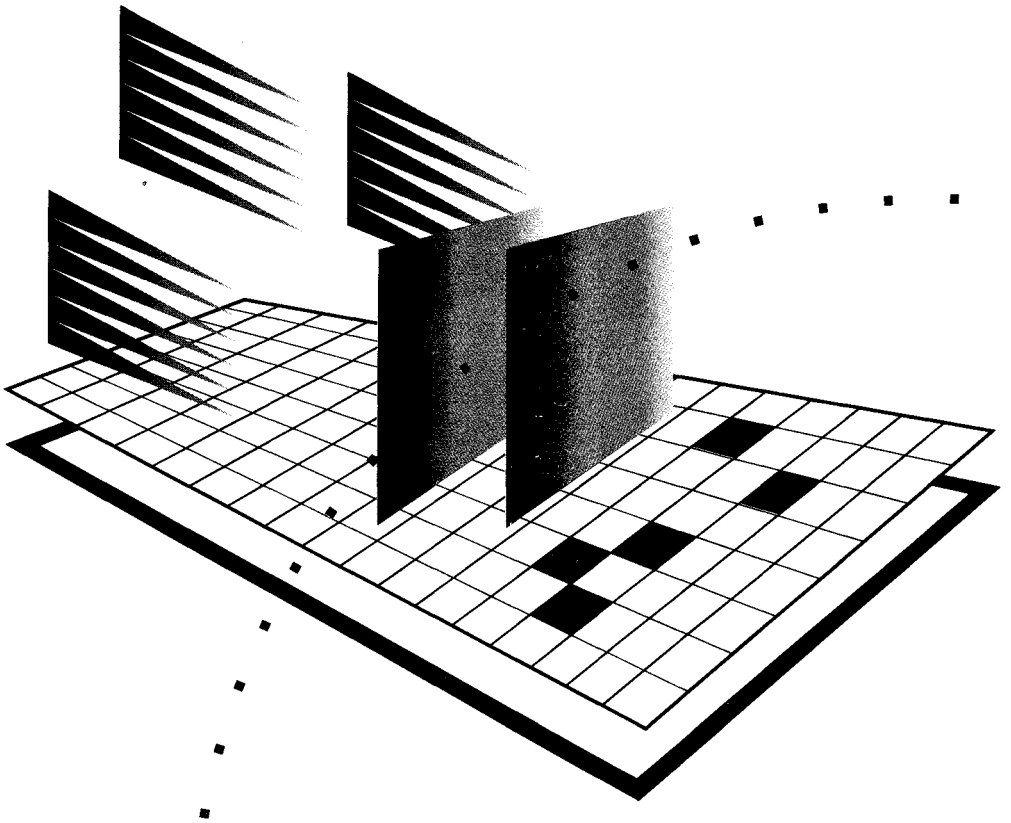

CONVEX

Exemplar SPP1000-Series Architecture

Fourth Edition

HEWLETT® PACKARD

Hewlett-Packard Company
Convex Technology Center
3000 Waterview Parkway
P.O. Box 833851
Richardson, TX 75083-3851
United States of America

Exemplar SPP1000-Series Architecture

Order No. DHW-014

Fourth Edition

May 1996

Hewlett-Packard Company
Convex Technology Center
Richardson, Texas
United States of America

Exemplar SPP1000-Series Architecture

Order No. DHW-014

Notice

This entire book is recyclable.

Printed in the United States of America

Revision Information for

Exemplar SPP1000-Series Architecture

Edition	Document No.	Description
Fourth	081-023430-003	Revised to include SPP1600 and its four-state coherency mechanism and improved I/O and performance monitors. Changed title from *Exemplar SPP1000/1200 Architecture* to *Exemplar SPP1000-Series Architecture.* Released May, 1996.
Third	081-023430-002	Revised to include SPP1200. Changed title from *Exemplar Architecture* to *Exemplar SPP1000/1200 Architecture.* Released May, 1995.
Second	081-023430-001	Revised nomenclature. Included description of CTI. Added a chapter describing globally shared memory features of the architecture. Released November, 1994.
First	081-023430-000	Initial release November, 1993.

Contents

Figures

Tables

Preface

The *Exemplar SPP1000-Series Architecture* reference manual documents the Convex implementation of massive parallel processing in the Exemplar product line.

Purpose and audience

The *Exemplar SPP1000-Series Architecture* reference manual provides technical information for the evaluation of the Exemplar product line and the optimization of applications that may be developed to run on the Exemplar product.

Using this guide

The *Exemplar SPP1000-Series Architecture* reference manual has eleven chapters. The chapters include a conceptual overview and specific information about the Convex implementation of a concept.

- Chapter 1, "Introduction," provides an introduction to the concepts used in the Exemplar product.
- Chapter 2, "Globally shared memory," describes the concept of globally shared memory.
- Chapter 3, "Physical memory," discusses the design of Exemplar memory systems and the specific implementation used in Convex Exemplar products.
- Chapter 4, "Virtual memory," discusses the virtual memory system for the Convex Exemplar series, including address translation, memory protection, and allocation of the virtual address space.
- Chapter 5, "Messaging," defines the messaging mechanisms provided by the Convex Exemplar series, including support for block data movement, the low-level programming interface which accesses those mechanisms, and the underlying hardware that supports those mechanisms within the Exemplar implementation.

- Chapter 6, "Synchronization," defines the synchronization mechanisms provided by the Convex Exemplar series, the low- level programming interface that accesses those mechanisms, and the underlying hardware that supports the mechanisms within the Exemplar implementation.
- Chapter 7, "Exceptions and interrupts," defines the exception and interrupt mechanisms provided by the Convex Exemplar series.
- Chapter 8, "I/O subsystem," defines the I/O subsystem provided by the Convex Exemplar series, the low-level programming interface that accesses the I/O subsystem, and the underlying hardware that provides the I/O system functionality within the Exemplar implementation.
- Chapter 9, "Performance monitors," defines performance monitoring mechanisms provided by the Convex Exemplar series, the low-level programming interface that accesses and implements those mechanisms, and the underlying hardware which supports them within the Exemplar implementation.
- Chapter 10, "Service processor," defines the service processor functions provided by the Convex Exemplar series, the low-level programming interfaces that access those functions, and the underlying hardware that supports those functions within the Exemplar implementation.
- Chapter 11, "Initialization and booting," defines the procedures and mechanisms provided by the Convex Exemplar series to support general system initialization and OS booting.
- The glossary contains brief definitions for terms used in this document.

Notational conventions

This section discusses notational conventions used in this book.

`Bold monospace`

In command examples, text shown in **`bold monospace`** identifies user input that must be typed exactly as shown.

`Monospace`

In paragraph text, `monospace` identifies command names, system calls, and data structures and types.

In command examples, `monospace` identifies command output, including error messages.

In command syntax diagrams, text shown in `monospace` must be typed exactly as shown.

Italic

In paragraph text, *italic* identifies new and important terms and titles of documents.

In command syntax diagrams, *italic* identifies variables that must be supplied by the user.

{ }

In command syntax diagrams, text surrounded by curly brackets indicate a choice. The choices available are shown inside the curly brackets and separated by the pipe (|) sign.

The following command example indicates that you can enter either `a` or `b`:

```
command {a | b}
```

[ ]

In command syntax diagrams, square brackets indicate optional data.

The following command example indicates that definition of the variable *output_file* is optional:

`command` *input_file* [*output_file*]

...

In command syntax, horizontal ellipsis shows repetition of the preceding item(s).

The following command example indicates you can optionally specify more than one *input_file* on the command line.

`command` *input_file* [*input_file ...*]

Associated documents

Using a Convex Exemplar system may require information not specific to the tasks described in this document.

- The *Guide to Exemplar Documentation*, order number DSW-851, provides a list and description of all Convex documentation including all applicable Exemplar documents.
- The *Exemplar Programming Guide*, order number DSW-067, provides a description of efficient programming methods in Convex C and Fortran on Exemplar.
- The *CXpa Reference*, order number DSW-605, enables you to profile optimized FORTRAN, C, and C++ programs.
- The *CXdb Reference*, Volume I and II, order number DSW-472, is a comprehensive reference for CXdb, a symbolic debugger for C, C++, and FORTRAN code
- The *CXdb Quick Reference*, order number DSW-474, lists all CXdb compiler options, commands and options, command line editing key sequences, command window key sequences, and performance report abbreviations.
- The *Exemplar Open Boot Quick Reference*, order number DSW-854, summarizes the commands associated with the Exemplar Open Boot firmware.

Specific architecture features of the Convex Exemplar systems are designed in accordance with IEEE standards. These standards include:

- *IEEE Std 1596-1992 Scalable Coherent Interface Specification.* This IEEE standard defines the protocol and address space definition used for the Exemplar internode mesh interconnect.
- *IEEE Std 1212-1992 Control and Status Register Architecture.* This IEEE standard defines the functionality and address space for control and status register (CSR) access across a range of current and future IEEE standard interconnects. The Exemplar CSR definition for CSRs accessible from the internode interconnect is a subset of the IEEE CSR definition.
- *IEEE Draft Std P1275/D3 Standard for Boot Firmware,* (Draft D3, dated March 1992). This is the IEEE standard version of the Open-Boot architecture.

Additional Hewlett-Packard documentation includes:

- *HP-UX Application Binary Interface for PA-RISC Systems.* This document contains information about the memory layout required to support ABI compliant PA-RISC applications.
- *The PA-RISC 1.1 Architecture and Instruction Set Reference Manual*; Third Edition. This manual contains generic

information about the PA-RISC architecture used as a basis for the Convex Exemplar.

The material presented here is intended to be self-sufficient, but the reader should refer to the above documents if more information is required.

Ordering documents

To order the current edition of these or any other Convex documents, send requests to:

Hewlett-Packard Company
Convex Technology Center
Customer Service
P.O. Box 833851
Richardson TX 75083-3851 USA

Please include the order number (DSW or DHW number) or the exact title of the document.

Technical assistance

If you have questions that are not answered in this book, contact the Hewlett-Packard Convex Technical Assistance Center (TAC) at the following locations:

Within the continental U.S., call 1 (800) 952-0379.

From Canada, call 1 (800) 345-2384.

All other locations, contact the local Convex Technology Center office.

You can also use the `contact` utility, if you would like to report any problems you may have with ConvexOS or its associated documentation. For more information refer to the contact(1) man page in *ConvexOS Man Pages for Users*, or the appendix "Reporting problems" in the *ConvexOS Primer* or *Managing ConvexOS: Operations Guide*.

Introduction 1

Overview

This chapter provides information about each aspect of the scalable parallel processing (SPP) architecture. It also discusses some of the objectives for the high-level design.

Most chapters in this manual present data that is *architectural* and some data that is Exemplar-specific. Exemplar-specific data is grouped into a separate section that is identified as Exemplar-specific.

Exemplar and the PA-RISC architecture

The Convex Exemplar product line uses SPP technology. SPP is an implementation of massive parallel processing (MPP) technology that is expandable as customer needs increase.

The Exemplar uses PA-RISC processors developed by Hewlett-Packard. Exemplar configurations of 4 to 128 PA-RISC processors are included.

The document describes three Exemplar products: The SPP1000, the SPP1200, and the SPP1600. The SPP1000 uses the HP PA7100 CPU and the other two use the HP PA7200 CPU. Most information in this book is common to all. Information specific to each product is contained in separate sections or is clearly noted in the text.

