," for a discussion of the machine-specific features of R&M bits.

Virtual memory protection

The *memory protection system* protects the virtual address space of a process, the operating system structures, and shared resources. The memory protection system is designed around the ring structure of the virtual address space. The functions of the memory protection system:

- Permit efficient implementations of virtual machine mechanisms
- Support the operating system located in user virtual address space
- Contain certain violations to a user's process that allow *only* the user's process to be modified

Two additional structures complete the basic structure of the memory protection system. These structures are the *access brackets* (for ring maximization) pertaining to the enforcement of the virtual address space ring structure and the *access field* (for access validity) contained within a page table entry (PTE).

Ring maximization

The access bracket structure directly implements *ring maximization*. Ring maximization means that the memory references are always at the access priority of the instruction.The memory reference compares the ring number in the access bracket (ring) field of the program counter, PC <31..29> (effective source), to the ring number of the referenced operand (effective target, if one exists) to determine the validity of the reference. In this ring mechanism, higher ring numbers have lower priority than rings with lower numbers.

Table 31 shows the valid virtual address references that satisfy the ring maximization.

Table 31 Ring maximization for source and target

	Effective Target				
Effective Source	**Ring 0**	**Ring 1**	**Ring 2**	**Ring 3**	**Ring 4**
Ring 0	Valid	Valid	Valid	Valid	Valid
Ring 1	*	Valid	Valid	Valid	Valid
Ring 2	*	*	Valid	Valid	Valid
Ring 3	*	*	*	Valid	Valid
Ring 4	*	*	*	*	Valid

The memory protection system uses two constructions called *effective source* and *effective target*, which have the following properties:

- If direct addressing is specified:
  - The effective source is the ring of the program counter (PC).
  - The effective target is the ring of the address of the referenced operand.
- If indirect addressing is specified:
  - The effective source is the ring of the program counter (PC).
  - The effective target is the ring of the effective address referenced by the indirect pointer.

A memory reference that satisfies ring maximization is a valid access with respect to memory protection. If an invalid reference is detected, a system exception occurs and an error code is loaded into address register A5.

Access Validation

After a memory reference has been found to satisfy ring maximization, the validity of the access is checked using the access protection bits in the corresponding PTE. If an invalid reference is detected, a system exception occurs and an error code is loaded into address register A5.

First, the valid bit of the PTE is examined. If this bit is set, the reference is valid. If this bit is cleared, the reference is invalid, a segment out-of-bounds error is detected, and a system exception occurs.

Next, the PTE's read, write, or execute bit is examined, depending on the type of memory reference. If the reference is a read, the read access bit of the valid PTE is examined. If it is set, the read is permitted. If it is cleared, the read is not permitted and a system exception occurs. If the reference is a write, the write access bit of the valid PTE is examined. If it is set, the write access is permitted. If it is cleared, the write is not permitted and a system exception occurs. If the reference is an instruction fetch, the execute access bit of the valid PTE is examined. If it is set, instructions can be fetched and executed from this page. If it is cleared, instruction execution is not permitted and a system exception occurs.

After all validity checks have passed, the resident bit in the PTE is checked to determine if the referenced page is currently resident in memory.

Memory protection notes

The following notes will help users take maximum advantage of the virtual memory protection mechanisms:

- Addresses relative to the PC are granted no special privileges. The appropriate read, write, and execute privileges apply, as previously specified.
- Access checking is only performed if the PTE associated with a virtual address is valid. If the PTE is invalid, the state of the read, write, execute, and resident bits is ignored.
- An access violation can be detected for nonresident pages.
- If an access privilege is changed for a process after that process has already established a context in the ATU, the ATU must be purged upon completion of the alteration. ATU entries are not altered automatically when a PTE is modified.
- If an instruction specifies an immediate operand, (for example, `add immediate`), the read access privilege of the page containing the immediate operand is not interpreted. It is treated as an execute access.
- A ring check is not performed for instructions that produce effective addresses but do not immediately use them. For example, if a load effective address instruction executed in ring 3 develops a ring 1 address, no ring violation occurs. If that ring 1 address is subsequently used by a ring 3 program to make an operand reference, a ring violation occurs.
- The intermediate addresses of all instructions that can make multiple memory references (for example, `vector load`) are always ring maximized with the current ring to determine the validity of the reference, that is, the address of each vector element is loaded.
- When indirection is specified, the page containing the indirect pointer must satisfy ring maximization and permit read access. This read access is independent of the instruction type, for example, `load`, `store`, or `jump`.
- I/O space operands must be addressed as single bytes or halfwords. Refer to Chapter 7, "Implementation-specific features," for the sizes of operands when accessing registers located in I/O address space. A protection violation occurs if a valid I/O reference is made using a nonvalid operand.

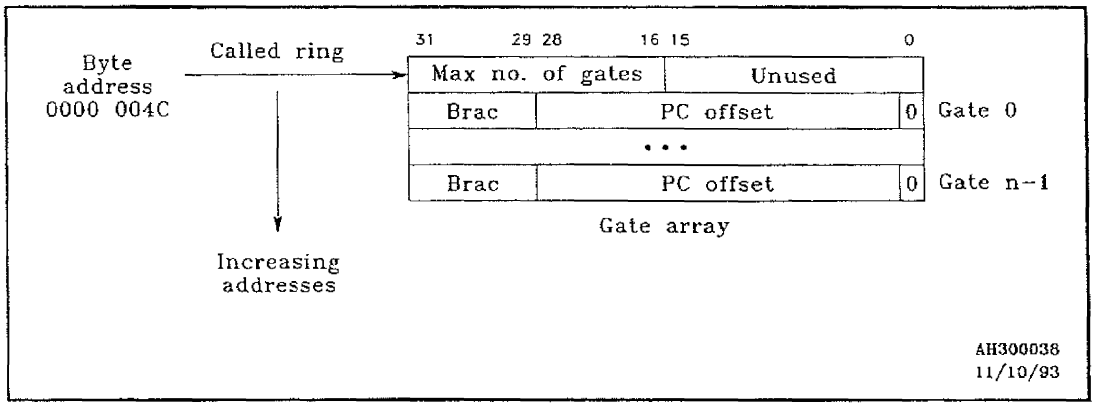
Inter-ring procedure call and return

Ring crossing can only occur as a result of an explicit attempt by a program control instruction to cross rings, or by a system exception. The explicit program control instruction is a system call (`sysc`) instruction, a return instruction (`rtn` with PSW(FRL) = 01, extended return block), or the privileged instruction (`rtnc`, return from context block). All other program control instructions stay within the current ring of execution, that is, the ring of the program counter. The appropriate higher order bits of the target effective address are essentially ignored.

The direction of a system call is inward, toward ring 0. Outward calls are trapped as ring violations. The direction of all system returns is outward, away from ring 0. Inward returns are trapped as system exceptions only when the return block is an extended return block. Short and long subroutine calls always return within the same ring.

The immediate field of the system call instruction is interpreted as an index into a table within the called ring. This index is referred to as a *gate number*. The table contained within the called ring is referred to as a *gate array*. The base of the gate array is pointed to by the segment entry point contained in byte address `0000 004C` of page 0 of the called ring. Figure 41 shows the gate array structure.

Figure 41 Gate array structure



Inward ring crossings function in the following manner:

1. The *gate index field* (G field) of the `sysc` instruction indicates the desired entry point. See the `sysc` instruction description in *CONVEX Assembly Language Reference Manual (C Series)*, "Instruction set," for a more detailed description of `sysc`.

2. This G field is compared with the Max number of gates (bits <31..16>) in the first word of the gate array pointed to by the segment entry point contained in address 0000 004C, page 0 of the target ring.

   If the G field is greater than or equal to bits <31..16>, a ring violation occurs and the ring crossing does not occur (the gate is not defined).

3. If the G field is less than bits <31..16>, the ring number of the segment containing the `sysc` instruction (current ring) is compared with the bracket field (Brac), bits <31..29> of the referenced *gate*.

   If the current ring is greater than the bracket field, the PC is not loaded, the ring crossing does not take place, and a system exception occurs.

4. If the current ring is less than or equal to the bracket field, then bits <28..1> of the gate are loaded into the PC, and bits <31..29> of the PC are loaded with the target ring.

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

This mechanism permits individual segments to have entry points with unique gate brackets. Thus, a particular operating system call can be restricted to a particular ring of origin. These actions are performed by the CPU. There is no software overhead or operating system kernel involvement unless an explicit kernel call is made.

Corrupted pointers

Corrupted pointers can occur on system calls when a passed pointer references the operating system's data space. The system process invoked as part of the inward ring call uses a passed pointer as part of system call processing. The system process expects these pointers to reference the virtual address space of the caller, that is, the ring of the user executing the `sysc` instruction. If a passed pointer references the system process data space, unexpected (and usually undetected) disasters can occur. The following facilities are provided to prevent such problems:

- An instruction that checks to see that the ring maximization is satisfied for passed pointers (`compare immediate`)
- A `load physical` instruction to obtain the access bits of appropriate PTEs
- Instructions that access data backwards (decreasing virtual memory) to always perform the ring maximization dynamically, ensuring that a corrupted pointer is not created

All of these actions can occur outside the operating system (OS) kernel. One of the objectives of the memory management and protection structures is to reduce the size of the OS kernel. Since the OS kernel is smaller, it is more reliable and secure. In addition, virtual machine structures are easier to construct.

Generally, there is no algorithm to guarantee that corrupted pointers do not occur. To minimize corrupted pointers, copy arguments into the called ring's virtual address space, then initiate corrupted pointer checking.

Reserved virtual memory

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

Note

The virtual address `0x0000 0000` is reserved and may not be used by software. In addition, the virtual address `0xn000 0000` is a reserved location in virtual page 0 of each ring.

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

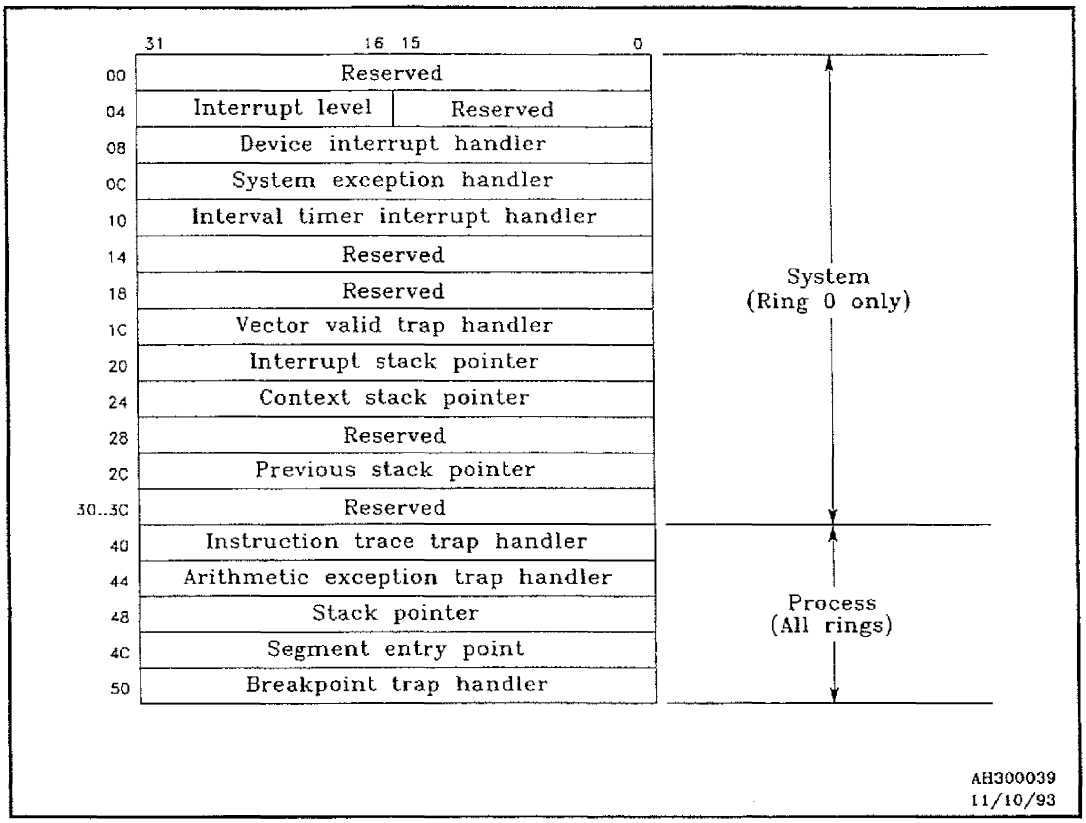
Page 0 is used in one of two ways, depending on the classification of the exception (trap or fault) that has occurred. The two types of exceptions that access page 0 are *process exceptions* and *system exceptions*. Interrupts also access page 0.

Page 0 - C100

Figure 42 shows the C100 Series virtual memory organization of page 0. Refer to Chapter 6, "Exceptions and interrupts," for some operational definitions of the page 0 locations.

EDTX 2-04-CV-120
51056DOC065452

Figure 42 Page 0 virtual memory organization—C100 Series CPUs



The entries from 0000 0000 to 0000 003C are valid for ring 0 only. In all other rings, these entries are reserved. All other entries are valid for all rings.

Each entry in Figure 42 is defined in ascending order according to the byte address offset associated with each entry. The high order nibble of several addresses in Figure 42 is marked x in order to denote that these byte addresses are applicable to any virtual memory page 0 regardless of the ring indicated in the segment field of the program counter.

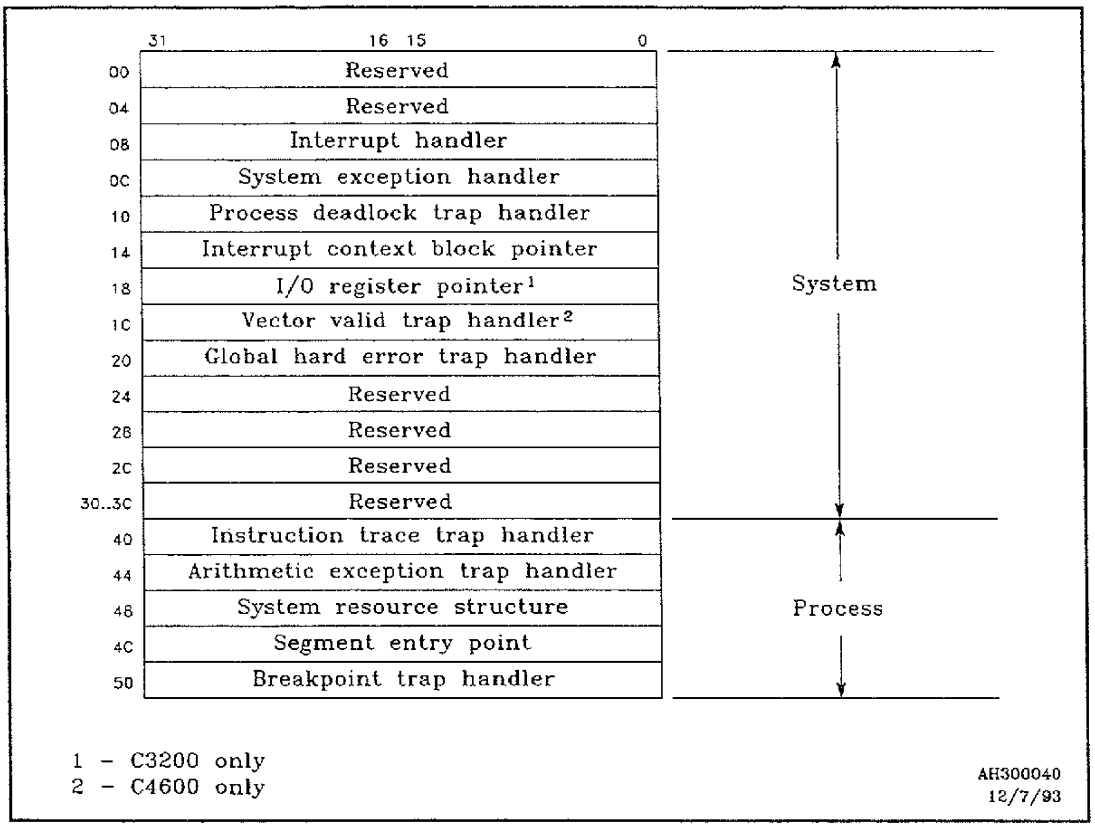
0000 0000	**Reserved**—Should not be used by software.
0000 0004	**Interrupt level**—A 16-bit memory-based counter that indicates the number of nested interrupts currently being processed. If the interrupt level is 0, no interrupts are being processed. If the interrupt level is not 0, interrupts are being processed and the ring 0 stack is the interrupt stack.
0000 0008	**Device interrupt handler**—A byte pointer to the handler for device interrupts other than the interval timer.
0000 000C	**System exception handler**—A byte pointer to a system exception handler. The exceptions that transfer control to this system exception handler are: error exit trap, undefined op code trap, ring violation, PTE violation, and nonresident page.
0000 0010	**Interval timer interrupt handler**—A byte pointer to the interrupt handler that responds to an interval timer interrupt.
0000 0014	**Reserved**—Should not be used by software.
0000 0018	**Reserved**—Should not be used by software.
0000 001C	**Vector valid trap handler**—A byte pointer to a trap handler that responds to the vector valid trap. A vector valid trap occurs if an attempt to execute a vector instruction occurs and the vector valid bit is 0. A vector instruction is an instruction that manipulates the V, VL, VS, or VM registers.
0000 0020	**Interrupt stack pointer**—A byte pointer that specifies the stack to be used when an interrupt occurs.
0000 0024	**Context stack pointer**—A byte pointer that specifies the stack to be used when a system exception occurs.
0000 0028	**Reserved**—Should not be used by software.

0000 002C	**Previous stack pointer**—A save area used for interrupt processing. When an interrupt first occurs and the ring 0 stack is initialized to the value of the interrupt stack pointer, the previous stack pointer is saved in byte address 0000 002C. This ensures that there is a proper linkage for stack switching in ring 0 for interrupt processing.
0000 0030	**Reserved**—Should not be used by software.
0000 0034	**Reserved**—Should not be used by software.
0000 0038	**Reserved**—Should not be used by software.
0000 003C	**Reserved**—Should not be used by software.
x000 0040	**Instruction trace trap handler**—A byte pointer to the handler that responds to an instruction trace trap. Refer to the "Instruction trace trap" section of Chapter 6..
x000 0044	**Arithmetic exception trap handler**—A byte pointer to the handler that responds to an arithmetic exception. The PSW contains bits that indicate the type of arithmetic exception(s) that occurred.
x000 0048	**Stack pointer**—A save area that maintains the stack pointer for cross ring call processing.
x000 004C	**Segment entry point**—A byte pointer to the base of the gate array defined in the called ring. Each ring has a unique entry point and associated gate array.
x000 0050	**Breakpoint trap handler**—A byte pointer to the handler that is executed when the bkpt instruction is executed. Refer to the "Breakpoints" section of Chapter 6..
x000 0054– x000 0FFF	**Reserved**—Available for use by software.

Page 0 - C3200/C3400/C3800/C4600

Figure 43 shows the virtual memory organization of page 0 that is reserved for the multiprocessing implementation. Refer to the "Interrupt context blocks" section in Chapter 6, "Exceptions and interrupts," for some operational definitions of the page 0 locations.

Figure 43 Page 0 virtual memory organization—multiprocessing C-Series CPUs



The entries from 0000 0000 to 0000 003C are valid for ring 0 only. In all other rings, these entries are reserved. All other entries are valid for all rings.

Each entry in Figure 43 is defined in ascending order according to the byte address offset associated with each entry. The high-order nibble of several addresses in Figure 43 is marked x to denote that these byte addresses are applicable to any virtual

memory page 0, regardless of the ring indicated in the segment field of the program counter.

0000 0000	**Reserved**—Should not be used by software.
0000 0004	**Reserved** —Should not be used by software.
0000 0008	**Interrupt handler**—A byte pointer to the handler for all interrupts including the interval timer.
0000 000C	**System exception handler**—A byte pointer to a system exception handler. The exceptions that transfer control to this system exception handler are: error exit trap, undefined op code trap, ring violation, PTE violation, and nonresident page.
0000 0010	**Process deadlock trap handler**—A byte pointer to a trap handler that is called whenever a process deadlock occurs.
0000 0014	**Interrupt context block pointer** — A pointer to a set of interrupt context blocks that contain all of the CPU specific context required for interrupt processing. Refer to the "Interrupt processing - C100" section in Chapter 6.
0000 0018	**I/O register pointer (C3200 Series only)**—A virtual address that is mapped to the timer registers located within I/O address space. Timer registers are described in the "Timers" section in Chapter 7.
0000 001C	**Vector valid trap handler**—A byte pointer to a trap handler that responds to the vector valid trap. A vector valid trap occurs if an attempt to execute a vector instruction occurs and the vector valid bit is 0. A vector instruction is an instruction that manipulates the V, VL, VS, or VM registers.
0000 0020	**Global hard error trap handler (C4600 only)**—A byte pointer to a trap handler that is called whenever a hard error on any CPU occurs. Refer to Chapter 6, "Exceptions and interrupts," for more on the global hard error trap error.
0000 0024	**Reserved**—Should not be used by software.

0000 0028	**Reserved**—Should not be used by software.
0000 002C	**Reserved**—Should not be used by software.
0000 0030	**Reserved**—Should not be used by software.
0000 0034	**Reserved**—Should not be used by software.
0000 0038	**Reserved**—Should not be used by software.
0000 003C	**Reserved**—Should not be used by software.
x000 0040	**Instruction trace trap handler**—A byte pointer to the handler that responds to an instruction trace trap. Refer to the "Instruction trace trap" section of Chapter 6.
x000 0044	**Arithmetic exception trap handler**—A byte pointer to the handler that responds to an arithmetic exception. The PSW contains bits that indicate the type of arithmetic exception(s) that occurred.
x000 0048	**System resource structure**—The virtual address of a communication register that contains a pointer to a list of available stack pointers. Refer to the discussion of system resource structures for the multiprocessing implementation in this chapter for more information.
x000 004C	**Segment entry point**—A byte pointer to the base of the gate array defined in the called ring. Each ring has a unique entry point and associated gate array.
x000 0050	**Breakpoint trap handler**—A byte pointer to the handler that is executed when the `bkpt` instruction is executed. Refer to the "Breakpoints" section of Chapter 6.
x000 0054– x000 0FFF	**Reserved**—Available for use by software.

Power up addressing mode

Physical addresses are normally generated by virtual-to-physical address translation. The exception to this process occurs during the bootstrap process at powerup. Prior to cold start, the service processor unit (SPU) must create a bootstrap page table entry (PTE) structure. The PTE structure is used by the CPU's cold start microcode to make the address translation mechanism operational.

Multiprocessor management

5

The C3200, C3400, C3800, and C4600 Series systems implement a multiprocessing architecture. These systems support a multiple instruction stream/multiple data stream (MIMD) parallel architecture in which each CPU operates as an independent 64-bit supercomputer.

The automatic self-allocating processors (ASAP) multiprocessor management scheme binds the CPUs of a CONVEX complex into a tightly coupled, shared memory set. With ASAP, the CPUs function independently by automatically allocating themselves.

This scheme also provides a simple and flexible set of instructions for dynamic CPU allocation, deallocation, and communication. It allows the user to exploit software parallelism by allocating multiple threads of execution within a single process. The operating system provides simultaneous execution of multiple processes for system load balancing and timesharing.

The C-Series architecture defines and supports the following:

- **Complex**—The entire set of one or more physical CPUs in a configuration
- **Subcomplex**—Any subset of a complex
- **Process**—A collection of one or more threads executing within a single virtual address space
- **Thread**—Any single instruction stream executing within a process
- **Multiprocessing**—The creation and scheduling of processes on any subcomplex

The C-Series multiprocessor management architecture

- Provides a set of instructions for thread creation, which requires no knowledge of the physical CPU configuration. These instructions implement fast thread creation and termination functions so users can take advantage of small regions of parallelism within programs without involving the operating system.
- Optimizes CPU execution cycles. The instruction set allows each CPU never to wait for another CPU to become available.
- Supports any number of CPUs within a complex for configuration, software compatibility, performance, and future expansion.

Tightly-coupled symmetric multiprocessing

The C-Series architecture provides a tightly-coupled, symmetric multiprocessor system. All CPUs within a complex share the same physical address space, and are equivalent in design and function.

The instruction set supports parallel execution but does not require it. Also, parallel execution is independent of the number of CPUs available. Therefore, the instruction set supports writing software for the C-Series architecture that is independent of the number of CPUs.

The C-Series architecture does not guarantee that CPUs will be allocated to a process when requested. CPUs will join a process if they are currently idle. However, OS calls exist to guarantee that all CPUs not currently executing the process are made idle whenever that process is scheduled to execute, thereby guaranteeing parallel execution when requested.

Automatic self-allocating processors

The automatic self-allocating processors (ASAP) mechanism assigns CPUs to processes and threads using both hardware and software during process execution. The multiprocessing C-Series architecture implements ASAP as a special case of dynamic scheduling through distributed control. The CPUs determine which processes or threads to execute next. User software also has some degree of resource control, requesting and releasing CPU resources using the CPU control (forking) instructions.

The ASAP mechanism switches a CPU from one process context to another without operating system knowledge or intervention so that both serial and parallel processes may execute simultaneously. Parallel processes may use multiple CPUs within parallel code regions without idling the CPUs during serial code regions. For example, a process could be executing on an entire complex, with one thread executing on each CPU.

The fundamental principle of operation for multiprocessing is that each CPU within the complex is solely responsible for scheduling itself, that is, associating and disassociating itself from an executing process. A master process that finds idle CPUs and schedules processes or threads on them does not exist. Each thread posts the need for another CPU to join in its computation, or in the case of the operating system, switch from one process context to another.

Communication registers

The multiprocessing C-Series architecture provides a set of hardware structures called *communication registers* (CMR) to assist in the multithreaded execution of a process across multiple CPUs in the complex. The communication registers are a single, globally shared, special-purpose register set used by the entire complex for communication between threads. The register set can be accessed equally by all CPUs in the complex. Threads communicate by sending and receiving data through these communication registers using communication register instructions.

The communication registers are a form of semaphored memory, available in much smaller quantities than virtual memory. One of the primary functions of communication registers is giving software a means to relocate frequently accessed data from virtual memory into a semaphored location.

Memory duals of the communication instructions can relocate data from the communication registers by performing primitive functions that are analogous to functions that manipulate communication registers. Using virtual memory, software can use these memory duals to create critical data structures in memory, then relocate these dual structures to a communication register set. These data structures and related operations on them are described in more detail in the "Process structures" section on page 74.

A communication register is visible from software as a word- or longword-addressable register with an associated hardware-maintained lock bit. The lock bit is used by both software and hardware as a binary semaphore on the contents of the register, and manipulated by the communication register instruction set to control and synchronize access by multiple CPUs to each communication register. The data portion may also be manipulated by some instructions that do not examine the state of the lock bit.

The C3200 Series hardware implements the communication registers as 1,024 64-bit longwords, with associated lock bits, modified bits, and parity. This provides eight communication register sets of 128 registers each.

The C3400/C3800/C4600 Series hardware implements the communication registers as 4,096 64-bit longwords, with associated lock bits and parity. This provides 32 communication register sets of 128 longword registers each. The C3400/C3800/C4600 Series CPUs do not implement the modified bits.

Communication index register

The *communication index register* (CIR) is the focal point of a ConvexOS process. Each CPU in a complex has one CIR. The CIR, which is not accessible by a user process, defines the address mapping between a particular physical communication register subset (partition) and the communication register addresses generated by a process. The CIR points to a partition of communication registers that is located within the communication register virtual address space. Only a single partition of the communication registers is visible to a user process at a time.

The value contained in the CIR is the *CIR index*. The CIR index points to one of *n* identical partitions of the communication register set (*n* equals 8 for C3200 Series complexes, and *n* equals 32 for C3400/C3800/C4600 Series complexes), each of which contains the current state of a single process. Since the CIR defines a UNIX operating system process, the communication register partition associated with that CIR index becomes a part of the process state.

Except for a special physical addressing scheme that is independent of the CIR, the CIR restricts each processor to one partition of the communication registers. The CIR index translates to a base address of a unique region of communication registers.

A CIR index is a virtual process identifier managed by the CPU, under the direction of either the operating system or automatically via the ASAP mechanism. Each executing process is associated with its own CIR index. When the CIR index changes, the entire process context changes.

A process is *mounted* on the CPU complex when its current state is represented by a partition of the communication register set. Any CPU in the complex can execute any mounted process by changing the index value in its CIR to reference the partition describing the process. This action *binds* a communication register set to a CPU when a CPU mounts and begins executing a thread. When multiple CPUs in the complex are executing multiple threads of a parallel code region of a process, these threads must perform atomic operations on this register partition to communicate and synchronize with each other.

Both the operating system and the hardware ASAP mechanism may access the communication registers either *virtually*, that is, under control of the current CIR index, or *physically*, that is, independent of the CIR. The virtual and physical method of CIR

access is described in the "Communication register address translation" section on page 148.

The SDRs, resource structures, and other frequently accessed data structures are stored in the communication registers. With respect to communication registers, the CIR performs a function similar to the function of the SDRs and PTEs in the virtual memory management scheme. This function of the CIR defines which SDRs are in use by a process, since the SDRs reside in the communication registers.

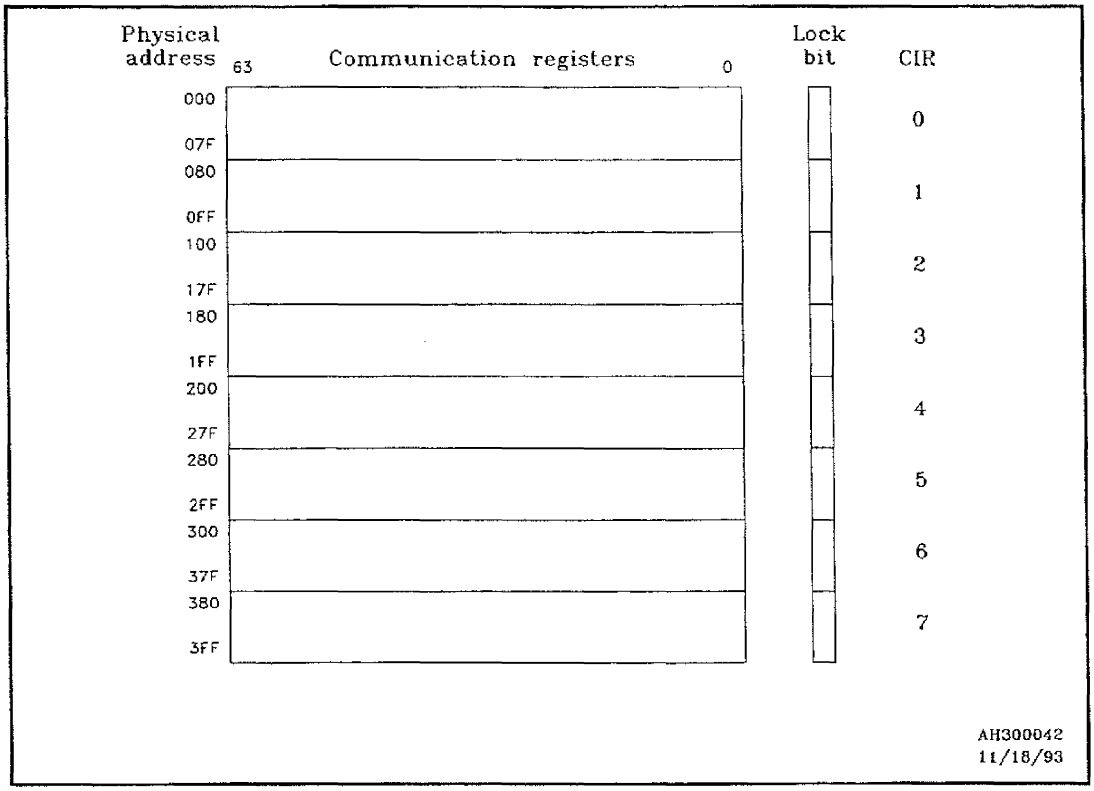
Within any multiprocessing C-Series architecture, at least twice as many physical communication register sets as CPUs exist. This number of register sets permits one communication register partition (pointed to by a CIR index) to be mounted and executing, and another communication register partition to be loading, so the operating system can quickly reschedule a thread by changing the CIR index.

CIR - C3200

In a C3200 Series complex, the CIR is a 3-bit register. Therefore, a CIR may contain any one of eight different index values from 0-7. A 3-bit CIR does not limit a C3200 Series complex to managing and executing a total of only eight processes. Rather, a maximum of eight processes can be mounted at any given time.

Figure 44 shows the division of the communication registers for the C3200 Series.

Figure 44 Communication register partitions by CIR index—C3200 Series CPUs

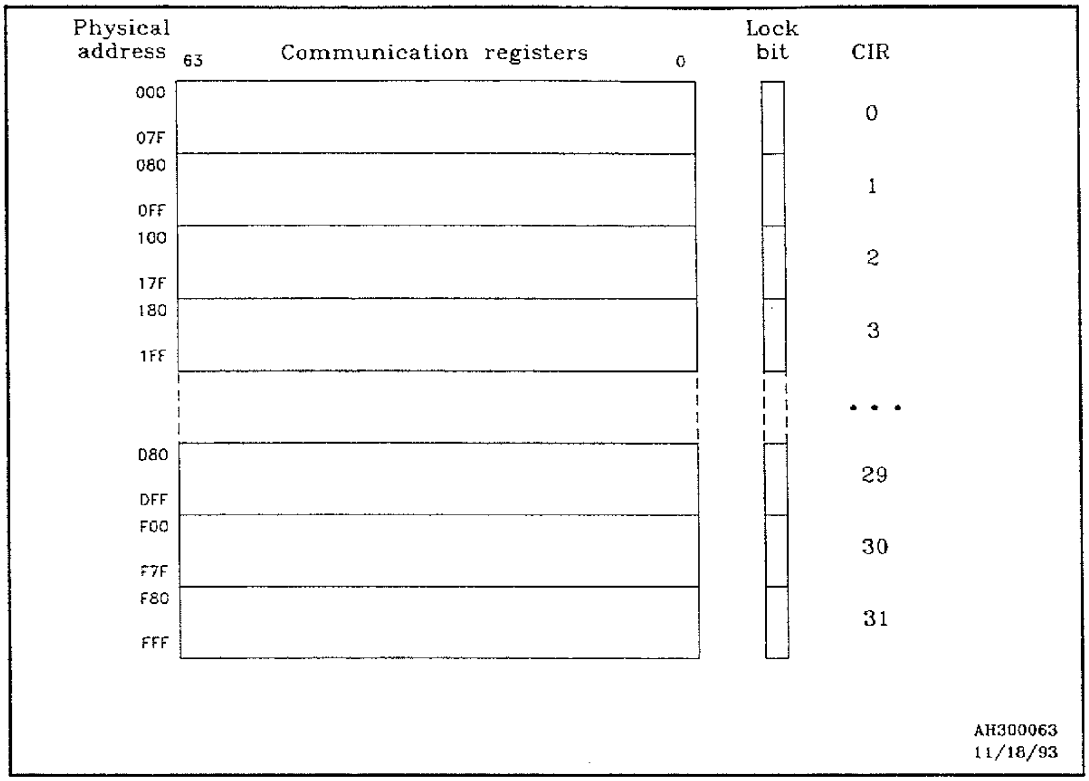


CIR - C3400/C3800/C4600

In the C3400/C3800/C4600 Series complexes, the CIR is a 5-bit register. A CIR may therefore contain any one of 32 different index values from 0-31. A 5-bit CIR index does not limit these complexes to managing and executing a total of only 32 processes. Rather, a maximum of 32 processes can be mounted at any given time.

Figure 45 shows the division of the communication registers for the C3400/C3800/C4600 Series CPUs.

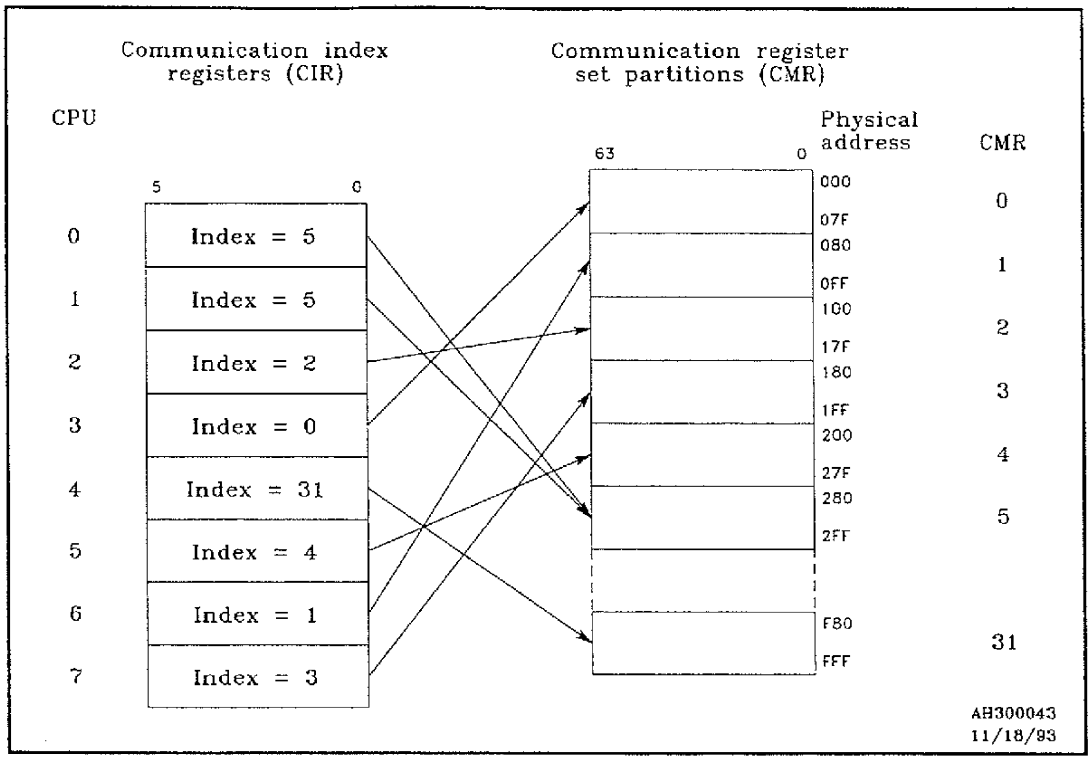
Figure 45 Communication register partitions by CIR index—C3400/C3800/C4600 Series CPUs



Two or more CPUs can execute threads of a process by binding the same communication register partition. Conversely, multiple CPUs are bound to the same communication register partition when each CPU loads its CIR with the same index value.

Figure 46 shows the register binding relationships of the CIR index and the communication register partitions for a C3400/C3800 Series system. Since C3200 Series CPUs only have 8 CIRs, and the C3200/C4600 Series systems have only four CPUs, the binding for those CPUs is a subset of this relationship.

Figure 46 Binding a communication register set to a CPU



Communication register virtual addressing

Communication registers are addressed with a 16-bit virtual address supplied in the instruction, the communication register effective address (*Ceffa*). This virtual address is mapped, using the CPU's CIR, to a physical communication register address. This physical address defines a unique 64-bit communication register.

Each communication register has an associated lock bit that is maintained by hardware or independently manipulated with communication register lock instructions. The lockbit allows semaphore operations on communication registers. The logical interpretation of the physical sense (1 or 0) of the lock bit depends on the operation involved.

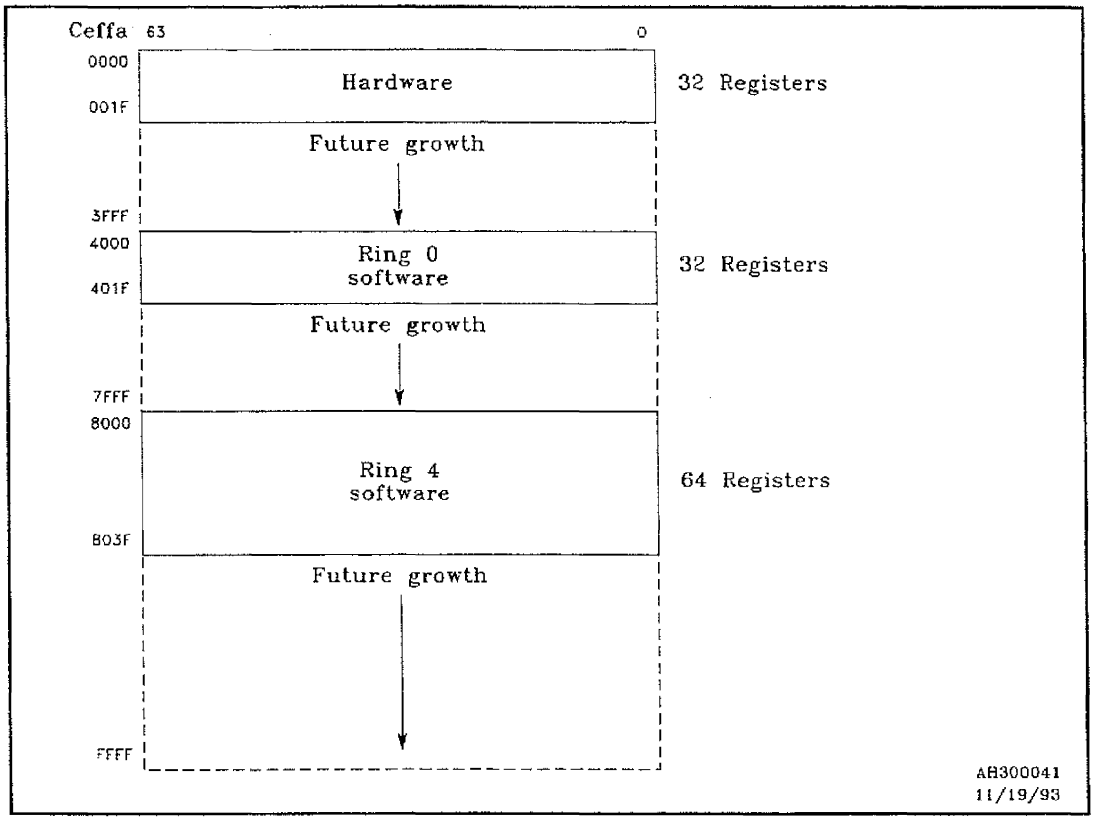
The *Ceffa* is developed in the same way as a memory address, using any of the addressing modes described in the "Addressing modes" section on page 73. The 16 least-significant bits of the developed address become the *Ceffa*, while the 16 most-significant bits are ignored. A communication register address may be generated using a 32-bit immediate, but code that always uses 16-bit immediates operates identically to code that uses 32-bit immediates and is more efficient.

A communication register set is divided into three distinct regions within the communication register virtual address space. One region of the address space is allocated to Ring 4, another to ring 0, and the last to hardware.

Virtual memory addresses have five rings: 0, 1, 2, 3, and 4. Communication register virtual addressing has only two distinct rings: ring 4 and ring 0. A process executing in rings 1, 2, 3, or 4 only has access to ring 4 communication registers. There are 32 registers each in the hardware and ring 0 regions, and 64 registers in the ring 4 region.

Figure 47 shows the ring partitioning of the communication register virtual address space.

Figure 47 Communication register virtual address space



The hardware-specific registers are allocated from address 0000 towards 3FFF, and the registers for ring 0 software are always allocated from 4000 towards 7FFF. This convention ensures that for any given hardware implementation of the C-Series architecture, the hardware register requirements do not conflict with the software register requirements. The ring 4 communication registers are located from 8000 to FFFF, which ensures sufficient space is available for additional user communication registers.

Communication register addressing is governed by a protection scheme similar to that of memory addressing. All communication register virtual addresses generated explicitly by instructions are checked by the hardware. If an address is out of range, a system exception occurs.

An `invalid communication address` system exception is generated if a process executing in rings 1, 2, 3, or 4 violates this ring protection. For example, if a user (ring 4) process (thread) references the communication addresses `0000` or `4000`, the thread will cause a system exception.

The multiprocessing C-Series hardware does not define the virtual address ranges `0020` to `3FFF`, `4020` to `7FFF`, and `8040` to `FFFF`. Attempting to access these address ranges also causes an invalid communication address trap. There are two exceptions: the C3200 Series CPUs use the address range `3C00` to `3FFF` and the C3400/C3800/C4600 Series CPUs use the addresses range `3000` to `3FFF` to implement the special physical addressing scheme. See the next section.

Refer to the communication register instruction definitions and related material in the *CONVEX Assembly Language Reference Manual (C Series)* for more information.

Communication register physical addressing

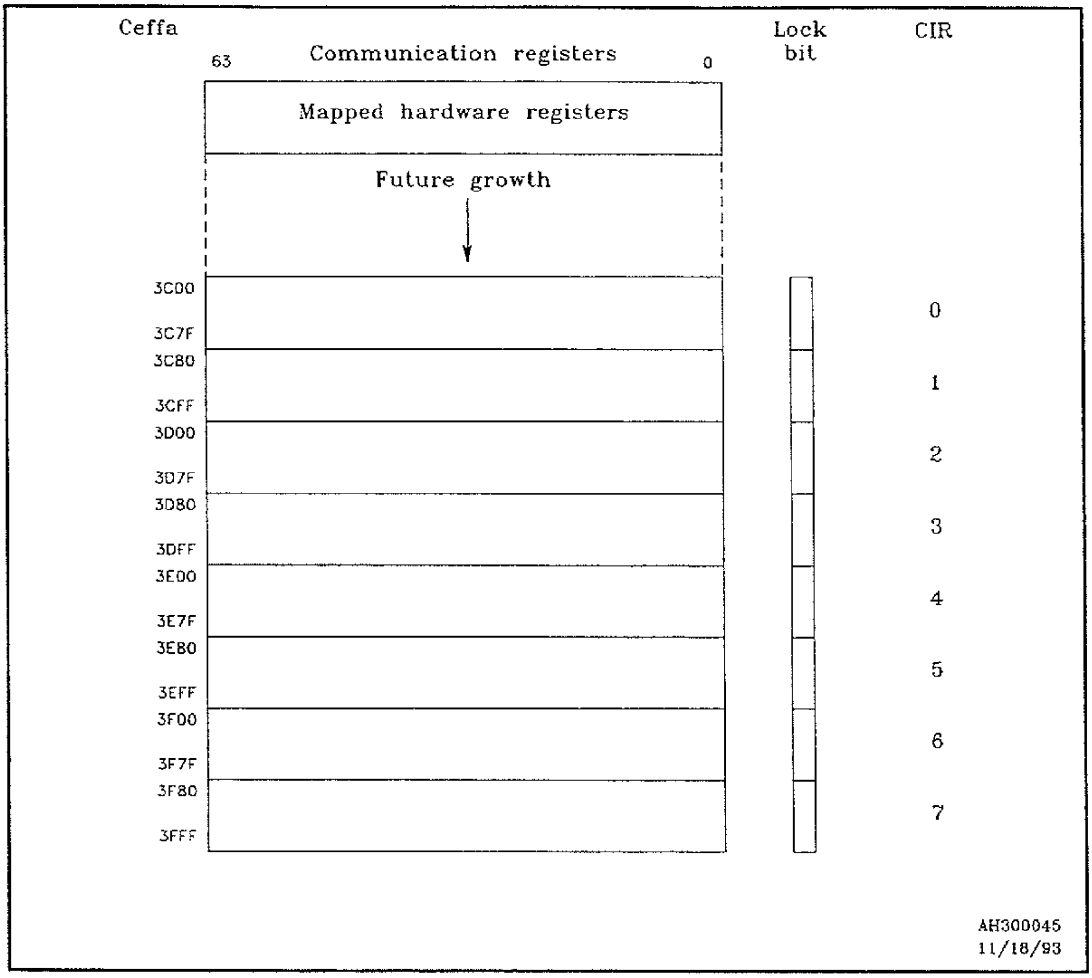
Although each CPU has direct access to a portion of the communication registers based on the CPU's current CIR index, hardware and ring 0 software can also access communication registers in partitions other than their own via an additional special address mapping called *communication physical addressing*. This type of physical addressing should not be confused with the physical address that results from the virtual-to-physical memory address translation process described in Chapter 4.

Each physical communication register set is accessible through the communication register virtual address space from ring 0, regardless of the current communication register set mapping. Physical addressing is accomplished by defining a fixed virtual-to-physical translation for a portion of the virtual address space assigned to the hardware communication registers. Figure 48 shows the fixed virtual-to-physical mapping of all register sets within the hardware virtual address space for C3200 Series CPUs. Figure 49 shows the mapping for C3400/C3800/C4600 Series CPUs. This mapping allows ring 0 software to access all communication register sets, regardless of which communication register set is currently bound to the CPU through its CIR index.

Therefore, each communication register is addressable with two communication addresses. One is CIR-based (virtual) and one is CIR-independent (physical). The C-Series architecture sets aside a range of virtual communication register addresses spanning enough address space to provide a second one-to-one address mapping. Since this one-to-one address mapping is independent of the CIR, this range of addresses may be physically addressed by ring 0 software. In addition, this second mapping always ends at virtual address `3FFF`, which places it in the (privileged) ring 0 communication address space.

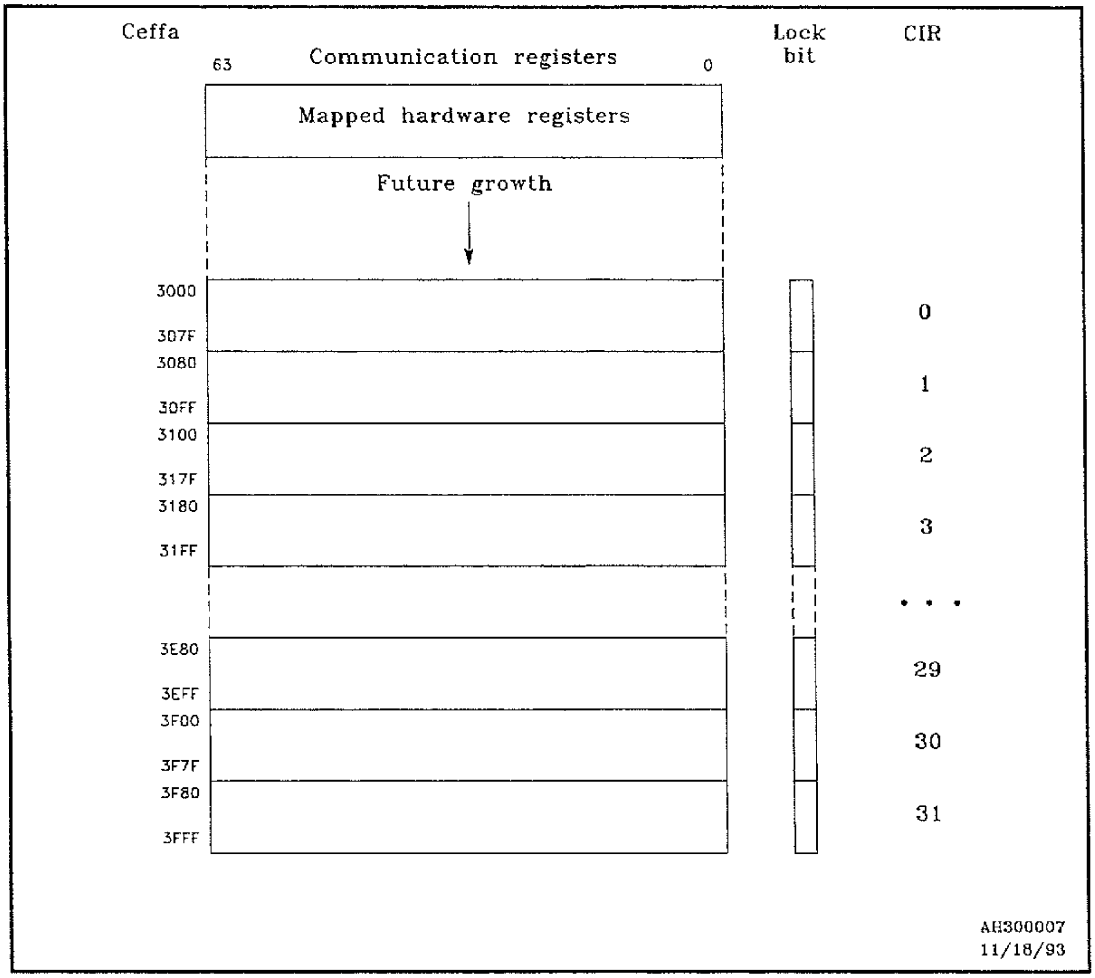
The C3200 Series hardware has 1,024 communication registers. The physical address mapping is located in the range of `3C00` to `3FFF`. Figure 48 shows this mapping broken into 128-register partitions, assigned by CPU (CIR).

Figure 48 Physical communication register address mapping—C3200 Series CPUs



The C3400/C3800/C4600 Series have 4,096 communication registers. The physical address mapping is located in the range of 3000 to 3FFF. Figure 49 shows this mapping broken into 128-register partitions, assigned by CPU (CIR).

Figure 49 Physical communication register address mapping—C3400/C3800/C4600 Series CPUs



Communication register address translation

Virtual-to-physical communication register address translation is a function of the CIR and communication register offset.

CMR address translation - C3200

In C3200 Series complexes, the 1,024 physical communication registers are accessed using a 10-bit physical register address. These physical registers are divided into eight sets of 128 communication registers, which are accessed using a 3-bit CIR index. The communication register virtual address and the 3-bit CIR index are combined in various ways to result in the 10-bit physical address.

Table 32 shows how a hardware register address is developed by using a 16-bit virtual address and translating the virtual address into a 10-bit physical address.

Table 32 Communication register address mapping—C3200 Series CPUs

Virtual address (hexadecimal)	Virtual address bits used	Physical address (binary)	Description
0000 - 001F	0000 0000 000a aaaa	ccc00aaaaa	Ring 0 hardware
0020 - 3BFF	NA	NA	Reserved[1]
3C00 - 3FFF	0011 11aa aaaa aaaa	aaaaaaaaaa	Ring 0 software (CIR independent)
4000 - 401F	0100 0000 000a aaaa	ccc01aaaaa	Ring 0 software (CIR dependent)
4020 - 7FFF	NA	NA	Reserved[1]
8000 - 803F	1000 0000 00aa aaaa	ccc1aaaaaa	Ring 4 (CIR dependent)
8040 - FFFF	NA	NA	Reserved[1]

[1]Any access to these address ranges violates the communication address ring protection and will cause an *invalid communication address* system exception.

The first column in Table 32 defines a range of virtual communication addresses that are mapped to a valid physical communication address.

The second column shows which virtual address bits are used to form the physical address.

The third column shows how the bits in the 16-bit virtual address are used when the virtual communication address is

translated to a physical communication address (where the c bits are the CIR index, and the a bits are the relevant bits from each virtual address range).

The physical address base used in the communication physical addressing mechanism for each 3-bit CIR is shown in Table 33.

Table 33 C3200CIR physical address base assignment—C3200

CIR index	Physical address base assignment
0	3C00
1	3C80
2	3D00
3	3D80
4	3E00
5	3E80
6	3F00
7	3F80

CMR address translation - C3400/C3800/C4600

In C3400/C3800/C4600 Series complexes, the 4,096 physical communication registers are accessed using a 12-bit physical register address. These physical registers are divided into 32 sets of 128 communication registers, which are accessed using a 5-bit CIR index. The communication register virtual address and the 5-bit CIR index are combined in various ways to result in the 12-bit physical address.

Table 34 shows how a hardware register address is developed for the C3400, C3800, and C4600 Series complexes by using a 16-bit virtual address and translating the virtual address into a 12-bit physical address.

Table 34 Communication register address mapping—C3400/C3800/C4600 Series CPUs

Virtual address (hexadecimal)	Virtual address bits used	Physical address (binary)	Description
0000 - 001F	0000 0000 000a aaaa	ccccc00aaaaa	Ring 0 hardware
0020 - 2FFF	NA	NA	Reserved[1]
3000 - 3FFF	0011 aaaa aaaa aaaa	aaaaaaaaaaaa	Ring 0 software (CIR independent)
4000 - 401F	0000 0000 000a aaaa	ccccc01aaaaa	Ring 0 software (CIR dependent)
4020 - 7FFF	NA	NA	Reserved[1]
8000 - 803F	1000 0000 00aa aaaa	ccccc1aaaaaa	Ring 4 (CIR dependent)
8040 - FFFF	NA	NA	Reserved[1]

[1]Any access to these address ranges violates the communication address ring protection and will cause an *invalid communication address* system exception.

The physical address base used in the communication physical addressing mechanism for each 5-bit CIR is shown in Table 35.

Table 35 CIR physical address base—C3400/C3800/C4600

CIR index	Physical address base assignment
0	3000
1	3080
2	3100
3	3180
•••	•••
1D	3E80
1E	3F00
1F	3F80

Communication register modified bits - C3200

Since the CIR defines an operating system process, the communication registers become a part of the process state. Therefore, the communication registers are saved and restored by the operating system when the process is rescheduled, that is, the process relinquishes its CIR (communication register partition) and another process's state is mounted in the communication register partition indexed by the CIR.

On a C3200 Series complex, the communication register hardware provides and maintains a structure called *modified bits* to accelerate save and restore operations for the communication registers. These modified bits are similar in function to the memory referenced and modified bits. The hardware uses these bits to save and restore only those communication registers that have been modified.

Note

The C3400/C3800/C4600 Series CPUs *do not* implement the modified bits.

In general, each register does not have a modified bit. Instead, a modified bit covers a subregion of the communication register address space. Any time a communication register or lock bit in the particular region is modified with `put`, `lck`, `ulk`, `snd`, or `rcv` primitive operations, the modified bit corresponding to that region is set. The following instructions will set the modified bit whenever executed, regardless of success of the operation.

```
put.l, put.w
lck, ulk
snd.l, snd.w
rcv.l, rcv.w
inc.l, inc.w
```

In addition, any instruction that internally performs any of the preceding operations (for example, `pfork`), will set the modified bits associated with the communication registers that the instruction uses.

The C-Series architecture provides two privileged instructions to implement operations for saving and restoring communication registers: `the stcmr` instruction copies communication registers for a specified CIR to memory, and `ldcmr` loads a specified CIR's communication registers from memory.

The `stcmr` instruction examines the communication register modified bits to store only the modified region of communication registers, and then stores the modified bits, along with the

associated communication registers. Since a receive operation is used to implement the read operation, all of the lock bits are cleared after the `stcmr` is complete.

The subsequent `ldcmr` instruction that restores the communication registers will only load the registers that were actually saved by the `stcmr`. The memory copy of the modified bits are referred to as the *valid bits* to avoid confusion. The operating system may alter these bits in memory to force a `ldcmr` instruction to restore more or less of the communication registers than the previous `stcmr` instruction saved.

The C3200 Series CPUs contain 1,024 communication registers, with 128 allocated to each of the eight CIRs. Up to 128 registers per CIR are stored and loaded, along with two longwords of lock bits and one long word containing the communication register set modified bits. The CPUs implement 16 modified bits, two per CIR. These modified bits are maintained by the CPUs that control which registers are stored to (via `stcmr`) and loaded from (via `ldcmr`) memory.

For each C3200 Series CIR, one modified bit corresponds to the hardware and ring 0 communication registers, and another modified bit corresponds to the ring 4 communication registers. When the modified bits are stored to memory, bit <0> of the register set valid bits longword is the hardware/ring 0 modified bit, and bit <32> is the ring 4 modified bit. The remaining bits are hardware reserved.

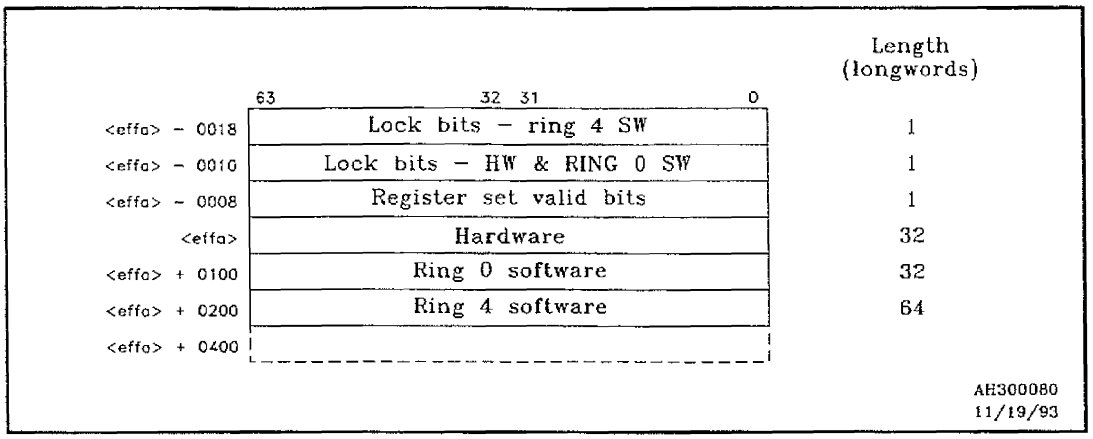
For C3200 Series CPUs, eight of the ten reserved hardware communication registers are not stored by `stcmr` or loaded by `ldcmr`. These registers are reserved as physical CPU-bound quantities. The other two reserved registers may be process-bound, and are saved.

The longword of lock bits corresponding to hardware and ring 0 software from the MSB has eight 0s (reserved). The longword contains the lock bits for the hardware registers (virtual address `0008` to `001F`) in bits <55..32>. It contains the lock bits for ring 0 software (virtual address `4000` to `401F`) in bits <31..0>.

The longword of lock bits corresponding to ring 4 contains the lock bit for the lower communication register address (`8000`) in the MSBs and lock bit for the highest communication register address (`803F`) in the LSBs.

Figure 50 shows the memory format of the communication register lock and valid (modified) bits, with respect to the `stcmr` and `ldcmr` instructions, that are used by the C3200 Series CPUs.

Figure 50 `ldcmr/stcmr` memory map



For C3200 Series CPUs, the valid bit longword contains the memory copy of the modified bit for ring 4 registers in bit <32> and the modified bit for ring 0 registers in bit <0>. The block of longwords that store the data portion of the communication registers are arranged with numerically lower addressed communication registers in numerically lower memory.

C3400/C3800/C4600 Series CPUs use the same memory map shown in Figure 50, but ignore the register set valid bits longword.

Hardware communication registers

The C-Series architecture allocates half of ring 0 communication address space for hardware (see Figure 47). These registers are used by the hardware and ring 0 software to provide multithreaded execution. In the following figures, the notation *Ceffa* denotes a communication register address xxxx..

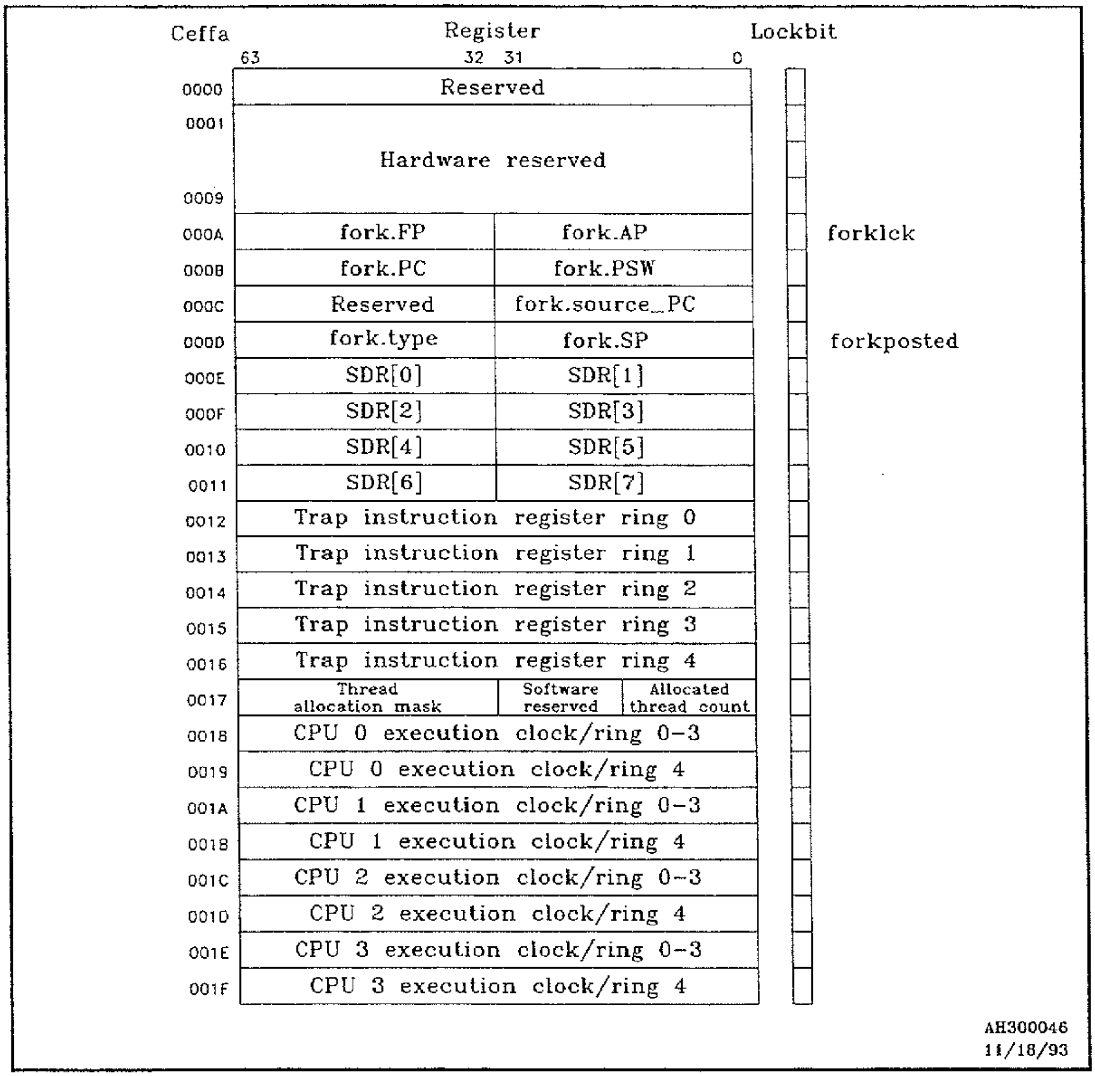
The hardware communication register set contains all process-specific states necessary to schedule a process and create or terminate executing threads. This register set is only accessible from ring 0 and is the primary structure for process scheduling. Hardware enforces protocols on the sense of the lock bits on some of these registers.

The communication registers include a set of universal (except for implementation-specific addressing) registers that are applicable to all multiprocessing C-Series complexes, plus specific control registers for C3200 Series implementations. The C3400 Series CPUs have an additional set of control registers that are integrated into the communication register address space. The C3800/4600 Series CPUs also have an additional set of control registers, but these are not integrated into the communication register address space.

Hardware communication registers - C3200

C3200 Series CPUs use the hardware communication registers illustrated in Figure 51, which include the hardware reserved registers illustrated in Figure 65.

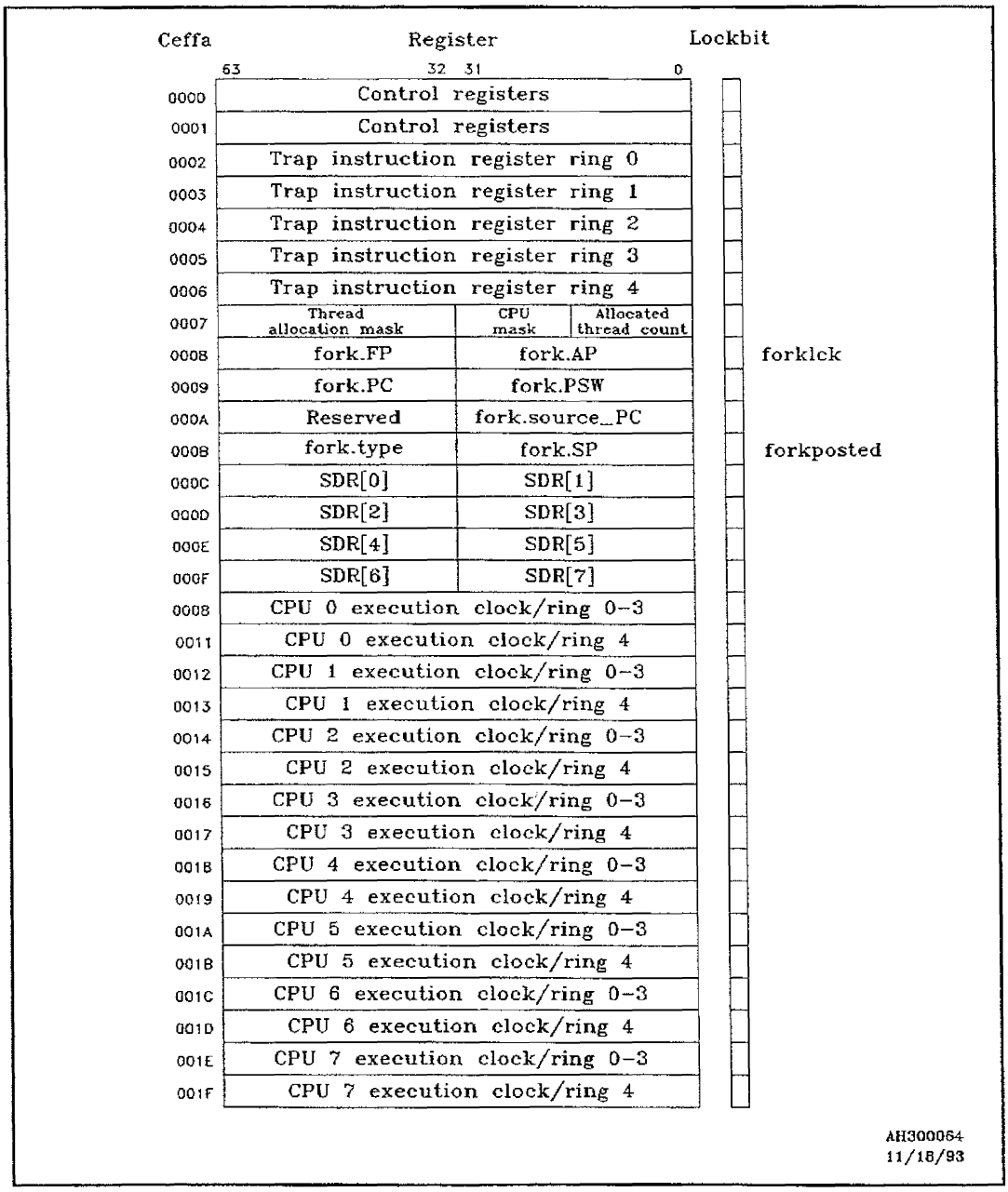
Figure 51 Hardware communication registers—C3200 Series CPUs



Hardware communication registers - C3400/C3800

C3400/C3800 Series CPUs use the hardware communication registers illustrated in Figure 52.

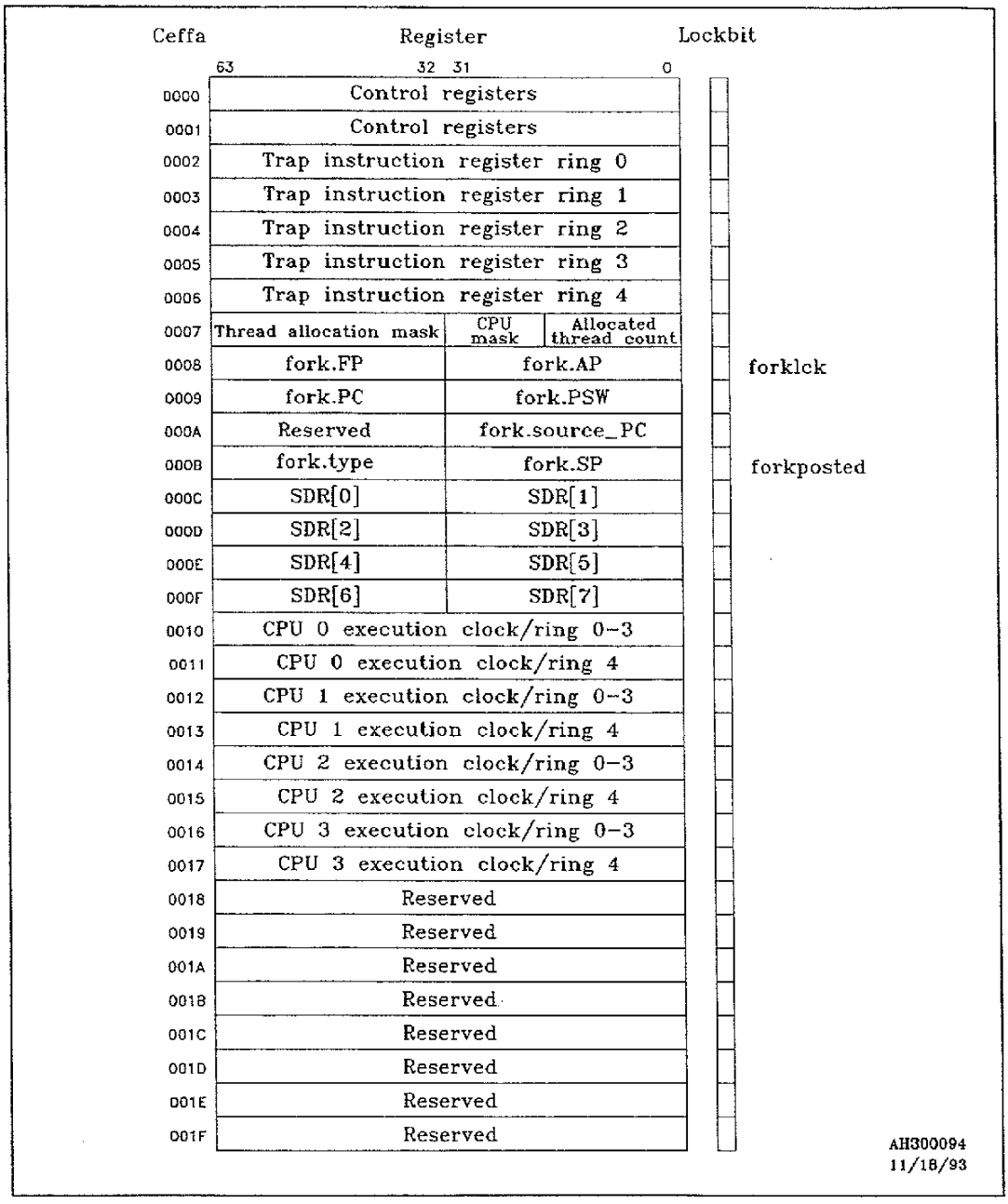
Figure 52 Hardware communication registers—C3400/C3800 Series CPUs



Hardware communication registers - C4600

C4600 Series CPUs use the hardware communication registers illustrated in Figure 53.

Figure 53 Hardware Communication Registers—C4600 Series CPUs



Fork event communication registers

The fork event registers are used for holding information required to create an independent thread of execution. Figure 54 and Figure 55 show the fork event registers.

Figure 54 Fork event registers—C3200 Series CPUs

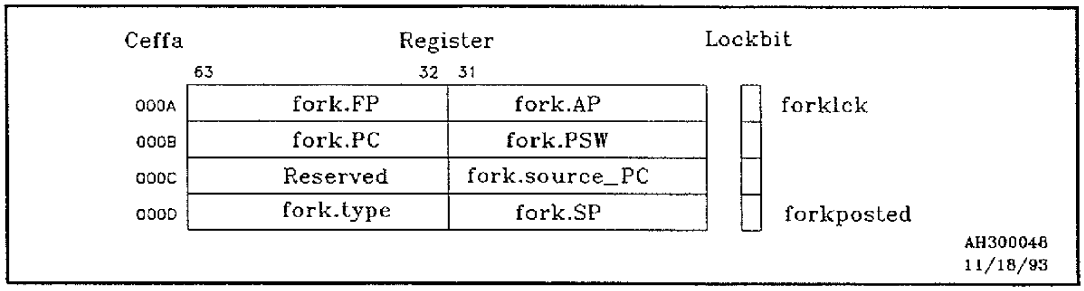
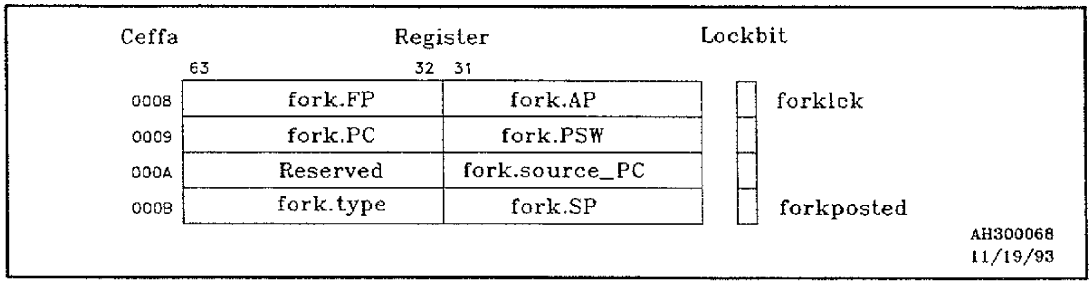


Figure 55 Fork event registers—C3400/C3800/C4600 Series CPUs



One process executing on a CPU requests another CPU by storing information in the fork event registers. An idle CPU then creates a thread and executes it on behalf of the process by loading this information from the fork event registers into its own state registers.

The fork event registers are defined as:

- `fork.FP`—The initial frame pointer for the thread.
- `fork.AP`—The initial argument pointer for the thread.
- `fork.PC`—The program counter to begin execution of the thread.
- `fork.PSW`—The initial PSW for the thread.
- `fork.source_PC`—The PC for the thread posting the fork.
- `fork.type`—A parameter passed from posting to acceptance of the fork.

The possible hexadecimal `fork.type` values are:

- PFORKED 0000 0000
- SPAWNED 0000 000A
- STOPPED 0000 000B

This parameter defines the fork type of a posted fork, so `pfork`, `spawn`, and `join` instructions cannot be mixed in a multithreaded process. Refer to the "Multithreaded execution" section in this chapter, and to the `pfork`, `spawn`, and `join` instruction definitions in the *CONVEX Assembly Language Reference Manual (C Series)* for more information.

- **`fork.SP`**—The initial stack pointer for the thread.

When a fork is posted with `pfork` or `spawn`, the PC of the instruction following the `pfork` or `spawn` instruction is loaded into `fork.source_PC`, which is located in the fork event registers.

When a fork is taken, the value in `fork.source_PC` is loaded into an idle CPU's PC to establish a current ring of execution as the CPU transitions to the active state. A current ring of execution must be established since an idle CPU has no state. The CPU then performs a jump to `fork.PC` with ring wrapping enabled. The ring wrapping prevents a `pfork 0,SP` instruction in ring 4 from accessing ring 0.

The lock bits on the fork event registers, called `forklck` and `forkposted`, are used to convey the state of the fork during its transitions from cleared to posted to taken and to cleared. The fork event registers are defined as:

- **`forklck`**—This bit is a lock bit on the `fork.FP` and `fork.AP` combination. When this lock bit is 1, the hardware is transitioning the fork from clear to posted or posted to taken.
- **`forkposted`**—This bit is a lock bit on the `fork.type` and `fork.SP` combination. When this lock bit is 1, there is a valid fork posted to be taken.

This protocol implies that the first and last fork event registers are manipulated with `snd` and `rcv` operations. Refer to the "Forking operations" section on page 198 for more information on how the `forklck` and `forkposted` lock bits are manipulated.

Segment descriptor registers

The segment descriptor registers (SDR) define the extent of the virtual address space associated with a process. Locating the SDRs in the communication registers causes the entire address translation for a CPU to change whenever the CIR index is

changed. Lock bits on these registers are ignored. These registers should be accessed with `put` and `get` operations.

A `put` or `send` to the SDRs in a communication register *does not change* the copies of the SDRs that a CPU may have accelerated, because the SDRs are accelerated only when the CPU changes the CIR.

Figure 56 and Figure 57 show the segment descriptor registers.

Figure 56 Segment descriptor registers—C3200 Series CPUs

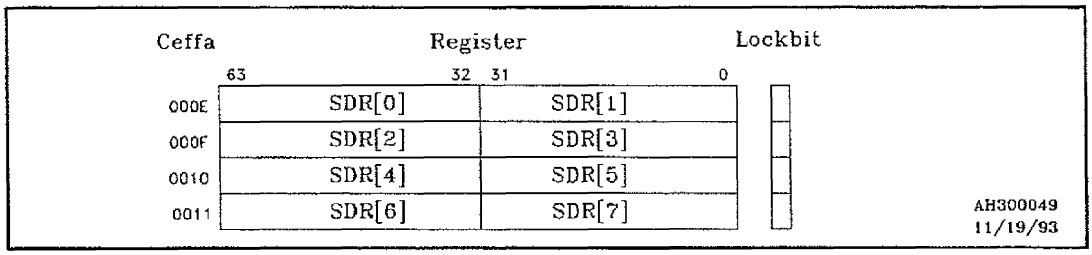
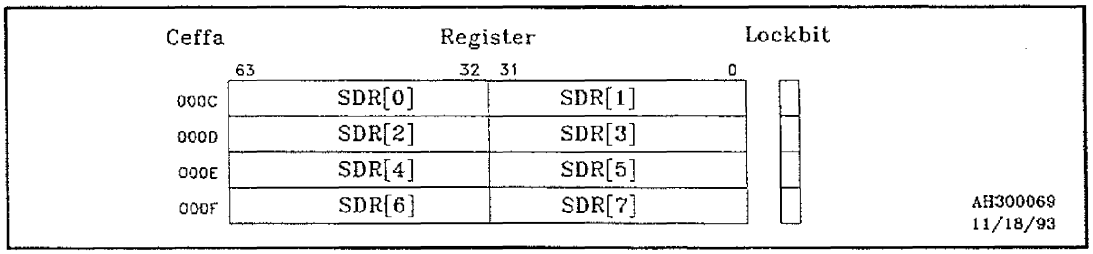


Figure 57 Segment descriptor registers—C3400/C3800/C4600 Series CPUs



Trap instruction registers

There is one 64-bit trap instruction register (TIR) for each ring. It is used by the `trap` and `pbkpt` instructions, which can set specific bits in this register to cause a process-wide system exception. The lock bits on the TIRs are ignored. The TIR is primarily used for asynchronously trapping thread breakpoints or for thread scheduling. The TIR and the `trap` and `pbkpt` instructions are described in detail in the Chapter 6, 'Exceptions and interrupts,' on page 209. Figure 58 and Figure 59 show the trap instruction registers.

Figure 58 Trap instruction registers—C3200 Series CPUs

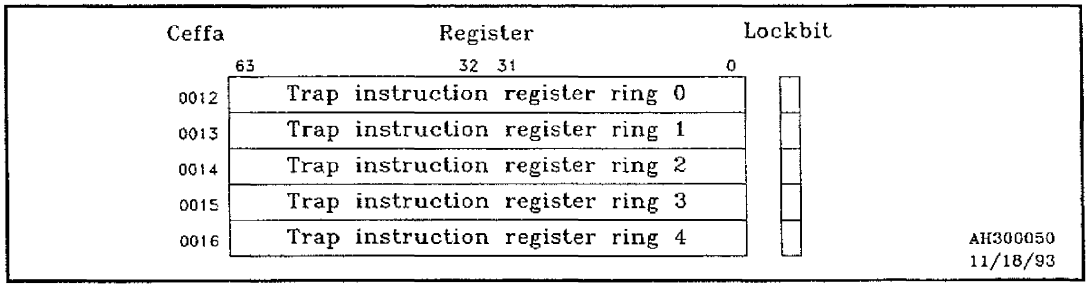
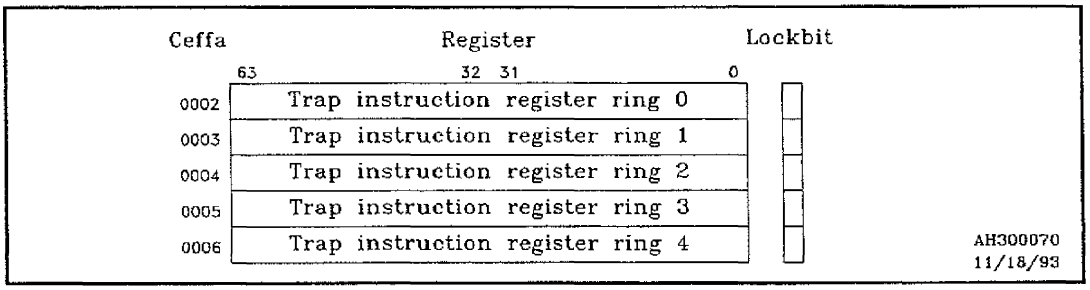


Figure 59 Trap instruction registers—C3400/C3800/C4600 Series CPUs



Thread allocation mask and count

The thread allocation mask is a 32-bit mask and is the primary means for defining the multithreading extent of a process. Each bit position in the thread allocation mask represents a unique thread ID that allows a process to create up to 32 unique threads. Each bit that is set defines a thread that can be created. By limiting the number of bits in the mask, a process is forced to run with a limited number of threads.

Figure 60 and Figure 61 show the thread allocation registers.

Figure 60 Thread allocation register—C3200 Series CPUs

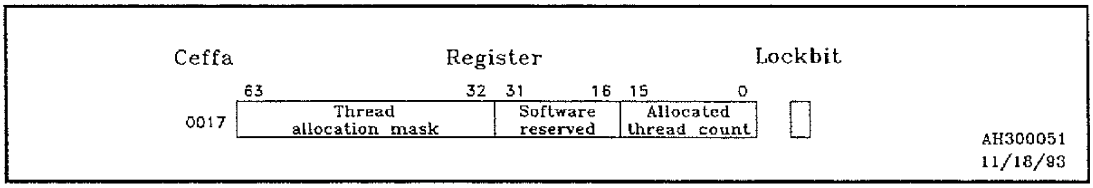
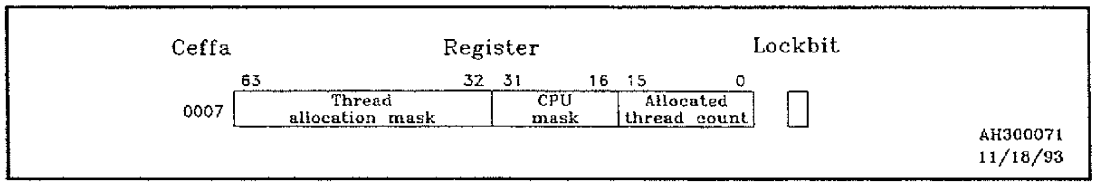


Figure 61 Thread allocation register and CPU mask—C3400/C3800/C4600 Series CPUs



In order for a thread to be created (CPU transition from idle to allocated), a unique thread ID is generated. This is done by atomically clearing a single bit in the thread allocation mask as a function of the CPU idle loop. The CPU idle loop searches the communication register sets for a posted fork event. The CPU's thread ID (TID) register is then loaded with the allocated thread ID to identify the new thread throughout its existence.

When a CPU transitions from allocated to idle, it atomically sets the bit associated with the CPU's TID register in the thread allocation mask as a function of the CPU idle loop.

The allocated thread count is a 16-bit integer. It is a count of the number of thread IDs allocated from the thread allocation mask. When a thread is created, the thread count is incremented, and when a thread terminates, the thread count is decremented. The thread allocation count is used by the `join` instruction. It can also be used to determine the current extent of process multithreading.

The lock bit for the thread mask and thread count (thread allocation register) is interpreted as a "valid" bit, allowing this

register to be manipulated with `snd` and `rcv` operations. This lock bit is the central synchronization point of all fork operations. An idle CPU waits until it can successfully receive this register to ensure that a valid fork is taken, then allocates a thread to the fork. By locking the thread mask or count (that is, making the thread count or mask unreceivable), software can ensure that no forks are accepted in that communication register set.