The PA7200 CPU used in the SPP1200 and SPP1600 possesses several enhancements over its predecessor:

- Faster clock rate—The PA7200 runs at 120MHz.
- The Runway bus—an innovative split transaction bus capable of 768 MBytes/sec bandwidth.
- On-chip performance monitor.
- Fully-associative on-chip data cache—hides memory latency and minimizes cache miss rates.

A detailed description of the PA-RISC architecture is presented in the Hewlett-Packard *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*. The *Exemplar Architecture* does *not* attempt to

duplicate information in that reference manual. Instead, it presents detail of the Exemplar product line and the the integration of PA-RISC processors.

The primary difference between SPP1200 and SPP1600 platforms is that the SPP1600 implements a four-state cache mechanism. The SPP1200 (as well as the SPP1000) uses a three-state cache.

The SPP1600 also has an enhanced I/O subsystem and improved performance monitor capabilities.

SPP features

Several fundamental objectives define the design of the Convex SPP architecture. These include:

- The machine is easy to use. A significant problem with SPP systems is the difficulty of programming and administering, resulting in a lack of available applications. This requirement dictated decisions to provide a scalable shared memory architecture and to concentrate on the Fortran, C, and C++ programming environments.
- The machine exhibits an exceptional price-to-performance ratio. This is achieved by employing a RISC CPU technology developed by Hewlett-Packard. The HP PA-RISC architecture has consistently demonstrated exceptionally high scalar and floating-point performance and has proven to be scalable across a broad range of systems.
- The machine is scalable, from small configurations (4 CPUs) to large configurations and exhibits excellent performance throughout the range.
- The machine emphasizes time-to-solution as well as throughput. By emphasizing time-to-solution, it can effectively employ its resources to solve quickly a large problem. In general, as the problem size increases, it becomes increasingly effective to employ parallel processing to speed the solution of a single problem.
- The machine is highly reliable.

The hypernode

The fundamental building block in the Convex SPP architecture is the hypernode. A hypernode is a symmetric multiprocessor (SMP). An SPP is a group of hypernodes sharing a low-latency interconnect.

An SMP architecture effectively exploits fine-grain parallelism. Many programs are optimized for SMP machines, so it is important that the Convex SPP in small configurations (one hypernode) run these programs cost-efficiently.

A conceptual hypernode block diagram is shown in Figure 1.

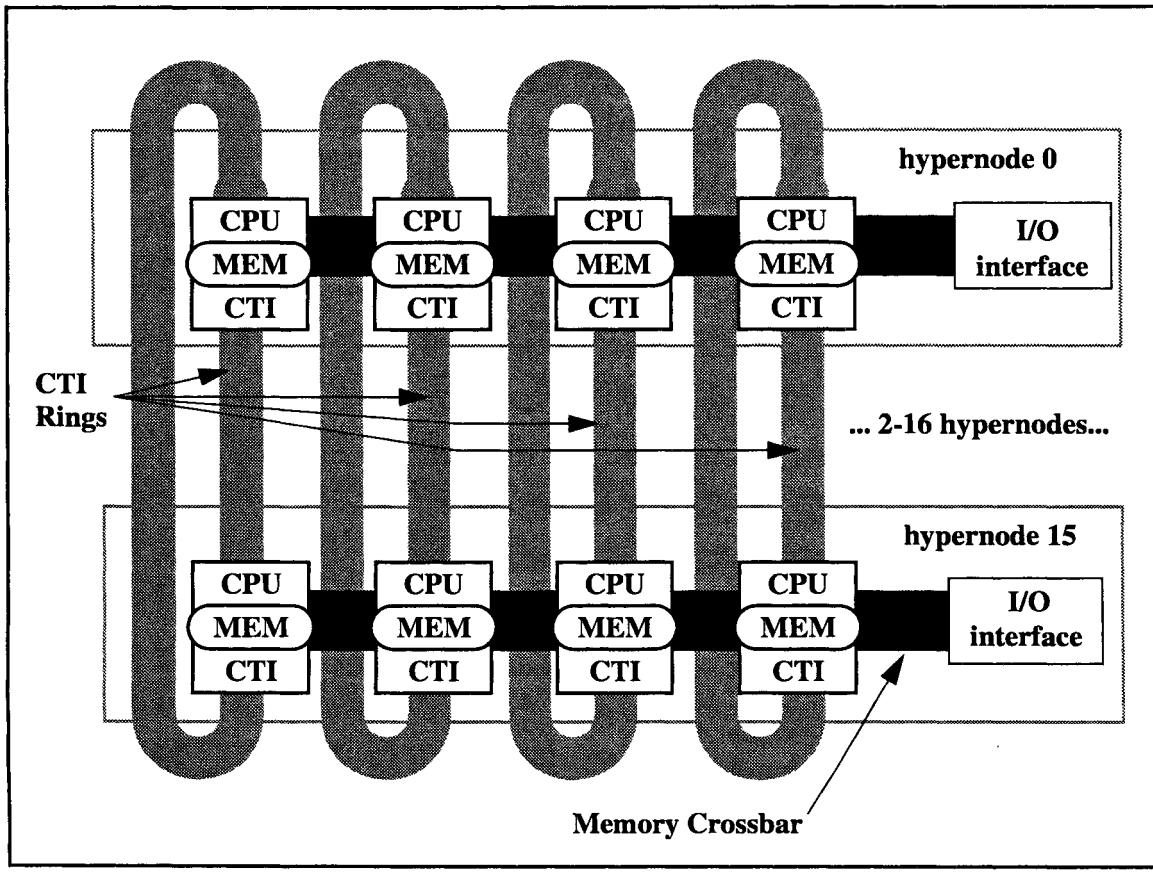

Figure 1 Conceptual hypernode block diagram

Each hypernode contains four CPU blocks. The CPU block contains the following (described in this chapter):

- Two PA-RISC processors
- An associated data and instruction cache(s)
- A CPU agent
- A CPU-private memory
- Convex Toroidal Interface (CTI)

Each hypernode also contains one or more hypernode-private memories that can be accessed only by any CPU within the hypernode (hypernode-private memory is not accessible from other hypernodes). Multiple hypernode-private memories operate independently and may be hardware-interleaved to provide greater bandwidth.

Each hypernode contains one or more global memory blocks. Global memory blocks provide global memory accessible to all hypernodes in the system, including the one containing it. A CTI interface used to connect to other hypernodes, and a CTI cache. The CTI cache encaches all global memory data imported by this CTI interface from other hypernodes on the CTI.

A hypernode contains an I/O adapter, which connects the hypernode interconnect to I/O devices.

The number of CPUs provided on a node is a balance between parallel speed-up and memory latency/interconnect scalability. It is easiest to achieve low-memory latency in a single CPU system, because there is a simple, direct path with few components between the CPU and memory system. When two or more processors are supported, an interconnection scheme is required so that any processor can access the shared memory.

The hypernode interconnect connects each of the components of the hypernode CPU blocks, hypernode-private memory, global memory blocks, and the I/O adapter. Interconnection schemes face complex trade-offs between latency, bandwidth, scalability, and cost. To make an effective SMP hypernode, low latency to memory is required, which ultimately limits the number of supported CPUs. The hypernode interconnect is typically implemented as a bus or crossbar connection.

Memory hierarchy

Exemplar uses a hierarchical memory architecture. Several types of memory are available with different sharing and latency characteristics. The following types of memory are provided, listed in order of increasing memory latency:

1. *CPU-private* memory is provided for data accessed only by a single CPU. Because this memory is implemented on the same board with the CPU, it has the lowest latency.
2. *Hypernode-private* memory is provided for data shared only by CPUs within a single hypernode. The hypernode-private memory of one hypernode may not be accessed from a different hypernode. Hypernode-private memory may be implemented as an interleaving of all the CPU-private memories on the hypernode.
3. *Near-shared* memory is globally accessible from all hypernodes, but has affinity for its home hypernode. Accessing near-shared memory from any hypernode other than the home hypernode suffers a latency penalty. Near-shared memory is allocated from the globally shared memory of its home hypernode.
4. *Far-shared* memory is globally accessible from all hypernodes and is accessed with equal latency from any hypernode participating in the application. Far-shared memory may be implemented as an interleaving of all near-shared memories of the hypernodes participating in the computation. Far-shared memory is allocated from the globally shared memories of several hypernodes.

The term hypernodes above refers to all hypernodes participating in a single computation. The operating system may dynamically configure the hypernodes of an SPP together to form a subcomplex, in which all the hypernodes are working on the same problem. In this case, far-shared memory is interleaved among all the hypernodes in a subcomplex.

Exemplar hypernode architecture

The Exemplar implementation uses the following architectural features:

- The hypernode interconnect is implemented as a five-port crossbar for maximum bandwidth and minimal CPU-to-memory latency.
- The CPU blocks, global memory blocks, and hypernode-private memories are combined to form a single functional block that provides the functionality of the separate conceptual blocks. One memory unit in the block holds hypernode-private memory data, global memory data, and CTI cache data. Four functional blocks comprise a hypernode.
- CPU-private memory is not physically implemented; the operating system partitions hypernode-private memory used as CPU-private memory for each of the CPUs. Implementation of a physical CPU-private memory would not result in substantially lower CPU-to-memory latency, and the latency-to-hypernode-private memory would be increased.
- Two CPUs share a single CPU agent. There are eight CPUs per hypernode.
- The four memories in a hypernode functional block are interleaved, providing higher bandwidth and less contention than accessing a single memory. Interleaving is a scheme where sequential memory references (by linearly ascending physical addresses) search the four memories on a round-robin basis. Each memory returns 64 bytes for sequential reads.
- The CTI interface is an extension of a scalable coherent interface (SCI) that is IEEE P1212-compliant. Each hypernode has four CTI rings to the other hypernodes, one in each functional block.

The block diagram of a Exemplar hypernode is shown in Figure 2.