The CPU mask is an 8-bit mask that enables a CPU to pick up forks in that process. Bit 0 refers to CPU 0, bit 1 refers to CPU 1, and so forth. If the CPU's bit is set, then the CPU may pick up the fork.

CPU execution clock registers

Each CPU has a 64-bit microsecond clock register that provides the exact execution time per CPU within each ring. The CPU execution clock registers are updated on ring crossings, CIR changes, and communication register state stores. The `ctrsg` instruction forces an update of these registers. Software must ensure that these registers are updated before they are examined. These clock registers are maintained on a per-CPU basis, so synchronization is not necessary for operations that update them. The lock bits for the CPU execution clock registers are ignored.

Figure 62, Figure 63, and Figure 63 show the CPU execution registers.

Figure 62 CPU execution clock registers—C3200 Series CPUs

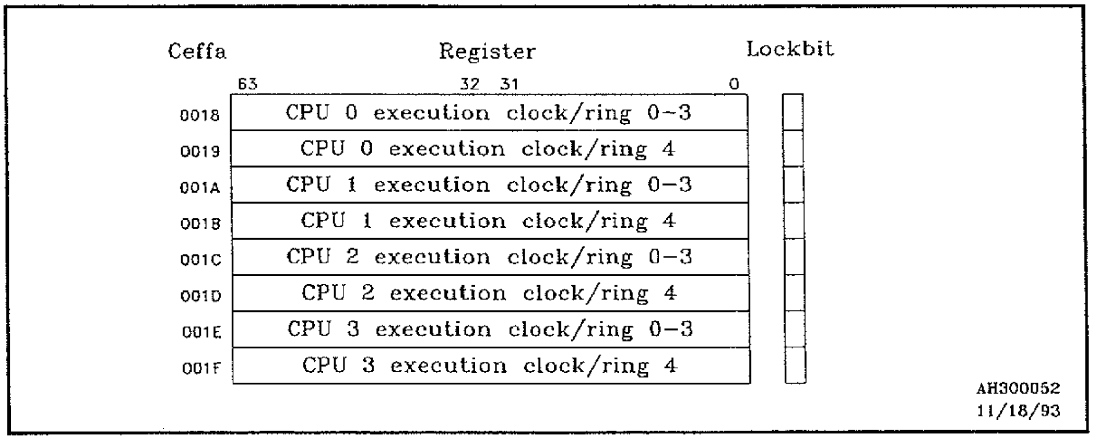


Figure 63 CPU execution clock registers—C3400/C3800 Series CPUs

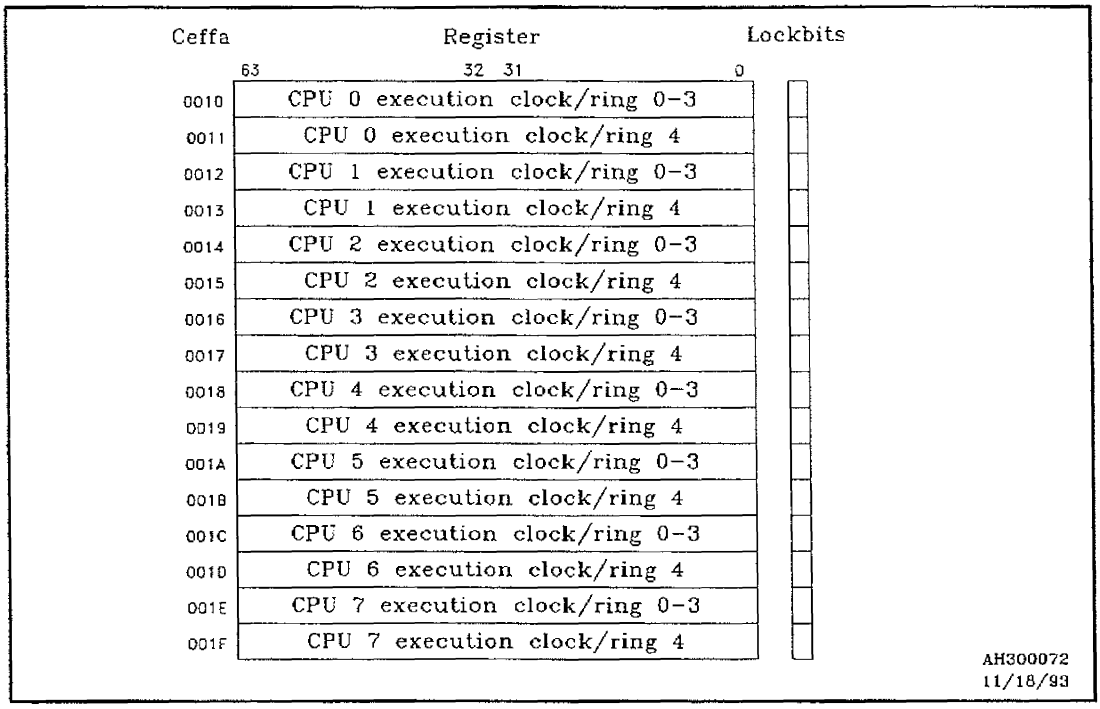
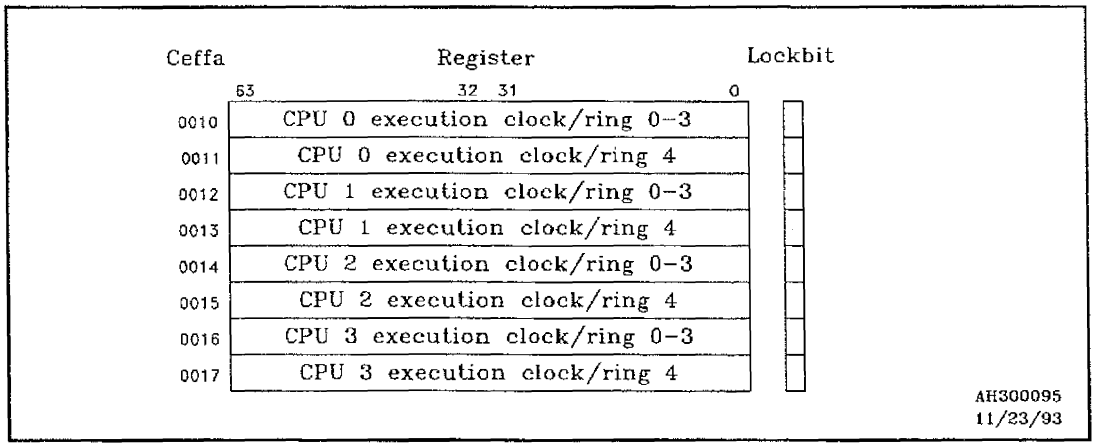


Figure 64 CPU execution clock registers—C4600 Series CPUs

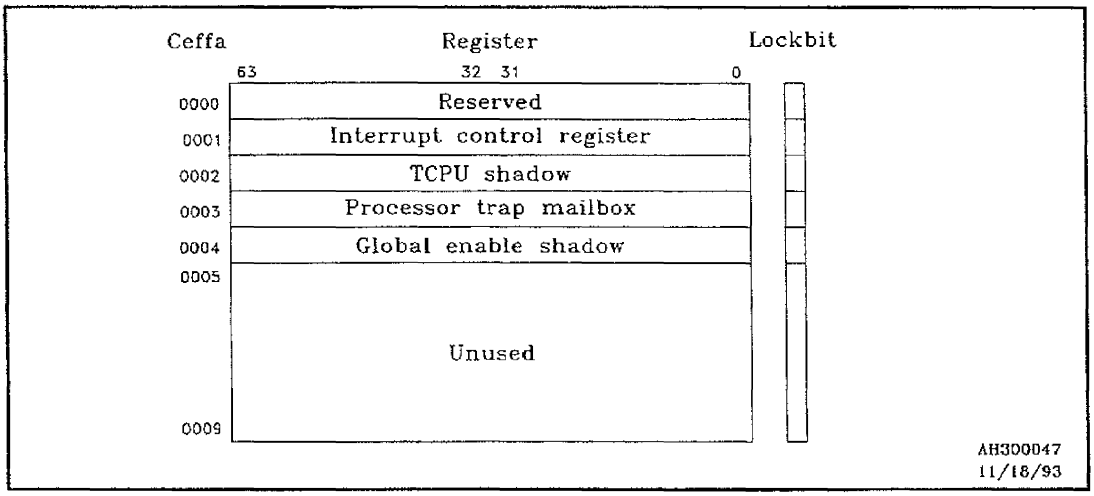


Refer to the "CPU execution timer"section in Chapter 5 for more information about the CPU execution clock registers.

Hardware reserved CMR - C3200

For the C3200 Series CPUs, the communication virtual address space from 0000 to 0009 is reserved for hardware reserved registers that are used primarily as *shadow* copies of complex-wide write-only registers. These registers (in the communication address range) are indexed by CIR = 0, as shown in Figure 65. The communication virtual address range 3C00 to 3C09 places these registers at the communication address range 0000 to 0009 indexed by CIR = 0.

Figure 65 Hardware reserved communication registers—C3200 Series CPUs



For example, the target CPU register (TCPU) is a hardware reserved register in the interrupt logic. This register is explained in Chapter 6, 'Exceptions and interrupts,' on page 209. The C-Series architecture defines instructions to both write (`mov Sk, TCPU`) and read (`mov TCPU, Sk`) this register. When the TCPU is written with `mov Sk, TCPU`, the register `Sk` is also written to one of the hardware reserved communication registers. This shadow copy value is independent of the CIR. A `mov TCPU, Sk` instruction writes register `Sk` from the shadow copy in the hardware reserved communication registers.

Control registers - C3400

Control registers used by C3400 Series complexes are not contiguous, and are integrated within the communication register space. The control registers are the first two entries in each communication register set (CIR). They should be accessed with physical addresses, since the functions of the control registers are independent of the CIR set of which they are a part. Most registers are accessed with microcode (firmware) invoked by dedicated assembly language commands.

Figure 66 describes the mapping of the C3400 Series control registers from within the communication registers.

Figure 66 Control register mapping—C3400 Series CPUs

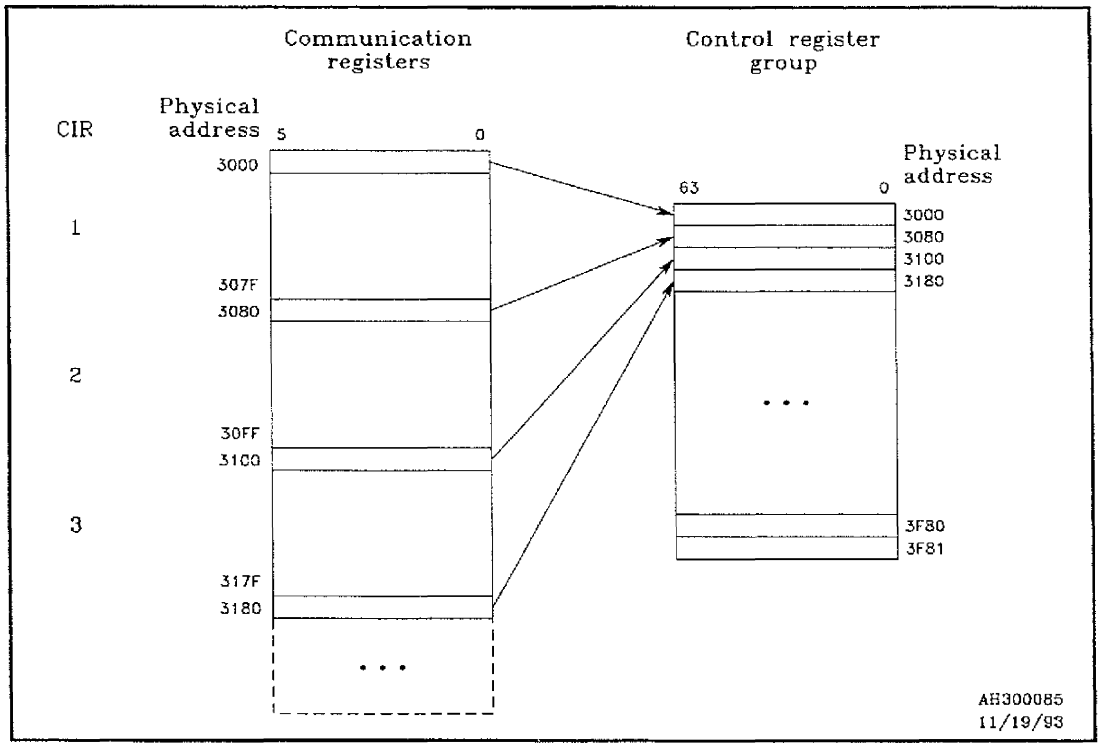
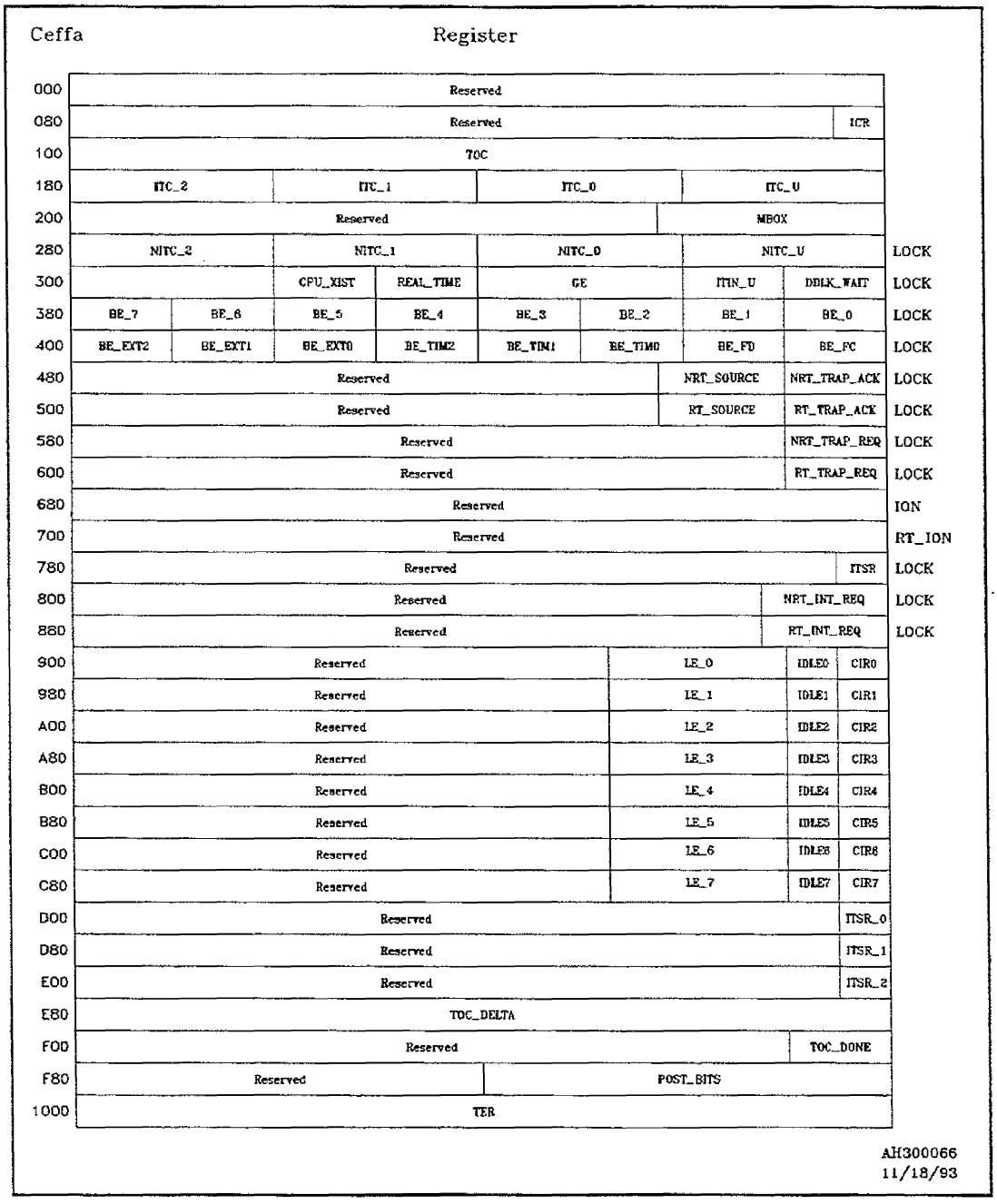


Figure 67 describes the layout of the C3400 Series control registers.

Figure 67 Control register layout—C3400 Series CPUs



Control registers refer specifically to the registers the C3400 Series CPU and utilities board (CUJ) use to implement global functions. These registers are used to control interval timers, request system interrupt bus (SIB) interrupts, store time of century (TOC), and provide status for process deadlock detect.

The following subsections describe the control register bit assignments within communication register space as shown in Figure 67.

Interrupt control register

The interrupt control register (ICR) contains the interrupt mode and interrupt CIR. Refer to the "Interrupt system" section in Chapter 6. The interrupt mode portion is a shadow of a write-only hardware register. The interrupt CIR is not a shadow. It is implemented entirely in communication registers.

```
ICR<4..0>
```

Time of century register

The time of century (TOC) register is a 64-bit counter that can be loaded by the CPUs. It is always on.

```
TOC<63..0>
```

Time of century delta time register

The time of century delta time (TOC_DELTA) register is a 64-bit counter that accurately maintains timers across TOC writes.

```
TOC_DELTA<63..0>
```

Interval timers

When the CUJ decrements the 16-bit interval timers, it updates this control register. The ITC_U interval timer is intended to support the same functions as the C3200 Series interval timer. It is clocked every 10 µs. The remaining 3 interval timers are intended to support the new realtime functions. Their clocks are individually selectable to be 10 µs or 100 µs. This selection is made in the interval timer status register (ITSR).

```
ITC_U<15..0>
ITC_0<15..0>
ITC_1<15..0>
ITC_2<15..0>
```

Interval timer indicators

As with C3200 Series complexes, NITC is the load value for the interval timer. The NITC_U interval timer is intended to support the same functions as the C3200 Series interval timer. The remaining three interval timers are intended to support the new realtime functions.

```
NITC_U<15..0>
NITC_0<15..0>
NITC_1<15..0>
NITC_2<15..0>
```

Process trap mail box

The processor trap mailbox (MBOX) is used by the hardware to communicate between CPUs for instructions such as `patu` where one CPU executing the instruction must cause other CPUs to perform an action. MBOX stores SIB interrupt channel bits (one bit per channel), and instructions for CPU traps.

```
MBOX<17>      Deadlock
MBOX<16..9>   SIB interrupt channel
MBOX<8>       CPU trap request
MBOX<7..3>    reserved
MBOX<2..0>    Encoded CPU trap
```

MBOX bits <16..9> correspond to CPU7-CPU0.

MBOX bits <2..0> are listed in Table 36.

Table 36 MBOX action codes—C3400 Series CPUs

Code	Meaning
7	Trap instruction
6	TOC write
5	Update CPU timers
4	Purge ATU reference bits
3	Purge ATU modified bits
2	Purge ATU
1	Purge ATU entry

Interval timer interrupt indicators

The system interval timer sends an interrupt over the system interrupt bus (SIB) when it is on and underflows. ITIN_U is intended to support the same functions as the C3200 Series ITIN, that is, it stores the SIB channel to be used for this interrupt.

```
ITIN_U<7..0>
```

CPU exist indicators

CPU_XIST stores CPUs currently in the system. The CUJ uses this information to determine which CPU-specific control registers to monitor. In C3200 Series complexes, this register is read-only and is loaded by scan. In C3400 Series complexes, this register is in communication register space, and can be written by the CPUs.

```
CPU_XIST<7..0>
```

CPU_XIST bits <7..0> correspond to CPU7-CPU0.

Realtime indicators

A CPU updates a control register bit to indicate whether it is a realtime or non-realtime (timeshare) CPU. The CUJ uses this information to determine which CPUs receive realtime PROC_TRAPs and which receive non-realtime PROC_TRAPs.

```
REAL_TIME<7..0>
```

REAL_TIME bits <7..0> correspond to CPU7-CPU0.

Deadlock indicators

Once the CUJ has detected a deadlock in a CIR, it counts down from this number before requesting a deadlock trap. This is done to allow time for any transient deadlock conditions to settle.

```
DDLK_WAIT<7..0>
```

Global enable register

The global enable (GE) register can globally enable or disable any interrupt that is destined for the CPUs. The GE register accommodates the additional realtime interrupts in C3400 complexes. The bit assignments are the same as local enable register.

Local enable registers

C3200 Series CPUs have local enable (LE) registers to enable or disable specific interrupts to each CPU. This includes all realtime and non-realtime system interrupt bus (SIB) interrupts, the realtime interval timer interrupts, and the realtime external interrupts.

```
LE_0<15..0>
LE_1<15..0>
LE_2<15..0>
LE_3<15..0>
LE_4<15..0>
LE_5<15..0>
LE_6<15..0>
LE_7<15..0>
```

The local enable bit assignments are listed inTable 37.

Table 37 Bit assignments—global and local enable registers

Bit number	Description
0	SIB 0
1	SIB 1
2	SIB 2
3	SIB 3
4	SIB 4
5	SIB 5
6	SIB 6
7	SIB 7
8	SIB FC
9	SIB FD
10	TIM_0
11	TIM_1
12	TIM_2
13	EXT_0
14	EXT_1
15	EXT_2

Broadcast enable registers

The broadcast enable (BE) registers replace the TCPU (target CPU) and MODE (broadcast/target mode) registers in the C3200 Series CPUs. The BEs make the interrupt broadcast selection more flexible. This includes SIB interrupt channels, realtime interval timers, and realtime external interrupts.

The eight bits can be set in any combination to allow interrupts to be broadcast to any set of CPUs. If none of the BE bits are set, the CUJ will choose a target CPU, based on which CPUs are idle. It is assumed that the BEs will be set to broadcast to only realtime CPUs or to only non-realtime CPUs.

```
BE_0<7..0>
BE_1<7..0>
BE_2<7..0>
BE_3<7..0>
BE_4<7..0>
BE_5<7..0>
BE_6<7..0>
BE_7<7..0>
BE_FC<7..0>
BE_FD<7..0>
BE_TIM0<7..0>
BE_TIM1<7..0>
BE_TIM2<7..0>
BE_EXT0<7..0>
BE_EXT1<7..0>
BE_EXT2<7..0>
```

BE bits <7..0> correspond to CPU7-CPU0.

Interrupt/trap source indicators

In C3400 Series complexes, there is only one interrupt/trap (PROC_TRAP) signal on each CPU. Before the CUJ issues a PROC_TRAP to the CPUs, it first writes the source of the PROC_TRAP to one of these registers. The sources could be deadlock traps, CPU traps, SIB interrupts, interval timer interrupts, or external interrupts. Thus, when a CPU receives a PROC_TRAP, it must check a source register to determine the appropriate action to take. The CPUs may use another communication register to communicate the source of a CPU trap. Realtime CPUs should check RT_SOURCE and non-realtime (timesharing) CPUs should check NRT_SOURCE. The bit assignments of these registers correspond to the priorities, with CPU-requested traps at bit 0.

```
NRT_SOURCE<9..0>
RT_SOURCE<9..0>
```

The interrupt/trap source bit assignments are listed in Table 38.

Table 38 Bit assignment—Interrupt/trap source registers

Bit number	NRT_TRAP_SOURCE	RT_TRAP_SOURCE
0	CPU trap	CPU trap
1	SIB 0	SIB 0xFC
2	SIB 1	SIB 0xFD
3	SIB 2	Interval timer 0
4	SIB 3	Interval timer 1
5	SIB 4	Interval timer 2
6	SIB 5	External interrupt 0
7	SIB 6	External interrupt 1
8	SIB 7	External interrupt 2
9	Deadlock trap	Deadlock trap

Interrupt/trap acknowledge indicators

When the CUJ sends a non-realtime PROC_TRAP (interrupt/trap) to a set of CPUs, it sets the non-realtime NRT_TRAP_ACK bits corresponding to those same CPUs. The CPUs acknowledge receipt of the PROC_TRAP by clearing their assigned NRT_TRAP_ACK bit. When all of the NRT_TRAP_ACK bits are cleared, the CUJ is free to send another non-realtime PROC_TRAP. acknowledgment

Realtime RT_TRAP_ACK bits function identically.

Non-realtime CPUs should only acknowledge non-realtime PROC_TRAPs, and realtime CPUs should only acknowledge realtime PROC_TRAPs.

```
NRT_TRAP_ACK<7..0>
RT_TRAP_ACK<7..0>
```

TRAP_ACK bits <7..0> correspond to CPU7-CPU0.

Interrupt/trap request indicators

These registers are used for CPU-requested traps. A CPU must first check the lock bit. If it is clear, the CPU may load a broadcast enable (BE) into bits 0-7 and set the lock bit. If it is set, it indicates a pending CPU trap request. The CUJ uses this information in its PROC_TRAP arbitration.

Once the CPU-requested trap has won the PROC_TRAP arbitration, the CUJ clears the lock bit. Non-realtime CPUs should use NRT_TRAP_REQ, and realtime CPUs should use RT_TRAP_REQ.

A communication register can be set so that the CPUs post whether they are available to accept traps. The requesting CPU can then set up its BE based on which CPUs are available. The BE for NRT_TRAP_REQ should select only non-realtime CPUs and the BE for RT_TRAP_REQ should select only realtime CPUs.

```
NRT_TRAP_REQ<7..0> + LOCK
RT_TRAP_REQ<7..0> + LOCK
```

TRAP_REQ bits <7..0> correspond to CPU7-CPU0.

SIB interrupt request indicators

The CPUs use these registers to request SIB interrupts. A CPU must first check the lock bit. If it is clear, the CPU loads an interrupt channel into bits 0-7 and sets the lock bit. The set lock bit indicates a pending SIB interrupt request. The CUJ then initiates an SIB interrupt using the predefined SIB protocol. Once the interrupt has been accepted, the CUJ clears the lock bit. If the requesting CPU cancels its SIB request, it sets bit 8 of this register.

The CUJ then cancels the request at the next available opportunity by clearing the lock bit and bit 8. If the SIB request is serviced before the CUJ is able to cancel it, the CPU is not notified. Non-realtime CPUs should use NRT_INT_REQ and realtime CPUs should use RT_INT_REQ.

```
NRT_INT_REQ<9..0> + LOCK
RT_INT_REQ<9..0> + LOCK
```

ION bit

This (lock) bit is used to enable or disable non-realtime (timesharing) SIB interrupt channels 0-7. It serves the same function in C3200/C3800 Series complexes. The ION bit is global, as is the RT_ION bit. Non-realtime CPUs use ION and realtime CPUs (C3400 Series only) use RT_ION.

```
ION
```

RT_ION bit

This (lock) bit (C3400 Series CPUs only) is equivalent to ION, but is used to enable or disable realtime interrupts. This includes SIB interrupts `0xfc` and `0xfd`, three interval timer interrupts, and three external interrupts.

```
RT_ION
```

Interval status register

ITSR is the status register for the interval timer.
In C3200 Series complexes, the ITSR is a global register in I/O space and bits 0-1 are cleared when the register is read. In C3400 Series complexes, the ITSRs are in communication register space, which means that a copy of the ITSR resides on each CPU.

In order to clear the ITSR on all of the CPUs simultaneously, a CPU must perform a write. The write automatically occurs whenever a CPU reads its copy of the ITSR, so this difference is transparent to the operating system.

The ITSR timer is intended to perform the function of the C200 Series complex's interval timer. The remaining timers are intended to support the new realtime functionality.

Bits 0-2 have the same meaning as in C3200 complexes. ITSR<0> indicates that an underflow has occurred since the last read of this register. ITSR<1> indicates that 2 or more underflows have occurred since the last read of this register. ITSR<2> turns the timer off or on. ITSR<3> selects the clock interval for the realtime interval timers (0=10 µs, 1=100 µs).

```
ITSR<3..0>
ITSR_0<3..0>
ITSR_1<3..0>
ITSR_2<3..0>
```

Idle indicators

A CPU updates a control register bit when it switches in or out of the idle state. This information is used on the CUJ to select a target CPU for an interrupt or trap, if none has been specified in the broadcast enable (BE) register.

```
IDLE0<7..0>
IDLE1<7..0>
IDLE2<7..0>
IDLE3<7..0>
IDLE4<7..0>
IDLE5<7..0>
IDLE6<7..0>
IDLE7<7..0>
IDLE bits <7..0> correspond to CPU7-CPU0.
```

Communication interrupt registers

Each CPU writes its current CIR (process) to a corresponding control register. Five bits are needed because there are 32 possible CIRs. The CUJ uses this information to detect CIR deadlock.

```
CIR_0<5..0>
CIR_1<5..0>
CIR_2<5..0>
CIR_3<5..0>
CIR_4<5..0>
CIR_5<5..0>
CIR_6<5..0>
CIR_7<5..0>
```

TOC write complete

These bits provide synchronization for TOC write bits.

```
TOC_DONE<7..0>
```

TOC_DONE bits <7..0> correspond to CPU7-CPU0.

Post bit register

If the POST_BITS bit corresponding to a CIR is set, the CIR is examined by idle heads for work otherwise, the CIR is skipped. This speeds up the idle loop processing.

```
POST_BITS<31..0>
```

POST_BITS bits <31..0> correspond to CIR31-CIR0.

TER trap enable register

A single TER exists for the complex, and is manipulated by the `diag` instruction subcodes listed in Table 39.

Table 39 TER operations `diag` instruction subcodes

diag subcode	Description
37	read TER: S0 = TER
38	write TER: TER = S0
39	pate local head only, address in A5
40	patu local head only

TER bits <7..0> correspond to CPU7-CPU0. On C3200 Series CPUs (maximum of four CPUs), bits <7..4> do not apply.

If a bit in the TER is clear (0), the following traps or commands are disabled on the corresponding CPU:

- `pate`
- `patu`
- `ctrsg` timer update
- `trap` and `pbkpt` handling
- `pref` (C3400/C3800 Series only)
- `pmod` (C3400/C3800 Series only)

If the bit in the TER is set, the traps or commands are enabled. All traps are enabled on cold start.

The processor executing the trap-generating instruction does not check the TER. For example, if CPU0 has traps disabled via the TER and executes a pate instruction, its PTE cache is purged.

The TER is not controlled by an internal locking semaphore. It is assumed the diag instruction to write the TER is executed by the kernel in a single-threaded region.

Control registers - C3800/C4600

Control registers used by C3800/C4600 Series complexes are contiguous and are not integrated within the communication register space. The control registers are in an address space separate from the communication registers, referred to as X *space*. The registers are accessed with microcode (firmware) invoked by dedicated assembly language commands.

Table 40 describes the layout of the C3800/C4600 Series control registers in X space.

Table 40 C3800/C4600 Series control registers in X space

Addr <9..3>	Register name	Description
00	LCKB	Lockbit shift register and lock bit
01	TOC	Time of century register
02	TRPCMD	Trap command register and lock bit
03	PCIR	Posted thread CIR (ASAP) and status bit
04	NITC	Next ITC value
05	ITC	Interval timer counter
06	ITSR	ITC status register
07	ITIN	ITC interrupt channel
08	Spare	Unused
09	IO INSTALL	Map of ports containing NIAs
0A	CPU INSTALL	Map of ports CPUs
0B-0F	Spare	Unused
10-17	P0-P7 CIR	CIR index, CPUs 0-7
18-1F	Spare	Unused
20-27	P0-P7 IDLE	Idle status, CPUs 0-7
28-2F	Spare	Unused
30	GP	Global pending traps
31	GE	Global channel enables
32	MBP	Memory base pointer
33-37	Spare	Unused
38-3F	P0-P7 LE	Local channel enable, CPUs 0-7
40-47	Spare	Unused
48-4F	L0-L7 BE	CPUs broadcast enable, channels 0-7
50-7F	Spare	Unused

Control registers refer specifically to the registers the C3800/C4600 Series CPUs and utilities board use to implement global functions. These registers control interval timers, request system interrupt bus (SIB) interrupts, store time of century (TOC), provide status for process deadlock detect, and so on.

The following subsections describe the control register bit assignments within communication register space as shown in Table 40.

Lockbit shift register

The lockbit shift register (LCKB) is used when dumping, restoring, or initializing a bank of 64 CMRs. It contains 64 bits, one for each lockbit in a CIR block. This register itself has a lockbit to prevent corruption when multiple processors try to use it.

Time of century register

The time of century (TOC) register is a 64-bit counter which counts in microseconds. There is no enable to start and stop the counting, but the CPUs may write or read it at any time. The TOC is always on.

Trap command register

The trap command register (TRPCMD) contains 32 bits used to issue commands to the trap logic.

Posted thread CIR

The posted thread CIR (PCIR) is not really a register, but the read-only value of the ASAP accelerator output calculated during the read request. An idle CPU typically reads the PCIR to find a CIR index that has a thread available for that processor. If the read status is 0, no threads are available for that particular processor. If the read status is 1, the value of PCIR is the CIR index that has the available posted thread.

Next ITC register

The Next ITC register (NITC) contains a 16-bit value. When the interval timer counter (ITC) reaches zero, it is reloaded with the value in the NITC.

Interval timer counter

The interval timer counter (ITC) is a 16-bit value that decrements every ten microseconds. When it reaches zero, it may trigger an interrupt. It will automatically reload with the value in the NITC register, or it may be written at any time. The ITC will only count when bit <2> of the ITSR is set.

ITC status register

The ITC status register (ITSR) is a 3-bit register. The most significant, bit <2>, may be written, and the lower two are read only. When bit <2> is set, the ITC is enabled. Bit <0> is set when the ITC rolls over. Bit <1> is set if the ITC rolls over again. It indicates that the ITC has rolled over more than once without being serviced. When the ITSR is read, the two least significant bits are reset.

ITC interrupt channel register

The ITC interrupt channel register (ITIN) contains an 8-bit interrupt channel number. If the ITC rolls over to zero, it will set an interrupt flag. When the TRPCMD register is available (unlocked), it is loaded with an XMTI trap command with the channel number from the ITIN.

IO INSTALL register

The IO INSTALL register contains a 9-bit value. Each bit location corresponds to each of the nine ports where an NIA can reside. Bits <8..0> correspond to ports 8-0. Ports 7-0 can contain CPUs or NIAs. Port 8 always contains an NIA. A bit is set only when an active NIA is attached to the corresponding port. This register determines where XMTI traps above channel 8 can be sent.

CPU INSTALL register

The CPU INSTALL register contains an 8-bit value. Each bit location corresponds to each of the eight ports where a CPU can reside. Ports 7-0 correspond to bits <7..0> in this register. A bit is set only when an active CPU is attached to the corresponding port. This register determines which CPUs can participate in trap dispatches and deadlock checking.

Communication index registers

There are eight communication index registers (CIR), one for each of the eight possible CPUs in the complex. Each register is five bits wide for the 32 possible CIR values. The processors maintain these registers to shadow the values in the CIRs. These registers are used for deadlock checking on the CU.

IDLE registers

There are eight IDLE registers, one for each of the eight possible CPUs in the complex. Each register is one bit wide. The CPUs maintain the IDLE registers and set a register when the corresponding CPU is idle. The IDLE registers are used in the interrupt arbitration.

Globally pending interrupt register

The globally pending interrupt register (GP) is an 8-bit, read-only register. Bits <7..0> correspond to interrupt channels 7-0. Bits are set by XMTI commands being sent to the TRPCMD register for the corresponding channels. Bits in this register are reset when the corresponding interrupts are dispatched. Interrupt channels greater than seven are sent as XMTI traps to the NIA(s) for the IO system and have no effect on this register.

Global enable register

The global enable register (GE) is an 8-bit, read/write register. Bits <7..0> are used to enable interrupts on channels 7-0, respectively. When a bit is set, the corresponding interrupt channel is globally enabled. The interrupt channel is not necessarily enabled for dispatching, since the LE and BE registers also control the channels. This register provides the capability to enable interrupts on a channel by channel basis.

Memory base pointer register

The memory base pointer register (MBP) is a 64-bit read/write register used to store the memory base pointer, an OS variable. It is only a centrally located storage register and performs no special function as a control register.

Local enable registers

There are eight Local enable registers (LE) that have eight bits each. Each register corresponds to one of the eight possible CPUs. Bits <7..0> of each register enable a corresponding interrupt channel 7-0 to issue interrupts to that processor.

For example, if P4 LE has a value of `0x31`, then channels 4, 5, and 0 may interrupt processor 4. Channels 1, 2, 3, 6, and 7 cannot interrupt processor 4, even if these channels are otherwise enabled.

Broadcast enable registers

There are eight Broadcast enable registers (BE) that have eight bits each. Each register corresponds to one of the eight interrupt channels. Usually (in non-broadcast operations) an interrupt channel only interrupts one processor, even if several are locally enabled on that channel. In the case of broadcast, several processors receive the interrupt when it is dispatched. A channel is placed in broadcast mode by having a nonzero value in its BE register. In this case, the eight bits in each BE register correspond to the eight possible CPUs that can receive the interrupt when it is dispatched.

For example, if L6 BE has a value of 0x58, then processors 4, 6, and 3 will get a simultaneous interrupt when channel 6 interrupts, if these processors are all locally enabled on channel 6. If they are not all locally enabled, the interrupt will remain pending.

Traps and interrupts

Traps allow the CPUs and I/O Adapters (IA) to signal each other about events that need to be serviced. Interrupts are basically traps in the C3800/C4600 Series complexes. Interrupts to CPUs have some channel arbitration functions, but they are dispatched as traps.

In order to issue a trap, a CPU performs a microcode operation (SND_X) to the TRPCMD register. If a status of 1 is returned, the write was successful and the CPU can assume that the trap will be dispatched eventually. If a 0 status is returned, the TRPCMD register is locked, and the CPU must try again to issue the trap.

When a trap is dispatched, one or more trap ready (TRAP_RDY) signals are sent to the CPU ports or IA ports. The trap type is determined by the value written to TRPCMD <27..24>.

When a port has finished processing its trap, it sends back a trap complete (TRAP_COMP). This clears its busy bit in the trap state machine. When all of the ports have completed their traps, the CPU Utilities (CU) subsystem may return an MT_COMP.

Traps are usually sent to all ports with active CPUs, except for the port that initiated the trap. The active CPU ports are identified in the CPU INSTALL register. An exception to this is the DLCK trap, which goes only to the deadlocked CPU(s). Another exception to this is the XMTI trap, which invokes a complex and flexible arbitration scheme.

Communication register primitive operations

This section describes the set of primitive operations performed on the communication registers and on associated structures indirectly. The multiprocessing implementation includes instructions that execute these primitive operations directly, as well as some instructions that use more than one of these operations to perform their function. These primitive operations and related instructions are:

- **put**—Write the communication register, regardless of lock bit. The instructions are `put.w Ak,` *Ceffa* and `put.l Sk,` *Ceffa*.
- **get**—Read the communication register regardless of lock bit. The instructions are `get.w` *Ceffa*`,Ak` and `get.l` *Ceffa*`,Sk`.
- **send**—Write the communication register if the lock bit is clear, then set the lock bit. The send operation fails if the lock bit was already set, indicating valid data was in the register. A carry (C or SC) status of 1 is returned if the send operation is successful (that is, the lock bit was initially clear), and 0 if the send operation fails (that is, the lock bit was initially set, indicating data was already sent there). The instructions are `snd.w Ak,` *Ceffa* and `snd.l Sk,` *Ceffa*.
- **receive**—Read the communication register if the lock bit is set, then clear the lock bit. The receive operation fails if the lock bit was clear, indicating no valid data was in the register. A carry (C or SC) status of 1 is returned if the receive operation is successful (that is, the lock bit was initially set), and a status of 0 is returned if the receive operation fails (that is, the lock bit was initially clear, indicating the register contained no valid data to receive). The instructions are `rcv.w` *Ceffa*`,Ak` and `rcv.l` *Ceffa*`,Sk`.
- **lock**—Set the lock bit. The lock operation fails if the lock bit was already set. A carry (C) status of 1 is returned if the lock bit is successfully set (that is, the lock bit was initially clear), and a status of 0 is returned if the lock bit could not be successfully set and the operation fails (that is, the lock bit was initially set, indicating the bit was already locked). The instruction is `lck` *Ceffa*.
- **unlock**—Clear the lock bit. The unlock operation fails if the lock bit was already clear. A carry (C) status of 1 is returned if the unlock operation is successful (that is, the lock bit was initially set), and a status of 0 is returned if the unlock operation fails (that is, the lock bit was initially clear, indicating the lock bit was already unlocked). The instruction is `ulk` *Ceffa*.

- **test**—Read the lock bit into the carry (C) bit. The instruction is `tst` *Ceffa*.

Communication registers are managed in two ways:

- **lock and unlock operations**—These operations execute `lck` and `ulk` primitive operations on structures. The hardware bit is considered a lock bit. When it is set to 1, the register is locked, which means the register is inaccessible or in transition.
- **send and receive operations**—These operations execute `snd` and `rcv` primitive operations on structures. The hardware bit is considered a valid bit. When it is set to 1, the register is receivable, which means the register contains valid data.

With one exception, these two classes of operations, the lock and unlock operations, and the send and receive operations, should not be mixed in the same communication register. The lock and unlock operations may be used to establish appropriate initial conditions for send and receive operations. Otherwise, a `lck` operation and a `snd` and `rcv` operation *should not* be performed together on a communication register, or proper synchronization may not be maintained.

The status returned for instructions such as `rcv.w` *Ceffa*`,Ak` is loaded into the carry (C) or the scalar carry (SC) bit, and the flow of execution may change in the usual manner (using a branch instruction), based on the returned status.

A receive or branch pair of instructions is typically used to pass information with semaphoring. For example, one CPU may wait for data from another CPU by executing the sequence:

```
1$: rcv.w Ceffa,Ak
    bra.f 1$
```

This sequence will fall out of the loop with a successful receive when another CPU executes a `snd.w Ak,` *Ceffa* instruction.

An example of a communication register instruction that uses multiple primitive operations is `inc.w` *Ceffa*`,Ak`. This instruction increments the communication register at address *Ceffa* by the contents of `Ak` if the communication register is receivable (that is, the lock bit is set, indicating valid data has been sent). This instruction is implemented internally with a receive, add, and send combination, with the add and send primitives executed only if the receive primitive succeeds.

Locking memory structures

The locking operations provided in the communication registers are used to synchronize structures located in memory. Since the communication register and memory pipes are disjointed, memory must be synchronized by software before the communication register lock can be manipulated.

For example, the lock bits on two communication registers can be used when passing valid data between two or more CPUs operating in a producer-consumer relationship. The following code sequence, while appearing to be correct, actually contains a memory synchronization problem, discussed in the text following the example.

```
Producer                    Consumer

P1: lck 0x8000              C1: ulk 0x8001
    bra.f P1                    bra.f C1
    (store to memory)           (load from memory)
    lck 0x8001                  ulk 0x8000
```

This code sequence was adapted from Dykstra [Dykstra, E. W. "Co-operating Sequential Processes," *Programming Languages*, edited by F. Genuys. 1968. Academic Press. pp. 43-112.]. `L(8000)` and `L(8001)` control two critical regions, within which the communication occurs. Only one producer or one consumer critical region may be active at a time.

When `L(8000)` is unlocked, the producer is free to store new data. The producer cannot store new data until the consumer has loaded the data previously stored by the producer. When `L(8001)` is locked, the consumer is free to load new data. The consumer cannot load new data until the producer completes storing the new data. The initial conditions are that `L(8000)` and `L(8001)` are unlocked. These conditions force the consumer to wait for the producer to store data to memory.

The producer enters its critical region by locking `L(8000)`, producing (stores) its data, and locking `L(8001)` to indicate that data has been produced. The consumer, which may have been spinning at `C1` while waiting for the producer to store data to memory, is now free to enter its critical region and consume (load) the data. When the consumer completes loading of the new data, it unlocks `L(8000)` to signal the producer to produce new data.

In the previous code, however, the consumer could see the lock set on the communication register located at `L(8001)` and load old data from the memory system before the producer's memory

store reached the memory system. Likewise, the producer could see the lock clear on the communication register located at `L(8000)` and store new data to the memory system before the consumer's memory load was performed.

To remedy this problem, `msync` instructions must be inserted just *after* the memory operations. The `msync` instruction waits for the CPU to complete all store operations to memory, and for all data from load operations to arrive from memory. The correct code is shown in the following code sequence:

```
     Producer                Consumer

P1: lck 0x8000    C1: ulk 0x8001
    bra.f P1          bra.f C1
    (store to memory) msync (C3400/C3800/C4600 only)
    msync             (load from memory)
    lck 0x8001        msync
                      ulk 0x8000
```

If this is a single producer-consumer data transfer, the consumer does not need to synchronize memory, since the producer will not be trying to re-execute the `lck 0x8000`. When the consumer sees `L(8001)` set, the producer's store has reached memory (the producer executed an `msync`) and the consumer can load the data the producer stored.

If, however, the producer code is continually sending data to the consumer through the same memory locations whenever the lock bit at `L(8000)` is clear, the consumer code must perform an `msync` instruction before unlocking `L(8000)`. By unlocking the communication register lock bit at `L(8000)`, the consumer informs the producer that loading from memory is complete, so the producer may store more data in memory.

On the C3400, C3800, and C4600 series CPUs only, a second `msync` must be inserted in the consumer code. This `msync` is required to purge shared data from the data cache. Without the `msync`, the load from memory could read stale data from the CPU's data cache, instead of the correct data which resides in memory.

Multithreaded execution

A thread is any single-instruction stream executing within a process. Multithreaded execution occurs when more than one CPU is executing on behalf of the same process. The ASAP scheduling mechanism used by the system to implement multithreaded execution is called forking.

The ASAP mechanism performs two major tasks: it divides the workload and allows programs to exploit process multithreading. A process that takes ten seconds of CPU time will take ten seconds of wall clock time on a single CPU (assuming the process is allowed to execute without intervention). Under ideal conditions, if two CPUs are available and the work is equally divided, each half of that same process will execute in five seconds of wall clock time. When programs exploit multithreading, processes run on one or any number of processors with no software modification.

The multithreading mechanism is closely associated with the operating system. The hardware allows the operating system to observe and maintain some control over the thread, since the operating system must also schedule threads to perform process multiplexing. The control instructions can create and terminate threads without operating system involvement, and vice versa, since these instructions are not privileged.

The multithreading mechanism is a software and hardware interface, not just instructions for the operating system. Both sides of the hardware and software interface have protocol that must be observed to ensure that parallel execution functions correctly.

In order for the multiprocessing management system to support upwardly compatible software, each CPU within a complex operates as an independent CONVEX 64-bit supercomputer. All CPU events generated locally, such as system calls or page faults, are processed within the CPU that initiated those events. Externally generated events, such as interrupts, are delivered to any available CPU that is currently accepting interrupts. This permits each CPU to execute independently of every other, without requiring a master CPU.

CPU states

A CPU within the C-Series architecture functions in one of two states:

- **Allocated**—A CPU is executing a thread within a process
- **Idle**—A CPU is attempting to find a posted fork and creates a thread of execution within a process

An allocated CPU always contains valid thread state (statefull) and requires all local CPU states to be saved prior to preempting the currently executing thread on the CPU. Idle CPUs are always stateless, and require only that a CPU state be loaded to begin execution of a thread.

CPU scheduling

A CPU is scheduled using one of two independent types of scheduling in order for a CPU to begin execution on behalf of a process. The first type is the transition of the CPU from idle to allocated (thread creation). The second type is the transition from executing on behalf of one process to executing on behalf of another (context switching).

The hardware communication registers provide these two transition types. All process context necessary for a thread's execution becomes available to a CPU when a communication register set is bound to it. The fundamental action of binding a CPU to a communication register set establishes a process context for creating a thread of execution on behalf of a process. A communication register set is bound by loading the communication index register located in the CPU with a communication register set index. Once the CIR is loaded, it immediately shares all process context with any other CPU whose CIR contains the same index.

CPU allocation and deallocation

In the C-Series architecture, the act of allocating another CPU to initiate an additional thread on behalf of a process is called forking. The fork event allows a user to post the need for a CPU to execute on behalf of a process (create a thread), clear the need for a CPU, or force the current CPU back into the idle state (terminate a thread). C3200 C3200

The fork event has two possible states: cleared or posted. Posted means there is a current need for another processor to begin a thread. Cleared means there is no pending work to be done.

A CPU posts a fork when more CPUs can assist with the parallel execution of a process. A posted fork means that a CPU *requests* assistance from other CPUs. If there are no available CPUs, the posting CPU does not wait until another CPU becomes available. The CPU continues with its thread of execution. This mechanism allows a parallel process to execute as a single thread if only one CPU is available. Refer to the subsequent sections on forking operations for more information.

All user CPU management functions are supported by the following five instructions:

- `pfork`—Post a fork event (request allocation of a CPU)
- `spawn`—Post a fork event for multiple CPUs (request allocation of CPUs)
- `cfork`—Clear a fork event (clear need for CPU allocation).
- `wfork`—Wait for a fork event (deallocate CPU and wait for a `pfork`)
- `join`—Wait for a fork event if the executing thread is not the last thread (conditional deallocation of a CPU)

A CPU can post two types of forks by using the preceding instructions. The first type is a request for a single CPU to initiate a thread and is posted by the `pfork` *effa*`,Ak` instruction. The second type is a request for all available CPUs to initiate threads and is posted by the `spawn` *effa*`,Ak` instruction. These instructions load a group of communication registers (the fork event registers) with enough process state to start a thread. This state consists of a PC from which to execute, an initial value of PSW, and stack, frame, and argument pointers to define local memory structures.

The communication register addressing is based on a CIR index, so the fork is posted relative to a particular process. Idle CPUs scan through the fork event registers in each CIR as a function of

the CPU idle loop, looking for a posted fork. If a posted fork is found, the idle CPU binds itself to that CIR by loading the state contained in that CIR's fork event registers into its own CPU registers. If the fork was posted with `pfork`, the fork is cleared. If the fork was posted with `spawn`, it is left posted in the fork event registers for other available CPUs to take.

The `cfork` instruction explicitly clears a fork posted by a `pfork` instruction or a `spawn` instruction. However, proper synchronization between threads may not be maintained if a fork is explicitly cleared with `cfork` and not `join`.

At the end of thread execution, each CPU may terminate its thread (relinquish and deallocate the CPU). There are three instructions to do this—`wfork`, `join`, and the privileged `idle 5k` instruction. Refer to the descriptions of forking operations in this chapter for a discussion of mixing multithreading "models," that is, symmetric and asymmetric. These instructions are described in detail in subsequent sections.

The `wfork` instruction terminates a thread begun with the acceptance of a fork posted through `pfork`, that is, a single thread of execution. The CPU returns to the idle state, where it looks for more posted forks in other CIRs.

Forks posted through `spawn` should be terminated with `join`. If the processor is not the last thread to reach the `join`, the CPU returns to the idle state. If it is the last thread, execution continues at the instruction following the `join`. The process continues executing as a single thread after the `join` instruction executes.

These CPU control instructions allow a process to allocate and deallocate CPUs without operating system intervention. The operating system is involved only when the user requires an operating system service. It does not intervene when a CPU joins a process that has executed a `spawn` or `pfork` instruction, executes the thread, and then returns the CPU via `join or wfork`.

This mechanism provides a fast CPU allocation/deallocation scheme that permits the parallelization of small code regions. If at any time one of the CPUs within the process requires operating system services, only that CPU must incur the overhead required for such a request, not the entire process.

ConvexOS/Secure

The ConvexOS/Secure operating system includes additional control over the creation of a thread. When an additional thread is created, the vector valid (VV) flag is cleared.

If the ConvexOS/Secure operating system determines that a vector valid trap has occurred, the operating system reserves the address and scalar registers, in addition to reserving the vector registers, to the user process. The address, scalar, and vector registers for the created thread are initialized to a known state.

For more information about the VV flag and the vector valid trap, refer to Chapter 6, "Exceptions and interrupts."

Parallel processing

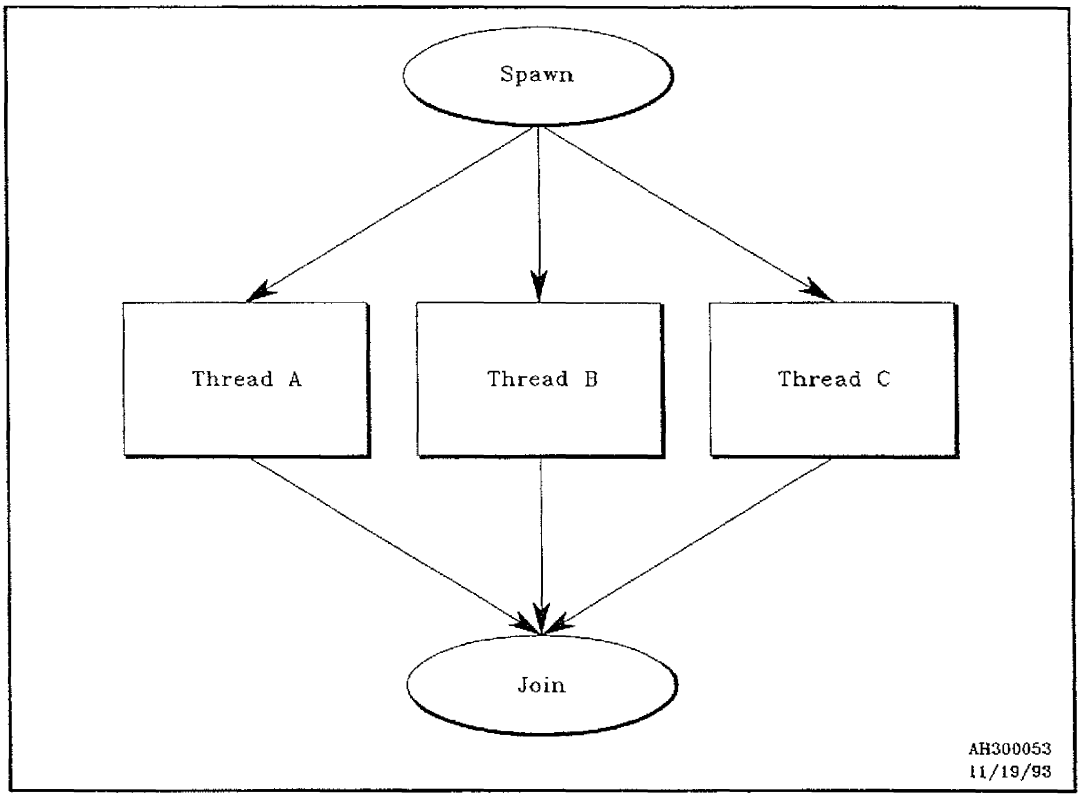
The forking instructions provide two types of parallel processing.

Symmetric parallel processing

The `spawn` instruction is used to implement *symmetric parallel processing*, in which threads in a process execute the *same* instruction stream. Upon completion, each thread executes a `join` instruction that forces a multithreaded process back to a single thread.

Figure 68 shows the concept of a symmetric parallel process.

Figure 68 Symmetric parallel processing



In Figure 68, a single process transitions from sequential (a single thread) to parallel (multithreaded) and back to sequential. The initial single thread posts a need for more threads to enter the process by executing a `spawn` instruction. Each thread that enters the process in response to the `spawn` leaves the fork posted.

The code between the spawn and join instructions must be such that all necessary operations are completed, regardless of the number of threads actually created.

When the first thread completes its job, it executes the join instruction. The join instruction marks the fork event registers by setting fork.type to STOPPED. This indicates that no more threads are required, so that no additional threads will accept the fork. Successive join instructions eventually bring the process back to a single thread, when the last thread executes a join instruction which clears the fork. A symmetric process using the symmetric processing paradigm that is shown in Figure 68 is also shown in Figure 69.

Figure 69 Example of a multithreaded symmetric process

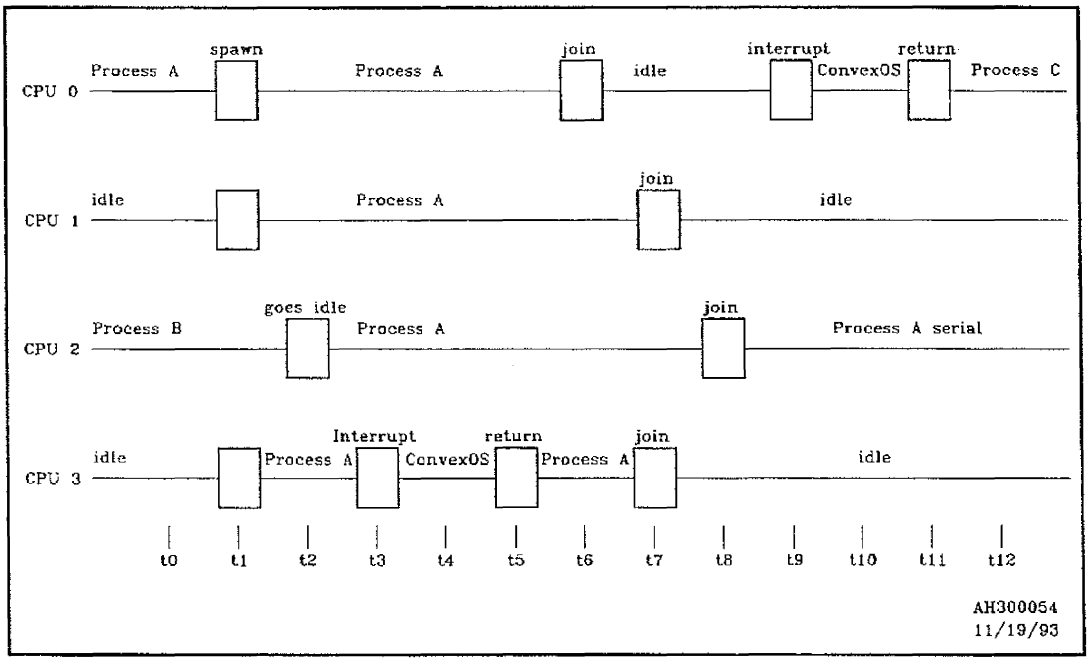


Figure 69 shows one possible time sequence for a parallel region of code executing in a four-CPU complex.

At time t0, process A is executing a single threaded process on CPU0 and a second process, B, is executing on CPU2. At t1, process A executes the spawn instruction, which is immediately accepted by CPUs 1 and 3, which were idle, and process B performs a normal termination, which idles CPU2. At t2, CPU2 also enters process A. At t3, CPU3 accepts an interrupt, suspends its thread of execution, enters ring 0, and executes operating

system code. After processing the interrupt, CPU3 is returned to process A at `t5`. At `t6`, the thread executing on CPU0 completes and performs the first join, followed by CPUs 1 and 3 at `t7`. At `t8`, CPU2 completes the execution of the last thread in process A, executes the `join` instruction, and resumes execution of the serial thread of process A. At `t9`, an interrupt occurs and frees process C to resume execution. The interrupt is taken by idle CPU0, and process C is resumed at `t11`.

Asymmetric parallel processing

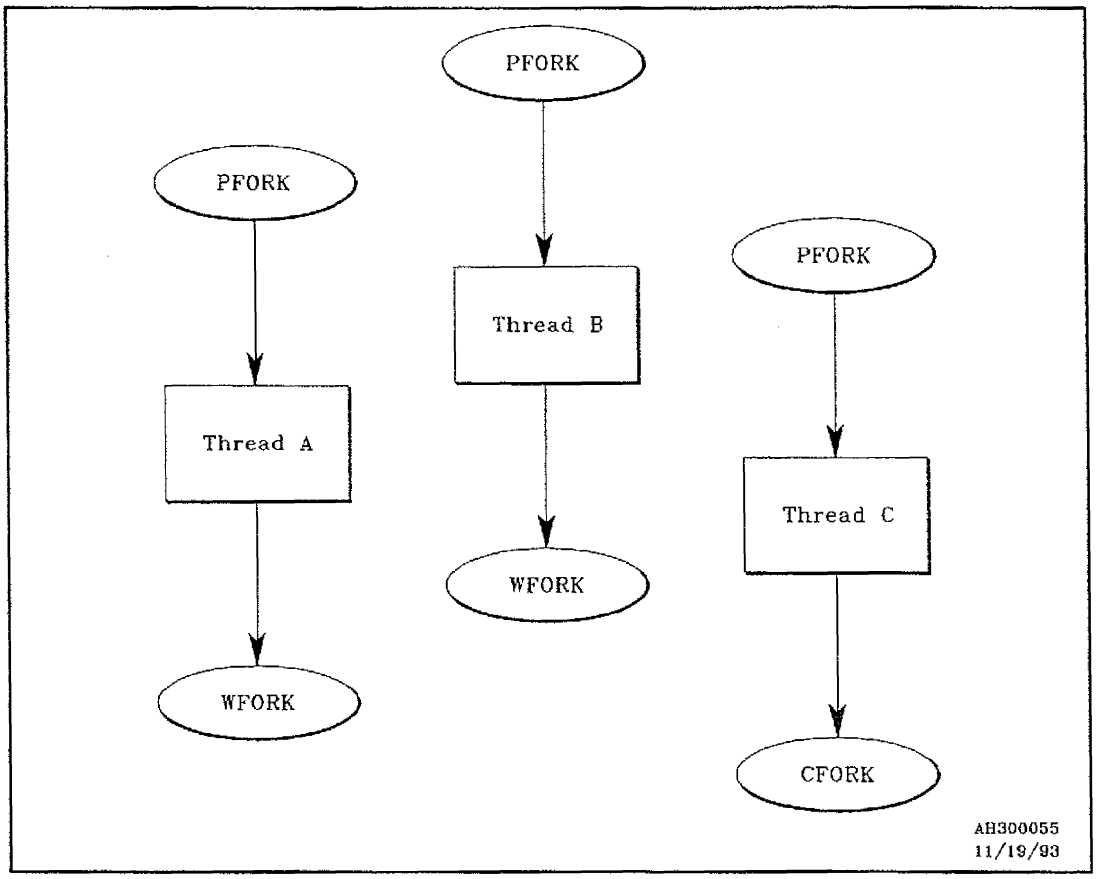
The `pfork` instruction is used to implement *asymmetric parallel processing*, which creates an additional thread when another CPU takes the fork. Each thread in an asymmetric process executes a different instruction stream from the other threads in the same process. Multiple threads within a process execute different functions by creating a single additional thread of execution analogous to the `fork` system call. The child thread may or may not communicate with the parent. The parent thread may terminate while leaving the child thread in execution. Either thread may fork additional threads.

An asymmetric process is forced back to a single thread with the `wfork` or `cfork` instructions.

In general, asymmetrical threads are disjointed, executing different code streams with different data, but within the same process address space. Asymmetric processing differs from the multithreaded execution in symmetric processing in that the posting thread usually requires another thread to accept the fork to perform the specific task, then communicates with the created task. An asymmetrical thread is initiated with a CPU requesting the execution assistance of *one* other CPU as opposed to a CPU requesting execution assistance of *all* available CPUs as implemented by the `spawn` or `join` instruction pair.

Figure 70 shows a three-threaded asymmetric process.

Figure 70 Asymmetric parallel processing



In Figure 70, thread B posts the need for an independent (disjointed) thread to execute concurrently. Thread A is started, and notifies B that it is executing when the microcode sets the lock bits on the fork event registers. Thread B completes execution and terminates by relinquishing the CPU (`wfork`) after determining that other threads in this process are still executing (threadcount > 1). Thread A posts the need (`pfork`) for a new thread C to perform another task. Thread C starts, posts a fork, and notifies thread A that it is executing. Thread A completes execution (`wfork`) and terminates in the same manner as thread B. Thread C determines all other threads have completed execution and examines the fork event registers to check the status of the fork that it posted. The fork was never taken, so

thread C clears the need for another thread by clearing the fork (`cfork`), and continues on as a single-threaded process.

As indicated in Figure 70, the asymmetric processing paradigm requires additional software overhead to establish thread synchronization and communication than does symmetric processing (`spawn` or `join`). The asymmetric processing paradigm assumes that each child thread communicates all status information with its parent and each parent thread communicates all status information with its child via send or receive operations. In addition, all threads that post a fork via `pfork` are assumed to check the status of the fork, and clear the fork that it posted upon termination with `cfork` if that fork was not taken by another CPU. Otherwise, all parent threads are assumed to terminate with `wfork` if, upon examination of the fork event registers, it finds that the posted fork was taken.

Privileged CPU control operations

The C-Series architecture defines several privileged instructions associated with the CPU control functions that manage the execution of a C3200 Series complex. These privileged instructions are:

- `mov CIR,Sk`—Moves the contents of the CIR into Sk. This instruction enables the operating system to determine the current communication register set binding. It is not privileged, but is included in this section since it is most often used by the operating system.
- `mov Sk,CIR`—Moves the communication register set index in Sk into the CIR. It enables the operating system to transition the current CPU context from one process to another by switching hardware communication register sets.
- `ldcmr` *effa*`,Ak`—Loads the communication register set indexed by Ak from memory.
- `stcmr Ak,`*effa*—Stores the communication register set indexed by Ak to memory.
- `idle Sk`—Forces a CPU to enter the CPU idle loop to search for another posted fork. This instruction is privileged and is used by the operating system (ring 0).

The operating system uses the `ldcmr` *effa*`,Ak` and the `stcmr Ak,`*effa* instructions to context switch a communication register set by saving and restoring the communication register set to and from memory. To ensure that the register set is atomically saved, the operating system makes the thread count unreceivable, which prevents more forks from being taken and forces the process to become single threaded.

Forking operations

This section explains the forking instructions (`pfork`, `spawn`, `cfork`, `join`, `wfork`, `idle`) and the microcode idle loop in detail, providing information to augment the descriptions of these instructions in the *CONVEX Assembly Language Reference Manual (C Series)*.

These instructions all perform basic operations on the fork event registers. Figure 54 and Figure 55 show the format of the fork event registers.

The lock bits, `forklck` and `forkposted`, are used to control access to the fork event registers between CPUs that may be attempting to post or accept forks at the same time. These two lock bits operate as a two-part semaphore.

CPUs attempting to post forks must first successfully lock the top, (`forklck`), write fork state into the communication registers, then lock the bottom, (`forkposted`). The locking operations performed by the `pfork` and `spawn` instructions are a `snd` operation to `forklck`, and, if successful, a `snd` operation to `forkposted`. CPUs attempting to accept forks must first unlock the bottom (`forkposted`), read all the fork state from the communication registers, then unlock the top (`forklck`).

The microcode idle loop uses forks as a way of searching the CIRs for posted fork events. This is true, in some cases, for the `wfork` and `idle` instructions, described later in a subsection. The locking operations performed during fork acceptance are a `rcv` from `forkposted` and, if successful, fork a `rcv` from `forklck`. This protocol makes the fork event registers operate like a queue, that is, they are written from the `forklck` end and read from the `forkposted` end.

The four possible states of the two lock bits are:

- `forklck` = 0, `forkposted` = 0 — No fork posted, ready for forks to be posted
- `forklck` = 1, `forkposted` = 0 — In transition in one of two ways:
  - **CPU beginning to post a fork** — Any other CPU attempting to post will fail, any CPU attempting to receive will fail until the fork is completely posted.
  - **CPU beginning to accept a fork** — Any other CPU attempting to post will fail until the fork is completely accepted, any CPU attempting to receive will fail.

- `forklck = 1, forkposted = 1` — Fork posted ready for acceptance. Any other CPU attempting to post will fail, and any CPU attempting to receive will succeed.
- `forklck = 0, forkposted = 1` — Undefined state

The thread allocation register contains the allocated thread count and thread allocation mask. It is manipulated as part of fork acceptance. The thread count is incremented each time a thread is created by fork acceptance, decremented each time a thread is terminated, and it may range in value from 0 to 32. The thread count allows the microcode to determine whether the thread is the last in the process. This directly affects the operation of the `wfork` and `join` instructions. It also allows the operating system to determine how many threads are currently in existence.

The thread mask is a 32-bit mask with each bit corresponding to one of the possible threads in a process. The least-significant bit corresponds to thread 0, the next to thread 1, and so on up to thread 31.

If a bit in the thread mask is set, the corresponding thread may be allocated by the microcode when creation of a thread is required by a process. If the bit is clear, the thread is already allocated so microcode may not allocate it again.

A `snd` or `rcv` locking protocol governs the thread mask and thread count. This locking protocol forces the microcode or the operating system to receive the registers before manipulation and send them back when the manipulation is complete as part of the process to examine or modify these registers.

The microcode performs the following actions when attempting to create a thread:

1. The mask or count is received.
2. The mask is searched for the least significant set bit, using a trailing zero count operation.
3. If there are no set bits, there are no threads available for creation. The remaining steps are skipped. If a set bit is found, it is cleared, marking it allocated.
4. The bit index of the set bit is written to the CPU thread ID (TID) register.
5. The thread count is incremented.
6. The thread mask and thread count is sent back to the communication registers.

When a thread is terminated, the following occurs:

1. The mask or count is received.
2. The contents of the TID register are used as a bit index to set a bit in the thread mask.
3. The thread count is decremented.
4. The thread mask and thread count is sent back to the communication registers.

The operating system can block any further creation of threads in a process by receiving the thread mask or count in that CIR, clear the entire thread mask, and send it back. Since the thread mask has been cleared, no more threads can be allocated.

Forking commands

The following commands are used to initiate and control forking operations.

pfork <*effa*>,Ak
This instruction posts a fork for a single CPU to accept if available. The new child thread begins with a PC value of *effa*, using a stack pointer specified in `Ak`, and inheriting the PSW, FP, and AP from the parent thread. First, the current frame pointer and argument pointer are assembled into a longword and sent to the `fork.FP/fork.AP` fork event register, attempting to lock `forklck`.

If this send operation fails, a fork has already been posted, so carry (C) is cleared, indicating failure and the instruction is complete. This failure status allows the thread to determine that it was unable to post a fork.

If the `snd` succeeds, posting continues by assembling the *effa* from the instruction and the current PSW into a longword and putting it into `fork.PC/fork.PSW`. Next, the current program counter (PC) is `put` into `fork.source_PC`. This action sets the ring bits of the PC when the fork is accepted. Finally, the constant PFORKED is concatenated with `Ak` for assembly into a longword, and sent to `fork.type/fork.SP`, setting the lock on `forkposted`. The return status on this `snd` is not checked since it always succeeds. The success status from the first `snd` is returned in carry (C), signaling that the `pfork` instruction successfully posted the fork.

spawn <*effa*>,Ak
This instruction works identically to the `pfork <effa>,Ak` instruction except the constant SPAWNED is put in `fork.type`. This lets accepting CPUs know the fork was spawned and is intended for multiple CPUs if any are available.

cfork
The `cfork` instruction explicitly clears a posted fork, that is, it removes it from the fork event registers without accepting it and creating a new thread. This instruction first receives the value in the `forkposted` register. If this `rcv` operation fails, there is no posted fork to clear, so the `cfork` instruction returns a failure status of 0 in carry (C). If the `rcv` succeeds, this success status is returned and the forklck is also received, making the status of the fork event register lock bits `forklck` = 0, `forkposted` = 0.

wfork
The `wfork` instruction terminates the current thread of execution and returns the CPU to the idle state. First, the `wfork` attempts to receive the thread mask or count. If the `rcv` operation fails, the `wfork` cannot continue since the mask or count is required to deallocate the current thread. The `wfork` instruction does not wait for it because the CPU would not be able to take interrupts, since interrupts are only delivered to the CPU at instruction dispatch boundaries. The microcode restarts the `wfork` instruction until the `rcv` operation succeeds.

When the `rcv` eventually succeeds, the thread count is checked for a value of 1. This means the terminating thread is the last thread in the process, so the current CIR is checked for a posted fork. If the posted fork is of type PFORKED, it is taken.

Although a thread has been deallocated and reallocated, the microcode still accepts the fork in the current CIR as the current TID. The specifics of fork acceptance are discussed in the "Idle CPU allocation" section on page 203.

If a posted fork of type SPAWNED or STOPPED is found, it means the thread was started with a `spawn`, and should have been terminated with a `join`. Since the `spawn` and `pfork` instructions should not be mixed in the same process without proper synchronization, a deadlock is reported to the operating system through a system exception. Deadlocks are described in the "Process deadlock" section on page 206. If there is no fork posted at all, the process has ended erroneously, so a deadlock trap is invoked.

If the thread count > 1, it means a child thread is terminating correctly. The thread count is decremented and the bit in the thread mask corresponding to the current TID is set, indicating the terminating thread is now available for allocation. The thread mask/count are sent back to the communication registers, and the CPU enters the microcode idle loop.

join
The `join` instruction reduces the process to a single thread of execution. All threads in the process that have accepted the spawned fork reach a `join` at the execution termination. Each CPU either terminates its current thread of execution and returns to the idle state or continues execution after the `join` as a single threaded process.

The spawn/join mechanism may be viewed as a race from `spawn` to `join`, where the first *N*-1 threads to reach the `join` terminate, and the *N*th thread continues executing after the `join` instruction. Once one thread has joined, any idle CPUs do not accept the fork, since the region of code between the `spawn` and `join` instructions has been completed by one thread and should soon be completed by all threads.

First, the `join` instruction attempts to receive the thread mask/count. If the `rcv` fails, the instruction is restarted as described in the "`wfork`" section. When the `rcv` succeeds, the CPU waits for all stores in the current CPU to reach the memory system by performing an `msync` operation. This ensures that the single thread continuing after the join will see all operands stored by all spawned threads if the single thread loads these operands. Next, the constant value STOPPED is sent to the `fork.type` for event register. This value signals other threads within the process that at least one thread has reached the `join` instruction, meaning the process is reducing to a single thread which prevents other CPUs from accepting the fork. Next, the thread count is checked for a value of 1. If the thread count > 1, a spawned thread is correctly terminating. The thread count is decremented and the bit in the thread mask corresponding to the current TID is set, indicating an allocatable thread. The thread mask/count are sent back to the communication registers, and the CPU enters the microcode idle loop. If the thread count is 1, it means this thread was the last to reach the `join` so the thread mask/count are sent back to the communication registers and the thread continues after the `join` instruction.

idle Sk
The privileged `idle Sk` instruction allows the operating system to send a CPU to the idle state. The idle state is a microcode loop that searches all CIRs for posted forks.

The idle loop is the common end of the `wfork`, `join`, and `idle Sk` instructions. The `wfork` and `join` instructions search the current CIR for a posted fork before idling. The `idle Sk` instruction searches the CIR specified by `Sk` before idling, and does not deallocate the thread the `idle` instruction executed from.

The following steps describe the `idle` instruction:

1. The CIR is switched to the value specified in `Sk`.
2. The `idle` instruction attempts to receive the thread mask and thread count. If the `rcv` operation fails, the idle loop is entered.
3. The thread mask is searched for the leftmost set bit, indicating an allocatable thread. If no bits are set, the thread mask and thread count are sent back to the communication registers and the idle loop is entered.
4. If an available thread is found, the CPU attempts to accept a fork by first receiving `forkposted`. The specifics of fork acceptance are discussed in the "Idle CPU allocation".
5. If a fork of type STOPPED is found, then the process in that CIR is attempting to join, so the fork is ignored, the thread mask or count is sent back, and the idle loop is entered.
6. If a fork of type SPAWNED or PFORKED is found, it is taken in the new thread. The updated thread mask and thread count (thread allocated and count incremented) are sent back to the communication registers and execution begins at the PC of the fork.

Idle CPU allocation

The idle state of a CPU consists of a microcode sequence referred to as the `idle loop`.

Note

The following idle loop sequence details the implementation on the C3200 Series CPUs as an example of idle loop processing. Idle loop processing is not guaranteed to be identical to this for all multiprocessing CPUs. The exact ordering of the idle loop is implementation specific, not architecturally defined.

The algorithm loops sequentially through all CIRs in search of a posted fork. There are four conditions that must be met in order for a fork to be taken. The C3200 Series CPUs check these conditions in the following order:

1. The thread mask/count communication register must be receivable, that is, have its lock bit set.
2. The thread mask must be nonzero, indicating that at least one thread is available for allocation.
3. There must be a fork posted, that is, `forkposted` must be locked.
4. The posted fork must not be of type STOPPED.

If any of these conditions are not met, the CIR is skipped and the next CIR is searched.

If these conditions are all met, the fork is accepted in the following manner:

1. The successful `rcv` of `forkposted` not only shows the availability of a fork but also fetches the `fork.type` and `fork.SP`.
2. The CPU's address register A0 (stack pointer) is loaded with `fork.SP`, and `fork.type` is checked to ensure it is not of fork type STOPPED.
3. If `fork.type` is a STOPPED type, the process running in the candidate CIR is attempting to `join` to a single threaded state, so the fork is ignored.
4. Assuming the fork is not ignored, the `fork.source_PC` is fetched with a `get` and loaded into the program counter. This sets the ring bits for the new thread of execution.
5. The `fork.PC` and `fork.PSW` are fetched with a `get` operation and loaded into the program counter and PSW. The PC is loaded the second time with the least significant 29 bits only, that is, the ring bits are not loaded.

   To understand why the program counter is loaded twice, consider the case where a `pfork 0,sp` is executed in ring 4. If a `fork.PC` of value 0 was simply loaded into PC by the accepting CPU, an illegal entry into ring 0 would be implied. To avoid an illegal ring 0 entry, the program counter of the posting thread is written to the `fork.source_PC` to set the ring bits (establishing a current ring of execution) for the accepting CPU. This makes the posting and acceptance of forks consistent with the `jmp` instructions.
6. If the fork is of type SPAWNED, the accepting CPU's frame and argument pointers are loaded with a `get` from `fork.FP` and `fork.AP`. The `forkposted` is locked with a `lck` operation, leaving it posted.
7. If the fork is of type PFORKED, the accepting CPU's frame and argument pointers are loaded with an `rcv` of `fork.FP` and `fork.AP`. This `rcv` clears `forklck`, which clears the fork.
8. A thread is allocated as described earlier, and the thread mask/count is sent back to the communication registers. The new thread begins execution at `fork.PC` in the ring of `fork.source_PC`.

Fork acceptance is one of two events that can forces a CPU to leave the idle state. The second event is an I/O interrupt. An idle CPU is always able to respond to interrupts. Interrupt processing is fully described in Chapter 6, "Exceptions and interrupts."

The CPU microcode idle loop employs an equitable *round-robin* scheduling algorithm. CPUs take forks from different processes (CIRs) by binding to the interrupt service process context (setting the CPU's CIR index equal to the interrupt CIR index) and looping through the other communication register sets using physical communication addressing.

If the idle CPU always began searching for forks in the fork event registers at CIR = 0 and progressed sequentially through to CIR = 7, the lower CIR index values would be treated more favorably. To circumvent this inequity, each time a CPU accepts a fork, it saves the CIR index from which it accepted the fork. The next time that CPU goes idle, it begins searching at the next one.

The CPU checks for interrupts after one complete pass of the eight CIRs. If there are no pending interrupts, another pass of the CIRs begins. The interrupt CIR (ICIR) must be entered during the idle loop to give the idle CPU enough context from which to take an interrupt or `trap` instruction trap.

CPU deadlock detection

C3200 Series CPUs (only) are capable of detecting when the currently executing threads within a process have reached a deadlock condition. A deadlock occurs when all the currently executing threads of a process are performing a "synchronization" instruction followed by a branch back to that instruction. Synchronization instructions are those instructions attempt to change the value of a lock and return status on the success or failure of that operation. Examples of such instructions are the `tas`, `snd`, `rcv`, and `inc` instructions.

Deadlocks are system exceptions and pass through page 0 of ring 0 to the process deadlock handler. The system then determines if any other threads within the process can be run and schedules them accordingly.

Process deadlock

A *process deadlock* usually occurs when all threads of an executing process are in a synchronization instruction sequence. When any of the deadlock detection instructions are followed by a branch back to the same instruction, they have the potential of triggering a process deadlock. The group of instructions listed in Table 41 are classified as synchronization or deadlock detection instructions.

Table 41 Deadlock detection instructions

Instruction	Description
`casr`	Compare and swap a word between a resource structure and memory
`getr`	Get contents of resource structure into a register
`inc`	Increment a communication register
`incr`	Increment the data field of a resource structure
`lck`	Lock a communication register
`mat`	Compare an address register with a communication register
`matr`	Compare a register with the contents of a resource structure
`popr`	Pop an address register off a resource structure
`pshr`	Push an address register onto a resource structure
`putr`	Copy contents of a register into a resource structure
`rcv`	Receive a value from a communication register

Table 41 Deadlock detection instructions

Instruction	Description
`rcvr`	Receive a value from a synchronized resource structure in memory
`snd`	Send a value to a communication register
`sndr`	Send a value to a synchronized resource structure in memory
`tas`	Test and set a memory byte
`tac`	Test and clear a memory byte
`ulk`	Unlock a communication register

Note

C3800/C4600 Series CPUs do not implement hardware-detected deadlock on these or any other instructions.

Synchronization instructions are those instructions that attempt to set the value of a lock and return status on the success or failure of the lock. All data representations of these basic instructions are implemented in the deadlock detection instructions, for example, `snd` includes `snd.w` and `snd.l`.

Note

These instructions all perform some type of semaphore or synchronization operation and return status in carry (C) or scalar carry (SC).

Whenever a CPU executes one of these instructions *immediately* followed by a branch back to the same instruction (that is, the same op code at the same program counter where the branch displacement must be the negative of the size of the synchronization instruction), the thread is deadlocked. If all threads currently executing in a process are deadlocked, the entire process is deadlocked.

When a deadlocked process is detected, each thread within the process immediately enters the ring 0 process deadlock handler pointed to by location `0000 0010` of ring 0 page 0. The process deadlock handler schedules other threads within the process to resolve the deadlock condition. An example code sequence would be:

```
1$: rcv.w    0x8000,a2
    bra.f    1$
```

If the `rcv` instruction fails in this code sequence and returns a carry (C) of 0, a backward branch is taken. The deadlock handler is dispatched, instead of retrying the `rcv` operation.

The concept of deadlock also extends to certain cases of thread termination and fork acceptance. Deadlock is detected if the last thread in a process attempts to terminate, or a thread that should have executed a `join` executes a `wfork` instead. The process deadlock mechanism is used with a separate qualifier code to notify the operating system of these cases.

Specifically, if the last thread in a process executes a `wfork` instruction (that is, the entering thread count is 1) and a fork is not posted in the CIR, a last thread termination deadlock is signaled. However, if a fork is found posted by the last thread executing a `wfork` instruction, and the fork is a STOPPED or SPAWNED type, the process executed a `spawn` or `join` instruction pair mixed with a `wfork` instruction without proper synchronization.

The last thread of a process should never execute a `wfork` since the process cannot continue. The acceptance of a fork in the current CIR is provided as a last opportunity to avoid deadlock, but if the fork is of the wrong type, it still causes deadlock.

Exceptions and interrupts 6

The C-Series architecture allows the operating system with a means to control process disruptions, and other asynchronous events that require changing the explicit flow of control. These events are called exceptions. An *exception* is an event that disrupts the execution of a thread, process, system, CPU, or CPU complex.

Note

Unless specifically stated otherwise, descriptions of exceptions and exception processing are applicable to all implementations of the C-Series architecture.

Exception system

Exceptions occur as a result of some asynchronous event that disrupts an executing process or thread, such as arithmetic inconsistencies or address translation faults. They also occur within an executing process, such as deadlock or execution of a *trap* instruction. An exception event results in the transfer of control to a predefined routine known as an *exception handler*. The starting addresses of the exception handlers are located in jump tables in reserved virtual memory referred to as "virtual memory page 0." Process and system state information is saved on the appropriate stacks.

The primary objectives of exception processing for the C-Series architecture are

- Involvement of the operating system (OS) kernel is kept to a minimum.
- The hardware is structured so that the exceptions are as asynchronous OS kernel calls.
- The hardware indicates the cause of the exception, if possible.
- Some exceptions that are under user control can be masked out by hardware.

The primary objectives of exception processing for the C-Series multiprocessing implementations only are

- Only the single thread causing an exception in a CPU complex is involved in that thread's exception processing. Other threads within the process may continue execution.
- Exceptions in each exception class can be one of the following:
  - **Local**—Local exceptions are related to the currently running thread within a process and may be handled with an exception handler in that process.
  - **Global**—Global exceptions are related to every thread within a process and must be handled by the operating system in ring 0.

The C-Series architecture exceptions are grouped into one of following three classes:

1. Process exceptions
   - **For all C-Series CPUs**—These exceptions belong to an executing process and may be handled with an exception handler in that process. The exception handler is in the current ring of execution.
   - **For multiprocessing C-Series CPUs**—Each exception handler is invoked by the thread causing the exception in the current ring of execution.
2. System exceptions
   - **For C100 Series CPUs**—System exceptions cannot be handled by an executing process and require intervention by the kernel executing in ring 0.
   - **For multiprocessing C-Series CPUs**—System exceptions are either local or global and cannot be handled by an executing process. Local system exceptions are thread specific and require explicit operating system intervention. Global system exceptions are associated with an entire process and require intervention by the kernel executing in ring 0.
3. Machine exceptions
   - **For all C-Series CPUs**—Machine exceptions include fatal errors in the system that cannot be handled by operating system software.

In general, when an exception occurs, the value of the program counter (PC) is the address pointing to the next executable instruction that would have been executed if the exception had not occurred. The PC is saved in a return block and pushed on the process stack before the exception handler begins executing. The formats of the return blocks are described in Chapter 4.

If exceptions of different classes are simultaneously pending, machine exceptions have the highest priority, followed by system exceptions, and finally process exceptions. Exceptions may also be subdivided according to how the exception handler normally treats them. In many situations, the exception handler can correct the underlying cause of an exception and signal the original process to resume execution.However, in some situations, the exception handler may not be able to correct the cause of the exception and cannot return control to the original process.

There are two distinct types of exceptions:

- A trap is an exception that occurs at an instruction boundary. All state information necessary to resume execution is architecturally defined and contained in the extended return block that is pushed when the trap occurs.
- A fault is an exception that cannot occur at an instruction boundary. State information, in addition to the extended return block, must be saved in order to later resume execution. Currently, all faults are caused by problems that occur during the virtual-to-physical address translation process for memory and communication register addresses.

A *trap frame* is a stack frame containing an extended return block (extended frame) that is pushed on the process stack as result of a trap. For example, when an arithmetic exception occurs, the hardware pushes an extended frame and jumps to the trap handler. This extended frame is referred to as a trap frame.

Process exceptions

All process exceptions occur at the process level (C100 Series CPUs) or thread level (multiprocessing C-Series CPUs). The user can handle these exceptions without operating system intervention. The exception handler that resides in the current ring of execution is invoked by the process (or thread) that caused the process exception. The stack frames are pushed on the process stack in the current ring before entering the exception handler. In addition, many process exceptions, such as arithmetic traps, can be disabled (masked out). The process exceptions are

- **Arithmetic trap**—This exception type results when a process produces arithmetic errors. The exception handler can return control to a user process after the trap has been processed. This trap can be masked out.
- **Instruction trace trap**—This feature allows a single instruction to execute between each exception.
- **Breakpoint**—This feature uses a breakpoint (`bkpt`) instruction to cause a transfer of control when executed.
- **Process breakpoints**—This feature uses a process breakpoint (`pbkpt`) instruction to cause a transfer of control when executed.

Instruction trace, sequential execution, and breakpoints all support process debugging. The following subsections describe each process exception.

Arithmetic trap

An arithmetic trap occurs when an arithmetic operation encounters or produces an illegal value. An illegal value is one that is not within the representable range of numbers for the machine. Refer to Chapter 2 for a description of the bit representation of arithmetic floating-point operands. The following are characteristics of each type of arithmetic exception:

- **Integer overflow**—This occurs when a result is too large to occupy the specified destination. When an integer overflow occurs, the AIV or SIV bit in the PSW is set. The result loaded into the destination is correct in the least significant bits.

  When an integer overflow exception occurs for integer longword multiplication (64 bits), the result is correct in the least significant 53 bits. Bits <63..53> are undefined.