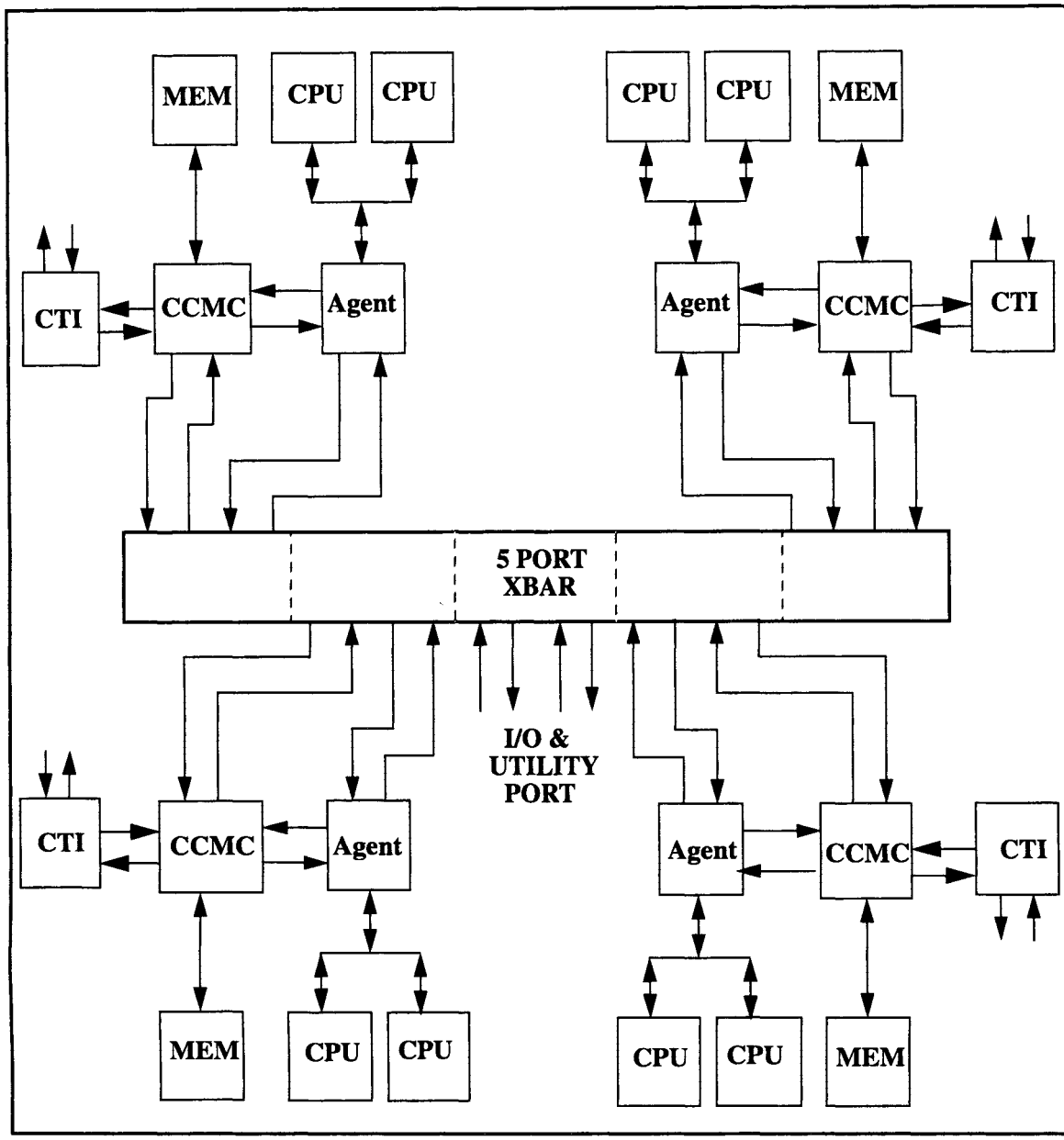

Figure 2 Exemplar hypernode block diagram

Convex Toroidal Interconnect (CTI)

Multiple hypernodes are interconnected with one or more low-latency interconnections. The CTI refers to a collection of rings used by hypernodes to access near-shared and far-shared memories.

It supports access to global memory on a cache line basis. A cache line is the size of memory moved over the CTI by hardware in response to load, store, or flush operations. Typically, the cache line size is smaller than a page size. The CTI cache line size may or may not be the same as the CPU cache line size. In the Exemplar products, the CTI cache line size is 64 bytes, while the CPU uses a cache line size of 32 bytes.

The CTI also supports messaging protocols. Messages are used by the operating system and distribute memory applications for interhypernode communication. Message data movement incurs somewhat less overhead than cache line movement due to the necessity of maintaining cache coherence for the cache lines.

Convex Exemplar provides hardware support for globally shared memory access, whereas a true distributed memory machine can only emulate shared memory by moving pages from hypernode to hypernode under software control. Cache lines and pages migrate between hypernodes in the Exemplar without software intervention, resulting in lower overhead to programs using shared memory.

The Exemplar system uses four CTI rings attached to each hypernode as the CTI between hypernodes. Four rings provide higher interconnection bandwidth, lower interhypernode latency, and redundancy in case of ring failure.

Sequential memory references to global memory (by linearly-ascending physical address) are interleaved across the four rings. This is accomplished using the ring in the same functional unit as the target memory, because the memories are interleaved on a 64-byte basis. The four CTI rings are interleaved on this basis as well; 64 bytes is the CTI cache line size. This ring interleaving tends to balance the traffic across all four rings. (global memory references from a CPU to the global memory on the same hypernode do not use the hypernode CTI and are not encached in the CTI cache.)

The following block diagram shows how multiple hypernodes are connected in Exemplar:

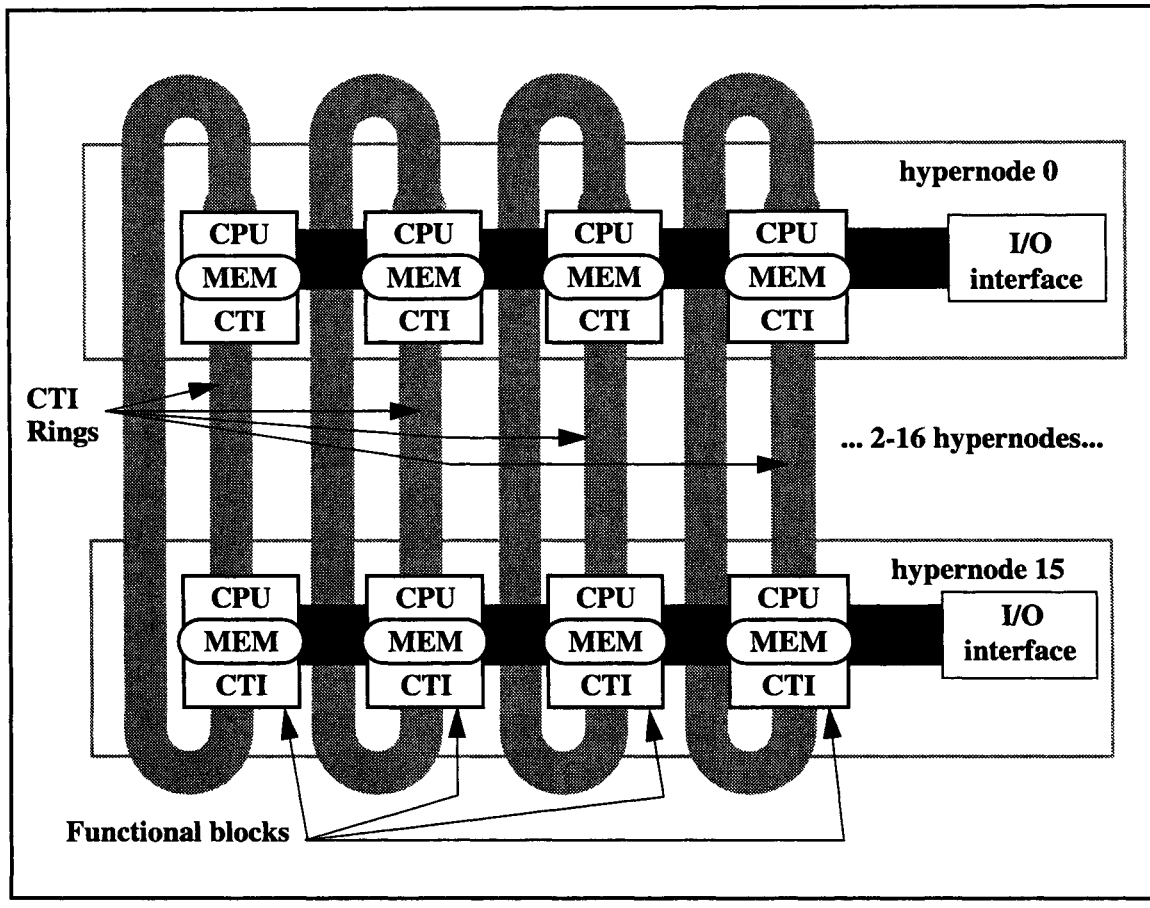

Figure 3 Exemplar CTI ring interconnect

CTI description

Convex CTI is an extension of the scalable coherent interface (SCI) defined in IEEE standard 1596-1992. The SCI standard defines an interface that provides a memory-address based, cache-coherent communication mechanism for massively parallel systems with large numbers of processors. Convex chose performance enhancements over compatibility where necessary to obtain performance objectives required to meet system specifications.

The CTI is physically defined as a pair of 18-bit, differential ECL, unidirectional links clocked at 150 MHz. Each link provides 16 bits of data along with a clock and a "flag" used as a framing delimiter for a total of 18 signals. Data is sampled on both the rising and falling edges of the clock for an equivalent 300 MHz data rate for a two-byte wide path, or a peak raw rate of 600 Mbytes/second.

The SCI specification does not define a specific interconnect, only the interface to an interconnect. The point-to-point nature of the interface and the communication protocols suggest a simple, unidirectional ring as the base interconnect.

All CTI communication is in packets containing a 16-bit destination hypernode identifier, a 16-bit source hypernode identifier, and a 48-bit address within the destination hypernode. The base protocol supports forward progress, delivery, fairness, and basic error detection and recovery.

The CTI coherency protocol, overlaid on the base protocol, provides a scalable linked list form of coherency directory. The base CTI cache line size is fixed at 64 bytes; the base protocol is write-back and invalidate. The CTI cache mechanism supports simultaneous linked-list modification by all the processors in a sharing list for maximum concurrency. There are no locks and no resource choke points in the protocol, allowing it to scale in a linear manner to large numbers of processors.

Availability features of the hypernode CTI

If a CTI ring fails, Exemplar is capable of degraded operation with one to three active CTI rings. Disabling a CTI ring also disables all CPUs and memories connected by the ring. For example, disabling one ring makes two CPUs and one quarter of the memory on each hypernode unavailable.

It is possible to deconfigure a failing hypernode from all rings. In this case the system operates with the remaining hypernodes. Memory and I/O devices connected to a nonconfigured hypernode are unavailable.

Reconfiguration of CTI rings or hypernodes requires a restart of the operating system.

Memory and cache coherence

The Convex Exemplar product line provides shared memory within a hypernode (hypernode-private memory) and globally shared memory across the entire array of Exemplar hypernodes. Shared memory makes the system easy to program and supports a wide variety of programming models.

To speed CPU references for memory, copies of the memory data are encached in each CPU instruction or data caches and in the CTI cache, if the copy is from the global memory of another hypernode. The hardware copies data between caches and memory in cache-line-sized parcels (64 bytes on Exemplar). The copies are coherent. If any of the CPUs reference any given memory lines, they "see" up-to-date copies. To maintain updated copies, the hardware follows these rules:

- Any number of read encachements of a cache line may be made at a single time. The cache line may be read-shared in multiple caches.
- To write data (store) into a cache line, the cache line must be owned exclusively. This implies that any other copies must be deleted.
- Modified cache lines must be written back to memory before being deleted, or the contents can be forwarded directly to the new owner.

The architecture does not specify the type of implementation required for cache coherence. In the Exemplar, cache coherence protocols are different within a hypernode than across the CTI interconnect.

Cache coherence can result in additional memory latency, because memory control logic may have to force write-backs of modified data before allowing a cache line to be copied into a CPU or CTI cache. Providing cache coherence on hardware, however, also provides benefits:

- It avoids the requirement for the programmer to explicitly flush the cache.
- It avoids unnecessary synchronizations.

Refer to the code in Figure 4. This pseudo-code does a simple matrix multiply. The algorithm calls *spawn* to fork *nCPUs* number of threads. Each thread receives a unique thread identifier, *itid*. Each iteration of the outer *do j...* loop is only executed by one CPU as determined by the following *if* statement. The CPUs are allocated in cyclical fashion to the loop iterations.