- **Integer divide by zero**—When the divisor is zero, the processor sets the appropriate divide by zero bit (ADZ or SDZ) in the PSW. The output of the divide is the dividend.
- **Floating divide by zero (native, IEEE)**—When the divisor is zero, the processor sets the FDZ bit in the PSW. The output of the divide is a reserved operand in native mode, or Not a Number (NaN) in IEEE mode.
- **Floating-point overflow (native, IEEE)**—When the resulting exponent requires more positive precision than is allowed, a floating-point overflow occurs. The resultant operand is forced to a reserved operand in native mode, or infinity in IEEE mode. The overflow (OV) bit in the PSW is set.
- **Floating-point underflow (native, IEEE)** —When the resulting exponent requires more negative precision than is allowed, a floating-point underflow occurs. The resulting operand is forced to true zero. True zero is forced regardless of the value of the underflow trap enable bit. The underflow (UN) bit in the PSW is set.
- **Reserved operand (native)**—When an input to a floating-point arithmetic operation has a sign = 1 and an exponent = 0, a reserved operand exception is detected. The fraction value is irrelevant. The output of an arithmetic operation with a reserved operand input is a reserved operand output. A reserved operand output has a 0 fraction. The reserved operand (RO) bit in the PSW is set.
- **NaN (IEEE)**—When an input to a IEEE floating-point arithmetic operation has an interpreted value equal to an IEEE NaN value, an arithmetic trap occurs. The reserved operand (RO) bit in the PSW is set.
- **Infinity (IEEE)**—When an input to a IEEE floating-point arithmetic operation has an interpreted value equal to an IEEE *infinity* value, an arithmetic trap occurs. The reserved operand (RO) bit in the PSW is set.

Enable bits in the PSW can selectively enable groups of arithmetic exceptions. These groups are listed in Table 42. Exceptions may be ignored by clearing the appropriate trap enable bit in the PSW.

Table 42 Arithmetic exceptions and corresponding PSW bits

PSW arithmetic trap enable bit	Trap description	PSW arithmetic exception bit
IVE	Integer overflow trap enable	SIV, AIV
DZE	Integer divide by zero trap enable	ADZ, SDZ
FE	Floating-point trap enable	OV, RO, FDZ
FUE	Floating-point underflow enable	UN
INE[1]	Intrinsic error enable	FIN

[1]Does not apply to C100 Series CPUs.

Once an arithmetic exception bit in the PSW is set, it remains set until cleared by software. This permits the PSW to record the occurrence of an exception that is masked out, but is explicitly tested later.

Floating-point exceptions are serviced by two trap enables, one for underflow and one for all other floating-point exceptions. These two trap enables exist because continued computation may still be possible after an underflow occurs. Underflow forces a true zero result that is sufficient in most circumstances. All other floating-point exceptions force either a reserved operand result (native) or a NaN or infinity result (IEEE). Reserved operands are generally markers for other trap handlers. These two trap enables allow the application programmer to choose the appropriate reaction to a floating-point exception.

The processing sequence for arithmetic traps that prepare a process to enter an exception handler are

1. The CPU sets the appropriate bits in the PSW to indicate the type of arithmetic exception. Since a C-Series CPU has multiple arithmetic units, arithmetic operations can result in multiple arithmetic exceptions. These types of arithmetic exceptions are reflected in the CPU state by setting the respective bit in the saved PSW. Multiple exceptions types can be identified simultaneously, because each exception type has an assigned bit in the PSW.
2. When a CPU detects an arithmetic exception that requires a trap (that is, it is enabled by the PSW), it suspends all pending instructions.
3. Because of the pipelined nature of the machine, more than one instruction may be executing when a trap occurs, so

the CPU completes execution of *all* currently executing instructions.

4. The CPU recognizes the trap only after completing Steps 1 through 3, and only if there are no events pending with a higher priority (such as interrupts).
5. The CPU pushes an extended return block onto the current stack (no ring crossing occurs).
6. The CPU clears the following bits of the newly generated PSW: C, SC, AIV, ADZ, UN, OV, FDZ, RO, SIV, SDZ, FRL, and FIN.
7. Instruction execution for the arithmetic exception trap handler begins at the address contained at byte address `0000 0044` of page 0 of the current ring.

Instruction trace trap

The instruction trace trap allows a single instruction to execute between each trap. Trace traps can be directly controlled by setting the appropriate (implementation-specific) enable bits in the PSW. After the execution of each instruction, the processor pushes an extended return block onto the stack in the current ring. The pushed PC references the next instruction to be executed. The trace trap handler is located at the address contained at address offset `0000 0040` of the current ring. Since the trace trap handler is located in the current ring, no ring crossing occurs and the operating system is not involved in trap processing.

Note

On some C-Series implementations, instruction trace traps fail to occur around certain instructions, causing a failure to *single-step* correctly. C3200 systems cannot single-step through a `sysc` instruction. C3800 systems cannot single-step through a `jump`, `call`, or `return` instruction.

When the PSW 's TR bit is set, instruction tracing is enabled and a trace trap occurs after an instruction is executed. In addition, the PSW's SEQ bit must be set for the instruction trace to function properly. SEQ forces instructions to execute one at a time without overlap.

The following paragraphs describe an additional capability that exists on the multiprocessing CPUs.

The trace thread concurrency bit, PSW (TTC), can be used to monitor all thread creation and termination that occurs within a process as a result of `pfork` or `spawn` instructions when the TTC bit is set. An instruction trace trap occurs after the execution of a `wfork`, `idle`, or `join` instruction prior to the CPU entering the

hardware idle loop. The PC that is pushed on the stack references the next instruction to be executed, (the instruction after the `wfork`, `idle`, or `join` instruction). The thread is not deallocated by the `wfork`, `idle`, or `join`, because the trap handler must have a thread identification in order to actually process the trap.

Before the trace trap handler returns, it is expected to backup the PC in the extended return block to the `wfork`, `idle`, or `join` instruction and re-execute the instruction again with TTC cleared (to allow the thread to properly terminate).

The TTC bit being set also causes an instruction trace trap to occur prior to the first instruction executed by a newly created hardware thread that accepted either a `pfork` or a `spawn`.

The trap class qualifier loaded into address register A5 may be used to decode the cause of the trace trap. Table 43 lists the class codes and qualifiers placed in register A5 for each type of trace trap.

Table 43 Trace trap class codes and qualifiers

Trace trap type	Class (byte 2) (hexadecimal)	Qualifiers (byte 3)	Priority
Instruction Trace	00	None	Highest
Trace thread concurrency	04	0-thread creation (`pfork`/`spawn` accepted) 1-`join` instruction executed 2-`wfork` instruction executed 3-`idle` instruction executed	↕
Thread initialization trap	08	None	Lowest

Note

The test for trace thread concurrency is performed based on `fork.PSW` after the fork is taken. The PC in the trap frame is `fork.PC`, the starting address of the new thread.

The conditions which can cause each kind of trace thread concurrency (TTC) trap are

Code	Condition
0400	Any of the following: — An idle CPU picks up fork and sees PSW (TTC) set in `fork.PSW`. — An `idle` instruction is executed, a fork found in CIR specified in Sk, and PSW (TTC) is set in `fork.PSW`. — A `wfork` instruction is executed by the last thread, a fork is found in the current CIR, and PSW (TTC) is set in `fork.PSW`.
0401	The `join` instruction was executed by a thread other than the last thread, so the CPU traps instead of entering the idle loop.
0402	The `wfork` instruction was executed by a thread other than the last thread, so the CPU traps instead of entering the idle loop.
0403	The `idle` instruction finds no fork to take in CIR Sk and goes to the CPU idle loop, *after* determining that the CPU's PSW (TTC) is set, and traps rather than enter the CPU idle loop. When a CPU cannot find a fork to take, this means — No fork is posted, — The CPU cannot receive (`rcv`) thread count or mask, — No allocatable threads exist in the thread mask, or — The posted fork is marked *STOPPED* (another thread has joined).

In order for a process to continue after all its threads are joined in the trap handler and to keep the PSW (TTC) set after the last thread continues as the serial thread, the trap handler should initiate the following trap processing sequence:

1. A `join` instruction executes in the trace trap handler.
2. The trace trap handler returns from the TTC trap frame.

For example, consider a two-threaded trap process. The first thread executes a `join` instruction and traps before terminating, so the trace handler is called *without deallocating the thread*. The second thread (normally the last thread) encounters the `join` instruction, sees the thread count of 2, and also trace traps. Since all threads have trace-trapped, the serial thread continues

execution after the `join` in the trap handler completes with the PSW (TTC) bit cleared.

A `join` instruction must always be executed to allow all but one thread to terminate with PSW (TTC) clear. All threads are joined in the trace trap handler and the trace trap handler returns with a serial thread. Only the last thread executes the return. This thread continues after the join operation with the PSW (TTC) bit set, which was reloaded when the extended frame popped.

The thread initialization trap allows a thread state to be initialized as the thread begins execution. This trap is primarily for vector registers permitting each thread to be forced to start with a known vector register state. If the PSW (TIT) bit is set when a CPU picks up a fork, a trace trap is taken to allow a user-defined handler to initialize the desired state. Table 43 lists the class code and qualifier placed in address register A5.

Note

The thread initialization trap is based on the PSW contained in the fork block (`fork.PSW`) located in the communication registers. The TIT is a user trap—this trap occurs in the ring where it was executed. The PC in the trap frame is `fork.PC`, the starting address of the new thread. A CPU *does not* have to be in sequential mode for TIT traps to function correctly.

Sequential execution

Although sequential execution does not cause an exception, it affects the operation of the machine and perhaps the specific conditions that exist when an exception occurs. The PSW has two bits that control sequential execution: the SEQ bit and the SQS bit.

SEQ bit

All overlapped execution in the processor is disabled when a process sets the PSW (SEQ) flag. The PSW (SEQ) flag forces serial execution for both hardware and software. The numerical results produced by any arithmetic operations are the same, regardless of the setting of the SEQ flag. Setting SEQ only affects performance and the serial nature of the execution. This bit may be freely set or reset.

SQS bit

When the sequential store bit SQS is set, memory store operations are done in instruction execution order. This ability to force all memory stores to be sequential allows debugging parallel executing programs that rely on memory store order. The rules for locking memory structures still apply for multiple CPUs executing a multithreaded process. Only performance and the serial nature of the execution are affected. The user may freely set or reset this bit.

Breakpoints

Although the breakpoint instruction (`bkpt`) is not a true exception, execution of this instruction results in a trap. Execution of the `bkpt` instruction

- Causes a call to the breakpoint trap handler pointed to by the byte address pointer at `0000 0050` of the current ring, and
- Pushes an extended return block on the stack

For multiprocessing C-Series CPUs, execution of the process breakpoint (`pbkpt`) instruction results in a system exception, because these breakpoints cause a ring crossing to the ring 0 exception handler. The process breakpoint instruction is described in the "Process traps and process breakpoints" section on page 226.

System exceptions

A *system exception* normally cannot be handled by the user process and must involve the operating system. Examples of system exceptions are address translation faults and ring-crossing traps.

All system exceptions have the following characteristics:

- System exceptions are not maskable.
- System exceptions always result in a ring crossing to ring 0.
- Ring 0 has residency and alignment requirements. The ring 0 stack must always be aligned on a 32-bit (word) boundary, and ring 0, page 0 must be in resident memory. A machine exception results if either requirement is not met.

The processing sequence for system exceptions is:

1. The hardware performs a ring crossing to ring 0.
2. A return block is pushed on the ring 0 process stack. The return block saved in each case is either an *extended* return block (FRL = 01) or a *context* return block (FRL = 00).

For multiprocessing C-Series CPUs, almost all system exceptions are local to a CPU. The only global system exceptions are *process deadlock*, and *process trap*. Local and global system exceptions are defined at the beginning of this chapter.

The following sections describe each of the system exceptions. Refer to Tables 45, 47, and 47 for the corresponding exception class codes and qualifiers.

Error exit trap

An error exit trap occurs if the CPU encounters an all-zero op code. This trap occurs, assuming that the memory in question has been previously cleared, if the CPU attempts to execute code from memory that resides beyond the boundaries of a program. An error exit trap results in a system call to the system exception handler pointed to by the address `0000 000C` of page 0 of ring 0.

Undefined op code trap

An undefined op code trap occurs whenever the CPU attempts to execute an illegal (undefined) instruction and results in a system exception. An undefined op code is a syntactically correct instruction with an op-code field (binary bit pattern) that has no associated definition of an actual machine instruction.

An undefined op code trap results in a system call to the system exception handler pointed to by the pointer at address `0000 000C` of page 0 of ring 0. A class code of 1 is loaded into byte 2 of address register A5 after an extended return block is pushed on the stack. No qualifier code is loaded into byte 3 of address register A5.

Vector valid trap

Under control of the vector valid (VV) flag, a vector valid trap can be programmed to occur the first time a vector instruction is used. A vector instruction is any instruction that reads or writes a vector register (V0 - V7 or V0 - V15), VL, VS, VM, or VF (C4600 only). This includes implicit reads, as in the `ste` instruction. The occurrence of a vector valid trap permits the operating system to save and restore the vector registers on demand (for any process or thread that uses vector instructions). Refer to Chapter 3 and the definitions of the `mov Sk,VV` and `tstvv` instructions in the *CONVEX Assembly Language Reference Manual (C Series)* for more information on the vector valid trap and the VV flag.

For example, assume that ten processes are running, but that only two use the vector registers. Upon interrupt during one of these two processes, the system does not save the vector registers since the interrupt service routine does not use them. If subsequent processes do not use the vector registers (either statically because there is no need, or dynamically because the particular code segment is not vector in nature), no CPU time is wasted in saving the vector machine state.

However, if one of these subsequent processes attempts to use vector instructions (which would alter the vector machine state), a recoverable vector valid trap occurs. When this trap occurs, the operating system saves a previous process's vector machine state. Once this machine state is saved, the affected process resumes.

Note

The description of the algorithm used to process vector valid traps is a function of the operating system implementation and *not* part of the C-Series architecture.

A vector valid trap may be generated in all C-Series CPUs when a process uses vector instructions. The state of the VV flag determines whether a vector valid trap is generated. When the operating system determines that the vector trap has occurred, the operating system reserves the vector register set (VM, VL, VS, and V0–V7) to the user process. The general algorithm used to process the vector valid trap may behave differently depending on the number of CPUs involved.

The vector valid trap occurs if the following two conditions are met:

- The vector valid (VV) flag is clear.
- The CPU attempts execution of a vector instruction.

If the two preceding conditions are met, the vector valid trap is processed in the following sequence:

1. The hardware performs a ring crossing to ring 0, and pushes an extended return block on the ring 0 stack.
2. The CPU jumps to the starting address of the exception handler pointed to by the byte address pointer located at `0000 001C` of page 0 of ring 0.

Ring violation traps and faults

Ring violation traps and faults are system exceptions concerning invalid access to rings.

The following ring violations are defined for all C-Series CPUs, arranged by the qualifier code returned in address register A5:

0 **Privileged instruction**—A CPU attempted to execute a privileged instruction outside of ring 0.

1 **Inward address**—A CPU attempted to reference an address contained in an inner ring.

2 **Outward system call**—A system call (`sysc`) attempted to call an outer ring. All system calls must call the current ring or an inner ring.

3 **Inward return**—A return instruction attempted to return to an inward ring. All returns must be to the same or to an outward ring. This violation occurs only when the return block is an extended frame. Short and long return blocks always return within the same ring.

4 **Invalid gate**—An incorrect gate number is specified in a `sysc` instruction.

5 **Invalid frame length on return instruction**—A return instruction encounters a frame length which does not agree with the type for the return.

The following ring violations are defined for multiprocessing C-Series CPUs only:

6 **Invalid communication register access**—A reference is made to an inner ring's communication register, or to an invalid virtual communication register address. This is valid only on C3400/C3800/C4600 Series CPUs.

7 **Invalid trap instruction**—A trap instruction is executed with an invalid ring field or invalid bit field.

Page table entry violation faults

PTE violation faults encompass a group of illegal PTE accesses. The following PTE violations are defined for all C-Series CPUs, arranged by the qualifier code returned in address register A5.

1 **Read protect**—The system attempted a read access to a page whose valid PTE did not allow reads.

2 **Write protect**—A CPU attempted a write access to a page whose valid PTE did not have write enabled.

3 **Execute protect**—A CPU attempted an instruction fetch on a page without execute enabled in its valid PTE.

4 **Invalid SDR**—A CPU attempted a memory access to a segment whose SDR valid bit was not set.

5 **Invalid level-1 PTE**—A memory reference was attempted to an address that had an invalid first-level PTE. A first-level PTE is not valid if the PTE valid flag is not set.

6 **Invalid level-2 PTE**—A memory reference was attempted to an address that had an invalid second-level PTE. A second-level PTE is not valid if the PTE valid flag is not set.

The following PTE violation is defined for multiprocessing C-Series CPUs only:

7 **Invalid level T PTE**—If the corresponding level-T PTE for an address is not valid, an invalid level-T PTE exception occurs.

The following violation is defined for C3200/C3400 Series CPUs only:

8 **Invalid I/O access**—If an I/O access is not valid, an invalid I/O access exception occurs.

Nonresident page faults

A nonresident page fault occurs when a CPU attempts to reference a memory location that is part of the virtual address space, but is not part of the physical address space. The system initiates a page fault only after it has interpreted the validity and appropriate access bits in a PTE. The nonresident page fault has two forms:

- **Nonresident data page**—The actual data page that corresponds to the virtual address is not in physical memory.
- **Nonresident page-table page**—A CPU attempted to reference a virtual address that accessed a nonresident page table (when translated to a physical address). A nonresident page table may be a second-level page table or a thread-level page table.

Note

If a CPU detects another page fault while responding to a page fault as described in the preceding sequence (faults on a reference pushing the context frame), a machine exception (hard error) occurs. This check prevents generation of an infinite number of page faults.

Process deadlock traps

A process deadlock trap results in a system call to the process deadlock trap handler pointed to by the pointer at address `0000 0010` of page 0 of ring 0. A class code and qualifier for an exception are placed in address register A5. The class codes and qualifiers for the deadlock exceptions are listed in Table 44. Refer to "CPU deadlock detection" section in Chapter 5 for more information on the deadlock process.

Table 44 Process deadlock class codes and qualifiers

Process deadlock	Class (Hexadecimal) (Byte 2)	Qualifiers (Byte 3)	Priority
Last thread termination	0	None	Highest ↕ Lowest
Hardware deadlock detected	4	0-All threads branching to synchronizing instruction 1-Mixed `wfork` and `join`	

Invalid communication address exception

An invalid communication register address exception is generated when a communication register operation is executed using an invalid communication register address. This exception is implemented as a trap on the C3200 Series CPUs and as a fault on the C3800/C4600 Series CPUs. In general, an invalid communication register address can be one of the following:

- **Unimplemented address**—For example, specifying the address `0x8040` on the C3200/C3400/C3800/C4600 Series complexes (the implemented address range ends at `0x803F`).
- **Ring-protected address**—For example, a ring 4 (user) program specifying the address `0x0000`, which may be referenced by ring 0 only.

When an invalid communication register address is detected on a C3200 Series CPU, the PSW (CAT) bit is set. At the next instruction boundary, the microcode jumps to a trap routine that pushes an extended frame, clears the PSW, places the appropriate class code in A5, and enters the ring 0 system exception handler. The exception is deferred through the PSW because of the pipelined nature of C3200 Series machines. A ring 0 operation that pushes the PSW may be dispatched before a ring 4 operation that contains an invalid address completes.

For example, consider the following sequence:

```
get.l   0x0000,s0
sysc    #0,#1
```

If the `sysc` had been dispatched before the invalid address was detected, the crossing to ring 0 may have already been made. By placing the trap condition in the PSW, the `sysc` pushes the PSW with the CAT bit set, and the exception is deferred until after the `rtn` from the `sysc`.

Note

When the system exception handler is entered, A5 contains the code `0x000000806`, identifying the type of exception, but the invalid communication register address is not specified.

When an invalid communication address is detected on a C3400 Series CPU, the trap occurs immediately after the communication register instruction that caused the trap and before the next sequential instruction is executed. Therefore, the PSW (CAT) bit is not used in the C3400 Series CPUs. When the trap occurs, the hardware pushes an extended frame onto the appropriate ring 0 stack, clears the PSW, places the appropriate class code in A5, and enters the ring 0 system exception handler.

When an invalid communication register address is detected on a C3800/C4600 Series CPU, the hardware pushes a fault context block onto the appropriate ring 0 stack, clears the PSW, places the appropriate class code in A5, and enters the ring 0 system exception handler.

Process traps and process breakpoints

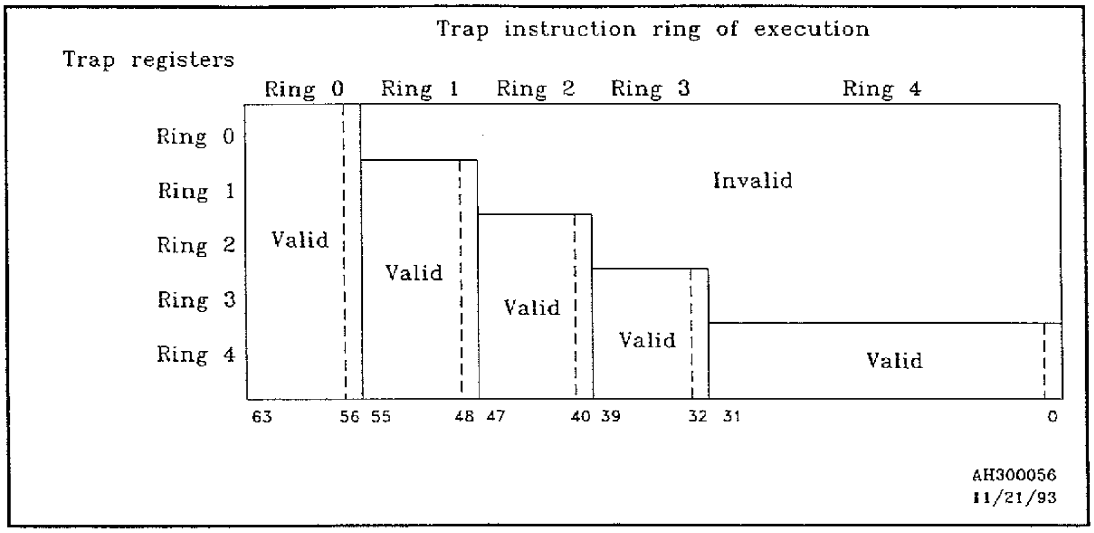
The trap instruction, `trap #rm,#b`, and the process breakpoint instruction, `pbkpt`, provide the only means, other than deadlock, for gaining control of all threads within a process in a timely fashion. The `trap` and `pbkpt` instructions are detailed in the *CONVEX Assembly Language Reference Manual (C Series)*.

When a `trap` instruction is executed, each CPU executing the process (within the rings specified) immediately traps to the ring 0 system exception handler. Any CPU that subsequently attempts to enter a ring that has a process trap exception pending enters the ring 0 system exception handler.

The `#rm` field is a 5-bit ring mask used to select which rings it is forced to trap. This mask defines which of the five 64-bit trap instruction registers (TIR) in the hardware communication register set are to be modified. The most significant bit of `#rm` specifies ring 4, down to the least significant bit which specifies ring 0. The `#b` field is a bit number between 0-63 that is set in all the trap instruction registers specified by the #rm field. Refer to Chapter 5 for more information on the trap instruction registers.

All accesses to the TIRs by the `trap` instruction are protected by ring maximization as shown in Figure 71.

Figure 71 Trap instruction register partitioning



The height of the boxes show the validity of the target ring references based on ring of execution. The width shows which bits may be set by the `trap` instruction within each ring of execution. The occurrence of any set bit in the TIR causes a trap of all threads associated with, or entering, the ring shown on the left.

The execution of a `trap` instruction can only change the TIRs of greater than or equal to rings. Each ring may only modify a fixed set of bits within any TIR. The bit positions associated with the dashed boxes are reserved for use by the process breakpoint (`pbkpt`) instruction. For example, a trap instruction executed in ring 3 can only set bits <39..33> in the TIRs for rings 3 and 4. Any trap instruction executed with an invalid ring or bit field causes the executing thread to enter the system exception handler with an invalid trap instruction exception code.

By providing the protection in this way, ring 0 is able to determine which ring executed the trap instruction by the position of the bit in the TIR.

When a valid trap occurs, the system exception handler is entered with a trap instruction class code in A5. The trap condition remains outstanding until the associated bit that caused the exception is cleared in the TIR. The program counter of the ring that trapped and the value of the TIR can be used to determine the trap instruction register and permit clearing of the exception.

No locking protocol exists on the TIRs. In order to clear the TIRs in a communication register set, the exception handler reduces the process running in that communication register set to a single thread to ensure that no traps are missed. Therefore, the exception handler must wait until all threads have entered the exception handler before clearing the TIRs.

The process trap mechanism is also used to implement a process breakpoint facility. A 16-bit process breakpoint instruction `pbkpt` is defined that causes all rings greater than or equal to the current ring of execution to trap. This is done by setting the appropriate bits (reserved for process breakpoint) in the TIRs for all rings greater than or equal to the ring of execution. Figure 71 shows which bit for each ring of execution is set by the process breakpoint instruction. For example, if ring 2 executes a `pbkpt` instruction, bit <40> is set in the trap instruction registers for rings 2, 3, and 4.

The TIRs set by the `trap` and `pbkpt` instructions are checked upon entering a ring, except when entering ring 0 for an interrupt. Whenever a ring crossing is executed, the TIR for the target ring is checked for any set bits. If any of the TIR bits of the target ring are set, the system exception handler is entered with a trap instruction class code in A5, instead of performing the original function of the ring crossing (enter exception handler, do *sysc*, and so on). The trap condition remains outstanding until all bits in the TIR are cleared. If a thread attempts to enter a ring that has any bits set in its TIR, the thread immediately enters the ring 0 system exception handler. The PC in the trap frame for this exception points to the intended target instruction of the original ring crossing.

For example, a two-threaded process running with one thread in ring 0 and one thread in ring 4.

1. The ring 0 thread executes a `trap` instruction to trap ring 0, but the other thread does not trap, because it is in ring 4.
2. Next, the ring 4 thread takes a page fault.
3. While jumping to the exception handler for the page fault, the ring 4 thread determines that the TIR bit is set, pushes a frame pointing to the system exception handler (to handle the page fault), and enters the system exception handler.
4. After the TIR trap is processed, the return from the exception handler causes a second entry to the exception handler for the page fault.
5. After the page fault is processed, the context return sends the thread to ring 4.

Although a `trap` instruction cannot set the "invalid" bits shown in Figure 71, a `put` instruction can set them in ring 0. On the next ring crossing into that ring, a trap occurs because the hardware checks the TIR on ring crossings. If the TIR is nonzero, the CPU traps.

Note

The invalid bits should not be set with a `put` instruction, as it subverts the intent of the `trap` or `pbkpt` mechanism.

System exception processing

When a CPU detects a system exception for C-Series CPUs, the appropriate exception handler is determined by the following:

- If the exception is a vector valid trap, the vector valid trap handler pointed to by address `0000 001C` of page 0, ring 0 is executed.
- If the exception is a process deadlock trap, then the process deadlock trap handler pointed to by address `0000 0010` of page 0 is executed.
- If the exception is a global hard error trap, the global hard error trap handler pointed to by address `0000 0020` of page 0, ring 0 is executed.
- All other system exceptions are served by the system exception handler pointed to by address `0000 000C` of page 0, ring 0.

The information passed in address registers A5, A4, and A3 describes the exact type of exception. As soon as the CPU pushes a return block on the current ring 0 stack, it loads a 32-bit exception code into register A5. Bytes 0 and 1 of this code are always 0. Byte 2 specifies the class of the exception, and byte 3 is an optional qualifier for that particular class. In addition to the class codes loaded into address register A5, the CPU loads the virtual address of the failure into address register A4 and the length of the fault context block into A3 for certain types of exceptions.

Exceptions - C100

C100 Series CPUs exceptions are listed on Table 45. It lists the class codes and qualifiers placed in address register A5 for each exception. Usually, a return block of some type must be pushed on the ring 0 stack. The CPU loads the number of bytes stored in the return block into address register A3 if a context block is saved (FRL = 00). The 2-bit FRL field of the corresponding PSW

that is pushed on the stack specifies the type of return block (extended or context).

Table 45 System exception class codes and qualifiers—C100 Series CPUs

System Exception type	Class (Byte 2) (Hexadecimal)	Qualifiers (Byte 3)	Memory fault	Priority
Error exit	00	None	No	Highest
Undefined op code	04	None	No	
Ring violation	08	0-Privileged instruction 1-Inward address 2-Outward system call 3-Inward return 4-Invalid gate 5-Invalid frame length on return instruction	No Yes No No No No	
PTE violation	0C	0-Read protect 1-Write protect 2-Execute protect 4-Invalid SDR 5-Invalid level 1 PTE 6-Invalid level 2 PTE	Yes Yes Yes Yes Yes Yes	
Nonresident page	10	0-Level 1 PTE2 page 1-Level 2 DATA page	Yes Yes	Lowest

Exceptions - C3200/C3400/C3800

C3200/C3400/C3800 Series CPU's exceptions are listed in Table 47. It lists the class codes and qualifiers placed in address register A5 for each exception.

Table 46 System exception class codes and qualifiers—C3200/C3400/C3800 Series CPUs

System exception type	Class (Byte 2) (Hexadecimal)	Qualifiers (Byte 3)	Memory fault	Priority
Error exit	00	None	No	Highest
Undefined op code	04	None	No	
Ring violation	08	0-Privileged instruction 1-Inward address 2-Outward system call 3-Inward return 4-Invalid gate 5-Invalid frame length on return instruction 6-Invalid communication register address 7-Invalid trap instruction	No Yes No No No No No/Yes[1] No	
PTE violation	0C	0-Read protect 1-Write protect 2-Execute protect 4-Invalid SDR 5-Invalid level 1 PTE 6-Invalid level 2 PTE 7-Invalid I/O access[2] 8-Invalid level T PTE	Yes Yes Yes Yes Yes Yes Yes Yes	
Nonresident page	10	0-Level 1 PTE2 page 1-Level 2 DATA page 2-Level 2 PTET page	Yes Yes Yes	
Trap instruction	14	None	No	Lowest

[1]For C3800 Series CPUs
[2]Does not exist on C3800 Series CPUs.

For all exceptions resulting in memory faults (defined in Table 47), or other exceptions that cross to ring 0, a ring 0 stack must be allocated from the ring 0 system resource structure. After a stack is allocated and a return block is pushed (if necessary), an exception code is loaded into address register A5.

Byte 0 (most significant byte) and byte 1 of this code are always a binary 00. Byte 2 specifies the class of the exception, and Byte 3 is an optional qualifier for that particular class.

For memory faults (defined in Table 47), in addition to the codes loaded into A5, the processor loads the virtual address of the failure into A4 and the number of bytes stored in the return block into A3, since a context block is saved (FRL = 00). The FRL field of the corresponding PSW specifies the type of return block. Whereas, the number of bytes in an extended return block is invariant, the size of a context block is implementation-specific.

The preceding exception processing sequence does not apply to a special case of process traps and process breakpoints. This special case is described in the following sequence:

1. Some type of fault occurs in an outer ring.
2. The CPU pushes a context return block onto the ring 0 stack, enters the exception handler in ring 0, and processes this fault.
3. A process trap or breakpoint occurs in the same outer ring in which the fault occurred.
4. The CPU finishes processing the fault and executes the `rtnc` instruction to return to the outer ring. The `rtnc` microcode detects a process trap pending in the outer ring.

In this situation, an extended return block is not pushed on the ring 0 stack as the exception handler is entered to handle the process trap, since the state of the outer ring has already been saved on the same stack during the original exception handler entry. Instead, the `rtnc` microcode jumps directly to the system exception handler with A5 set to indicate a process trap. The context return block is not popped, nor is an extended return block pushed. Register A3 is set to the size of the context frame.

For this reason, the system exception handler should always check the frame length bits in the PSW to determine if a return (`rtn`) or context return (`rtnc`) should be executed to exit the exception handler.

Exceptions - C4600

Table 47 lists the class codes and qualifiers placed in address register A5 for the C4600 Series CPUs. The C4600 Series CPUs also generate an *Undefined opcode trap* if an attempt is made to execute a nonlongword aligned `Format 8` instruction.

Table 47 System exception class codes and qualifiers—C4600 Series CPUs

Exception type	Class (Byte 2) (Hexadecimal)	Qualifiers (Byte 3)	Memory fault	Priority
Error exit	00	None	No	Highest
Undefined op code	04	None	No	
Ring violation	08	0-Privileged instruction 1-Inward address 2-Outward system call 3-Inward return 4-Invalid gate 5-Invalid frame length on return instruction 6-Invalid communication register address 7-Invalid trap instruction	No Yes No No No No Yes No	
PTE violation	0C	0-Read protect 1-Write protect 2-Execute protect 4-Invalid SDR 5-Invalid level 1 PTE 6-Invalid level 2 PTE 8-Invalid level T PTE	Yes Yes Yes Yes Yes Yes Yes	
Nonresident page	10	0-Level 1 PTE2 page 1-Level 2 DATA page 2-Level 2 PTET page	Yes Yes Yes	
Trap instruction	14	None	No	
Global hard error	3C	0 - dsi failed 1 - dsi succeeded	No	Lowest

In the C4600 Series CPUs, a portion of the fault context block is *not* pushed onto a stack acquired from the System Resource Structure. Instead, the *physical* address of the fault context area of each CPU is placed into its CXBASE registers. Refer to the "CXBASE registers" section on page 236 for a description of these registers. The context stored in this physical address space

is the non-architectural state (hardware context) when a fault occurs. The non-architectural state is reloaded when a `rtnc` instruction is executed. When a fault occurs, the OS is responsible for moving the physical address of a new context area into the CXBASE registers before any subsequent fault may occur.

The following steps occur when a fault is detected:

1. The processor stores most non-architectural state to the area of *physical* memory area pointed to by the CXBASE registers.
2. It then acquires a stack from the system resource structure, and pushes a context frame onto this stack. The context frame consists of an extended frame followed by all remaining non-architectural state. The FRL bits of the PSW in this extended frame are 00.
3. The size of the context block which has been pushed onto the stack is placed into A3, the faulting address is placed into A4, and the fault code is placed into A5.
4. The processor then begins execution of the system exception handler pointed to by address 0x0000000c of page 0 of ring 0.

A second fault *cannot* occur during step 1 above, since all addresses are physical. If a second fault occurs during steps 2 to 4 above, then the CPU takes a hard error.

The following steps occur when a `rtnc` instruction is executed:

1. The microcode verifies that the FRL bits of the PSW on the top of the current stack are 00.
2. It then pops the context frame from the current stack, and pushes the stack pointer onto the system resource structure.
3. The microcode will load the necessary non-architectural state from the physical memory area pointed to by the CXBASE registers.
4. If a fault occurs during steps 1 to 2 above, the CPU takes a hard error. A second fault *cannot* occur during step 3, since all addresses are physical.

Global hard error trap

When a hard error occurs on a C4600 Series CPU, the hardware issues a nonmaskable exception (trap) to all CPUs in the complex. This trap enables the operating system, executing on the surviving CPUs to try to recover from the hard error without taking the entire system down.

When an active CPU takes a global hard error trap, it pushes an extended return block on the ring 0 stack and jumps to the global hard error handler. Location `0x20` of page 0, ring 0 is a pointer to the global hard error handler.

When an idle CPU takes a global hard error trap, the idle stack pointed to by the ICB is used. Nothing else in the ICB is used for the global hard error trap. The microcode steps for taking a global hard error trap on an idle CPU are:

1. Set CIR = ICIR
2. Set TID = CPUID
3. Set SP to the idle stack address from the CPU's ICB.
4. Push an extended frame on this stack with a return PC of zero and with a PSW having FRL bits equal 01 (indicating an extended frame). The return PC == zero indicates that the global hard error trap occurred on an idle CPU. The rest of the extended frame is undefined, due to the idle CPU's lack of a real state.
5. Leave `SP=FP=idle_stack_base - extended_frame_length`. This is different than the use of the idle stack on an interrupt, where the interrupted CPU pushes the dummy state on the idle stack and loads SP with the interrupt stack.
6. Attempt to disable interrupts (issue a microcode `dsi`).
7. If the `dsi` succeeds (interrupts were previously on), set A5 = 0x00003c01. If the `dsi` fails (interrupts already off), set A5 = 0x00003c00.
8. Start execution at the PC referenced by the contents of memory location 0x20 in page 0, ring 0.

CXBASE registers

Three 32-bit CXBASE registers hold the physical memory addresses used for saving and restoring CPU dependent hardware context during faults and returns from faults. The first 4,096 bytes of context are stored in the physical page addressed by CXBASE[0], the next 4,096 bytes are stored in the physical page addressed by CXBASE[1], and the final 4,096 bytes are stored in the physical page addressed by CXBASE[2].

These registers are loaded from ring 0 only. The format of these registers is similar to the format of PTEs. Bits <31..12> specify physical address bits <31..12>. All other bits are unused. The CXBASE registers point to the start of physical pages.

Machine exceptions

Machine exceptions include hardware failures that cannot be corrected by the operating system, for example, memory errors or parity errors. The following conditions also result in machine exceptions:

1 **Page fault during a page fault**—A PTE violation trap or a nonresident page fault occurred while the machine is changing context to service one of these two exceptions. This exception prevents an infinite number of page faults.

2 **Nonresident data for SDRs**—A nonresident page fault occurred for the data read when either the kernel SDRs were loaded by a Load Kernel SDR (`ldkdr`) instruction or the process segment descriptor registers were loaded by an `ldsdr` instruction.

3 **Execution of `ldkdr` after virtual memory is enabled**—The load kernel SDRs (`ldkdr`) instruction was executed after virtual memory enabled.

4 **Invalid SDR0 after virtual memory is enabled** — The SDR0 register was accessed while invalid.

5 **Unaligned ring 0 stack**—The stacks in ring 0 are not aligned on a 32-bit boundary.

6 **Unaligned data for SDRs**—The data to be loaded into an SDR by either an `ldkdr` or an `ldsdr` is not word-aligned on a 32-bit boundary.

7 **Nonresident communication register data** — A nonresident page fault occurred while loading (`ldcmr`) or storing (`stcmr`) data in the communication registers in the current CIR.

8 **Ring 0 system resource structure underflow**—A fault occurred and a ring 0 stack could not be allocated to save machine context.

Table 48 lists the applicable machine exceptions (1 through 8) by CPU type.

Table 48 Machine exceptions

Machine Exception	Description	Architecture	
		C100 Series CPUs	Multiprocessor CPUs
1	Page fault during a page fault	Yes	Yes
2	Nonresident data for SDRs	Yes	Yes
3	Execution of `ldkdr` after virtual memory is enabled	Yes	Yes
4	Invalid SDR0 after virtual memory is enabled	Yes	Yes
5	Unaligned ring 0 stack	Yes	No
6	Unaligned data for SDRs	Yes	No
7	Nonresident communication register data	No	Yes
8	Ring 0 system resource structure underflow	No	Yes

Interrupt system

The C-Series architecture provides the operating system with a means to control I/O requests and other asynchronous events that require changing the explicit flow of control. These events are called interrupts. An *interrupt* is an asynchronous exception that requires a response by the operating system (ring 0 software) and not the executing process or thread. They belong to the system and not to the executing process.

Interrupts are processed on an *interrupt stack* in ring 0. They are nested if additional interrupts occur during interrupt processing. When an interrupt occurs, the processor jumps to a particular interrupt handler as a function of the source of the interrupt.

Note

Unless specifically stated otherwise, descriptions of interrupts and interrupt processing are applicable to all implementations of the C-Series architecture.

The interrupt system consists of

- Interrupt channels,
- Interrupt enable registers, and
- Interval timer

All C-Series CPUs have 256 interrupt channels, except the C3400 Series CPUs, which have 259. There are two types of channels

- Timesharing CPU virtual channels, and
- I/O virtual channels

Eight channels are allocated to the CPU and are addressed as channels 0 through 7 of the 256 system-wide channels.

The remaining 248 interrupt channels within the C-Series CPUs are allocated to I/O processors. The number of I/O processors and the number of virtual interrupt channels allocated to I/O processors are specific to each complex configuration. The instruction set for each processor includes instructions to allow any one channel to interrupt any other channel.

The C3400 Series CPUs have 259 interrupt channels. These channels are of three types

- Timesharing CPU virtual channels,
- Realtime CPU virtual channels, and
- I/O virtual channels

The group of CPUs dedicated to realtime applications is called the *realtime subcomplex*, and the group of CPUs running ConvexOS (non-realtime) is called the *timesharing subcomplex*.

Eight channels are allocated to the timesharing (non-realtime) subcomplex, as in all other multiprocessing C-Series CPUs, and are addressed as channels 0 through 7 of the 256 system-wide channels.

Another eight channels are specifically allocated to the realtime subcomplex, and are referred to as virtual channels (`0xf9-0xfd`, and `0x100-0x102`). Channels `0xfc` and `0xfd` can be accessed by any I/O device, or by any CPU in the complex. Interrupt channels `0xf9-0xfb`, and `0x100-0x102`, are reserved for interval timers and external interrupts.

Table 49 shows how the C3400 Series CPU realtime virtual channels are mapped.

Table 49 Realtime interrupt channels—C3400 Series CPUs

Virtual channel	Physical channel	Priority
0xfc	SIB interrupt 0xfc	Highest
0xfd	SIB interrupt 0xfd	
0x100	Interval timer 0	
0x101	Interval timer 1	
0x102	Interval timer 2	
0xf9	External interrupt 0	
0xfa	External interrupt 1	
0xfb	External interrupt 2	Lowest

The remaining 243 interrupt channels within a C3400 Series CPU are allocated to I/O processors. The number of I/O processors and the number of virtual interrupt channels allocated to I/O processors are specific to each complex configuration. The number of CPU virtual channels is independent of the number of actual I/O channels.

All external devices and controllers, regardless of their local intelligence, interrupt a multiprocessing C-Series CPU on one of the eight CPU virtual channels (or, on C3400 Series, one of eight timesharing or eight realtime CPU virtual channels, depending on the mode of the CPU).

For example, a physical I/O controller may use only one I/O channel to initiate interrupts using more than one CPU virtual channel. In some cases, the CPU may interpret one physical I/O controller as multiple I/O channels. Conversely, there may be up to 248 I/O virtual channels (243 for C3400 Series CPUs)

competing for eight CPU virtual channels. The CPU can individually interrupt any I/O channel by using the `xmti` instruction.

On multiprocessing C-Series CPUs, all virtual interrupt channels are *complex-wide virtual channels*. All external devices and controllers, regardless of their local intelligence, interrupt the CPU complex on one of eight complex-wide virtual channel ports.

Interrupt processing - C100

The `xmti` instruction enables a CPU in the complex to individually interrupt any of the I/O channels in the complex. In some cases, one physical I/O controller may be viewed as multiple I/O channels. Any CPU can interrupt another CPU in the complex by addressing channels 0 through 7.

The number of complex-wide virtual channels has no relationship to the number of actual I/O channels. For example, one I/O channel may initiate interrupts using more than one complex-wide virtual channel. Conversely, as many as all the I/O channels may be competing for eight complex-wide virtual channels.

The `mski` instruction is used by the C100 Series CPUs to mask out interrupts selectively from a particular CPU virtual channel.

There are two causes of interrupt: an I/O device via a virtual channel and an interval timer. When one of these causes initiates an interrupt, the following events take place:

1. A ring crossing to ring 0 is executed if the current ring is not ring 0.
2. The 16-bit halfword containing the CPU interrupt level, located at `0000 0004` of page 0, ring 0, is fetched. If this halfword is 0, the current interrupt is the first interrupt processed. This condition is referred to as *base-level interrupt processing*.
3. If this halfword is not zero, the current interrupt is not the first interrupt, and the CPU is already executing at *interrupt level*.
4. Once the CPU determines the interrupt level by interpreting the contents of the interrupt-level halfword, the interrupt-level halfword is incremented by one and stored back into the interrupt-level halfword. The CPU cannot be interrupted while the interrupt-level halfword is being incremented by one.

The fundamental difference between the two classifications of interrupt processing is the existence of an *interrupt stack*. When the interrupt level is zero, a unique stack is established in ring 0 for interrupts. This unique interrupt stack is different from the ring 0 process stack, in that it is used exclusively for interrupt processing.

Regardless of the level of interrupt processing, the program counter (PC) is pushed onto the stack. It references the

instruction that would have been executed had the interrupt not occurred. In addition, all further interrupts are kept pending by disabling interrupts—the interrupt on (ION) flag is cleared during the subsequent interrupt processing sequences.

Base-level processing

Base-level processing occurs when the interrupt level is 0. The actions that subsequently occur are determined by the current ring of execution.

When a return to base level is performed, the interrupt dismissal routine moves the previous process stack pointer (SP) (contained in byte address offset `0000 002C` of ring 0) to address register A0 prior to executing the `rtn` instruction.

Base-level processing—Ring 0

Assume that the stack pointer (SP) is already initialized to the ring 0 address space. Since the interrupt is at base level, stack multiplexing to the interrupt stack must occur, and the following interrupt processing sequence is initiated:

- The frame length, PSW (FRL), is set to `01`, indicating an extended return block is used.
- The extended return block is saved on the current ring 0 stack.
- The PSW is cleared.
- The updated stack pointer is saved in the previous stack pointer contained in byte address `0000 002C` of page 0 of ring 0. This previous stack pointer is at the top of the ring 0 process stack. This procedure is preparatory to stack multiplexing the interrupt stack.
- The SP (A0) and FP (A7) are loaded from byte address offset `0000 0020` of page 0 of ring 0. This is the *interrupt stack pointer.*
- A common hardware interrupt sequence is executed.

Base-level processing—Non-ring 0

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

- A ring crossing to ring 0 is executed (as though the `sysc` instruction were executed).
- The steps described in the base-level ring 0 processing are executed.

Interrupt-level processing

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

Interrupt-level processing—ring 0

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

Interrupt-level processing—non-ring 0

In this case, the following actions are taken:

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

Common interrupt processing sequence

The following processing sequence describes the actions taken by a C100 series CPU after a crossing to ring 0 and an interrupt stack is established:

1. Byte address offsets `0000 0008` and `0000 0010` of ring 0 contain the address of the appropriate interrupt handler. This interrupt handler address is selected by hardware and loaded into the PC.
2. The identification of the interrupting device is loaded into address register A5 after the return block is pushed. This identification value format is 29 zero bits followed by a three-bit encoding. This three-bit encoding identifies which CPU virtual channel initiated the interrupt. The channel number is always loaded in A5, since the interval timer uses one of the eight channels.
3. The interrupt handler executes the first instruction. Since all interrupts have been disabled, the interrupt handler must explicitly enable the interrupts during the course of execution.

General interrupt processing notes - C100

The following items should be taken into consideration during interrupt processing:

1. The interrupt return sequence determines whether or not the return is to base level or interrupt level as a function of the interrupt-level halfword in page 0 of ring 0.
2. The return to base level is achieved by executing a `rtn` instruction. The return to interrupt level is also achieved by executing an `rtn` instruction.
3. In order to return from an interrupt, the following steps must be taken by the software.
   - The interrupt level is decremented by one.
   - If the level is now zero, address register A7 (FP) is loaded from the previous stack pointer contained in byte address `0000 002C` of page 0, and the `rtn` instruction is executed.
   - If the level is not zero, the `rtn` instruction is executed. Register A7 need not be restored, since the ring 0 stack must still be the interrupt stack.
4. The process stack pointer contained in address `0000 0048` in page 0, ring 0 is not modified during hardware-initiated interrupt processing.
5. If the interrupt is initiated by an I/O device interrupting a CPU virtual channel, the CPU virtual channel interrupt is reset after the processor responds to the interrupt.

Interrupt Processing - Multiprocessing CPUs

On multiprocessing CPUs, the `xmti` instruction functions as on the C100 CPU.

Multiprocessing C-Series systems use additional registers to process interrupts, since more than one CPU are available for servicing them.

Interrupt enable flags
The ION flag is a single interrupt enable flag for the entire CPU complex. It is updated by the `eni` (enable interrupt) and `dsi` (disable interrupt) instructions. When ION is disabled, all interrupts sent to the CPU complex are deferred until ION is enabled. The `dsi` instruction atomically disables interrupts and returns the previous interrupt state. This enables `dsi` to be used as a simple lock for protecting critical code sections within the entire CPU complex.

In the C3400 Series CPUs, the ION flag is the interrupt enable flag for the timeshare CPU subcomplex. The RT_ION flag has the same function for the realtime subcomplex. Only one of these flags is used at a time.

The RT_ION flag (for C3400 Series CPUs) is a single CPU realtime subcomplex interrupt enable flag. It is updated by the `eni` (enable interrupt) and `dsi` (disable interrupt) instructions. When RT_ION is disabled, all interrupts sent to the realtime CPU subcomplex are deferred until RT_ION is enabled. The `dsi` instruction atomically disables interrupts and returns the previous interrupt state. This enables `dsi` to be used as a simple lock for protecting critical code sections within the entire CPU complex.

Interrupt enable registers
Each C3200/C3400/C3800/C4600 Series CPU has an 8-bit local enable register. The entire CPU complex shares an 8-bit global enable register. These registers selectively permit each channel to interrupt the CPU. Bits <7..0> correspond to virtual channels 7 to 0 respectively.

In the C3200/C3800/C4600 Series CPUs, the `enal` instruction updates the local enable register. The `enag` instruction updates the global enable register. If a bit is set, that channel is enabled. If a bit is clear, that channel is disabled (masked out) from that CPU.

Each C3400 Series CPU has a 16-bit local enable register and a 16-bit global enable register that selectively permit each channel to interrupt the CPU. The low order eight bits <7..0> of each register are for the timesharing subcomplex, and are mapped identically to the interrupt enable bits for other multiprocessing

C-Series CPUs (see Table 50). The high order eight bits <15..8> are for the realtime subcomplex.

Table 50 shows how the realtime subcomplex virtual channels are mapped in the local and global interrupt enable registers.

Table 50 Realtime virtual channels—C3400 Series CPUs

Virtual channel	Enable/disable bit
0x100	2
0x101	3
0x102	4
0xf9	5
0xfa	6
0xfb	7
0xfc	8
0xfd	9

If the CPU is in timesharing mode, the timesharing bits are used; if the CPU is in the realtime mode, the realtime bits are used.

In the C3400 Series CPUs, the `enal` instruction updates all 16 bits of the local enable register. The `enag` instruction updates all 16 bits of the global enable register. If a bit is set (1), that channel is enabled. If a bit is clear (0), that channel is disabled (masked out) from that CPU.

Target CPU register

An additional register, the interrupt target CPU (TCPU) register, is found *only* in the C3200 Series CPUs.

The TCPU is a single complex register that contains the identification of the target CPU that services all global interrupts. This register allows one CPU to serve as the nesting point for interrupts. The target CPU (TCPU) register is a 3-bit register that contains the CPUID of the CPU where all interrupts must be delivered. If the target CPU register contains all binary ones (the value -1), any CPU within the complex can be selected as the target CPU for interrupt delivery.

Interrupt control register

The interrupt control register (ICR) within the multiprocessing C-Series CPU complex defines the operating modes of each interrupt channel and the communication register set used during interrupt processing.

Figure 72 and Figure 73 show the format of the interrupt control register.

Figure 72 Interrupt control register (ICR)—C3200 Series CPUs

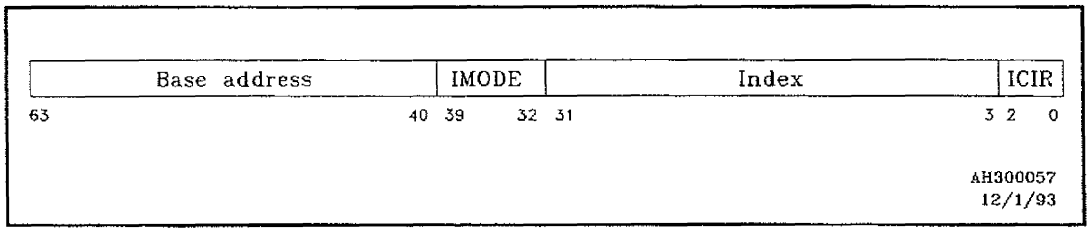
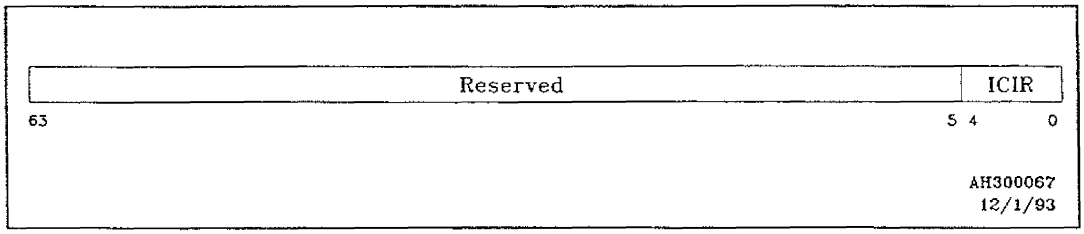


Figure 73 Interrupt control register (ICR)—C3400/C3800/C4600 Series CPUs



The interrupt mode (IMODE, C3200 Series CPUs only) in the interrupt control register is an 8-bit field in which bits <7..0> correspond to complex virtual channels 7 through 0. The interrupt mode controls the mode of operation for each CPU complex-wide interrupt channel. Any complex-wide virtual channel can be selected to operate in one of the following modes:

- **Local interrupt mode**—A single CPU within the complex is selected to receive the interrupt. The interrupt is delivered to the selected CPU when the interrupt occurs.
- **Broadcast interrupt mode**—All CPUs within the complex are selected to receive the interrupt. The interrupt is delivered to all CPUs when the interrupt occurs.

If the bit associated with the virtual channel is clear, the channel is a local interrupt channel. If the bit is set, the channel is a broadcast interrupt channel.

Both broadcast and local interrupts can be selectively enabled or disabled with one of the two interrupt channel enable

instructions. The global CPU enable instruction, `enag`, is used to enable or inhibit interrupt delivery to all CPUs within the complex. A global complex interrupt channel is enabled or disabled by setting or clearing, respectively, the appropriate bit in the *global enable* interrupt register.

For broadcast (global) interrupts, the global interrupt handler ensures that all CPUs have entered the interrupt handler before executing an `eni` instruction at the end of the handler and returning (idling). Otherwise, a hardware race condition could exist where one CPU may execute an `eni` instruction before the other CPUs have received the interrupt causing the other CPUs to lose the interrupt.

The local CPU enable instruction, `enal`, is used to enable or inhibit delivery to a single CPU. A local complex interrupt channel is enabled or disabled by setting or clearing, respectively, the appropriate bit in the local enable interrupt register. These two instructions allow any single CPU within the complex to enable or disable interrupt reception locally or for the entire complex.

The interrupt communication index register (ICIR) is a three-bit field (C3200 Series CPUs) or a five-bit field (C3400, C3800 C4600 Series CPUs) that defines the communication register set is mapped when servicing interrupts. This communication register set provides the process context necessary for an idle CPU to service an interrupt.

Broadcast enable registers

The C3400/C3800/C4600 CPUs contain a separate set of broadcast enable (BE) registers for the CPU interrupt channels 0 through 7. There are eight BE registers, one for each of the eight CPU interrupt channels. Each register contains one bit per CPU. If any bit in a BE register is set, the corresponding interrupt channel is a broadcast interrupt, and the bits set identify the CPUs to receive the broadcast interrupt. If no bits are set in a BE register, the corresponding interrupt channel is a single CPU interrupt. Bits 7 through 0 correspond to CPUs 7 through 0 respectively. Only bits 3 through 0 are used on the C4600.

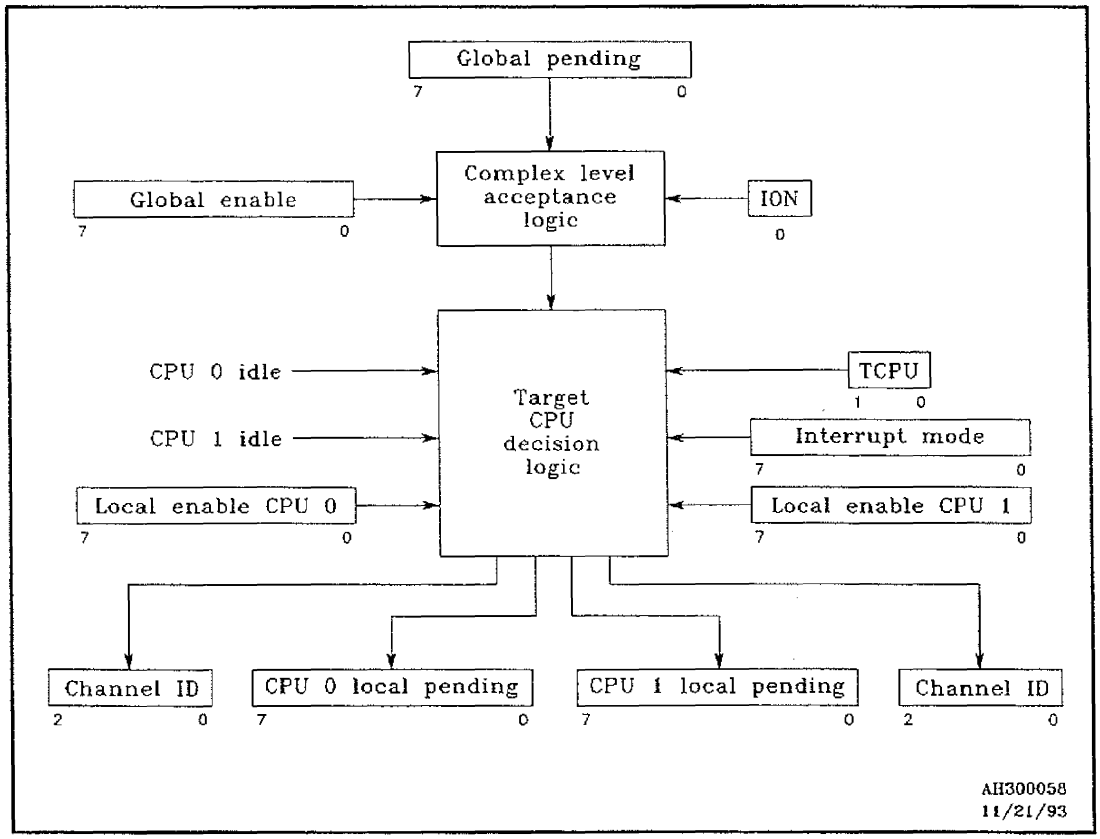
For example, by clearing bit <3> of register 2, CPU number three will not receive any interrupt broadcast on channel 2. The `mov Sk,BE(Sj)` and `mov BE(Sj),Sk` instructions are used for writing and reading these registers.

Interrupt flow - C3200

Figure 74 presents the basic implementation of the interrupt system for a C3200 complex. For clarity, only two CPUs are shown.

Interrupts enter the complex when the corresponding bit is set in the eight-bit global pending register. Each bit in this register corresponds to one of the eight virtual channels to which the CPUs respond. Before a target CPU can recognize an interrupt in the global pending register, it must perform a series of enables and destination checks. If all these checks are satisfied, the interrupt is registered in the local pending register of the target CPU(s).

Figure 74 Interrupt flow—C3200 Series CPUs



The first level of interrupt checking is at the complex level, shown as the complex-level acceptance logic in Figure 74.

1. If the ION flag is zero, interrupts are disabled for the complex and the interrupt stays in the global pending register. The ION flag must be cleared to allow software to modify the state of the subsequent interrupt control hardware.
2. The next check is the global enable register. Each bit in this register enables the corresponding bit in the global pending register.

When the complex-level checks are complete, the destination CPU is chosen by using the following decision logic:

1. If the bit in the interrupt mode register corresponding to the global pending bit is set, the interrupt is considered a broadcast interrupt, and *must* be sent to all CPUs.
2. If the target CPU (TCPU) register is set to 0, 1, 2, or 3, the interrupt *must* be sent to CPU 0, 1, 2, or 3, respectively. In this case, the target CPU must also have the corresponding local enable bit set.
3. If TCPU is -1 (binary `11`), either CPU may be chosen. In this case, an idle CPU (designated by the CPU "n" idle signal and shown in Figure 74) is chosen, if the correct bit in the idle CPU's local enable is set.
4. If all CPUs are idle and locally enabled, CPU 0 is chosen.

If all the preceding conditions are met, the bit in the selected CPU's local pending register is set, the bit is cleared in the global pending register and the ION flag is cleared.

The bit in the global pending register remains set if the interrupt cannot be taken all the way to a local pending register. These checks are performed on every system clock. If the interrupt is taken, the ION flag is reset to zero, disabling interrupts until they are explicitly re-enabled with the `eni` instruction.

Interrupt flow - C3400

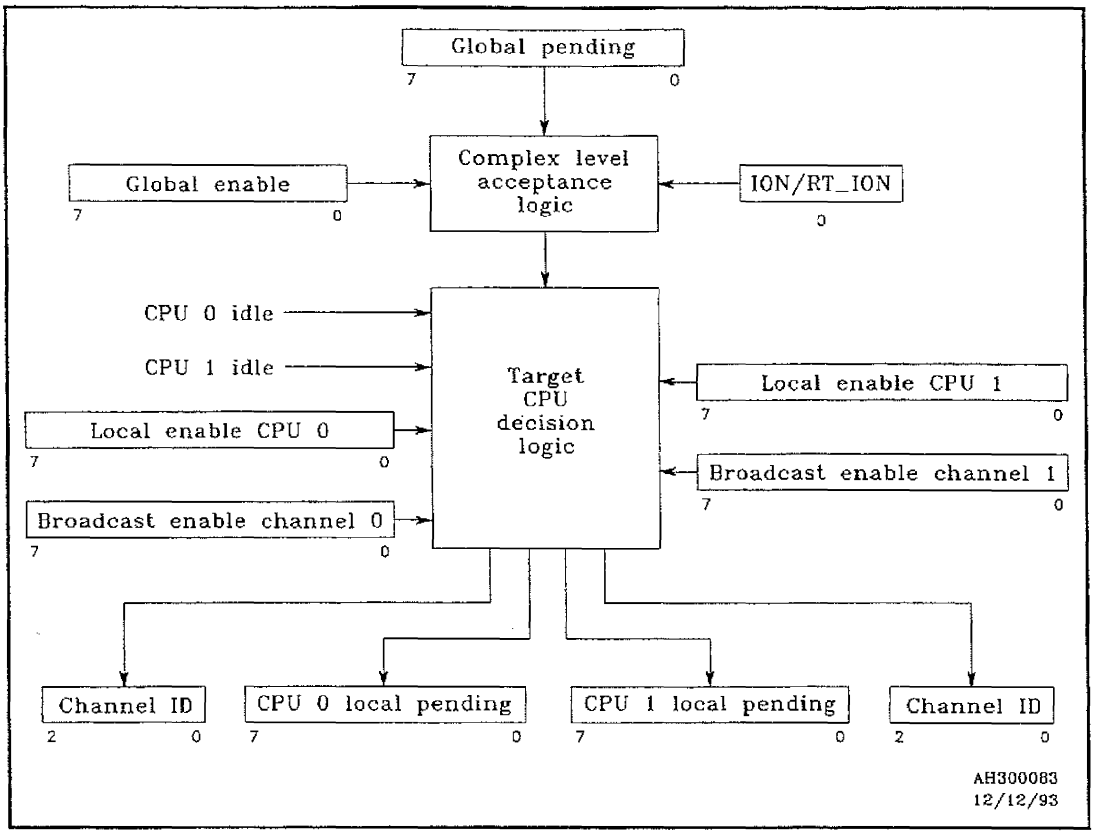
Figure 75 is representative of a C3400 Series complex. For clarity, only two CPUs are shown.

Interrupts enter the complex when the corresponding bit is set in the 16-bit global pending register. Eight bits are for the realtime subcomplex, and eight bits are for the timesharing subcomplex. These bits are indexed depending on the mode of the CPU. Each bit of the eight timesharing bits in this register corresponds to one of the eight virtual channels to which the CPU responds. Before a CPU can recognize an interrupt in the global pending register, a series of enables and destination checks must be made by the target CPU. If all of these checks are satisfied, the interrupt is registered in the local pending register for the CPU(s) that responded to the interrupt.

The bit in the global pending register remains set if the interrupt cannot be taken all the way to a local pending register. These checks are performed on every system clock. If the interrupt is taken, the ION flag is reset to zero, disabling interrupts until they are explicitly re-enabled with the eni instruction.

Figure 75 shows the flow of interrupts in a C3400 Series CPU.

Figure 75 Interrupt flow—C3400 Series CPUs



Interrupt flow - C3800/C4600

Figure 76 is representative of a C3800/C4600 Series complex. For clarity, only two CPUs are shown.

Interrupts enter the complex when the corresponding bit is set in the eight-bit global pending register. Each bit in this register corresponds to one of the eight virtual channels that the CPU responds to. Before a CPU can recognize an interrupt in the global pending register, a series of enables and destination checks must be made. If all of these checks are satisfied, the interrupt is registered in the local pending register for the CPU(s) that respond to the interrupt.

The first level of interrupt checking is at the complex level, shown as the complex level acceptance logic in Figure 76.

1. If the ION flag is zero, interrupts are disabled for the complex and the interrupt stays in the global pending register. The ION flag must be cleared to allow software to modify the state of the subsequent interrupt control hardware.
2. The next check is the global enable register. Each bit in this register enables the corresponding bit in the global pending register.

When the complex level checks are complete, the destination CPU is chosen by using the following decision logic:

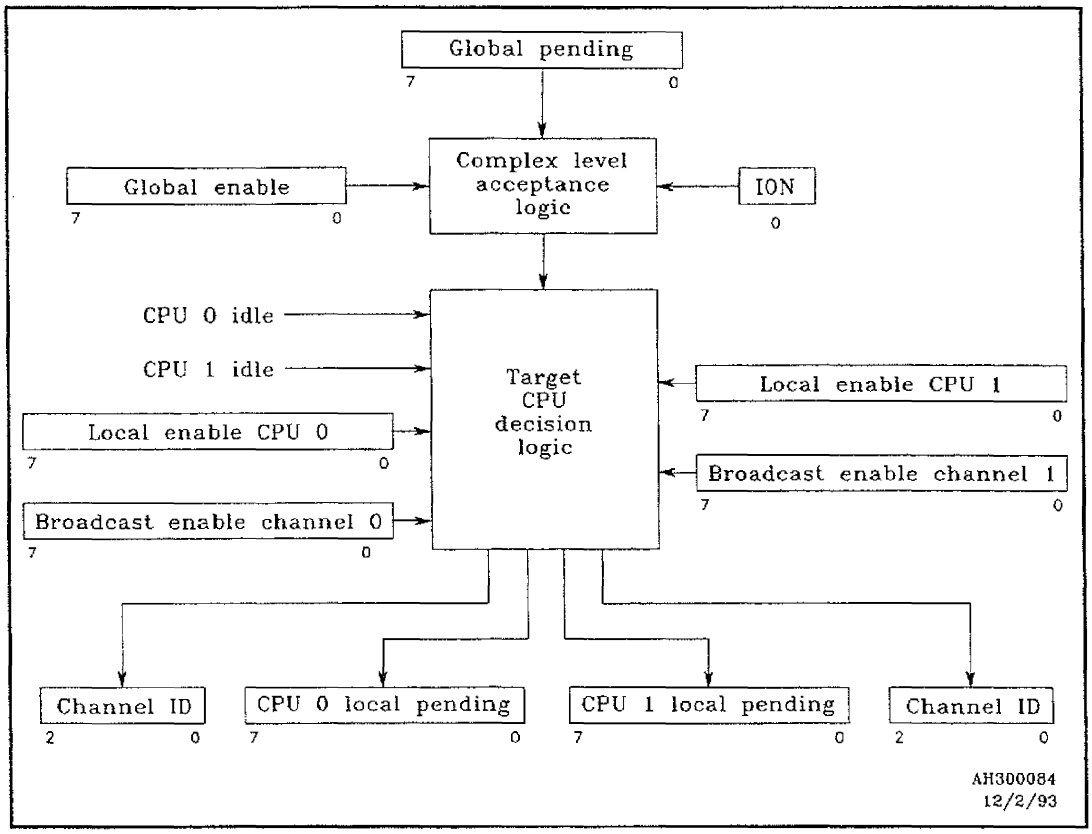
1. If the broadcast enable register corresponding to the global pending bit has any bit set, the interrupt is considered a broadcast interrupt. When all CPUs selected by the broadcast enable register have enabled the interrupt in their local enable register, then the interrupt is transferred to the local pending register of all selected CPUs.
2. If the broadcast enable register corresponding to the global pending bit is all zeros, the interrupt is considered non-broadcast, and a single CPU is selected as follows:
   - If any idle CPUs have the interrupt enabled in their local enable register, the lowest numbered idle CPU is selected.
   - If any active CPUs have the interrupt enabled in their local enable register, the lowest numbered active CPU is selected. Otherwise, the interrupt remains in the global pending register.

If all the preceding conditions are met, the bit in the selected CPU's local pending register is set, the bit is cleared in the global pending register and the ION flag is cleared.

The bit in the global pending register remains set if the interrupt cannot be taken all the way to a local pending register. These

checks are performed on every system clock. If the interrupt is taken, the ION flag is reset to zero, disabling interrupts until they are explicitly re-enabled with the `eni` instruction.

Figure 76 Interrupt flow—C3800/C4600 Series CPUs

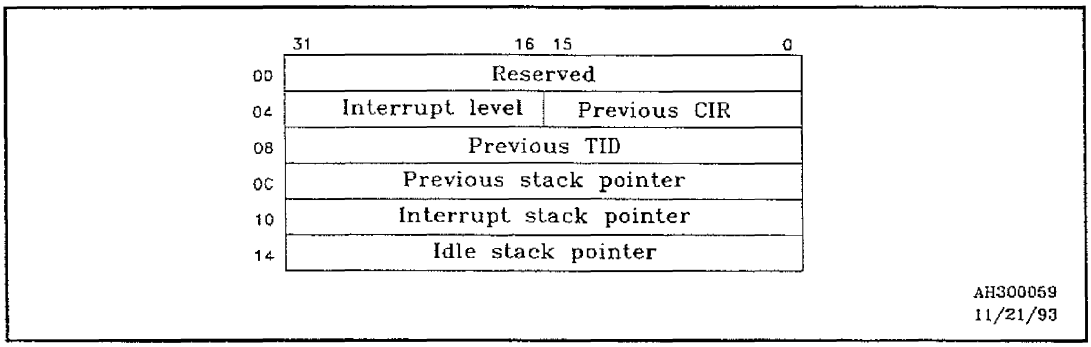


Interrupt context blocks

While servicing an interrupt, each multiprocessing C-Series CPU locates its associated interrupt context block (ICB) in unshared (that is, thread-local) memory. Each CPU indexes into memory based on thread ID (TID = CPUID) as described in Chapter 4. The pointer to the interrupt context block is located in the interrupt context block pointer contained in location 0000 0014 of ring 0 page 0.

Figure 77 shows the format of an interrupt context block.

Figure 77 Interrupt context block



Servicing interrupts

When an interrupt occurs on a multiprocessing C-Series CPU, the 16-bit halfword located at bytes 4 and 5 of the ICB is fetched. If this halfword is 0, then the interrupt is the first interrupt processed. This condition is referred to as base-level interrupt processing. If this halfword is not 0, then the interrupt is not the first interrupt, and the processor is already at interrupt level. The current ring 0 stack used is the interrupt stack. In effect, the ring 0 process stack pointer has temporarily become the interrupt stack pointer.

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

In the following sections, the program counter (PC) that is pushed onto the stack references the instruction that would have been executed had the interrupt not occurred. The interrupt handler is entered with all CPU complex interrupts disabled.

The complex virtual channel interrupt is always reset after a CPU responds to the interrupt.

Virtual memory restrictions

Some restrictions are placed on virtual memory mapping due to existence of a separate interrupt CIR. The interrupt service microcode must deal with both process context (the extended frame pushed on the ring 0 stack) and ICB context (interrupt level, and so on).

The ICBs *must* be resident at all times. A machine exception occurs if they are not resident. In addition, the ICBs *must* be mapped in all CIRs, so that while the interrupt service microcode is transitioning the CPU from the process CIR to the interrupt CIR, both process and ICB context may be accessed in the process CIR.

The interrupt CIR does not require access to all process context. Each process may have a unique page 0. However, the interrupt handler whose address is found in the process's page 0, must be mapped in all CIRs. When the interrupt handler is entered, the stack pointer is always equal to the frame pointer.

These memory-mapping restrictions become apparent in the following sections describing CPU interrupt processing. Note carefully where the transition from a process CIR to an interrupt CIR is made.

Idle CPU interrupt processing

When an idle CPU takes an interrupt, it must be at base level (interrupt level 0). An idle CPU has no state, so the CPU state is not saved or restored. An *idle stack* provides the operating system a consistent entry and exit mechanism. The process context referenced by an idle CPU is mapped to the interrupt CIR.

The interrupt service for an idle CPU performs the following interrupt processing sequence:

1. The interrupt CIR is fetched from the interrupt control register (ICR) and loaded into the CPU's CIR. The TID is set to the CPUID. The CPU assumes that this thread is always allocated. The TID is set before any memory references to the ICB occur to allow the ICBs to use unshared memory for multiple CPUs entering the interrupt handler simultaneously in different threads.
2. The interrupt context block (ICB) pointer is fetched from page 0 of the process address space. The ICB must be mapped into the interrupt CIR because the CPU has already entered the interrupt CIR.

3. The interrupt handler address is fetched from page 0 of the process address space. The interrupt handler must be mapped into the interrupt CIR because the CPU has already entered the interrupt CIR.
4. The stack pointer is loaded with the value contained in the idle stack pointer in the ICB. Whenever the interrupt handler is entered, the stack pointer is always equal to the frame pointer.
5. The interrupt level in the ICB is set to 1 (incremented from 0 or base level).
6. The previous CIR in the ICB is set to -1, which means that an idle CPU serviced the interrupt. Setting the CIR in the ICB to -1 means that the previous TID in the ICB is meaningless. In this case, the TID is ignored on the subsequent `rtn` or `idle` instruction.
7. A PC of zero and PSW (FRL) containing a binary 01 (extended frame) is pushed on the stack (idle stack). The stack pointer is updated as if an entire extended frame had been pushed.
8. The stack pointer (reflecting the push of the extended frame) is stored in the previous stack pointer in the ICB.
9. The stack pointer and frame pointer are now loaded from the interrupt stack pointer in the ICB, establishing the interrupt stack.
10. The PSW is cleared.
11. The channel ID of the interrupting virtual channel is placed in address register A5.
12. The PC is loaded with the interrupt handler address that was previously fetched. Execution continues in the interrupt handler (still within the interrupt CIR), with a newly established thread.

Active CPU interrupt processing

An active CPU responds to both base and interrupt-level interrupts. If the interrupt is base level, the CPU is not already using the interrupt CIR, and the referenced process context belongs to a different CIR. An active CPU may have to cross rings to enter ring 0. Interrupt-level interrupts are already in the interrupt CIR in ring 0. Both cases follow this common processing sequence:

1. The interrupt context block (ICB) pointer is fetched from page 0 of the process address space.
2. The interrupt level is fetched from the ICB and tested. If the interrupt level is 0, an active CPU continues with base-level processing. The interrupt-level fetch occurred in the process CIR, which requires that the ICB be mapped in all CIRs. If the interrupt level is nonzero, execution continues with interrupt-level processing.

Active CPU base-level processing

Active CPU base-level processing follows this sequence:

1. The interrupt CIR is fetched from the interrupt control register (ICR) and loaded into the CPU's CIR. The TID is set to the CPUID. The CPU assumes that this thread is always allocated. Before any memory references to the ICB occur, the TID is set to allow the ICBs to use unshared memory for multiple CPUs that are entering the interrupt handler simultaneously from different threads.
2. The incremented interrupt level is stored in the ICB (while still executing within the process CIR).
3. If the interrupted process was not executing in ring 0, the CPU crosses to ring 0 and allocates a ring 0 stack from the shared resource structure (SRS). If the interrupt occurred while in ring 0, a ring 0 stack already exists.
4. The address of the interrupt handler is fetched from page 0 of the process address space.
5. An extended frame is pushed on the ring 0 stack.
6. The updated stack pointer is stored in the previous stack pointer field of the ICB (still in process CIR).
7. The interrupt stack is established by initializing the stack and frame pointers from the interrupt stack field of the ICB (still in process CIR).
8. The CIR is loaded from the interrupt CIR field of the interrupt control register. A new thread is allocated and the

thread count incremented in the new (interrupt) CIR. The TID is set to the CPUID.

9. The PSW is cleared.
10. The channel ID of the interrupting virtual channel is placed in address register A5.
11. The PC is loaded with the interrupt handler address that was fetched earlier. Execution continues in the interrupt handler with a newly established thread (while still within the interrupt CIR).

Active CPU interrupt-level processing
During interrupt-level processing, an active CPU is already executing in ring 0, within the interrupt CIR, with the thread established at the base-level interrupt.

Active CPU interrupt-level processing follows this sequence:

1. The incremented interrupt level is stored in the ICB.
2. The address of the interrupt handler is fetched from page 0 of the process's address space.
3. An extended frame is pushed on the ring 0 stack.
4. The updated stack pointer is stored in the previous stack pointer field of the ICB.
5. The PSW is cleared.
6. The channel ID of the interrupting virtual channel is placed in address register A5.
7. The PC is loaded with the interrupt handler address that was previously fetched. Execution continues in the interrupt handler while still within the interrupt CIR.

Returning from a base-level interrupt

After an interrupt is processed, executing the `rtn` or `idle` (for C3200 Series CPUs) or the `eni_rtn` or `eni_idle` (for C3400/C3800/C4600 Series CPUs) instruction returns from a base-level interrupt.

The software performs the following operations:

1. If the previous CIR field in the ICB is -1, (the interrupt was serviced by an idle CPU), the CPU returns to the idle state by an `idle` instruction. Since an idle CPU has no state, an extended frame is not popped.
2. If returning to a CPU which was previously executing, the CIR and TID are restored with the previous CIR and previous TID fields in the ICB. Address register A7 is then loaded from the previous stack pointer in the ICB. A standard subroutine performs an extended return sequence, because the FRL bits in the pushed PSW indicate that an extended return block was pushed.
3. Software always leaves *N* threads allocated in the interrupt CIR (where *N* is the number of CPUs in the complex) to allow each CPU to set TID equal to the CPUID as it services an interrupt.

General interrupt notes - multiprocessing CPUs

1. The identification of the interrupting device loaded into address register A5 after the return block is pushed is a 32-bit value. This value takes the form of 29 zero bits followed by a three-bit encoding. This three-bit encoding identifies which CPU complex virtual channel initiated the interrupt.
2. All CPU-complex interrupts are disabled when the interrupt handler is entered. The interrupt handler must explicitly re-enable interrupts.
3. The interrupt return sequence determines if the return is to base level or interrupt level as a function of the interrupt level in the ICB.
4. A CPU returning from base level is returned to the idle state or the previous context based on the previous CIR in the interrupt context block.
5. In order to return from an interrupt, the following steps must be taken by software:
   a. The interrupt level is decremented by one.
   b. If the level is now 0, the frame pointer (A7) is loaded from the previous stack pointer in the ICB. The `rtn` or `idle` (for C3200 Series CPUs) or the `eni_rtn` or `eni_idle` (for C3400/C3800/C4600 Series CPUs) instruction is now executed.
   c. If the level is not zero, the `rtn` instruction is executed. Address register A7 is not restored since the ring 0 stack must still be the interrupt stack.

Implementation-specific features

7

The addressing methods for the physical address space differ slightly between the C100 Series CPUs and the multiprocessing C-Series CPUs.

The I/O address space is implementation-specific, resulting in significant differences between the C100 Series and the multiprocessing C-Series implementations.

C3200 Series CPUs implement some machine functions through registers physically located in the I/O address space. These registers are addressed in much the same way as elements of main memory. This allows access to a number of subsystems required for proper operation of the CPUs.

C3400/C3800/C4600 Series CPUs do not have explicit I/O space defined in physical memory, and use special instructions for these functions.

Physical address space

The physical address space in all C-Series CPUs spans physical memory. The physical address space definitions vary between the C100 Series CPUs and the multiprocessing C-Series CPUs. In C3200 Series CPUs, it also spans a set of I/O registers that implement numerous system control and overhead functions. Data access at the program level uses virtual addresses. However, these virtual addresses are translated to physical addresses in hardware to access the physical hardware features. The possible range of these physical addresses describes the physical address space.

Power-up addressing mode - C100

Physical addresses are normally generated by virtual-to-physical address translation. The exception to this process occurs during the bootstrap process at power up. Prior to cold start, the Service Processor Unit (SPU) must create a bootstrap Page Table Entry (PTE) structure. The PTE structure is used by the CPU's cold start microcode to make the address translation mechanism operational.

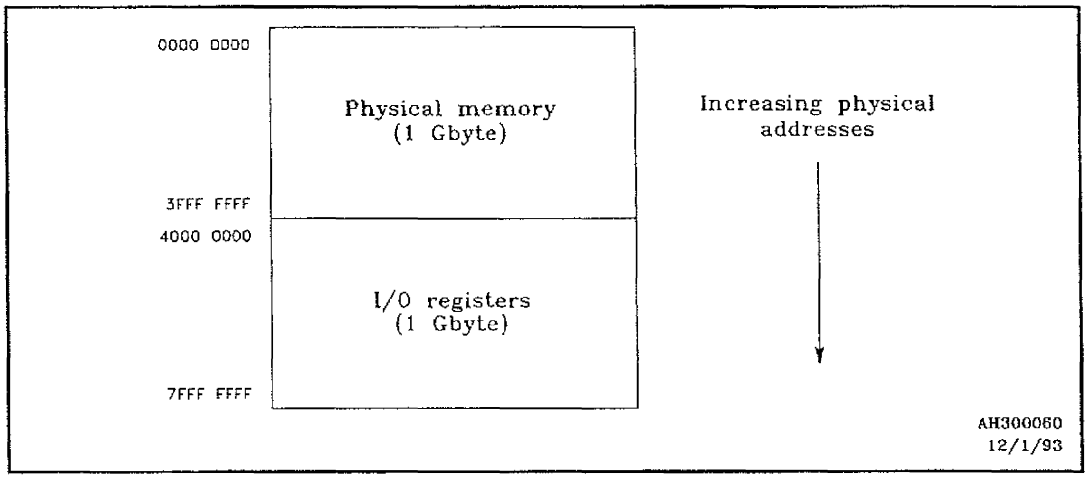
Physical address space - C100

C1 CPUs have a maximum physical memory configuration of 128 Mbytes. This memory may be accessed with physical addresses in the range 0000 0000 to 07FF FFFF.

The C120 CPU has a maximum physical memory configuration of 1 Gbyte. This memory may be accessed with physical addresses in the range 0000 0000 to 3FFF FFFF.

Figure 78 shows the physical address space for C120 CPUs.

Figure 78 Physical address space—C120 CPUs

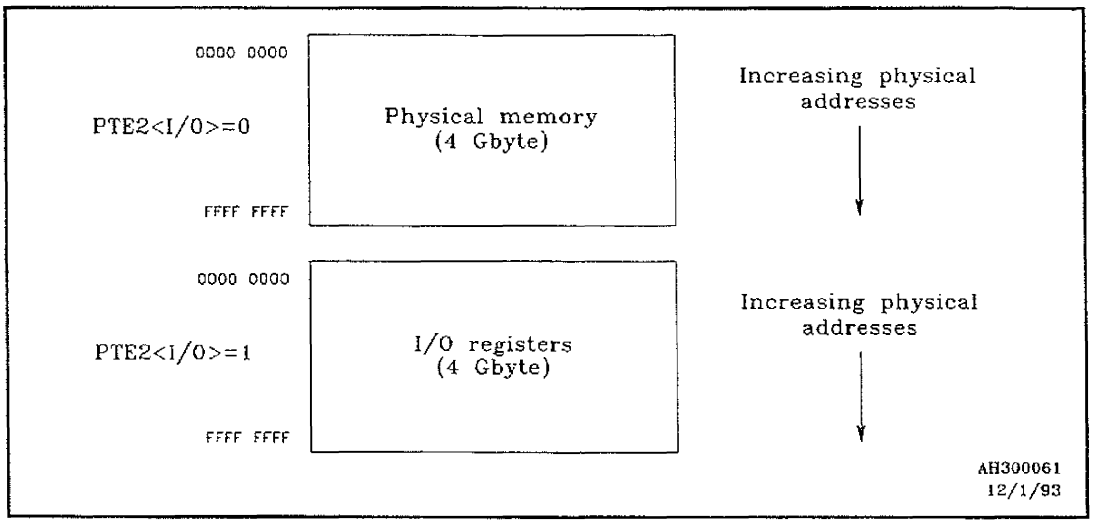


Physical address space - C3200

C3200 Series CPUs have a maximum physical memory configuration of 4 Gbytes. Physical memory may be accessed with physical addresses in the range 0000 0000 to FFFF FFFF. This address range references either physical memory or I/O registers, depending on the state of the I/O bit flag contained in the second-level PTE used to form a physical address. This I/O flag bit must be clear before physical memory can be accessed in C3200 Series CPUs.

Figure 79 shows the physical address space for C3200 Series CPUs.

Figure 79 Physical address space—C3200 Series CPUs



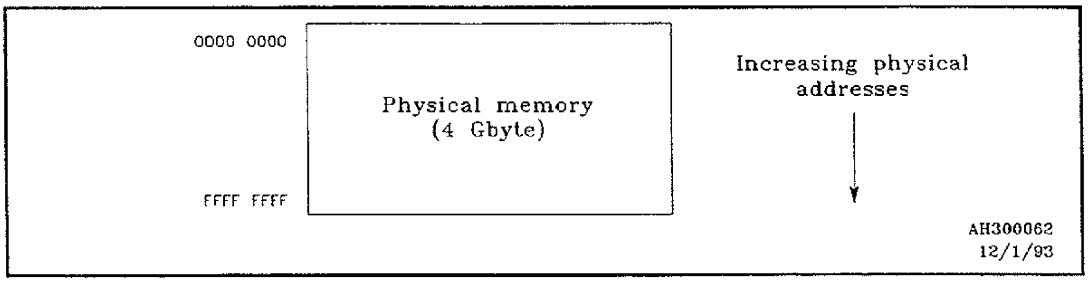
Physical address space - C3400/C3800/C4600

C3400/C3800/C4600 Series CPUs have a maximum physical memory configuration of four Gbytes. Physical memory may be accessed with physical addresses in the range 0000 0000 to FFFF FFFF. This address range references only physical memory.

C3800/C4600 Series CPUs do not have explicit I/O space defined in physical memory.

Figure 80 shows the physical address space for C3400/C3800/C4600 Series CPUs.

Figure 80 Physical address space—C3400/C3800/C4600 Series CPUs



C-Series physical memory is not always contiguous. Physical memory address space usually contains large areas of nonexistent memory, like physical memory that is not installed. The location of *installed* physical memory is maintained by hardware in the physical configuration map (PCM). Any attempt to access nonexistent memory causes a machine exception.

I/O address space

The I/O address space definitions vary between the C100 Series CPUs and the multiprocessing C-Series CPUs.

All I/O operations in C-Series CPUs are memory mapped. The architecture does not define instructions to directly reference I/O registers. I/O registers and status bits are referenced through virtual-to-physical address mapping. I/O register data is never accelerated into a cache, which eliminates multiple copy problems.

The I/O address space in the C-Series architecture allows access to a set of machine-specific functions and registers. Only a fraction of the available address space implements physical registers.