The code on the left works correctly for a cache-coherent machine. No synchronization is required because each outer loop iteration

computes disjointed data (no loop dependencies), and only one CPU executes the body of each iteration.

```
global c(idim, idim),
a(idim,idim)
global b(idim,idim), nCPUs
private i, j, k, itid

call spawn(nCPUs)

do j=1,idim
   if (jmod(j,nCPU).eq. itid)then
      do i=1,idim
         c(i,j) = 0.0
         do k=1,idim


            c(i,j) = c(i,j) +
               a(i,k) * b(k,j)
         enddo
      enddo
   endif
enddo

call join
```

```
global c(idim, idim),
a(idim,idim)
global b(idim,idim), nCPUs
private i, j, k, itid

call spawn(nCPUs)

do j=1,idim
   if (jmod(j,nCPU).eq. itid)
then
      do i=1,idim
         c(i,j) = 0.0
         do k=1,idim
            call flush (a(i,k))
            callflush(b(k,j))
            c(i,j) = c(i,j) +
               a(i,k) * b(k,j)
            enddo
         call flush(c(i,j))
      enddo
   endif
enddo

call join
```

Figure 4 Avoiding explicit flushes with hardware cache coherence

The code on the right is required for a machine that is not cache-coherent. The calls to *flush* on the *a* and *b* arrays are required to ensure that the executing CPU gets an up-to-date copy of the input arrays, because they may have been modified on a different CPU after being encached on the current CPU. The call to *flush* for the *c* array is required to write the output array back to memory, so that subsequent operations will see the result of the matrix multiply.

The code on the right may not work, however, depending on the relation of *idim* to the cache line size and the alignment of the *c*-array. The algorithm as written guarantees only that no two

CPUs will update the same *c (i,j)* value. This is all that is required from a logical point of view, but if multiple *c (i,j)* array elements are put into a cache line and two processors update different values in the same cache line at the same instant (no synchronization), the resulting data will be incorrect.

This problem can be alleviated by either of the methods shown in Figure 5. The left algorithm has a single binary semaphore mechanism. Lock and unlock calls around the update to *c (i,j)* and subsequent flush avoid two simultaneous updates to the same line. To reduce the size of the critical region, the update of *c (i,j)* was moved out of the inner loop. The left algorithm is still unacceptable from a performance perspective, however, since it creates a single critical region around the update.

```
global c(idim,
idim),a(idim,idim)
global b(idim,idim), nCPUs
private i, j, k, itid, tmp
semaphore isem

call spawn(nCPUs)

do j=1,idim
   if (jmod(j,nCPU).eq. itid)
then
      do i=1,idim
         tmp = 0.0
         do k=1,idim
            call flush (a(i,k)
            call flush(b(k,j))
            tmp = tmp +
               a(i,k) * b(k,j)
         enddo
         call lock(isem)
         c(i,j) = tmp
         call flush(c(i,j))
         call unlock(isem)
      enddo
   endif
enddo

call join
```

```
global c(idim, idim),
a(idim,idim)
global b(idim,idim), nCPUs
private i, j, k, itid, tmp
semaphore is(idim,idim)

call spawn(nCPUs)

do j=1,idim
   if (jmod(j,nCPU).eq. itid)
then
      do i=1,idim
         tmp = 0.0
         do k=1,idim
            call flush (a(i,k)
            call flush(b(k,j))
            tmp = tmp +
               a(i,k) * b(k,j)
         enddo
         call lock(c(i,j),
is(i,j))
         c(i,j) = tmp
         call flush(c(i,j))
         call
unlock(c(i,j),is(i,j))
      enddo
   endif
enddo

call join
```

Figure 5 Avoiding unnecessary synchronization with hardware, cache coherence

The code on the right exhibits better parallel performance by reducing contention over the critical section. Here a semaphore array is created, with an element in the semaphore array for each cache line. The lock and unlock functions are defined to compute the corresponding cache line address for their first argument, and to lock or unlock the semaphore in their second argument corresponding to that cache line. Thus lock and unlock ensure exclusive access on a cache line basis to the array being modified.

By including the flush in this critical section, software has correct cache coherence.

The algorithm on the right is what the cache coherence hardware implements: exclusive access for memory updates on a cache line granularity. The algorithm must be run to guarantee that the data is correct. Implementation of the algorithm in hardware reduces software and compiler complexity, as well as providing lower overhead than explicit software address computations and access to a parallel array of cache line locks.

Globally shared memory 2

Overview

Globally shared memory (GSM) provides the benefits of a scalable parallel processing system that maintains a familiar programming model. Convex uses a combination of hardware and software to provide a scalable distributed memory system that maintains the program model of a conventional shared memory system. This program model looks like a tightly-coupled shared memory machine.

This programming model allows the developer, compilers, and applications to view the system as a number of processors sharing a large physical memory and a number of high bandwidth I/O ports. Logically, the system appears as a multiple-processor (4 to 128), multiple-instruction, multiple-data (MIMD), shared memory system, as shown in Figure 6.

Convex compilers use GSM to provide automatic, efficient parallelization while viewing memory as a single contiguous virtual address space.

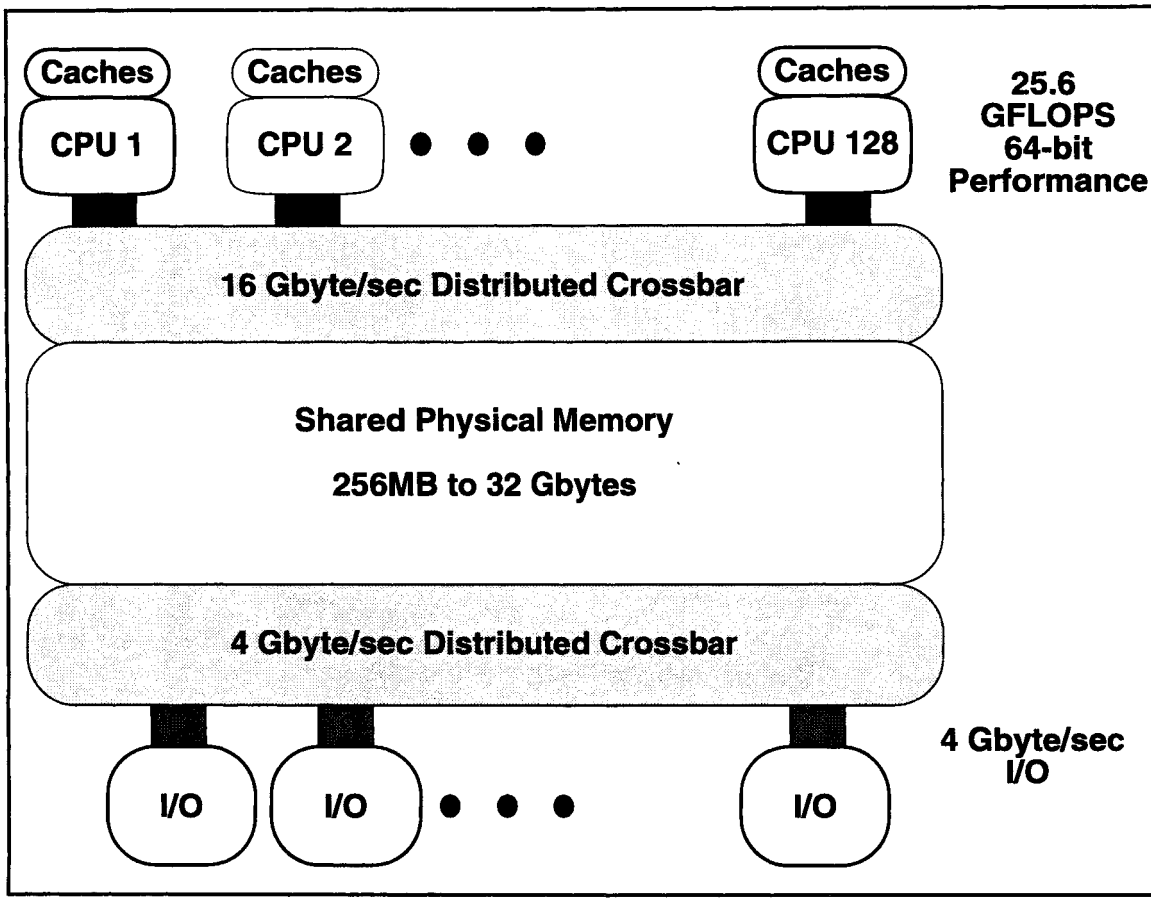

Figure 6 GSM from a user perspective

GSM hardware

The GSM hardware includes distributed physical memory, the memory crossbars, and Convex Toroidal Interconnect (CTI). These components provide access from any CPU to global memory.

Figure 7 shows the relationship among the hardware components that support GSM.

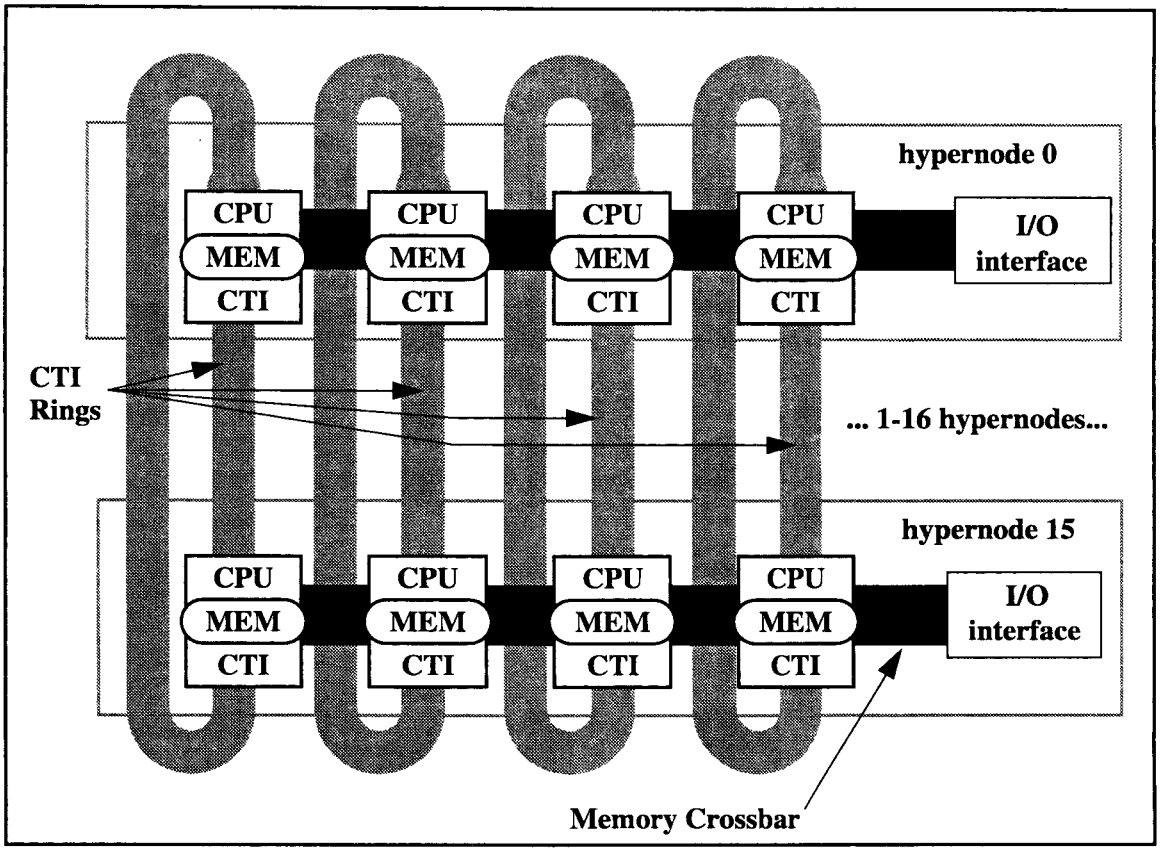

Figure 7 GSM hardware overview

GSM is scalable. As the complex size increases from 1 to 16 hypernodes, physical memory is expanded across the complex. Each hypernode includes I/O, memory, and CPUs as shown in Figure 8.

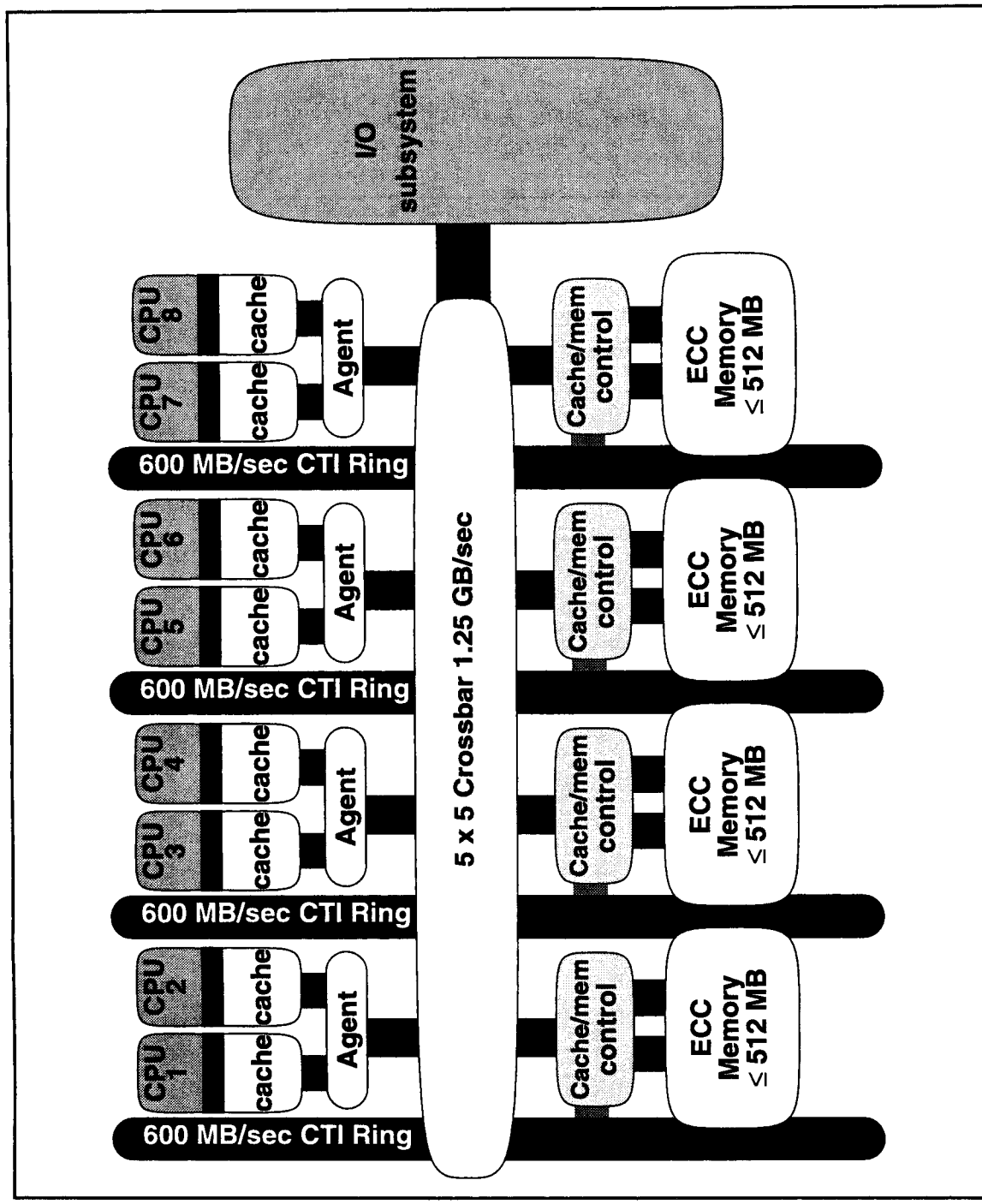

Figure 8 Hypernode hardware details

Memory subsystem

The Exemplar memory system is a nonuniform memory access (NUMA) architecture containing four types of allocatable memory. Specific latency for each type of memory is discussed in the Chapter 3.

GSM supports a two-level hierarchial memory subsystem where each level is optimized for a particular class of data sharing. The first level consists of 5x5 crossbar connecting memory, CPUs, and I/O in a single hypernode. The second level consists of a single dimensional interconnect (CTI) between hypernodes.

An Exemplar system uses a 5x5 crossbar in each hypernode to provide high bandwidth, low latency nonblocking access from CPUs and I/O channels to the hypernode local memory. The crossbar implementation prevents the performance drop-off associated with systems that employ a system-wide bus for CPU and I/O memory traffic.

The CTI is composed of four unidirectional CTI rings attached to each hypernode as shown in Figure 8. Four CTI rings are used to provide higher interconnection bandwidth, lower hypernode latency, and redundancy in case of individual CTI ring failures.

Sequential memory references to GSM (by linearly ascending physical address) are interleaved across the four rings. Since memory is interleaved on a 64-byte basis, the CTI interconnects are interleaved on a 64-byte basis as well. Using interleaving tends to balance traffic across the four CTI rings.

The CTI communicates in packets. Packets contain a 16-bit destination hypernode identifier, a 16-bit source hypernode identifier, and a 48-bit address.

References from a CPU to GSM on the same hypernode use the 5x5 crossbar, and references from a CPU to memory in another hypernode may use the CTI or a combination of CTI and crossbar.

Memory latency

Exemplar systems use hardware to determine access methods for each memory reference. Specific latency will vary depending on the proximity of data that is referenced. Figure 9 illustrates the various stages that may be required for memory reference instructions.

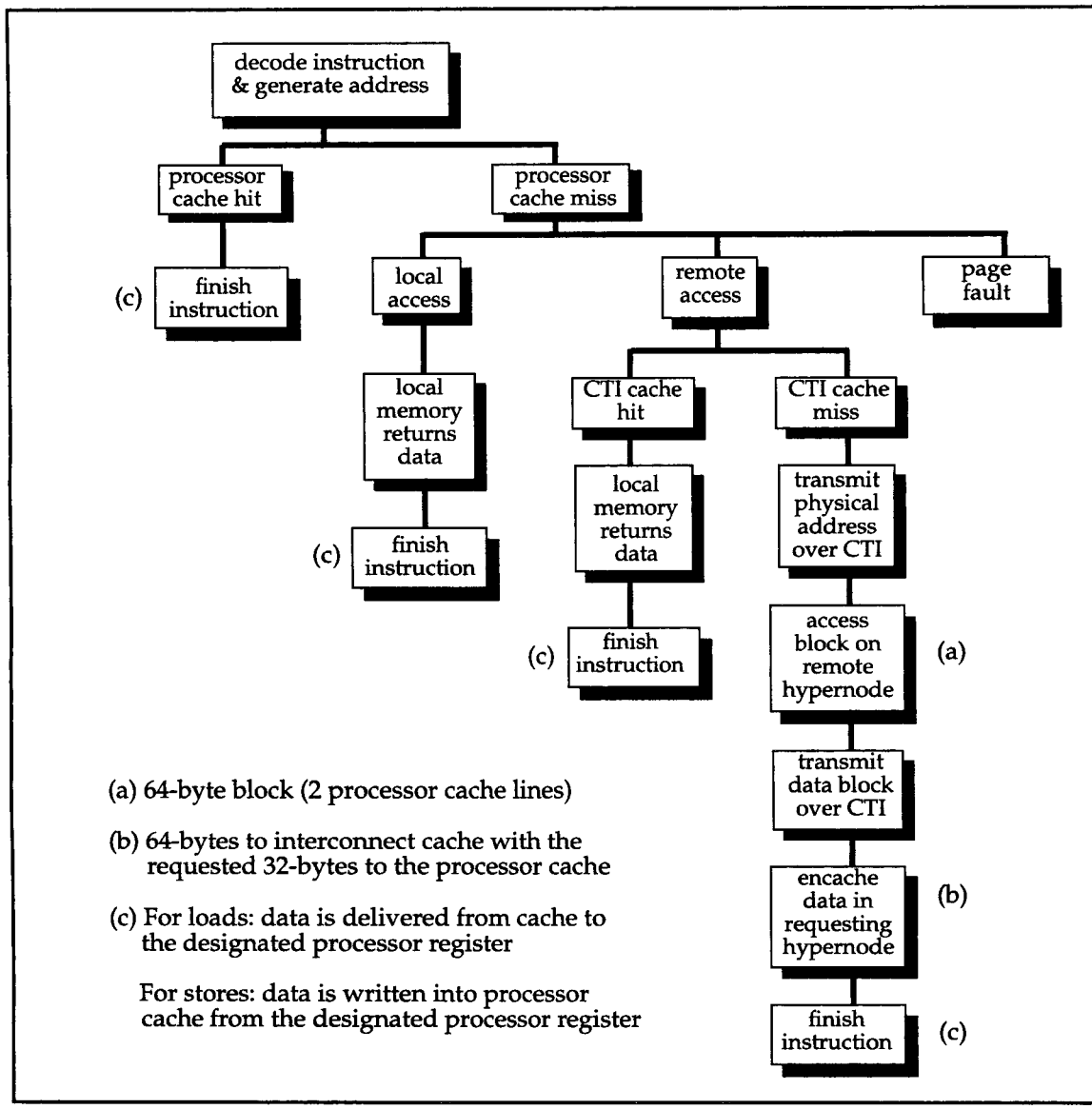

Figure 9 Hardware processing of a load/store

Each hypernode supports 128 (Exemplar/CD only), 256, 512, 1024, or 2048 Mbytes of 60ns DRAM, physically distributed on four memory boards. Each memory board has two memory banks that can transfer a 64-byte cache line (eight longwords) in eight clock cycles. GSM allows cache lines to be automatically copied and encached between hypernodes without software intervention. This translates to lower overhead for programs using GSM.

These memory types differentiate how they allocate and share data. GSM allows for the existence of different memory allocation, sharing, and latency without requiring different physical memory in a hypernode.

Table 1 shows the four different types of Exemplar memory.

Table 1 Exemplar memory types

Memory type	Physical layout	Shared by
CPU-private memory	Interleaved within hypernode by 64-byte cache line	One CPU
Hypernode-private memory	Interleaved within hypernode by 64-byte cache line	CPUs within hypernode
Near-shared memory	Interleaved within hypernode by 64-byte cache line	CPUs across hypernodes
Far-shared memory	Interleaved within hypernode by 64-byte cache line and across hypernodes by 4-Kbyte page	CPUs across hypernodes

GSM physical address space

The hypernode physical memory is divided into five separate regions to perform coherent memory access, noncoherent access of processor dependent code (PDC = boot and diagnostic code), and either I/O Channel or control and status register (CSR) access.

Addresses 0 through 0xEFFFFFFF access main memory using a protocol that ensures cache coherency. This address range is logically divided into CTI cache, GSM, and local memory partitions that can be adjusted to optimize performance.

Addresses 0xF0000000 through 0xF0FFFFFF access PDC and other diagnostic-related hardware. Addresses 0xF1000000 through 0xFFDFFFFF are available for kernel assignment to I/O channels, I/O configuration maps, and other large, noncoherent structures related to system I/O. Addresses 0xFFE00000 through 0xFFFFFFFF access control and status registers (CSRs). CSR accesses are direct. That is they are a read or write to a CSR within the hypernode or external to it.

Figure 10 shows the 5 regions of physical memory address space.