As a result of using memory mapped I/O, certain types of operand references may cause undesirable side effects. Generally, I/O operand references should be on an integral byte or halfword boundary, so that the least-significant address bits equal to the precision of the referenced operand are all zeros.

Depending on the CPU and architecture, the I/O address space is used to implement the following:

- A set of memory management referenced and modified bits (R&M bits)
- A physical configuration map (PCM) that identifies the amount and location of installed physical memory
- An interval timer (refer to the "Interval timers" section on page 279 for more information)
- A time of century (TOC) clock (refer to the "Time of century clocks" section on page 287 for more information)

Caution

The only operations allowed on I/O mapped pages are loads and stores of the correct size. I/O mapped pages are always protected for ring 0 access only. Test-and-set (`tas`) instructions executed on I/O pages outside of ring 0 are trapped and result in a fatal system exception.

I/O address space - C100

C100 Series CPUs define 1 Gbyte of I/O address space in the range from `4000 0000` to `7FFF FFFF`. The I/O address space is accessed by a 31-bit physical address where the most-significant bit is 0.

I/O address space - C3200

C3200 Series CPUs define four Gbytes of I/O address space in the range from `0000 0000` to `FFFF FFFF`. The I/O address space is accessed by a 32-bit physical address. This address space is accessed by a translated virtual address whose second-level PTE defines the page as being in I/O address space (the I/O flag bit is set).

Attempting to access a nonexistent I/O address causes a machine exception. The restrictions to the alignment and size of operands applying to I/O address space accesses are described in each I/O register functional description.

Referenced and modified bits

The physical address referenced and modified (R&M) bits are used by the operating system to track memory utilization. There is one reference bit and one modified bit for each 4-Kbytes page of physical memory. Each C-Series system implements these bits differently.

The R&M bits are implicitly modified by successful accesses to main memory. A read, write, or execute access sets the referenced bit. A write access also sets the modified bit. In addition, both R&M bits may be explicitly accessed by reading or writing a byte operand in I/O address space for C100/C3200 Series CPUs, or main memory for C3400/C3800/C4600 Series CPUs.

On C100 Series CPUs, the R&M bits reflect *only* memory accesses from the CPU. Memory access via the peripheral bus (PBUS) has no effect on the referenced or modified bits. They cannot be accessed explicitly by the I/O system.

The multiprocessing C-Series CPUs treat CPU and PBUS memory accesses differently. All multiprocessing CPU memory ports have explicit access to the I/O registers in general, and the R&M bits in particular. Although the PBUS has I/O access to the R&M bits, memory accesses through the PBUS are *not* reflected in the R&M bits.

R&M bits - C100

C1 CPUs have a maximum physical memory configuration of 128 Mbytes. The physical cache unit (PCU) contains 32 Kbytes of referenced bits and 32 Kbytes of modified bits. The addresses of these bits are byte granular and are located in previously undefined I/O address space. R&M bits are densely packed with eight bits in a byte, with bit <0> corresponding to the least-significant page address. For example, bit <0> of physical byte address `4000 0000` reflects the referenced history of page 0.

The data path is one byte wide, allowing the addresses to be accessed only through a byte *load* or *store* instruction. Attempts to read or write the addresses using other operand sizes produce incorrect and unspecified results. Attempts to access addresses with invalid contents produce a fatal system error.

C120 CPUs have a maximum physical memory configuration of 1 Gbyte. The physical cache unit (PCU) contains 256 Kbytes of referenced bits and 256 Kbytes of modified bits. The addresses of these bits are byte granular and are located in previously undefined I/O address space. The C120 mechanism is identical

in all respects to the C1, except for the larger number of bits required.

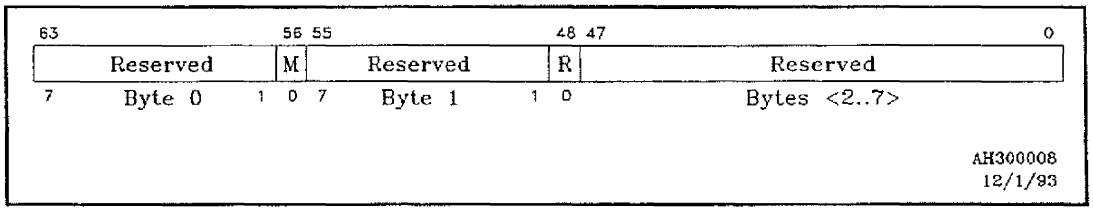
R&M bits - C3200

On C3200 Series CPUs, the R&M bits are implemented in the I/O address space. An access to the I/O address space in a C3200 Series CPU complex is defined by the I/O bit flag in the second-level PTE. Any I/O memory references through the PBUS do not affect the state of the R&M bits. When power is first applied, the state of the R&M bits is indeterminate.

The R&M bits are sparsely packed with one referenced bit and one modified bit per longword of I/O register space. Only byte 1 and byte 0 of each longword are defined. Bit <0> of byte 0 contains the modified bit, and bit <0> of byte 1 contains the referenced bit for a given page. *All other bits contain random data.* This data must be accessed with either a byte or a halfword operand. Word or longword accesses will produce a fatal system error.

The R&M bits for each page are located together in the most-significant 16 bits of each 64-bit I/O location, in the format shown in Figure 81.

Figure 81 Memory page referenced and modified bits—C3200 Series CPUs



C3200 Series CPUs have a maximum physical memory configuration of 4 Gbytes (1024 k pages). Since each page has one referenced bit and one modified bit, eight Mbytes (1024 k longwords) of I/O address space are required to accommodate 1 Mbit of referenced bits and 1 Mbit of modified bits.

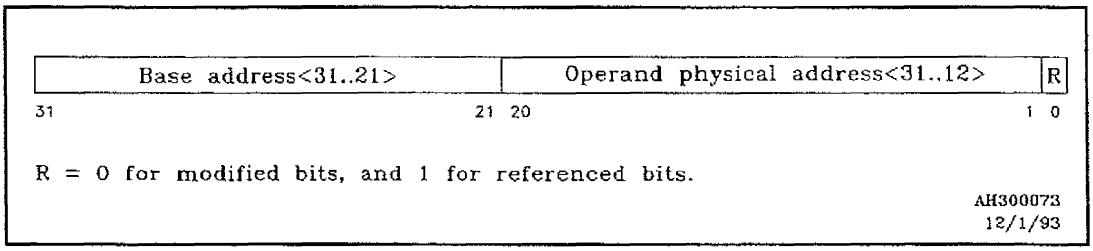
R&M bits are maintained by hardware. These registers are located in eight Mbytes of reserved I/O address space (from `0000 0000` through `007F FFF8`).

R&M bits - C3400/C3800

In the C3800/C3400 Series complexes, the R&M bits are located in a dedicated area of physical memory. This 2 Mbytes of main memory is contiguous and begins on a 2 Mbyte boundary, set by the SPU and loaded into a base address register of each CPU. Bits <31..21> of this base address are loaded into bits <31..21> of each R&M bit address. Bits <31..12> of the data operand are loaded into bits <20..1> of each R&M bit address.

Figure 82 shows the determination of the R&M bit addresses.

Figure 82 Referenced and modified bit addresses—C3400/C3800 Series CPUs



The R&M bits are sparsely packed with the modified bits residing in bit <0> of even addresses, and the referenced bits residing in bit <0> of odd addresses. The other bits of the referenced and modified bytes are indeterminate. Bit <0> of each R&M bit address is 0 (even) for the modified byte and 1 (odd) for the referenced byte. This type of interleaving allows the CPU to set both R&M bits for a page in a single access.

When a memory access could cause a change in the R&M bit status, the CPU issues an additional transparent (to the program) memory write cycle to properly maintain these bits.

The R&M bits are also accelerated in a CPU's PTE cache. When a CPU issues a write cycle to set an R&M bit, the corresponding bit in the CPU's PTE cache is also updated. This cache is used to determine if an access could cause a change in the R&M bit status. If the appropriate cache bit is set for an access, the R&M bits in memory are not altered. But if the cache bit is not set for an access, an additional write is issued and the bit in that CPU's PTE cache is set.

Two privileged instructions, `pref` and `pmod`, are available on C3400/C3800 Series CPUs to maintain cache consistency for the reference and modified bits. Anytime software clears the modified bits, `pmod` must be used to purge the modified bits in the PTE cache of the entire complex. If a referenced bit is cleared, `pref` must be used to purge the referenced bits in the complex.

Refer to the *CONVEX Assembly Language Reference Manual (C Series)* for specific information about these two instructions.

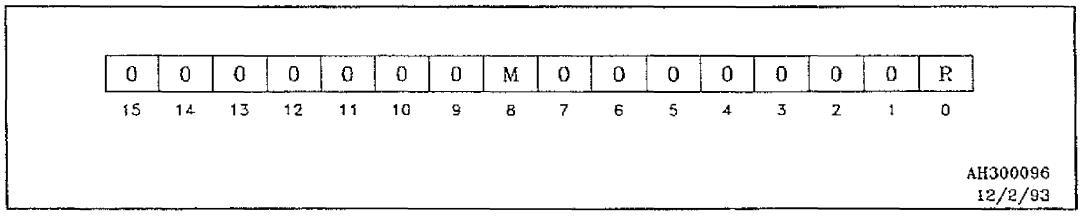
R&M bits - C4600

In the C4600 Series complexes, the R&M bits are located in a dedicated area of physical memory. The R&M Base address register (MRBASE) supplies bits <34..24>, and bits <34..12> of the data operand address are loaded into bits <23..1> of the R&M bit address. The R&M bits begin on a 16-Mbyte boundary. The MRBASE register can be set only by scan.

There is one modified and one referenced bit in main memory for each physical page available to the system. The modified bit is set for every successful write to a physical page, and the referenced bit is set for every successful read, write or execute to a physical page. A successful read, write or execute is one that does not result in a PTE access fault. In order to avoid a referenced or modified bit write for every memory reference, each CPU maintains referenced and modified flags in its PTE cache. The integrity of these flags must be maintained by software.

The format of the hardware modified/referenced bits for a single page is shown in Figure 83.

Figure 83 Referenced/Modified Bits



The referenced and modified bits are allocated one bit per byte shown above with modified bits occupying even byte locations and referenced bits occupying odd byte locations.

- **Modified** – When set, the physical page associated with this modified bit has been modified.
- **Referenced** – When set, the physical page associated with this referenced bit has been validly accessed.

To ensure the PTE cache's R&M bits agree with the R&M bits in memory, the PTE cache's R&M bits must be purged using the `patu` instruction anytime software changes the state of the R&M bits in memory. A `pate` instruction may be used to purge the R&M bits for a single PTE cache entry.

The `pref` and `pmod` instructions do not exist on the C4600.

Physical configuration map

In C-Series CPUs, the physical configuration map (PCM) in I/O address space contains the CPU type and the amount and location of physical memory installed. The size of this region (number of entries) is machine-specific, but all C-Series CPUs implement the same format for the PCM entry.

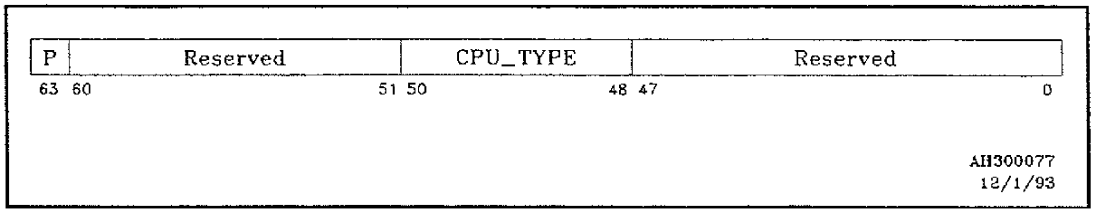
On C3400/C3800/C4600 Series CPUs, the PCM is not a software-readable entity.

Every PCM entry contains a 3-bit code describing the CPU type.

Memory is configurable in 2-Mbyte blocks. There is an entry in the PCM for each block. If the block is installed, the present (P) bit in the corresponding PCM entry is set.

Figure 84 shows the format of a PCM entry for a 2-Mbyte block of memory.

Figure 84 Physical configuration map entry



The following subsections define he PCM entry fields:

Bit <63>—Present (P)
This bit indicates whether or not the 2-Mbyte block of physical memory associated with this entry is installed in the machine.

Bit <62..51>—Reserved
These bits contain random data.

Bits <50..48>—CPU_TYPE
This 3-bit field indicates the CPU type. The hexadecimal codes are: 0x7 for C100 Series and 0x6 for C3200 Series. Any other code is undefined (there is no code for C3400, C3800, or C4600 Series CPUs).

Bits <47..0>—Reserved
These bits contain random data.

Physical configuration map - C100

C1 CPUs have a maximum main memory capacity of 128 Mbytes. The PCM on C1 CPUs is a 64-longword region from I/O address 6FFF FC00 to 6FFF FDF8. The P bit in the first location (6FFF FC00) corresponds to the 2-Mbyte block starting at physical address 0000 0000, and so forth.

This region is only accessible on C1 CPUs from the PBUS.

C120 CPUs have a maximum main memory capacity of 1 Gbyte. The PCM on C120 CPUs is a 512-longword region from I/O address 6FFF 8000 to 6FFF 8FFF. The P bit in the first location (6FFF 8000) corresponds to the 2-Mbyte block starting at physical address 0000 0000, and so forth.

To provide backward compatibility with C1 software, the I/O addresses from 6FFF FC00 to 6FFF FDF8 are mapped by the hardware to addresses 6FFF 8000 to 6FFF 81F8.

This region is only accessible on C120 CPUs from the PBUS (in the same manner as C1 CPUs).

Physical configuration map - C3200

C3200 Series CPUs have a maximum main memory capacity of 4 Gbytes. The PCM for these complexes is a 2,048-longword region from 6FFF 8000 to 6FFF BFF8. The P bit in the first location (6FFF 8000) corresponds to the 2-Mbyte block starting at physical address 0000 0000, and so forth.

The C3200 Series CPUs ignore bits <3..0> of the PCM access address. The four least-significant bits of any PCM access address can either be 0 or 8. For example, a write access to both 6FFF 8000 and 6FFF 8008 write to the same PCM longword, even though these two longword addresses apparently describe two different blocks of physical memory.

Note

Even though the granularity of the C3200 Series PCM is a longword, these CPUs restrict access to only the upper halfword. Any accesses to the PCM to determine CPU type should be done only to the upper halfword. Attempts to read or write a word or longword to this area will result in a fatal system error.

In addition to the explicit access method, an implicit read occurs during each access to main memory. Before any physical address is accessed, it is tested to ensure it contains a valid block. An I/O reference to nonpresent memory results in an aborted transfer and an error status being returned to the device initiating the request. A CPU reference to nonpresent memory results in a system fatal hard error.

Physical configuration map - C3400/C3800/C4600

The C3400/C3800/C4600 Series CPUs do not support a software-readable PCM.

Timers

The C-Series architecture has several timers provided to permit fine grain, accurate accounting of CPU execution time and to assist in process scheduling. There are four timers in the multiprocessing C-Series architecture.

- Interval timer counter (ITC)—All C-Series CPUs

  All C-Series CPUs contain an interval timer counter (ITC) used to interrupt the processor at a programmable rate. The implementation of the interval timer is processor-specific, but the logical structure of the interval timer is the same on all C-Series systems. Each system contains an ITC, a next interval timer count register (NITC), and an interval timer status register (ITSR). The ITC advances (increments or decrements) at a fixed processor-specific rate. Each time the ITC reaches zero, it is loaded from the NITC.

  There is only one ITC in each C-Series system, regardless of the number of CPUs in the system.

  The ITSR controls the operation of the counter and the generation of interrupts. The ION flag enables and disables the interval timer *only* in the C100 Series architecture.

- Time of century clock (TOC)—C100 CPUs

  In the C100 Series architecture, the operating system software implements a TOC clock via the C100 Series interval timers.

- Time of century clock (TOC)—All multiprocessing CPUs

  The multiprocessing C-Series architecture implements a TOC clock in hardware that can be both read and written. This clock keeps "wall clock time," not user time.

- CPU execution timer (CTR)—All multiprocessing CPUs

  The multiprocessing C-Series architecture includes CPU execution timers (CTR) for each process that maintain microsecond timing for each CPU's time spent in rings 0–3 (system time) and ring 4 (user time).

- Thread timer (TTR)—All multiprocessing CPUs

  The Multiprocessing C-Series architecture has a thread timer that measures the elapsed CPU time for executing a thread. This thread timer can be both read from and written to.

Interval timers

All C-Series systems contain an interval timer counter (ITC). The implementation of the interval timer is processor-specific.

Interval timers - C100

The C100 Series interval timer is *not* located in I/O address space. It is internal to the CPU, and the actual structure is only visible to microcode. The interval timer has a programmable interrupt frequency range of 1 MHz to slightly greater that 1 Hz. The ION interrupt flag enables and disables the C100 Series interval timer, depending on the logic sense of this flag.

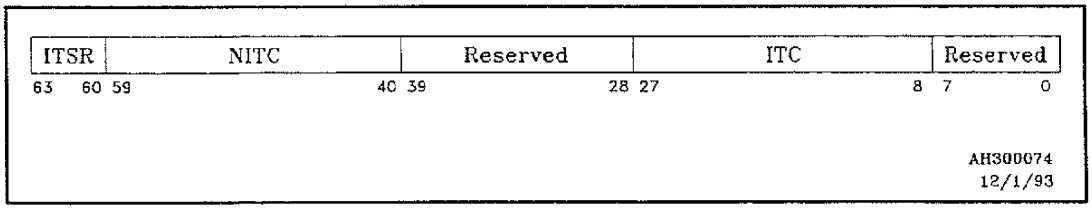
The C100 Series interval timer is implemented by a 20-bit interval timer counter (ITC), a 20-bit next interval timer register (NITC), and an interval timer status register (ITSR). A 2-MHz clock, divided by two, increments the C100 ITC counter every microsecond. When the counter increments to the maximum count value (`0FFF FF00`), the ITC is reloaded from the NITC on the next clock and the full bit in the ITSR is set. If the full bit is already set, the overflow bit is also set.

At the macro-instruction level, a single 64-bit register represents the interval timer. It is manipulated by a set of three instructions. Two of these instructions access the whole 64 bits, the third is used for interrupt servicing and writes only the control bits:

- `mov ITR, Sk`—The contents of the interval timer registers (ITC, NITC, and ITSR) are moved (copied) to a scalar register.
- `mov Sk, ITR`—The contents of a scalar register are moved to the interval timer registers (ITC NITC, and ITSR).
- `mov Sk, ITSR`—The four most-significant bits of a scalar register are moved (copied) to the ITSR register.

Figure 85 presents the format of C100 Series architecture interval timer registers.

Figure 85 Interval timer registers—C100 Series



The following subsections define these fields.

Bits <63..60>—Interval timer status register (ITSR)
Controls the operation of the interval timer counter and controls the generation of interrupts. The interval timer status register (ITSR) contains four individual bits. These bits are defined as:

- **Bit <63>—On**
  When this bit is set, the interval timer will count. When this bit is cleared, the interval timer stops.
- **Bit <62>—Interrupt enable**
  When this bit is set, an interrupt is signaled to the CPU when the full bit (<61>) is set. When bit <62> is cleared, no interrupt is generated.
- **Bit <61>—Full or decrement**
  This is the full status bit during reads and the decrement control bit during writes.
- **Bit <60>—Overflow**
  This bit is the overflow status bit.

Bits <59..40>—Next interval timer counter (NITC)

A 20-bit register loaded into the ITC when it reaches terminal count. The NITC register is not affected by loading the ITC with the value in the NITC. The value in the NITC can be used repeatedly.

Bits <38..28>—Reserved
Reserved for future use.

Bits <27..8>—Interval timer counter (ITC)
A 20-bit count-up timer that increments every microsecond. On the count that clears the ITC to zero, the ITC is loaded with the value from the NITC and signals the control logic that a terminal count (ITC cycle) has occurred.

Bits <7..0>—Reserved
Reserved for future use.

The overflow and full bits form a 2-bit pseudocounter, with the full bit as the least-significant bit of the counter. Whenever the ITC reaches terminal count, this pseudocounter counts up one step. Whenever the ITSR or ITR is written with data in which bit<61> is set, this pseudocounter counts down. The actual occurrence of interval timer interrupts has no effect on the full or overflow bits.

The values of the full and overflow bits and corresponding events are listed in Table 51.

Table 51 Full and overflow bit values and events—C100 Series

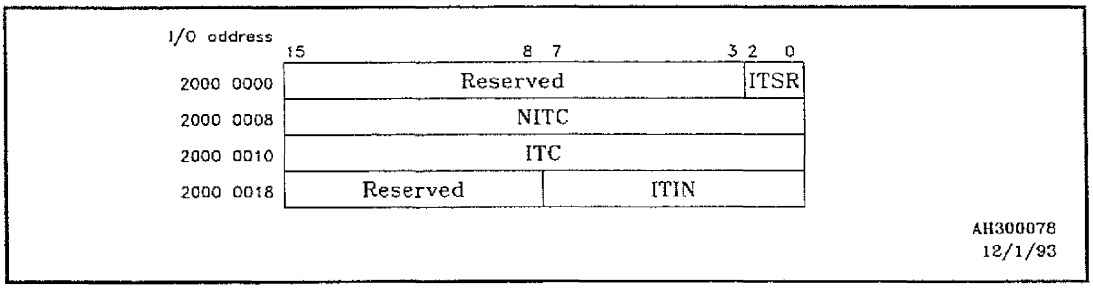
Event	Current values		New values	
	Overflow bit	Full bit	Overflow bit	Full bit
ITC terminal count	0	0	0	1
	0	1	1	1
	1	1	1	1
Write with ITSR<61> = 1	1	1	0	1
	0	1	0	0
	0	0	0	0

Interval timers - C3200

The interval timer on the C3200 Series systems is accessed via I/O address space. The interrupt frequency of this interval timer is programmable from 100 kHz to about 1.6 Hz. This timer is controlled by a set of four 16-bit registers, longword aligned at I/O addresses 2000 0000 to 2000 0018. The interval timer decrements by one at ten microsecond intervals. Because this timer is located in I/O address space, no special instructions are required to service it.

Figure 86 presents the address locations and format of the interval timer registers for the C3200 Series systems.

Figure 86 Interval timer registers—C3200 Series



Interval timer status register (ITSR)
A 16-bit register located at I/O address 2000 0000. Only the least-significant three bits of the ITSR are valid. These bits of the interval timer status register are defined as:

- **Bits <15..3>—Reserved**
  Reserved for future use.
- **Bit <2>—On**
  When this bit is set, the interval timer (ITC) is active and counts by decrementing in bit 0. When this bit is cleared, the interval timer is halted. Interrupts generated by the interval timer are disabled by disabling the virtual interrupt channel specified in the ITIN.
- **Bit <1>—Underflow**
  This bit is *read only* and is set when the ITC underflows and the Empty bit has already been set by a previous ITC cycle. The ITC completes a cycle by reaching the terminal count value located in the NITC. The underflow bit is cleared by reading the ITSR.
- **Bit <0>—Empty**
  This bit is *read only* and is set when the ITC underflows. This bit is cleared at the same time as the underflow bit (when the ITSR is read as a normal part of the interrupt service). This bit is set when the ITC completes a cycle by reaching terminal count and underflowing.

Next interval timer counter (NITC)
A 16-bit register located at I/O address 2000 0008. The NITC contains the count value to be loaded into the ITC upon underflow. It can be both read and written to.

Interval timer counter (ITC)
A 16-bit countdown timer located at I/O address 2000 0010. The ITC is loaded from the NITC each time it decrements to zero. It can be both read and written to.

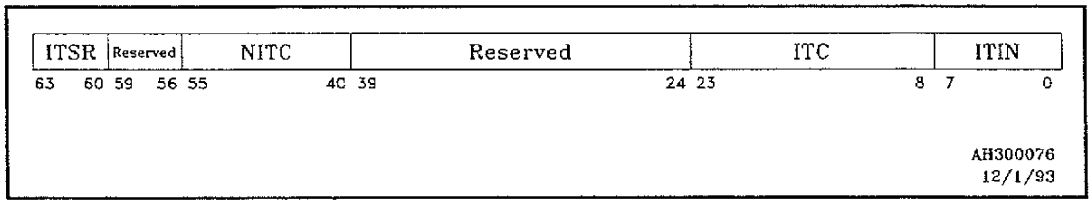
Interval timer interrupt number (ITIN)
A 16-bit register located at I/O address 2000 0018. Only the least-significant byte is valid, and it contains the virtual interrupt channel number that receives timer underflow interrupts. The interval timer interrupts can be disabled by disabling the corresponding virtual interrupt channel.

Interval timers - C3800/C4600

The interval timer on the C3800/C4600 Series is accessed via special instructions. The interrupt frequency of this interval timer is programmable from 100 kHz to about 1.6 Hz. This timer is controlled by a 64-bit register. The interval timer decrements by one at ten microsecond intervals. The instruction `mov ITR, Sk` is used to place the contents of the interval timer registers into a scalar register. The instruction `mov Sk, ITR` is used to write to the interval timer registers. The interval timer status register may be written separately with the instruction `mov Sk,ITSR,` which copies the highest four bits of `Sk` into `ITSR`

Figure 87 presents the format of the interval timer for the C3800/C4600 Series systems.

Figure 87 Interval timer registers—C3800/C4600 Series systems



Bits <63..60>—Interval timer status register (ITSR)
Controls the operation of the interval timer counter and controls the generation of interrupts. The interval timer status register (ITSR) contains four individual bits. These bits are defined as:

- **Bit <63>—Reserved**
  Reserved for future use.
- **Bit <62>—On**
  When this bit is set, the interval timer (ITC) is active and counts by decrementing in bit 0. When this bit is cleared, the interval timer is halted. Interrupts generated by the interval timer are disabled by disabling the virtual interrupt channel specified in the ITIN.
- **Bit <61>—Underflow**
  This bit is *read only* and is set when the ITC underflows and the empty bit has already been set by a previous ITC cycle. The ITC completes a cycle by reaching the terminal count value located in the NITC. The underflow bit is cleared by reading the ITSR.

- **Bit <60>—Empty**
  This bit is *read only* and is set when the ITC underflows. It is cleared at the same time as the underflow bit (when the ITSR is read as a normal part of the interrupt service. It is set when the ITC completes a cycle by reaching terminal count and underflowing.

Bits <59..56>—Reserved
Reserved for future use.

Bits <55..40>—Next interval timer counter (NITC)
Loaded into the ITC when it reaches terminal count. The NITC register is not affected by loading the ITC with the value in the NITC. The value in the NITC can be used repeatedly. Since the ITC is incremented every ten microseconds, the count period loaded into the NITC should be set at a multiple of 10 µs.

Bits <39..24>—Reserved
Reserved for future use.

Bits <23..8>—Interval timer counter (ITC)
A 16-bit countdown timer that is decremented every 10 microseconds. Each time the ITC decrements to zero, it is loaded with the value from the NITC. It can be both read and written.

Bits <7..0>—Interval timer interrupt number (ITIN)
An eight-bit field of the ITR. It contains the virtual interrupt channel number that receives timer underflow interrupts. The interval timer interrupts can be disabled by disabling the corresponding virtual interrupt channel.

Interval timers - C3400

The C3400 Series CPUs have four interval timers. One of these timers is dedicated for the timesharing subcomplex, and three are for the real-time subcomplex. The registers for these timers are in the control registers for the communication registers. See the "Control registers - C3400" section in Chapter 5 for details on the physical layout.

Timesharing interval timers
The timesharing subcomplex interval timer has four registers associated with it: ITSR_U, NITC_U, ITC_U, and ITIN_U.

- **Interval timer status register (ITSR)**

  The interval timer status register (ITSR) controls the operation of the interval timer counter and controls interrupt generation.

The timesharing ITSR contains three bits. These bits are defined as:

- **Bit <2>—On**
  When this bit is set, the interval timer (ITC) is active and counts by decrementing in bit 0. When this bit is clear, the interval timer is halted. Interrupts generated by the interval timer are disabled by disabling the virtual interrupt channel specified in the ITIN.
- **Bit <1>—Underflow**
  This bit is *read only* and is set by incrementing whenever the ITC underflows and the empty bit has already been set by a previous ITC cycle. The ITC completes a cycle by reaching the terminal count value located in the NITC. The underflow bit is cleared by reading the ITSR.
- **Bit <0>—Empty**
  This bit is *read only* and is set whenever the ITC underflows. It is cleared at the same time as the underflow bit, when the ITSR is read as a normal part of the interrupt service. I t is set when the ITC completes a cycle by reaching terminal count and underflowing.

- **Next interval timer counter (NITC)**

  This 16-bit register is loaded into the ITC when it reaches terminal count. The NITC register is not affected by loading the ITC with the value in the NITC. The value in the NITC can be used repeatedly.

- **Interval timer counter (ITC)**

  This 16-bit count-down register is decremented every 10 or 100 microseconds depending on bit 3 of the ITSR. Each time the ITC decrements to zero, it is loaded with the value from the NITC. It can be both read and written to.

- **Interval timer interrupt number (ITIN)**

  The interval timer interrupt number (ITIN) is a 16-bit field of the ITR. It contains the virtual interrupt channel that receives timer underflow interrupts. The interval timer interrupts can be disabled by disabling the corresponding virtual interrupt channel.

Realtime interval timers

The three realtime subcomplex interval timers (0, 1, 2) have three associated registers, ITSR, NITC, and ITC. These timers are dedicated to an interrupt channel, therefore an ITIN is not needed.

- **Interval timer status register (ITSR)**

  This register controls the operation of the interval timer counter and controls the generation of interrupts. The ITSR contains four individual bits. These bits are defined as:

  - **Bit <3>—Interval**
    This bit sets the clock interval for the interval timers. When this bit is set, the timers increment each ten microseconds. When this bit is reset, the timers increment every 100 µs.
  - **Bit <2>—On**
    When this bit is set, the interval timer (ITC) is active and counts by decrementing in bit 0. When this bit is clear, the interval timer is halted. Interrupts generated by the interval timer are disabled by disabling the virtual interrupt channel specified in the ITIN.
  - **Bit <1>—Underflow**
    This bit is *read only* and is set by incrementing whenever the ITC underflows and the empty bit has already been set by a previous ITC cycle. The ITC completes a cycle by reaching the terminal count value located in the NITC. The underflow bit is cleared by reading the ITSR.
  - **Bit <0>—Empty**
    This bit is *read only* and is set when the ITC underflows. It is cleared at the same time as the underflow bit, when the ITSR is read as a normal part of the interrupt service. It is set when the ITC completes a cycle by reaching terminal count and underflowing.

- **Next interval timer counter (NITC)**

  This 16-bit register is loaded into the ITC when it reaches terminal count. The NITC register is not affected by loading the ITC with the value in the NITC. The value in the NITC can be used repeatedly.

- **Interval timer counter (ITC)**

  This 16-bit count-down register is decremented every 10 or 100 microseconds, depending on bit 3 of the ITSR. Each time the ITC decrements to zero, it is loaded with the value from the NITC. It can be both read and written to.

Time of century clocks

The time of century clock (TOC) keeps "wall clock" time, not user time. It is not saved and restored during context switches. It is, therefore, not altered by context switches and will keep time indefinitely.

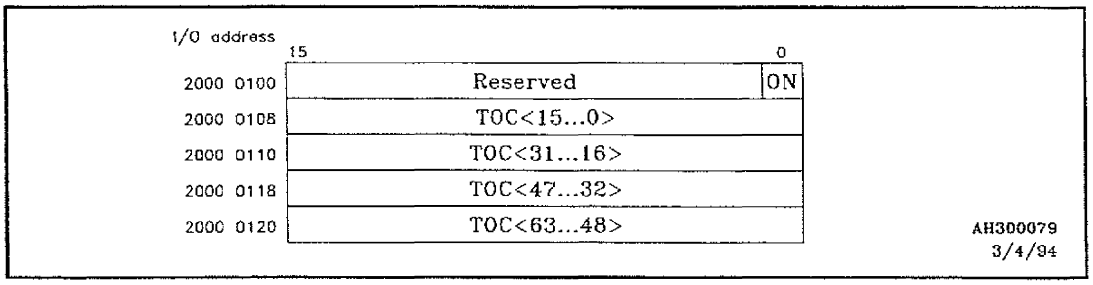
The multiprocessing C-Series CPUs implement a 64-bit time of century clock (TOC) that keeps time in 1-µs increments and overflows every 500,000 years.

TOC - C3200

The C3200 Series TOC clock is implemented as four 16-bit I/O locations for the counter, and one 16-bit I/O location for the counter status. All I/O addresses are developed from the page 0, ring 0 I/O register pointer. Refer to the "Virtual address space" section on page 70 for more information regarding the I/O register pointer. Refer to the"I/O address space" section on page 268 for information about other uses of I/O address space.

Figure 88 presents the logical structure of the C3200 Series TOC, with the actual physical I/O addresses for the C3200 Series architecture.

Figure 88 64-bit TOC clock—C3200 Series CPUs



The most-significant bit of the C3200 Series TOC clock is contained in the most-significant bit of I/O address `2000 0120` The least-significant bit is located in the least-significant bit of I/O address `2000 0108`.

The TOC increments every microsecond. Before the TOC can be written, it must be turned off by clearing the on bit (0). All four remaining clock locations should then be written to, and the TOC set to continue counting by setting the on bit (1).

The TOC should be disabled by clearing the control register (which disables counting) until loading the TOC is completed. This action avoids having the TOC incorrectly loaded while it continues to increment.

The TOC must be written to 16 bits at a time. To correctly load a 64-bit value in the TOC counter, the TOC must be written at four separate locations, with four separate instructions.

A special instruction, `mov TOC, Sk`, simplifies the process of reading the TOC. This instruction will atomically read the TOC information by performing the actual I/O accesses in microcode. The `mov TOC, Sk` instruction cannot be interrupted when the TOC is actually being read.

TOC - C3400/C3800/C4600

The C3400/C3800/C4600 Series implements the TOC in X-space as a single 64-bit counter. It can be initialized using the privileged instruction `mov Sk, TOC`. The TOC should be initialized immediately after the processor is powered up.

The TOC will count continuously thereafter by incrementing in TOC <0> in 1-µs intervals.

There is no way to turn off the TOC on C3400/C3800/C4600 Series CPUs.

A special instruction, `mov TOC, Sk`, is used for reading the TOC information. The `mov TOC, Sk` can be executed by user code, and it is an atomic operation.

CPU execution timer

In order to provide accurate accounting information to the operating system, the multiprocessing C-Series CPUs include CPU execution timers (CTRs) for each process that maintain microsecond timing for each CPU's time spent in rings 0–3 (system time) and ring 4 (user time). The CTRs are visible to the operating system in the hardware communication registers. In a CIR-dependent manner, there is a set of CTRs for each CIR. Refer to the "CPU execution clock registers" section in Chapter 5 for the format of these timers.

These timers are maintained by microcode on a *demand* basis. This means that if a CPU is running in ring 0 and executes a `get` of the ring 0 CTR for the CPU, the time will not be the up-to-date time. The demand events that cause the microcode to update a particular timer are primarily ring crossings and execution of the `ctrsg` instruction (a privileged instruction).

The operating system uses `ctrsg` to instruct all CPUs to update their CTRs, that is, the timer for that CPU and ring. For example, on a C3200 Series CPU, if one thread of a process is the operating system kernel executing on CPU0 (in ring 0) and the other thread is a user program in ring 4, the operating system thread can execute `ctrsg` and then read the ring 4 or ring 0 CTR and know that the time is accurate. This global timer updating is implemented with the processor trap mechanism.

On C3200/C3800/C4600 Series complexes, the CPU execution timer is implemented with a single, hardware delta timer on each CPU. This resettable timer counts up by microseconds. An update of the CTR is done by reading the current CTR value from the communication register, adding the current delta time to that value, writing the sum back to the CTR, and clearing the delta timer.

The C3400 Series CPUs use the TOC and a communication register (CTR). The delta value is the difference between the TOC and the CTR value. After access, the CTR is loaded with the current TOC to provide the delta value the next time it is accessed.

Thread timer

The multiprocessing C-Series CPUs have one 64-bit microsecond timer per thread, implemented in microcode (not in a register), that is accessed by nonprivileged instructions. The thread timer (TTR) allows each thread to determine the CPU execution time of any code region without the overhead of a system call. This register only reflects the CPU time on a ring-specific basis and cannot be used to time inner ring calls. It increments in bit <0> whenever a CPU is executing a thread. It can be read or written to at any time by the currently executing thread.

The C3200/C3800/C4600 Series complexes implement the thread timer on each CPU and update it using the same hardware delta timer used to implement the CTRs. The delta timer is a microsecond timer that exists on each CPU. It is used to time intervals between accesses to the TTR or CTR. The thread timer is updated by adding the delta timer to the current TTR's value and clearing the delta timer.

The C3400 Series complexes use the TOC and a communication register. On demand, the delta value is the difference between the TOC and the communication register value. After access, the communication register is loaded with the current TOC to provide the delta value for the next time it is accessed.

The `mov TTR,Sk` instruction reads the thread timer by updating the TTR's value and copying the updated value to Sk. The `mov Sk,TTR` instruction writes the thread timer by copying Sk to the current TTR's value and clearing the delta timer. Refer to the *CONVEX Assembly Language Reference Manual (C-Series)* for more detailed information on the instructions used to access the thread timer register.

The thread timer is primarily used for timing sections of code running in ring 4, without including time spent in asynchronous events such as interrupts and page faults. The thread timer is not as effective in ring 0, since there are many events that can change CIR and TID in ring 0 and not affect the thread timer. These events do not affect the TTR since the old CIR or TID's thread timer is not saved, as it is in the extended frame on ring crossings.

The thread timer register is saved on the stack on all cross-ring calls, and restored from the stack on all cross-ring returns. This enables the timer to track a particular thread's context if the thread migrates between CPUs during its execution.

On entry to the inner ring, the thread timer is cleared, although the CPU maintains the same TID. The outer ring timer value is saved in the return block, and the value is restored when control is returned to the outer ring. If an extended frame is pushed on the stack without a ring crossing (that is, a system call to ring 0 from ring 0, and so on), the thread timer value in the extended frame is undefined. The subsequent `rtn` instruction examines the PC and determines no ring crossing occurred, so the thread timer is not popped from the extended frame.

For example, if the thread takes an interrupt, the TTR is saved on the extended frame before entering the ring 0 interrupt handler. It is restored on the subsequent return from the interrupt handler.

CTR and TTR manipulation

The multiprocessing C-Series CPU execution timers (CTR) and the thread timers (TTR) timers are closely related since both timers are maintained with the same delta timer. When an event forces the CTR to be updated, the delta timer is cleared, so the TTR must be updated at the same time. Similarly, when an event forces the TTR to be updated, the CTR must be also updated.

Because these timers are so closely related, the specific order of actions that manipulate the CTR and TTR are organized by event, rather than by each timer, and are as follows:

- Power-up (cold start)—The TTR and delta timer are cleared.
- Push extended frame (interrupts, exceptions)—If a ring crossing occurs, the CTR and TTR are updated, the TTR is pushed, and the delta timer is cleared.
- Pop extended frame (`rtn`)—If a ring crossing occurs, the TTR is popped from the extended frame, the CTR is updated, and the delta timer is cleared.
- Page fault—The TTR is pushed to the context block. If a ring crossing occurs, the CTR is updated, and the TTR and delta timer are cleared.
- `rtnc`—If a ring crossing occurs, the TTR is popped from context block, the CTR is updated, and the delta timer is cleared.
- `ctrsg`—The CTR and TTR are updated, and the delta timer is cleared (all CPUs).
- `mov Sk,TTR`—The TTR is written, the CTR is updated, and the delta timer is cleared.
- `mov TTR,Sk`—The TTR and CTR are updated, and the delta timer is cleared.
- `mov Sk,CIR`—The CTR is updated (in the old CIR), and the delta timer is cleared. The TTR is *not* modified.
- `mov Sk,TID`—No action is taken. The TTR is *not* modified.
- `stcmr`—The CTR is updated (so the CTR is stored correctly), the TTR is updated, and the delta timer is cleared.
- `ldcmr`—The CTR is updated (after loading, in case the current CIR is loaded), the TTR is updated, and the delta timer is cleared.

- `idle Sk`—The CTR is updated (in the old, pre-Sk CIR), and the delta timer is cleared. If the fork is accepted in CIR Sk, the TTR is cleared.
- `wfork`—The current CIR accepts a fork. No action is taken.
- Enter the CPU idle loop (thread termination)—The CTR is updated. The event relies on subsequent thread creation to clear the delta timer.
- Idle CPU takes interrupt—The delta timer is cleared.
- Idle CPU accepts fork—The TTR and delta timer are cleared.
- Base level interrupt—The CTR is updated, and the delta timer is cleared, regardless of whether or not a ring crossing occurred.

Event counter - C4600

The C4600 Series CPUs have a 64-bit event counter (EVCNT) that counts the number of occurrences of an event selected by an event select register (EVSEL). EVCNT and EVSEL may be written to or read by the user at any time. The events that may be selected by EVSEL are shown in Table 52. These registers are saved on the extended return block during inward ring crossings, and restored during outward ring crossings.

Table 52 Values for EVSEL register

$EVSEL_{3..0}$	Event
0	Data Cache misses - increments by one for each Data Cache miss. A miss occurs any time that data is not found in the Data Cache for a *Data Cache candidate* read. A Data Cache candidate read includes: A and S register `ld`, `ld0`, `ld1`, `pop`; loads of VM and VLS; all indirect cell loads; all resource structure loads; and all return instructions.
1	PTE cache misses - increments by one for each PTE Cache miss.
2	Data Cache accesses - increments by one for each Data Cache candidate read (see Data Cache misses above).
3	Vector Processor load/store elements - increments by VL each time a vector `load`, `store`, `ldvi`, `stvi` or `ste` occurs. For instructions under mask, increments by the number of asserted VM bits within the first VL bits of the VM register.
4	System clocks - increments by one each system clock (every ~7.14 ns).
5	Instruction Cache misses - increments by one for each Instruction Cache miss.
6	Floating point operations - increments by: one for each floating point scalar `add`, `sub`, `mul`, `div`, `neg`, `frint`, `sqrt`, `eq`, `gt`, `ge`, `cvt` instruction; VL for each floating point vector `add`, `sub`, `mul`, `div`, `neg`, `frint`, `sqrt`, `eq`, `gt`, `ge`, `cvt`, `sum`, `prod`, `min`, `max` instruction; 2*VL for each floating point vector `axpy`, `xypa`, and `dot`. Also increments by an implementation dependant amount for scalar `sin`, `cos`, `atan`, `exp` and `ln`. For vector floating-point instructions under mask, increments by the number of asserted VM bits within the first VL bits of the VM register (or twice this number for `axpy`, `xypa`, `dot`).
7-15	Reserved

The extended return block for the C4600 Series CPUs is described in the "Stack operations" section in Chapter 4.

Memory and cache management

CONVEX CPUs have implementation-specific cache management mechanisms. These mechanisms require purging the instruction, logical, ATU (C100 series CPUs), and PTE (multiprocessing CPUs) caches under certain conditions. This section describes the cache management mechanisms for both the C100 series CPUs and the multiprocessing CPUs.

Cache management - C100

The C100 Series CPUs perform a variety of implicit cache purges. They purge the instruction cache (Icache) for the following traps:

- Instruction trace traps
- Vector valid traps
- Arithmetic traps
- Interrupts
- Conditions other than traps.

Table 53 shows the general conditions under which Icache, Dcache, and ATU purges are performed by the C100 CPUs.

Table 53 Icache, Dcache, and ATUcache purges—C100 Series CPUs

Condition	Icache	Dcache	ATUcache	ATU entry
`patu` execution	Yes	Yes	Yes	No
`pate` execution	Yes	Yes	No	Yes
`pich` execution	Yes	No	No	No
`plch` execution	No	Yes	No	No
`ldsdr, ldkdr` execution	Yes	Yes	Yes	No
`trap/ interrupt`	Yes	No	No	No

Cache management - C3200/C3400/C3800

In the C100 series CPUs, the ATU is responsible for the virtual-to-physical address translation. The results of an address translation are encached in the ATU cache for future access. In multiprocessing C-Series CPUs, this cache is referred to as the PTE cache.