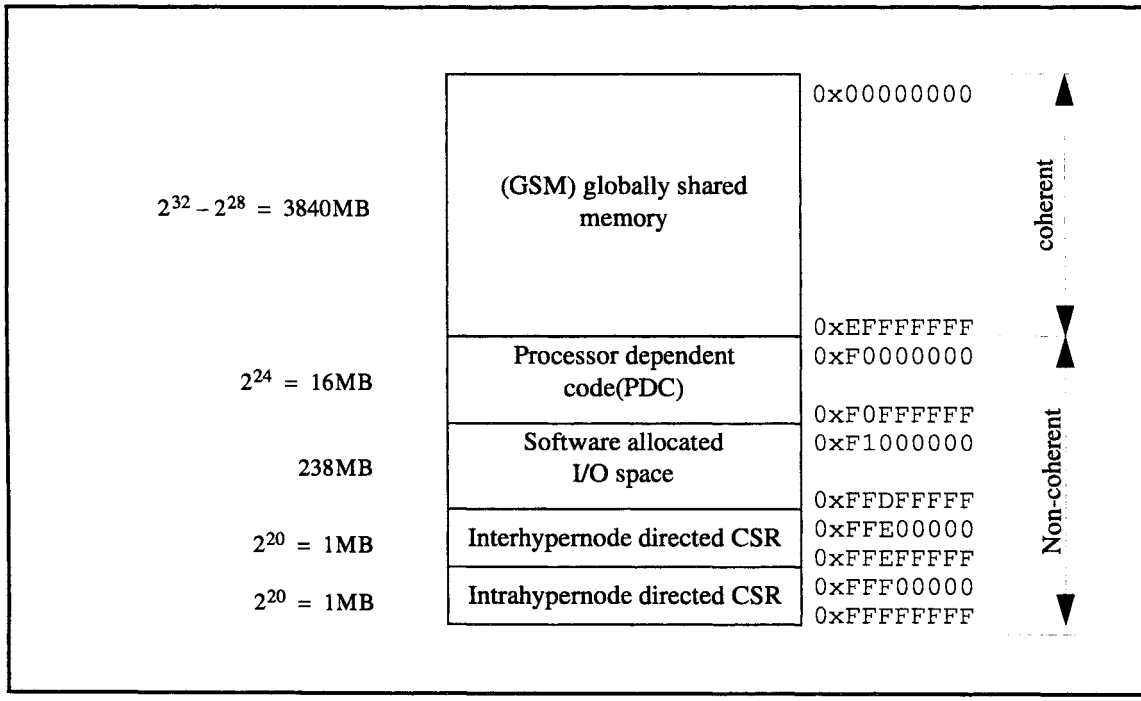

Figure 10 GSM physical address space

GSM programming model

The Exemplar programming model provides two perspectives from which a programmer can write code. The two perspectives are globally shared memory and message passing.

In the GSM model, compilers handle optimizations, and, if requested, parallelization. Data distribution is handled by hardware using the CTI cache.

Numerous compiler directives are available to further increase optimization. Since the shared memory approach is used on most non-MPP machines, Exemplar support of GSM allows programs from such machines to be easily ported to the Exemplar architecture. It is also easy to use the familiar shared memory paradigm for programs written from scratch for Exemplar, since it automates many of the tasks for which the message passing paradigm required explicit coding.

Refer to the *Exemplar Programming Guide* for examples of programming techniques that take advantage of GSM features.

Memory classes

In the GSM programming model, the Exemplar architecture provides two sets of memory classes.

Physical memory, described in the hardware section of this chapter, is associated with each hypernode functional block and expanded as the system complex size is increased. Physical memory is further divided into:

- **Hypernode-local memory**—is local to a hypernode and cannot be accessed by other hypernodes. This is where application and SPP-UX executables, as well as user process data that has been explicitly declared to be private, resides.
- **Subcomplex globally shared memory**—is accessible by all CPUs in a given subcomplex. This memory can be allocated by the system administrator during machine configuration.
- **CTI cache**—is used to store copies of global data fetched from other hypernodes. CTI cache memory can be configured by the system administrator.

Virtual memory is divided into five classes. The compilers will choose default classes for programs that provide good performance; programmers can also manually assign data to memory classes to improve data distribution and further increase performance. Manually assigning data to memory classes may require additional optimizations, such as loop parallelization, to be handled manually as well.

The virtual memory classes and their physical mapping are:

- **Thread-private**—This memory is private to each thread of a process. A thread-private data object has a unique virtual address for each thread within its hypernode. These address map to unique physical addresses in hypernode-local physical memory on each hypernode. Threads access the physical copies of thread-private data residing on their own hypernode when they access thread-private virtual addresses.
- **Node-private**—This memory is private to the threads on a given hypernode. A node-private data object has a unique virtual address by which all threads on all hypernodes access it. This address maps to one physical address per hypernode; when a thread accesses the data, it receives the value contained in the physical memory of its own hypernode.
- **Near-shared**—Data objects of the near-shared class have a single virtual address by which they can be accessed from any hypernode in the subcomplex. This address maps to a single physical address on a particular hypernode.
- **Far-shared**—Data objects of the far-shared class have a single virtual address by which they can be accessed from any hypernode in the subcomplex. Physically, far-shared data is distributed by pages in round-robin fashion to all the hypernodes in the subcomplex, so that the virtual address maps to a single physical address located randomly on one of the hypernodes.
- **Block-shared**—Data objects of the block-shared class have a single virtual address by which they can be accessed from any hypernode in the subcomplex. Physically, block-shared data is distributed in blocks equally among the hypernodes, one block per hypernode. Block-shared memory must be dynamically allocated; the programmer can then easily ensure that threads on a hypernode make the most of their accesses to the block residing on their hypernode.

For additional information on the use of memory classes in Exemplar refer to *Convex Exemplar Programming Guide.*

Private vs. globally shared memory

Private and shared data are differentiated by their accessibility and by the physical memory classes in which they are stored.

Both node-private and thread-private data are stored in hypernode-local memory and are therefore inaccessible to any hypernode other than the one on which they reside. Latency is identical for private data items that must be fetched from main memory. Near-shared, far-shared, and block-shared data are stored in subcomplex-global physical memory and therefore are accessible from any hypernode in the complex on which the process is running.

Main memory latency can vary for the GSM memory classes depending on whether the data is resident on the requesting hypernode.

Flexibility of GSM

SPP-UX and Convex compilers allow programmers to take advantage of GSM with program optimizations and resource management. GSM can be modified primarily through:

- Adding physical memory as the complex is expanded.The size of physical memory can be increased as the number of hypernodes increases.
- Adjusting the physical memory partitions to optimize performance for applications run on the system. The size of the CTI cache partition, the globally shared memory partition, and the local memory partition can be modified at boot time.
- Using Convex compilers that support memory class assignments to assign memory classes to data items.

In general, programmers will benefit from a large GSM configuration when they are using large data sets accessed randomly by different hypernodes. This occurs with programs that are highly parallelized and unsuited for domain decomposition.

If programs and associated data structures can be decomposed into specific domains, a larger near-shared configuration could provide better performance.

Figure 11 shows the memory partitions within a hypernode that can be configured to optimize application performance characteristics.

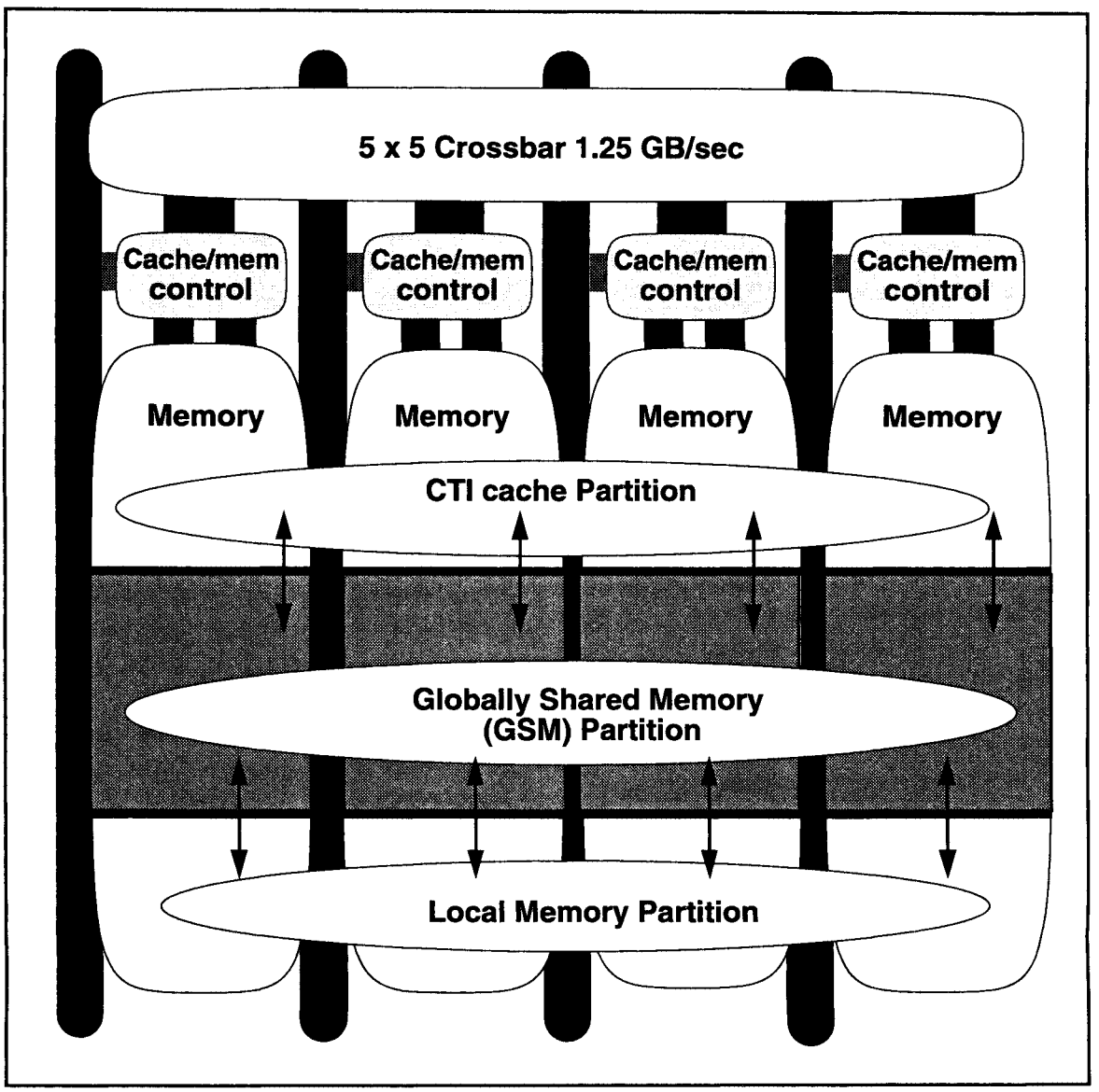

Figure 11 Adjustable hypernode memory partitions

A single-node Exemplar uses the crossbar to act as any other shared memory system. In a multinode Exemplar, the crossbar and CTI work together to provide the user with the same view of shared memory.

Physical memory 3

Overview

This chapter defines the physical memory system for the Exemplar product line, including the memory hierarchy, allocation of physical address space, memory interleaving, and ring mapping mechanism. This chapter also includes detailed definitions of all architectural aspects of the I/O address space, the utility and diagnostic address space, and the control and status register (CSR) address space.

Exemplar physical memory architecture

Exemplar architecture is based on commodity RISC processors, which derive the majority of their required data bandwidth from local instruction and data caches. These caches are small with respect to the size of an SPP application, so it is important to minimize the cache fault refill time to maintain high performance. In a small multiprocessor system (4-8 processors), all processors can share a common, high-bandwidth interconnect to reach memory within a reasonable latency. This interconnect can also provide cache-coherent access to shared data in a simple, cost effective way.

MPP systems with hundreds or thousands of processors must rely on some form of hierarchial memory structure to balance the simultaneous requirements of low latency and high bandwidth. These systems provide massive bandwidth by distributing the memory structure and providing simultaneous access to each segment of each memory element in the hierarchy. Latency is minimized through a hierarchy of caches that move the data closer to the processor transparently, and can dynamically copy the required data to memory elements closer to a particular processor.

Snooping coherency techniques do not scale in latency or cost as the number of processors increases. Likewise, directory-based approaches tend to create contention within the memory hierarchy and do not scale easily when many processors attempt to share a single cache line.

SPP memory hierarchy

Exemplar physical memory system addresses the problems discussed above with a unique hierarchy of memory and caches. Each of these is optimized for a particular class of data sharing. The Exemplar architecture is a two-level hierarchy. The first level consists of a multidimensional torus of identical processor/memory hypernodes (as described in Chapter 1). Convex expands this concept to two levels by making each hypernode a fully functional symmetric multiprocessor. Convex also expands the normal concept of a toroidal interconnect by splitting each link of the torus into four or more interleaved links for additional bandwidth (descibed in Chapter 1).

Each level is a separate cache-coherency domain. The intrahypernode domain implements coherency within hypernode processors and their memories, that is, within a single symmetric multiprocessor. The interhypernode domain implements coherency among the hypernodes in a system. The two domains are bridged by the CTI cache that tracks coherency in both domains. The interconnect and coherency mechanisms used for each domain may differ (between Exemplar architectures) to optimize communication separately at each level. The implementation of the CTI cache will ensure, however, that these differences are transparent to all application software.

This system organization is equivalent to a set of fully self-contained multiprocessor workstations interconnected by a high-performance, cache-coherent CTI. Processors within a hypernode are tightly coupled to support fine-grained parallelism. Hypernodes use coarse-grained parallelism with communication via shared memory and messages.

The Exemplar memory system is a Non-Uniform Memory Access (NUMA) architecture containing four types of allocatable memory, differentiated by the way data is allocated and shared. The existence of four different memories in allocation, sharing, or latency, does not imply that there must be four distinct physical memories. All four memories, as well as the CTI cache, may be implemented by the same physical memory on each hypernode.

Table 2 shows the four different types of memory associated with Exemplar.

Table 2 Exemplar memory types

Memory type	Physical layout	Shared by
CPU-private memory	Interleaved within hypernode by 64-byte cache line	One CPU
Hypernode-private memory	Interleaved within hypernode by 64-byte cache line	CPUs within hypernode
Near-shared memory	Interleaved within hypernode by 64-byte cache line	CPUs across hypernodes
Far-shared memory	Interleaved within hypernode by 64-byte cache line and across hypernodes by 4-Kbyte page	CPUs across hypernodes

There are up to eight tiers of latency in the NUMA SPP. Latency differences are related to communication latency (*within* hypernode versus *across* hypernodes) as well as coherency overhead due to data sharing. Table 3 lists each element in the memory hierarchy, sorted from least to greatest latency, and identifies the corresponding coherency domain. The interconnect mechanisms and the absolute value of each latency, however, may vary with each physical implementation. For a given implementation, the absolute value of two or more tiers of latency may be identical or very similar. For instance, an implementation (Exemplar) based on a one-dimensional mesh (a single set of interleaved rings) would not differentiate between latency tiers six, seven, and eight.

Table 3 Exemplar memory hierarchy

Element in hierarchy	Coherency domain	Latency domain	Approx. SPP1000 latency	Approx. SPP1200/1600 latency
Processor instruction cache	Intrahypernode	First tier	10 ns	8.33 ns
Processor data cache	Intrahypernode	First tier	10 ns	8.33 ns
CPU private memory	Intrahypernode	Second tier	500 ns	500 ns
hypernode private memory	Intrahypernode	Third tier	500 ns	500 ns

Table 3 Exemplar memory hierarchy —**(continued)**

Element in hierarchy	Coherency domain	Latency domain	Approx. SPP1000 latency	Approx. SPP1200/1600 latency
CTI instruction/data cache	Domain bridge	Fourth tier	500 ns	500 ns
Near shared memory	Interhypernode	Fifth tier	500 ns	500 ns
Far shared memory CTI level 1 CTI level 2 CTI level 3	Interhypernode	 Sixth tier Seventh tier Eighth tier	2.0 usec	2.0 usec

Note

Convex application compilers and programmers must be familiar with the latency tiers to optimize data distribution for performance.

Exemplar memory address space

The Exemplar physical address space is not uniform and contiguous. A system level (interhypernode) address implemented by the interhypernode interconnect references a hypernode (16 most significant bits) and an offset within that hypernode (48 least significant bits) as required by IEEE Std. #1596-1992 (SCI) and IEEE Std. 1212-1992 (CSR). This partitions the system address space into 64-Kbyte spaces of 256 Tbytes each, corresponding to 64-Kbyte hypernodes. See Figure 12.

SPP has a 64-bit physical address space. The Exemplar products (SPP1000, SPP1200, and SPP1600), however, provide only 32 bits of physical address.

Once a reference has entered the hypernode, the 48-bit hypernode offset is interpreted as a reference to a specific unit within the hypernode and an offset within that unit. A unit may be one of the following:

- Processor (interrupts)
- Memory
- Hypernode-level control and status register (CSR) space
- Any other entity Convex chooses to include in SPP systems

IEEE Std.1596-1992 (SCI) defines the least significant portion of the 48-bit offset (0x000000000000 - 0xEFFFFFFFFFFF) as a reference to the coherent memory of the hypernode and reserves the most significant 256 Mbytes of each hypernodes 48-bit offset (0xF00000000000 - 0xFFFFFFFFFFFF) for CSR access.

The PA-RISC architecture defines the least significant 15/16 of implemented physical address space in each hypernode to reference coherent memory space. The most significant 1/16 of implemented physical address space in each hypernode is reserved for noncoherent I/O and CSR access.

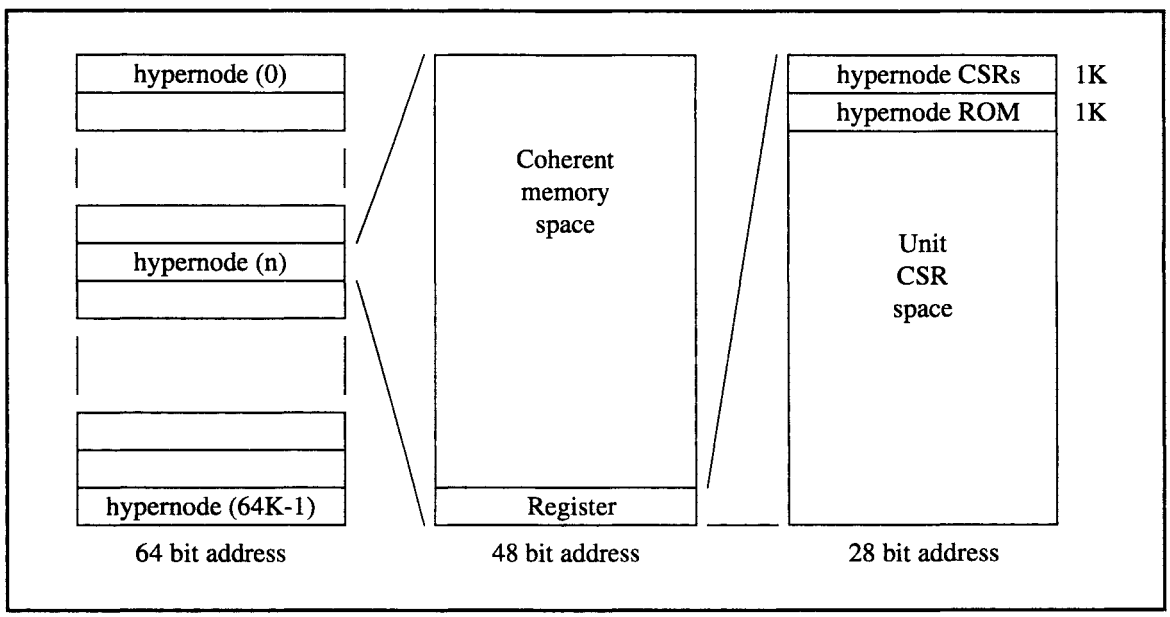

Figure 12 IEEE Std. 1596-1992 physical address space

The Exemplar products map the CTI coherent memory address space for each hypernode to the PA7100 coherent memory address space. At a minimum, it maps a portion of the PA7100 CSR address space to the first 4096 bytes of system-level CSR space for every hypernode in the system.

SPP1200 and SPP1600 address space

In addition to the information given in the preceding section, this section describes the address space unique to the SPP1200 and SPP1600.

The Runway bus

The PA7200 CPU connects directly to an enhanced system bus developed by HP and called the Runway. The Runway allows multiprocessor systems to interface to memory and I/O without additional processor interface components.

SPP1200 and SPP1600 addressing

The PA7200 CPU generates a 32-bit address internally and extends it to a 40-bit address when it issues a transfer onto the Runway bus. The SPP1200 and SPP1600 hardware, however, uses only the lower 32 bits of the Runway bus. Table 4 shows how the 32-bit address is extended to 40 bits.

Table 4 32-bit to 40-bit extension

32-bit address	40-bit Runway address	Hardware extension
0x00000000 - 0xEFFFFFFF	0x0000000000 - 0x00EFFFFFFF	0x00
0xF0000000 - 0xF0FFFFFF	0xF0F0000000 - 0xF0F0FFFFFF	0xF0
0xF1000000 - 0xFFFFFFFF	0xFFF1000000 - 0xFFFFFFFFFF	0xFF

Exemplar coherent physical address interleave

Exemplar systems interleave accesses to coherent memory in two ways. Within a hypernode, the coherent memory is divided into *n* equal sections, where *n* is any integer supporting modularity and fault tolerance. In practice, *n*-max will normally be a small binary power, such as 4 or 8. Intranode accesses are interleaved over the n-sections at a granularity of 64 bytes by address decode hardware. These memory sections are referred to by *ring number* because the Exemplar architecture associates an interhypernode interconnect *ring* with each memory section; see Chapter 1.

Coherent memory that is designated as far-shared (shared by and distributed across multiple hypernodes), is interleaved across the hypernodes on a per-page basis by operating system software allocation of the table entries.

Exemplar physical address space

The Exemplar CPUs are augmented by a mapping structure managed by the operating system (called the block descriptor table) to provide a 36-bit address consisting of a 4-bit hypernode identifier with a 32-bit intrahypernode offset. This address will be appropriately padded to create a 64-bit address for interhypernode accesses.

The remainder of this chapter will refer to three related address names:

- **HP physical address**—This is the 32-bit physical address produced by the PA7100 and the PA7200 CPUs. It refers to locations within a hypernode only. This address is sometimes referred to as simply the physical address.
- **Hypernode physical address**—This is the 36-bit physical address produced by attaching hypernode information to the HP physical address. This address can refer to any location within the system and is sometimes referred to as the CxBar address.
- **Interhypernode physical address**—This is the 64-bit address required for interhypernode access across the CTI ring. This address is sometimes referred to as the CTIRing address.

Intrahypernode addressing

The Exemplar architecture structures intrahypernode physical memory as linear, byte-addressable storage accessed by 32-bit absolute pointers. An absolute pointer is a 32-bit, unsigned integer whose value is the address of the most significant byte of the operand it designates.