Table 54 shows the general conditions under which Icache, and PTE cache purges are performed on C3200/C3400/C3800 series CPUs.

Table 54 Instruction and PTE cache management—C3200/C3400/C3800 Series CPUs

Condition	`pich`	`patu`	`pate`
`ldcmr` execution	Software invalidate CIR_PREV	Software invalidate CIR_PREV	NA
thread termination	Hardware	NA	NA
`mov Sk, CIR` execution	Software	NA	NA
`mov Sk, TID` execution	Software	NA	NA
`patu` execution	NA	Hardware	NA
`pate` execution	NA	NA	Hardware
`pich` execution	Hardware	NA	NA
`ldsdr, ldkdr` execution	Hardware invalidate CIR_PREV	Hardware invalidate CIR_PREV	NA
before `rtnc`	NA	NA	Software
interrupt handler	Software	NA	NA
idle	Hardware		

PTE cache management

The following sections describe PTE cache management for the C3200, C3400, and C3800 Series CPUs.

PTE cache management - C3200

Two sets of validity bits are maintained for the PTE cache. This means the entire PTE cache should not be purged more than once every 1,023 CPU cycles. The PTE cache entries are tagged with CIR and TID. The PTE cache does not require purging when these values are changed.

Conversely, the PTE cache must be purged when the SDRs change. The SDRs change implicitly when the `ldcmr` instruction is executed. Since the `ldcmr`, `ldsdr`, `ldkdr`, `patu`, or `pate` instructions do not purge the PTE, software has the flexibility to control how often the cache is purged.

For example, the operating system loads multiple communication register sets with repeated `ldcmr` instructions to initialize all processes before allowing them to execute. The required cache purging is performed before execution of the `mov Sk,CIR` instruction which transfers control to a new process.

Execution of a `pate` instruction does not require the instruction cache to be purged. The `pate` is usually due to the validation of a page of resident data or text (in which case the data is correct if it is in the cache), or the invalidation of a page (a fault will occur).

Note

The interrupt handler needs only to purge the instruction cache (because it has changed to the interrupt CIR) if page zero ring 0 interrupt context is not mapped globally to all CIRs. The interrupt handler executes with multiple threads and may execute different code streams.

PTE cache management - C3400

The C3400 Series CPUs have a two level PTE cache. The outer level cache contains a purgable cache RAM with 2 k entries. The inner cache is 256 entries, and is on the chip for the scalar unit.

PTE cache management - C3800

The C3800 Series CPUs have both a level one PTE cache and a level two PTE cache. The `pate` instruction purges one entry in the level two cache. This instruction does not modify the level one cache. If the level one page tables are modified, then a `patu` must be executed to purge the entire level one cache and the entire level two cache.

Instruction cache management

The scalar processor fetches both scalar and vector instructions from the instruction cache (Icache). Each CPU in the complex maintains an Icache of recently prefetched and executed instructions. This increases performance for programs that frequently access the same virtual memory locations, such as frequently called subroutines.

The entries in the Icache are associated by virtual address only. They are not tagged with CIR or thread ID, so the instruction cache must be purged any time these values change.

Sometimes the Icache is purged without an explicit software command, for example, when a CPU leaves the idle loop and takes a fork in a new CIR (with a new TID). Otherwise, the management of the Icache is primarily controlled by using the `pich` instruction.

The Icache is purged on a per-CPU basis, using the `pich` instruction. A process must be single-threaded to execute `pich` and to ensure that the Icache for the entire process is purged.

A CPU purges the Icache for a posted fork. The `idle Sk` instruction always purges the Icache, because the Icache is also purged when a `wfork`, `cfork`, or `idle` instruction kills the current thread (new_CIR is not equal to old_CIR).

The CPU purges the Icache if a fork is taken directly in `CIR Sk` (where `Sk` specifies which CIR to enter next). If there is no fork in `CIR Sk`, the CPU must purge the Icache before entering the idle state, even though the thread is not deallocated. Although a fork may not exist in `CIR Sk`, the CPU idle loop may take a fork in another CIR.

Instruction cache management - C3200
The C3200 Series CPUs maintain two sets of validity bits for the Icache entries. At any given time, one set is used for Icache accesses and the other is cleared. When the Icache is purged, the hardware switches to the second set and clears the other. The second set then becomes the first.

It takes 1,024 CPU cycles to clear the unused copy. If a purge is initiated within 1,024 machine cycles of a previous purge, execution is halted until a cleared copy can be prepared. Software should avoid this situation.

Instruction cache management - C3400/C3800
The C3400/C3800 Series CPUs use a purgable instruction validity RAM allowing the Icache to be purged quickly. The C3800 Series Icache is 16 Kbytes, and the C3400 Series Icache is 4k instructions.

The look-ahead address generator tries to remain ahead of the instruction processor. If the instruction processor must be restarted, the look-ahead address generator is also restarted.

Data cache management

The multiprocessing C-Series CPUs use a multiport memory system. The I/O subsystem has a separate port to main memory since it performs many of the same functions as a CPU. Because the memory system is multiported, a CPU complex with shared memory must maintain the consistency of each CPU's data cache. This is done by ensuring that when one CPU has loaded data and caused a local encachement, it is notified when another CPU stores data to that address location. In this manner, both CPUs maintain a consistent view of physical memory.

Data cache management - C3200

The data cache size on the C3200 is one page, or 4 Kbytes. C3200 CPUs use a technique called *remote invalidation*, in which each CPU monitors all memory ports for data stores and invalidates its local data cache when they occur. There are some cases when invalidates are missed. For example, if one CPU is in a loop, loading and comparing a word of memory, the data could become stuck in that CPU's data cache and not be invalidated when another CPU stores data to the same address location. This would also happen if the remote invalidate from the storing CPU was received by the loading CPU before the load data returned from memory and wrote the cache.

Following these software guidelines for communication through shared memory can avoid missing remote invalidates.

- A lock byte with full semaphoring must be maintained around the region of shared memory.
- A `tas effa` instruction must be successfully performed (the carry bit (C) returns as 1) before writing or reading the shared region.
- A `tac effa` instruction must be performed after reading or writing is complete. Since the `tas` instruction ensures the lock byte is set when it succeeds, the return status from the `tac` instruction need not be checked before continuing.

  Both of these instructions perform an `msync` instruction. They wait for all stores on the processor to reach the memory system boundary before performing the test and modify (set for `tas`, clear for `tac`).

The C3200 Series CPUs provide an explicit `msync` instruction in case it is needed for other shared memory applications. One such application is memory structures locked with communication registers, described in detail in Chapter 5.

Data cache management - C3400/C3800

The data cache size on a C3400/C3800 Series CPUs is four pages. The cache is tagged with a virtual address instead of a physical

address. To avoid an aliasing problem, an encache bit exists in the PTEs. This bit should be set when a physical page has more than one virtual mapping.

A separate set of validity bits is maintained for threaded and nonthreaded cache cells. Both sets of validity bits must be purged explicitly by software when the mapping between processes and CIRs change. All operations that require memory synchronization such as `msync`, `tas`, and `tac` will purge the nonthreaded validity set. The threaded validity set must be purged by software when the CPUs TID register is changed.

Cache management - C4600

Each C4600 CPU has a set of caches that is an integral part of the system complex. Each cache on a CPUs is tagged with enough information to ensure proper ownership of data contained within the cache.

The size, address, and tag of each C4600 cache are summarized in Table 55.

Table 55 C4600 cache summary

	PTE Cache	Data Cache	Instruction Cache
Cache Size	32K Entries	16K Bytes	256K Bytes
Cache Address			
Address	$VA_{31,26..12}$	$VA_{13..0}$	$VA_{17..0}$
CIR	Yes	No	No
TID	No	No	No
Cache Tag			
Address	$VA_{31..27}$	$VA_{31..14}$	$VA_{31..18}$
CIR	Yes	No	No
TID	Yes	No	No

PTE cache management

The C4600 PTE cache is a per CPU cache that maintains recently used virtual-to-physical address translations. Whenever a CPU makes a virtual address reference, the hardware examines the PTE cache to determine if the virtual address associated with the executing process and thread has already been translated to a physical address. If the virtual-to-physical translation does not exist in the PTE cache, the hardware traverses the page tables to determine the virtual-to-physical association. Once the virtual-to-physical association is made, the hardware places the second or thread-level PTE describing the virtual-to-physical association into the PTE cache and tags it with the CIR (and the TID if the page is thread-local)..

To insure proper virtual-to-physical translations when multiplexing different processes onto a CPU, a specific PTE cache entry can be purged using the `pate` instruction, while the entire PTE cache can be purged using the `patu` instruction.

PTEs are accelerated onto the C4600 PTE cache from memory in blocks of eight. A `pate` instruction removes all PTEs within a

block from the PTE cache. This is significantly different from the C3800, where PTEs are brought into the PTE cache one PTE at a time.

Instruction cache management

The instruction cache is a per CPU cache that contains the most recently fetched and executed instructions from memory. This cache enables quick access to recently executed instructions without having to access the main memory subsystem for the instructions. The instruction cache entries are tagged with virtual address only. This means that whenever CIR is changed, the entire instruction cache must be purged. (The assumption is that thread-local text segments do not occur.) The `pich` instruction will purge the instruction cache..

Data cache management

The data cache is a per CPU, write-through cache that contains the most recently used data from memory. This cache enables quick access to recently used data without having to access the memory subsystem for the data. The data cache entries are tagged with virtual address only. There is no attempt in hardware to maintain data cache integrity among multiple CPUs in a system complex. Data cache consistency with multiple CPUs is the responsibility of the software, and is performed in the same manner as for the C3800. The `pich` instruction will also purge the data cache..

EDTX 2-04-CV-120
51056DOC065630

Scalar loads that hit the data cache are *not* checked for read validity by the PTE cache mechanism. However, in order to maintain system security, the ring maximization check is performed on all memory accesses, including scalar loads that hit the data cache.

Cache coherency

C4600 systems maintain cache coherency by purging caches on specific events. Some of these purges are performed by the hardware (microcode), while others must be performed by the software.Table 56 identifies the events that cause purges of the C4600 caches. In this table, H indicates that the cache is purged in hardware by the indicated event, while S indicates that the cache was possibly made invalid by the event and may need to be purged by software.

Table 56 C4600 cache management

Instruction/Microcode Sequence	PTEcache	Dcache Thread	Dcache Non-Thread	Icache
tas/tac			H	
swap				
mat			H	
sndr/rcvr/getr/incr/matr			H	
pshr/popr			H	
casr			H	
msync			H	
plch		H	H	
psch			H	
pich				H
patu	H[9]	S[8]	S[8]	
pate	H-block[9]	S[8]	S[8]	
mov Sk,CIR (& accel SDRs)		S[7]	S[7]	S[7]
mov Sk,TID		H		
thread creation	[2]			[3]
join (and go idle)		H	H	
join (and continue)			H	
wfork		H	H	
idle Sk/eni_idle Sk		H	H	[3]
ldcmr[10]	S[4]	S	S	S
ldsdr[11]	H	H	H	H
System Resource Structure unlock		[5]	[5]	

[1]Where possible, on semaphoring instructions, purges will be avoided if the lock fails. For example, incr.w need not purge if the opening tas fails.

[2]Since the PTE cache is tagged with CIR and TID, it does not have to be purged when a CPU picks up a fork.

[3]Each CPU's microcode will retain the previous value of CIR. This value represents the state of the accelerated SDRs and Icache. When the microcode picks up a fork, the new CIR value is checked to determine if it is different from the previous value. If CIR is different, the SDRs are accelerated and the Icache is purged.

[4]Software purges the PTE Cache when a ldcmr is executed for a CIR that may have PTE entries encached anywhere within the system.

[5]Reads of the system resource structure (page 0 pointer and subsequent stack pointer) are done in cache bypass mode to avoid purging.

[6]The C4600 data cache size cannot be changed on ring crossing. It is assumed that any aliasing problems caused by a > 4kbyte data cache in the kernel will be solved via software.

[7]These were purged by microcode on C3200/C3400/C3800, but are left up to software on C4600 systems. This was done to allow the OS to avoid Icache/Dcache purges on a change of CIR when only Ring 0 data (that is, mapped the same in all processes) is to be touched.

[8]If the virtual-to-physical mapping is changed for a virtual page that may be accelerated into the data cache, then the data cache must be purged to avoid cache coherency problems; this is because the PTE cache is not searched on a data cache hit.

[9]Purges PTE cache or cache block immediately on all CPUs.

[10]It is assumed that no CPU is in the specified CIR when a `ldcmr` is executed.

[11]It is assumed that the process is single threaded when a `ldsdr` is executed.

Memory interleave

Memory interleave is the process of swapping address bits to determine which physical memory bank to access.

Interleave is desirable because the dynamic RAMs (DRAMs) in memory require multiple CPU clock cycles (depending on DRAM speed) to perform all read and full write operations. A full write refers to any word-aligned word or longword store. A word-aligned address is an address with the two least-significant bits equal to zero. For example, `0000 2000` or `8000 4FFC`.

Through interleaving, a different bank may be accessed on each clock cycle, allowing sequential requests to ascending banks to proceed at full speed. The C-Series memory subsystems may be interleaved in a variety of ways.

A minimum of eight independent memory banks (16 for C3800 Series CPUs) are required to make the memory system return data at the rate of one word per CPU clock cycle. The number of banks should be chosen to match the cycle time.

For example, a `read` instruction takes eight clock cycles to return data (clear a busy bank on C3800 Series CPUs). If eight successive read requests are made (at one per clock and one per bank), the first word of read data will return, beginning eight CPU clock cycles after the first request and one word more with each successive clock cycle. If as many banks as clock cycles exist, the system can sustain a data return rate of one word per clock.

If a multiprocessing C-Series CPU is making a stream of adjacent longword requests, each request goes to a different bank. This allows requests to be processed with some overlap.

Interleave - C100

A C100 Series memory subsystem supports 4-, 8-, 16-, and 32-way interleave. In order to achieve an interleave above 4, all memory access units (MAUs) must be the same size (all 16 Mbyte, all 32 Mbyte, or all 128 Mbyte).

Four-way interleave is achieved by switching between the four memory banks present on each MAU. Interleaves greater than four are achieved by switching between MAUs. In order for this interleaving scheme to work, the number of MAUs present must be a power of two. Otherwise, only 4-way memory interleave can be obtained.

If all (MAUs) are the same size, the interleave obtained is a function of the number of MAUs, as shown in Table 57.

Table 57 Memory interleave—C100 Series CPUs

Number of MAUs	Memory interleave
1	4
2	8
3	4
4	16
5	4
6	4
7	4
8	32

Interleave - C3200/C3400

The C3200/C3400 Series memory subsystem can be interleaved with more flexibility than the C100 Series. Each memory control module (MCM/MCM3) contains eight independent memory word banks with from one to four 32-bit wide rows of memory. A module or *board* with all of its banks is installed as either even or odd. There can be eight MCMs and 64 banks per complex.

There are a maximum of four memory ports on the C3400 Series complex. Normally, complexes with up to four CPUs have a port available for each CPU. Complexes with more than four CPUs share the four memory ports.

If two CPUs share a port, as is the case in a complex with more than four CPUs, and both CPUs on a port are attempting to access memory simultaneously, they must share the throughput on that port.

C3200/C3400 Series CPUs use address bits <7..3> to select memory banks for interleaving. Bits <7..6> select the memory board pair and bits <5..3> select the bank.

With a rate of one word per bank (or one longword per two banks), the C3200 Series complexes can have up to 64 word banks (or 32 longword banks), allowing up to four CPUs to receive data at the rate of one per clock. An eight-clock cycle allows eight words (=64 words per bank/8 clocks) of data, or four longwords (a longword from each memory board pair) of data to be returned per cycle from the memory.

With one word per bank, or one longword per two banks, the C3400 Series CPUs can have up to 64 word banks or 32 longword banks, allowing up to four CPUs to receive data at the rate of one per clock.

The eight memory banks on each MCM are completely independent. This independence allows the memory system to be interleaved to support a pipelined access rate of one cycle or one full write per clock cycle (not including memory refresh). Using a 40-ns clock rate, this corresponds to a memory bandwidth of 200 Mbytes per second (2 words/clock cycle) from a single pair of MCMs.

All processors (CPU or I/O) attempting to access the same pair of MCMs will be competing for the same 200 Mbytes per second of available bandwidth. The entire bandwidth is available to one processor only if no other processors make requests to a single port.

By interleaving memory at the board level as well as at the bank level, the bandwidth available from a fully configured system of four pairs of MCMs is 800 Mbytes per second. Although there are five ports (four CPU and one I/O) to access memory, only four can be in use at a time. Therefore, 800 Mbytes per second is the maximum bandwidth available. The 800 Mbytes per second access rate can be sustained provided that any accesses to the same bank of the 64 word bank memory system occur at least eight clock cycles apart.

A read cycle or a full write cycle can complete in eight clock cycles. Partial word write cycles and test-and-modify cycles take 11 clock cycles. A partial write refers to any byte or halfword store. A partial write is also any word or longword store to an address with either of the two least-significant bits *not* equal to zero. For example, a longword stored to `0000 2002`, or a word stored to `8000 EEE1`.

Although the C3200/C3400 Series memory system can consist of any number of MCM pairs up to a total of four pairs, installing less than four pairs of MCMs decreases the degree of bank-level interleaving, which causes a reduction in the available memory system bandwidth.

In addition, board-level interleaving is possible only over combinations of two or four pairs of MCMs. For example, if three pairs of MCMs are in use, the least-significant two-thirds of the address space would be 16-way interleaved over board pairs 0 and 1, and the most-significant one-third of the address space would be 8-way interleaved over board pair.

Although the memory system can consist of any number of MCM pairs up to a total of four pairs, installing less than four pairs of MCMs decreases the degree of interleaving which causes a reduction in the available memory system bandwidth. Additionally, board-level interleaving is possible only over combinations of two or four pairs of MCMs. If three pairs of MCMs are in use, the least-significant two-thirds of the address space would be 16-way interleaved over board pairs zero and one, and the most-significant one-third of the address space would be eight-way interleaved over board pair two.

Table 58 shows the available bandwidth in Mbytes per second and the interleaving possible for various combinations of MCMs for C3200/C3400 Series complexes.

Table 58 Memory subsystem bandwidth and interleaving—C3200/C3400 Series CPUs

Number of MCM pairs	Bandwidth (Mbytes per second)	Interleave factor	
		64 bit	32 bit
1	200	8	16
2	400	16	32
3	600	16 and 8	32 and 16
4	800	32	64

Interleave - C3800

The C3800 Series memory control module, referred to as the Memory Board (MB), contains 32 independent memory word banks (16 even and 16 odd) with from one to four 32-bit wide rows of memory. There can be eight NMBs and 256 word banks per complex.

C3800 Series CPUs use address bits <9..3> to select memory banks for interleaving. Bits <9..7> select the memory board pair and bits <6..3> select the bank.

With one word per bank, or one longword per two banks, the C3800 Series CPUs machines can have up to 256 word banks, or 128 longword banks, allowing up to eight CPUs to receive data at the rate of one per clock. A 12-clock cycle allows eight words or four longwords of data to be returned per cycle from the memory.

The 32 memory banks on each NMB are completely independent. This allows the memory system to be interleaved to support a pipelined access rate of one read or one full word write per clock cycle (not including memory refresh). Using a 16.67-ns clock rate, this corresponds to a memory bandwidth of 480 Mbytes per second (2 words/clock cycle) from a single NMB.

All processors (CPU or I/O) attempting to access the same NMB will be competing for the same 480 Mbytes per second of bandwidth available to a single port. The entire bandwidth is available to one processor only if no other processors make requests to that NMB.

By interleaving memory at the board level as well as at the bank level, the bandwidth available from a fully configured system of eight NMBs is the number of CPU ports times the bandwidth available to a single port, or 3840 Mbytes per second.

Although there are nine ports (typically, eight CPUs and one I/O) to access memory, only eight can be in use at a time. Therefore, 3840 Mbytes per second is the maximum bandwidth available. The 3840 Mbytes per second access rate can be sustained, provided that any accesses to the same bank of the 256 word bank memory system occur at least eight clock cycles apart.

A read cycle or a *full word write* cycle can complete in eight clock cycles. Partial write cycles and test-and-modify cycles take 20 clock cycles.

Although the C3800 Series memory system can consist of any number of NMBs (up to a total of eight), installing less than eight NMBs decreases the degree of interleaving, which causes a reduction in the available memory system bandwidth.

In addition, board-level interleaving is possible only over combinations of two, four, or eight NMBs. For example, if three NMBs are in use, the least-significant two-thirds of the address space would be 64 word-way interleaved over boards zero and one, and the most-significant one-third of the address space would be 32 word-way interleaved over board pair two.

Table 59 shows the available bandwidth in Mbytes per second and the interleaving possible for various combinations of NMBs for C3800 Series CPUs.

Table 59 Memory subsystem bandwidth and interleaving—C3800 Series CPUs

Number of NMBs	Bandwidth (Mbytes per second)	Interleave factor	
		64 bit	32 bit
1	480	16	16
2	960	32	32
3	1440	32 and16	64 and 32
4	1920	64	64 and 32
5	2400	64 and 16	128
6	2880	64 and 32	128and 32
7	3360	64, 32, and 16	128, 64, and 32
8	3840	128	256

Glossary

The terms in this glossary are defined as they are used at CONVEX. Standard acronyms are also included. Boldfaced The terms in this glossary are defined as they are used at CONVEX. Standard acronyms are also included. Boldfaced terms within a definition are found in separate entries.

A

A registers
See *address registers*.

ac power-controller
The device that regulates ac power from the cabinet circuit breaker to the computer's internal electronic and electromechanical components.

access modes
Any of the five processor access modes in which software executes. On the CONVEX system, processor access modes are (in order from most to least privileged and protected):

- Kernel (mode 0)
- Executive (mode 1)
- Supervisor (mode 2)
- Agent (mode 3)
- User (mode 4)

The operating system uses access modes to define **protection** levels for software executing in the context of a **process**.

address
A user-assigned number used by the operating system to identify a storage location.

address registers (A registers)
A set of registers intended primarily for memory address manipulation.

address space
Memory space, either physical or virtual, available to a process.

address translation faults (ATF)
Exceptions that results from a **page table entry** violation or a nonresident page.

address translation unit (ATU)
An address **cache** that accelerates the generation of **physical addresses.**

addressing modes
How the effective address of an instruction **operand** is calculated using the general registers.

agent
Processor access mode 3.

ALU
See *arithmetic logic unit.*

architecture
The physical structure of a computer's internal operations, including its registers, memory, instruction set, input/output structure, and so on.

argument pointer
An address register specifically dedicated (by convention) to point to the **subroutine** argument portion of a program. This program portion can either be in the **stack** or in part of **logical memory** pre-allocated by the **compiler.**

arithmetic logic unit (ALU)
A basic element of the **central processing unit** (CPU) where arithmetic and logical operations are performed.

array
An ordered structure of operands of the same data type. The structure of an array is defined as length, rank (or dimension), stride, and data type.

Atomic operation
An atomic operation is an indivisible operation. That is, once the operation begins, no other operation or event, such as **interrupts**, may intervene until the operation is complete.

ATF
See *address translation fault.*

ATU
See *address translation unit.*

B

b
See *byte.*

backplane
The circuitry and mechanical elements used to connect the boards of a system. Also called the motherboard.

backplane (VMEbus)
A printed circuit (PC) board with 96-pin connectors and signal paths that bus the connector pins. Some VMEbus systems have a single PC board, called the J1 backplane. It provides the signal paths needed for basic operation. Other VMEbus systems also have an optional second PC board, called a J2 backplane. It provides the additional 96-pin connectors and signal paths needed for wider data and address transfers. Still others have a single PC board that provides the signal conductors and connectors of both the J1 and J2 backplanes.

base-level interrupts
Interrupts that occurs when the kernel stack is the process stack; thus, a base-level interrupt occurs when no other interrupts are pending or currently being processed.

bit (b)
A binary digit.

bit complement
Exchanging 0s and 1s in the binary representation of a number. Also known as one's complement.

block
To stop the flow of execution. Execution cannot begin until the block no longer exists. Also called a **hazard.**

boot
The procedure (bootstrap) by which a program is initiated the first time. Typically, a boot is performed when power is first applied to the processor.

branch
A class of **instructions,** specifically relative to the program counter, used to transfer control of a program.

breakpoint
An instruction that aids in the debugging of a program. In particular, a breakpoint is a specific location in a program where one would desire to determine the various values of programmer-defined variables.

byte (b)
Eight contiguous **bits** starting on an addressable byte boundary. The smallest addressable unit in a CONVEX computer.

C

C language
The programming language of the ConvexOS operating system.

C
Address carry, PSW (C).

C shell
The standard shell provided with Berkeley standard versions of UNIX and ConvexOS.

cache memory
A small, high-speed buffer memory used in computer systems to temporarily hold a portion of the contents of the main memory that are, or believed to be, currently in use. CONVEX computers contain many separate caches, including **logical cache** (Lcache), **data cache** (Dcache), **instruction cache** (Icache), ATE cache, and PTE cache.

cache purge
The act of invalidating or removing entries in a cache memory.

central processing unit (CPU)
That portion of a computer that recognizes and executes the instruction set.

central processing unit bulkhead
A special panel on the CPU cabinet. Because the CONVEX supercomputer is electromagnetic interference (EMI) shielded, cables that connect internal components to the components or devices that are external to the CPU cabinet must pass through EMI shielded connectors mounted in a special panel called the CPU bulkhead.

chaining
The ability to overlap vector operations in the CPU. For instance, in the case of a **vector** load followed by a vector add, the add may be started as soon as the first operands are available, rather than waiting for the **load** to complete.

chassis
The physical box where the computer is housed.

compiler
A software tool used to translate the source code of a high-level language, such as C or FORTRAN, into object code (machine language), understandable to the computer.

context (processor)
The entire, current state of the machine associated with the executing process.

ConvexOS
The CONVEX version of the UNIX operating system.

CPU
See *central processing unit*.

D

d
See *double-precision*.

data types
The ways in which bits are grouped and interpreted. For processor instructions, the data types identify the size of the operand and the significance of the bits in the operand.

destination
The register or memory location that receives the result of the operation.

displacement
A derived 32-bit value used to indicate the distance in bytes between the referenced data and some base value. This base value can either be 0 or the contents of an address register. Note that 16-bit displacement values are sign extended to 32 bits.

double-precision (d)
This is a double-precision floating-point number that is stored in 64 bits. See also *single-precision*.

drawer bulkhead
The multibus drawer for the CONVEX supercomputer is electromagnetic interference (EMI) shielded. Cables that connect the internal components of the drawer to the components or devices that are external to the drawer must pass through EMI-shielded connectors mounted in a panel in the rear of the drawer. This panel is the drawer bulkhead.

E

EBUS
There are five ports on the memory system. These are referred to as ports A, B, C, D, and E. Ports A–D feed processors A–D; port E feeds the I/O system. Thus, EBUS is the bus to port E of the memory system.

electrostatic discharge (ESD)
The release of static electricity from a charged object to a grounded object.

EPROM
Erasable, programmable read-only memory.

EEPROM
Electronically erasable, programmable read-only memory.

ESD
See *electrostatic discharge*.

exceptions
Hardware-detected events that disrupts the running of a program, process, or system. See also *faults, interrupts*.

executive mode
Processor access mode **1**.

expansion cabinet
A secondary cabinet designed to house peripheral computer equipment, such as tape drives, disk drives, and controllers. See also *processor cabinet*.

F

faults
Exceptions that halt the instruction, but leave the **registers** and memory in a consistent state. The instruction can often resume its course when the cause of the fault is corrected. See also *exceptions*.

FIFO
Abbreviation for a first-in, first-out **queue**.

firmware
Software (computer programs) that reside in a physical device, such as **EEPROMs** or **ROMs**.

first-in, first-out (FIFO) queue
See *queue*.

flags
1-bit operands used to indicate the true or false results of an operation, or to enable or disable an operation.

floating point numbers
A numerical representation with a sign (positive or negative) bit, an exponent part, and a fraction part. The **fraction** is a fractional representation. The exponent is the value used to produce a power of two scale factor (or portion) that is subsequently used to multiply the fractions to produce an unsigned value. See also *fraction, guard bit*.

forced faulting mode
A mode of operation where the **CPU** diagnostics cause simulated **page faults** to occur. In forced faults mode, a bit is set in hardware so that some percentage of the time data is accessed in main memory, the entire context of the processor is saved off and then restored. This process thoroughly exercises the buses that are used to capture and restore the context of the machine as well as the entire memory system.

FORTRAN language
A high-level software language used mainly for scientific applications.

fraction
A part of a **floating point** number. The fraction is the unsigned fractional part that denotes the magnitude of the operand.

frame
See *page frame*.

`fsck` utility
A file systems check program used for maintenance and repair of data stored on disk.

function unit
A part of the CPU that performs a set of operations on quantities stored in **registers**.

G

gate arrays
Structure a used by the **ring** protection mechanism to define the entry points from a lower privileged ring to a higher privileged ring.

gather
Loading a vector register using another vector of indices instruction. See the `ldvi` instruction in the *CONVEX Assembly Language Reference Manual (C Series)*.

guard bits
A bit to the right (**least significant bit**) of a floating point fraction. The guard bit is used in intermediate calculations using floating point operands. See also *round bits*.

H

h
Abbreviation for **halfword**.

halfword (h)
Two bytes (16 bits). See also *longword; word*.

hazard
A block in the flow of execution that cannot be passed until the hazard no longer exists. Also called **block**.

Huffman's encoding
A binary encoding scheme that results in the densest packing of information.

I

Icache
See *instruction cache*.

immediates
These are literal **operands** (numbers) contained within the instruction stream.

indexing
The process of adding a **displacement** to the contents of an address register.

indirection
The process of obtaining the address of an operand by first referencing a word contained within memory.

input/output processor (VIOP)
The standard input/output device in the CONVEX supercomputer. The VIOP performs all the functions required to move data between main memory and the VMEbus subsystems, including logical-to-physical address translation and 64-bit-to-16-bit data path conversions.

instruction
Instructions are used by programmers to direct operations on the system's register set and memory.

instruction cache (Icache)
A cache that contains the most recently accessed instructions. The Icache accelerates the decoding of instructions to permit the

simultaneous decoding on one instruction with the execution of another instruction.

interrupts
Occurrences, other than **exceptions**, that change the normal flow of instruction execution. Interrupts originate from hardware, such as an I/O device. See also *maskable interrupts*.

interval timer counter (ITC)
A privileged register used to generate **interrupts** based on the passage of time.

J

jump
A departure from normal one-step incrementing of the program counter (PC). See *branch*.

K

kernel
The part of the ConvexOS operating system that resides in ring 0. The kernel typically manages process creation and deletion, scheduling, and other high-level, system-wide features.

keyswitch
A four-way electrical switch that controls the application of electricity to the central processing unit (CPU) boards.

L

l
See *longword*.

language specific information (LSI)
The area in the **stack** that is created as part of a **subroutine** call. It is language-dependent and may be **zero**.

last-in, first-out stack
See *stack*.

least significant bit (LSB)
The right-most bit in a field. The bit with the least weight in a calculation.

LIFO
Last-in, first-out stack. See also *stack*.

linker
A software tool that links separate software modules into one module.

load instruction
An instruction that moves data from memory to a register.

locality of reference
An attribute of a memory reference pattern that refers to the likelihood of an address of a memory reference being numerically close to a recent memory reference address, or the likelihood of a subsequent memory reference being identical to a previous memory reference within a given period of time.

logical cache (Lcache)
A cache that is accessed with **logical (virtual) addresses** for fast retrieval of data. It resides in the **CPU**.

longword (l)
Eight bytes (64 bits), the largest integer data type directly supported by hardware in the CONVEX computer. See also *halfword; word*.

LSB
See *least significant bit*.

LSI
See *language specific information*.

M

machine exceptions
Machine exceptions include fatal errors in the system that cannot be handled by the operating system. See also *exceptions*.

main memory
See *physical memory*.

maskable interrupts
Interrupts the operating system does not respond to at this time because they have been disabled.

Mbyte
See *megabyte*.

megabyte (Mbyte)
2^{20} (approximately one million) bytes.

memory management
The hardware and software features that control page mapping and **protection**.

microcode
A control program in firmware that resides within the CPU. Microcode provides the necessary control to map assembly language instructions onto processor hardware.

mode
See *access mode*.

mode switch
A three-way electrical switch that controls power to the system monitor board (SMB) on C100 Series models, or the system control monitor (SCM) on C200/C3200/C3400 Series models. The mode switch is located on the **ac power-controller**.

modified bits
Bits in I/O address space that record all valid write references to **page frames**. Modified bits are used by the operating system for **memory management**.

most significant bit (MSB)
The left-most bit in a field. The bit with the most weight in a calculation.

MSB
See *most significant bit*.

multiuser mode
The normal operating **mode** for ConvexOS, where the supercomputer is being run in a general timesharing environment with multiple users. See **single-user mode**.

N

negate
An **instruction** that performs a two's complement on a number.

normalization
This is the process of left-shifting a **fraction** until the leading bit is a one.

O

op code
The code or sequence of **bits** in an **instruction** that determines the operation to be performed.

operand
The code or sequence of **bits** in an **instruction** that references the register or memory location containing the data to be operated on.

optimize
Arranging instructions or data in storage so a minimum amount of machine time is spent accessing and executing those instructions or data.

orthogonal
The relationship of **instructions** and the **operands** they manipulate where a change in one property does not necessitate changes in other related properties.

P

packets
Groups of related data items, for example, groups of bytes being transmitted over a network.

page
A page is the unit of logical (virtual) memory controlled by the memory management algorithms. In the CONVEX computers, a page is a contiguous area of 4 kbytes. See **logical (virtual) memory.**

page fault
A page fault occurs when a process requests data that is not currently in main memory. The machine first saves off the state of all controllers onto a context stack in main memory. The operating system creates a free page of **physical memory** to bring the data in from the disk. The appropriate **page table entries** (PTEs) are set up so that the proper logical-to-physical translation occurs. The machine reads back from memory the state of the machine from the context stack, and restores the processor to the same state it was in when it determined that the data it needed was nonresident. The CPU then continues with normal operation of the process.

page frame
The unit of physical (main) memory in which pages are placed. Referenced and modified bits associated with each page frame aid in **memory management.**

page table entry (PTE)
A word in a page table that contains various **flags** and fields that are used in translation of logical-to-physical addresses. Address translation uses two levels of page table **indexing.**

PBUS
The primary internal interface between the I/O CCUs and other subsystem components.

physical addresses
Hardware-identified addresses in physical (main) memory consisting of the **page frame** number and the byte number within the page.

PC
See *program counter.*

physical cache
Any cache with physical addresses to access operands more quickly than in main memory.

physical memory
Main memory.

pipelining
An overlapping operating system cycle function used to increase the speed of computers by allowing multiple operations to occur concurrently.

pop
Retrieving an **operand** from a last-in first-out **stack**.

porting software
Moving software from one type of machine to another and making any required adjustments to the programs.

priority
The ordering of events. In ConvexOS the term may be applied to **protection** levels as well as to I/O interrupt levels.

privileged instructions
Instructions used by the operating system or privileged systems programs. They must execute in ring 0, or an **exception** occurs. See also *exceptions*.

process
The fundamental unit of a program managed by the job scheduler.

process exceptions
Exceptions that belong to the currently running process and may be handled with an **exception** handler in that process, in the current **ring** of execution.

processor cabinet
The cabinet designed to hold the **central processing units** (CPUs), as well as the ac and dc electrical devices, and a **system control module** (SCM).

processor status word (PSW)
A process structure that contains **flags** used to control and indicate the states of various computations and sequences within the processor.

program counter (PC)
A process structure that contains the address pointing to the next executable instruction of a process.

PROM
Programmable read-only memory.

prompts
A character or character string sent from a computer system to a terminal to indicate to the user that the system is ready to accept input. Typical CONVEX prompts are: `(fp)>`, `#`, `%`, `:`, and `(spu)>`.

protection
A mechanism provided by hardware and software that ensures that one user is protected from another user, or to ensure that a user does not perform an unsafe computation.

PSW
See *processor status word*.

PTE
See *page table entry*.

push
The act of storing an **operand** on a last-in, first-out **stack**.

Q

queue (FIFO)
A data structure in which data enters at one end and leaves out the other (first-in, first-out).

R

read
A non-destructive memory operation in which the contents of a memory location are accessed and passed to another part of the machine.

recursion
Continued repetition of the same operation or group of operations from within the operation itself.

reduced instruction set computer (RISC)
An architectural concept that applies to the definition of the **instruction** set of a processor. A RISC instruction set is an orthogonal instruction set that is easy to decode in hardware and for which a **compiler** can generate highly optimized code.

reduction
An arithmetic operation that performs a transformation on an **array** to produce a **scalar** result.

register
A hardware entity used to contain addresses, operands, or status.

reservation
The process of managing the various function units in the CPU. A reservation table is used to record the current status and availability of the function units.

reset
The process of establishing a known state in a machine register or flag.

RESET switch
A manually operated switch used to force a hardware reset on the **service processor unit** (SPU).

rings
The unit of logical memory used for **protection** purposes. See also *scan rings*.

There are five rings in CONVEX machines: four for system level usage and one for users. Each system ring (ring 0–ring 3) corresponds to one segment of logical memory (segment 0–segment 3), while the user ring (ring 4) contains four segments (segment 4–segment 7).

ring maximization
The mechanism used to enforce priority access in the logical (virtual) **address space**.

RISC
See *reduced instruction set computer*.

ROM
Read-only memory.

root directory
The base directory in ConvexOS from which all other directories stem, directly or indirectly.

round bits
One of the two **guard bits** used in the intermediate representation of a **floating point** number.

rounding
The process of transforming the intermediate representation of a **floating point** number to the memory representation. **Unbiased rounding** uses the round, guard, and **sticky bits** to determine the exact nature of this transformation. Truncation (as used in

converting floating point to fixed point integer) does not use the round, guard, or sticky bits.

runtime
A software module that is referenced as a procedure. A runtime routine represents a required function that is not directly supported by the hardware, but is required by the software.

S

scatter
Storing a **vector** register using another vector of indices. See the `stvi` instruction in the *CONVEX Assembly Language Reference Manual (C Series)*.

SCM
See *system control module*.

SDR
See *segment descriptor register*.

segment
The basic 512-Mbyte partition of the logical (virtual) memory space.

segment descriptor registers (SDR)
Each segment of **virtual memory** has a segment descriptor register (SDR) associated with it. Each SDR contains information pertinent to the access and mapping of virtual addresses.

segmented ALU
A logic design technique that permits multiple arithmetic operations of the same type to be **pipelined**.

service processor unit (SPU)
In a CONVEX CPU complex, an additional processor dedicated to monitoring the operation of the complex. Booting the complex, operator communication, and diagnostic software are three of the most common functions of the SPU.

shift instructions
A class of **instructions** used to shift the contents of a register right or left.

single-precision (s)
A single-precision **floating point** number stored in 32 bits. See also *double-precision*.

single-user mode
In ConvexOS, the mode of operation where the supercomputer is being controlled by a single system manager or operator. This

mode is used primarily for maintenance and system administrative functions. See also *multiuser mode*.

SMB
See *system monitor board*.

soft front panel
EPROM-based software that controls certain **booting**, internal testing, and communications functions in CONVEX supercomputers.

software device driver
A CONVEX-supplied or user-written program that controls the operation of attached I/O peripheral devices.

source
A **register** or memory location used as an input to a CONVEX **instruction**.

spatial reference
An attribute of a memory reference pattern that pertains to the likelihood of a subsequent memory reference address being numerically close to a previous address.

SPU tape cartridge
The magnetic tape cartridge containing the SPU programs, files, and utilities.

SPU tape drive
The tape drive installed on the service processor unit (SPU).

SPU OS
The CONVEX-developed, UNIX-based operating system software used to direct certain supervisory functions on the service processor unit (SPU) on CONVEX supercomputers.

stack
A data structure in which the last item entered is the first to be removed. Also referred to as last-in, first-out (LIFO) stack. In particular, stacks are used by the *call* and *return* instructions. See also *push* and *pop*.

sticky bit
A **bit** used in the intermediate calculation of a **floating point** operand. The sticky bit remembers whether any binary ones are shifted out during an alignment or partial product operation.

store
An **instruction** used to move the contents of **registers** to memory.

subroutine
A frequently used software module that is called from various places in a program.

superuser
The operating system term referring to the ultimate access and priority rights allowed anyone with the proper login and password, usually reserved for the **system manager**.

supervisor mode
Processor access mode 2.

system console
The CRT or printer terminal that serves as a communication device to the operating system on CONVEX supercomputers.

system control module (SCM)
An electronic safety mechanism that monitors hardware and environmental conditions on CONVEX C200 Series supercomputers. When an error condition is detected, the SCM transmits a hexadecimal status code to the system status display (SSD) on the processor cabinet front panel. See also *system status display (SSD)*.

system exceptions
Exceptions that cannot be handled by the current process, but require intervention by the **kernel** executing in ring 0. See also *exceptions*.

system manager
The person responsible for the management and operation of a CONVEX supercomputer.

system monitor board (SMB)
An electronic safety mechanism that monitors hardware and environmental conditions on CONVEX supercomputers. When an error condition is detected, one of the 16 LED indicators on the SMB is lit.

system status display (SSD)
A two-digit LED display located on the front panel of CONVEX supercomputers. It is used to display hexadecimal status codes transmitted by the **SCM**.

T

tag
A marker or label.

trace
A common debugging technique where the execution of every **instruction** of a program is tracked.

traps
Out-of-sequence branches due to the occurrence of an abnormal condition (such as the result of unexpected arithmetic results), a predetermined test condition, an **interrupt**, or an **exception**. See also *interrupts, exceptions.*

trojan horse pointers
An address that is passed from one ring to another as part of a system call. In particular, this passed pointer references the more privileged ring, as contrasted to the less privileged ring. This is unexpected and undesirable.

true zero
A **floating point** number with the sign bit with a value of zero, the exponent with a value of zero, and the **fraction** with a value of zero.

U

unbiased rounding
The process of interpreting the **round, guard,** and **sticky bits.** Unbiased rounding, as contrasted to biased rounding, rounds to even in the event that the intermediate floating point result is exactly midway between two floating point representations.

UNIX
An operating system developed by AT&T Bell Laboratories (now UNIX Systems Laboratories, Inc.). ConvexOS and SPU OS are both UNIX-based operating systems.

unsigned integer
An integer value that is always positive.

user
Processor access mode 4.

V

VIOP
See *input/output processor.*

valid bits
Bit used for the control of **caches.** The valid bit is used to determine if a cache entry contains an entry that can be used.

valid PTE reference
A reference that meets two requirements: First, the PTE must have the valid bit (bit <31>) set to 1; Second, the type of access

being made (read, write, or execute) must be allowed by the appropriate **protection** bits (bits <3..1> of the PTE).

vector
An **array** with one dimension.

virtual address
The **address space** seen by the application programmer.

virtual memory
That memory seen by the programmer. The logical (virtual) memory of a CONVEX computer is 4 Gigabytes. See also *page*.

VMEbus
A 16-/32-bit **backplane** bus.

W

word
Four bytes (32 bits), the fundamental data width of items in the CONVEX family of computers. See also *halfword*; *longword*.

working set
That portion (subset) of a user program currently in **physical memory**.

write
A destructive memory operation in which the contents of a memory location is replaced with new data.

Z

zero
In **floating point** number representations, zero is represented by the sign bit with a value of zero and the exponent with a value of zero. See also *true zero*.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

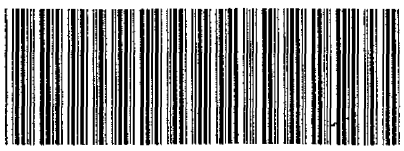
W

X

Z

ORDER NUMBER
DHW-300

DOCUMENT NUMBER
081-011830-001



CONVEX PRESS