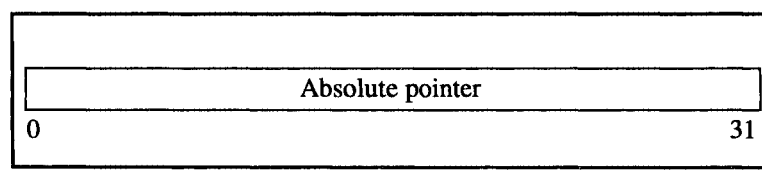

Figure 13 PA RISC absolute pointer

Memory is always referenced with byte granular addresses. Addressable units are bytes, halfwords (2 bytes), words (4 bytes), and double-words (8 bytes). Bytes in memory and bits within larger units are always numbered from 0, starting with the most

significant byte or bit, respectively. Bit or byte 0 is always the left-most, as illustrated in Figure 14.

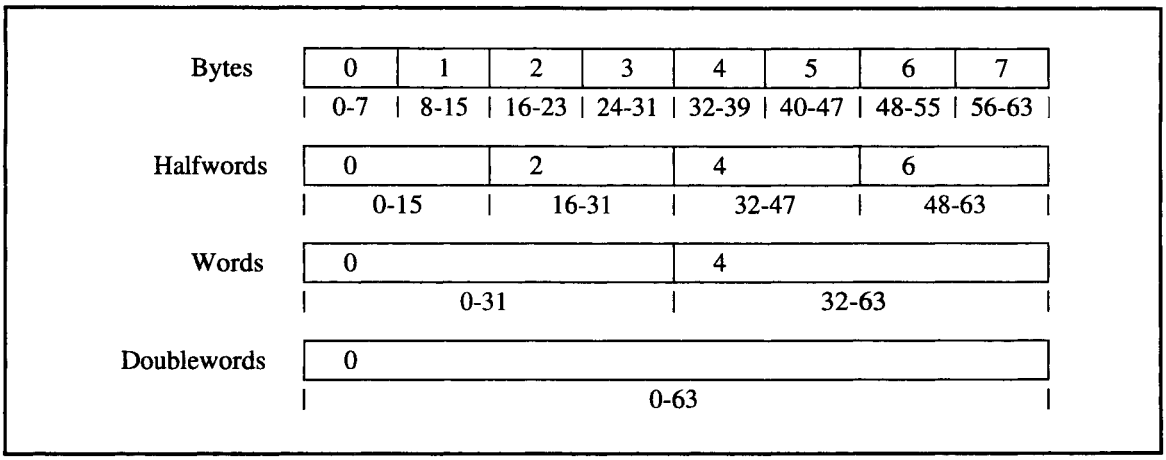

Figure 14 HP physical memory addressing and storage units

All addressable units must be stored on their naturally aligned boundaries. A byte may appear at any address, halfwords at any even address, words at any address that is a multiple of four, and double-words at any address that is a multiple of eight. If an unaligned virtual address is used to access memory, a trap occurs.

Exemplar allows intrahypernode CSR space to be referenced using bytes, halfwords or words with a strong preference for word-level access. Interhypernode CSR accesses are word-aligned.

Physical address space partitioning

The hypernode physical memory is divided into five separate regions (as illustrated in Figure 15) to perform coherent memory access, noncoherent access of processor dependent code (PDC = boot and diagnostic code) and I/O Channels, or control and status register (CSR) access. Address 0 through 0xEFFFFFFF access main memory using a protocol that ensures cache coherency. Addresses 0xF0000000 through 0xF0FFFFFF access PDC and other diagnostic-related hardware. Addresses 0xF1000000 through 0xFFDFFFFF are available for kernel assignment to I/O channels, I/O configuration maps, and other large noncoherent structures related to system I/O. Addresses 0xFFE00000 through 0xFFFFFFFF access control and status registers (CSRs). CSR accesses are *directed* (read/write of a single CSR either within a hypernode or within the "hypernode level" CSRs of another hypernode).

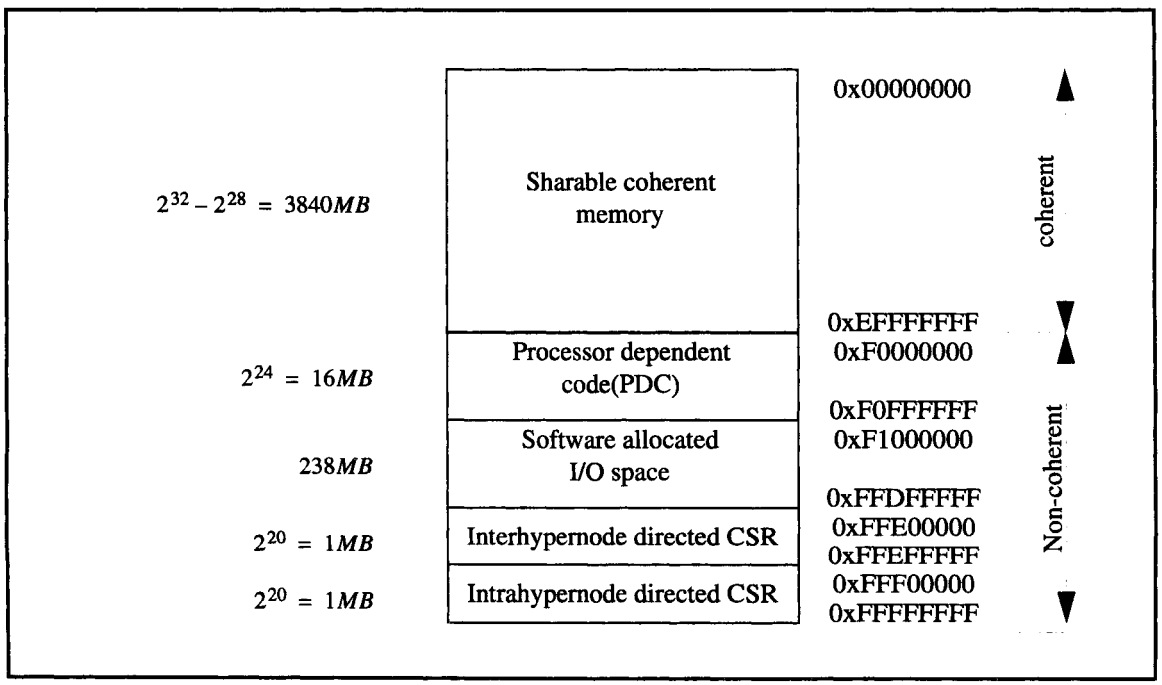

Figure 15 Hypernode physical address space partitioning

Coherent memory layout

Memory is physically located on memory blocks, each under the control of a coherent memory controller (CCMC). Coherent memory is allocated within a hypernode by 32-byte processor cache line pairs (and 64 bits of tag information) interleaved across the memory blocks. The pairs are further interleaved across two independently accessible memory banks in each block. Figure 16 represents a typical memory layout of four memory blocks per hypernode.

Two processor cache lines combine to make a single 64-byte CTI cache line. Processor transactions with memory occur on a processor cache line basis, while CTI transactions with memory occur on a CTI cache line basis. In Figure 16, processor cache lines 0 and 1 combine to form a single CTI cache line.

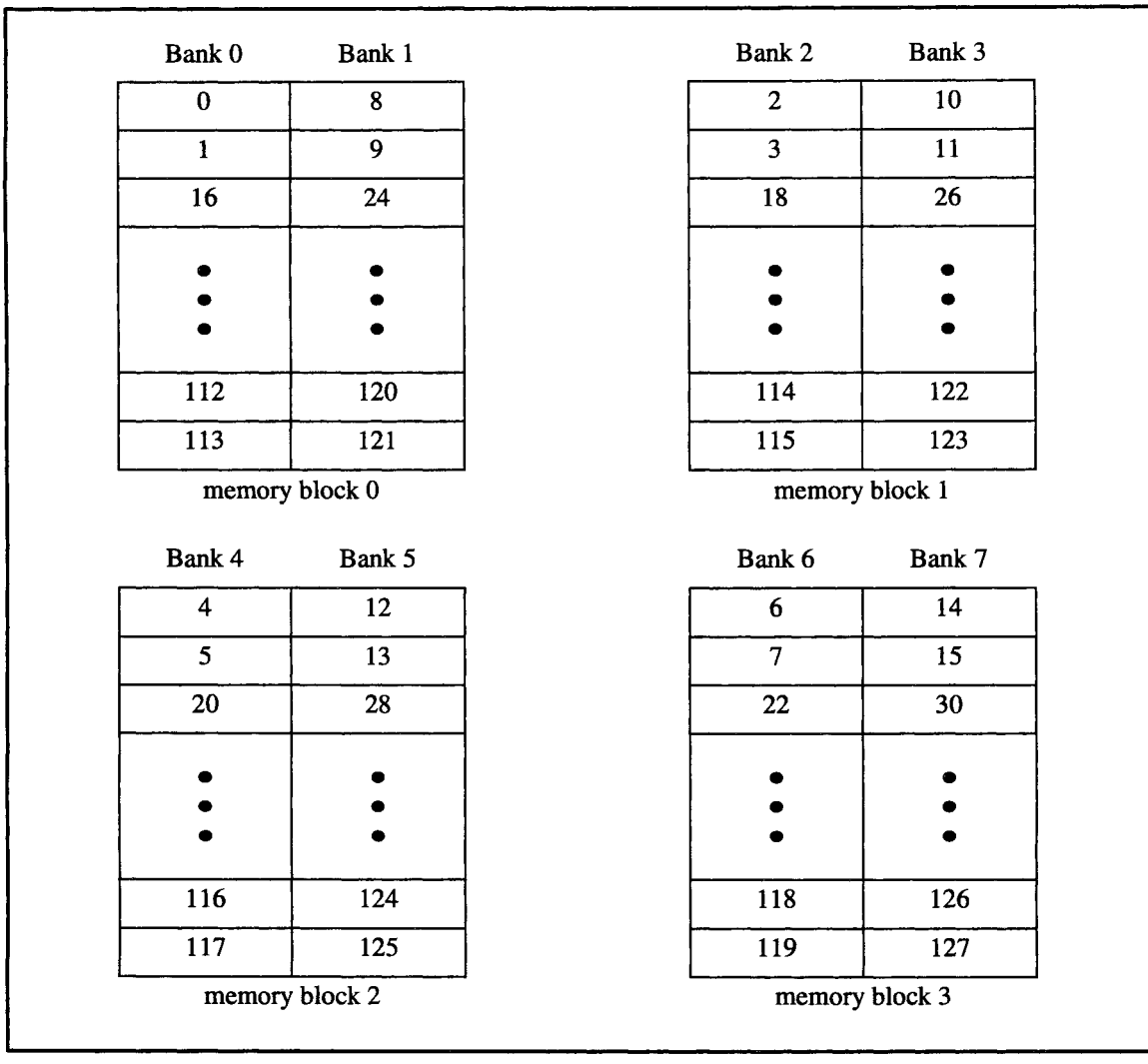

Figure 16 Processor cache line interleave

Coherent memory interleave

Memory interleaving is performed for all coherent memory accesses. In Exemplar, each of the four memory blocks per hypernode is always defined, but is not necessarily active or even physically present. Each memory block is associated with an interconnect ring (CTI Ring). Addresses are bound to cache lines in the *active* memory blocks through an interleave mechanism. Figure 17 illustrates the memory interleaving and virtual-to-physical ring mapping mechanisms.

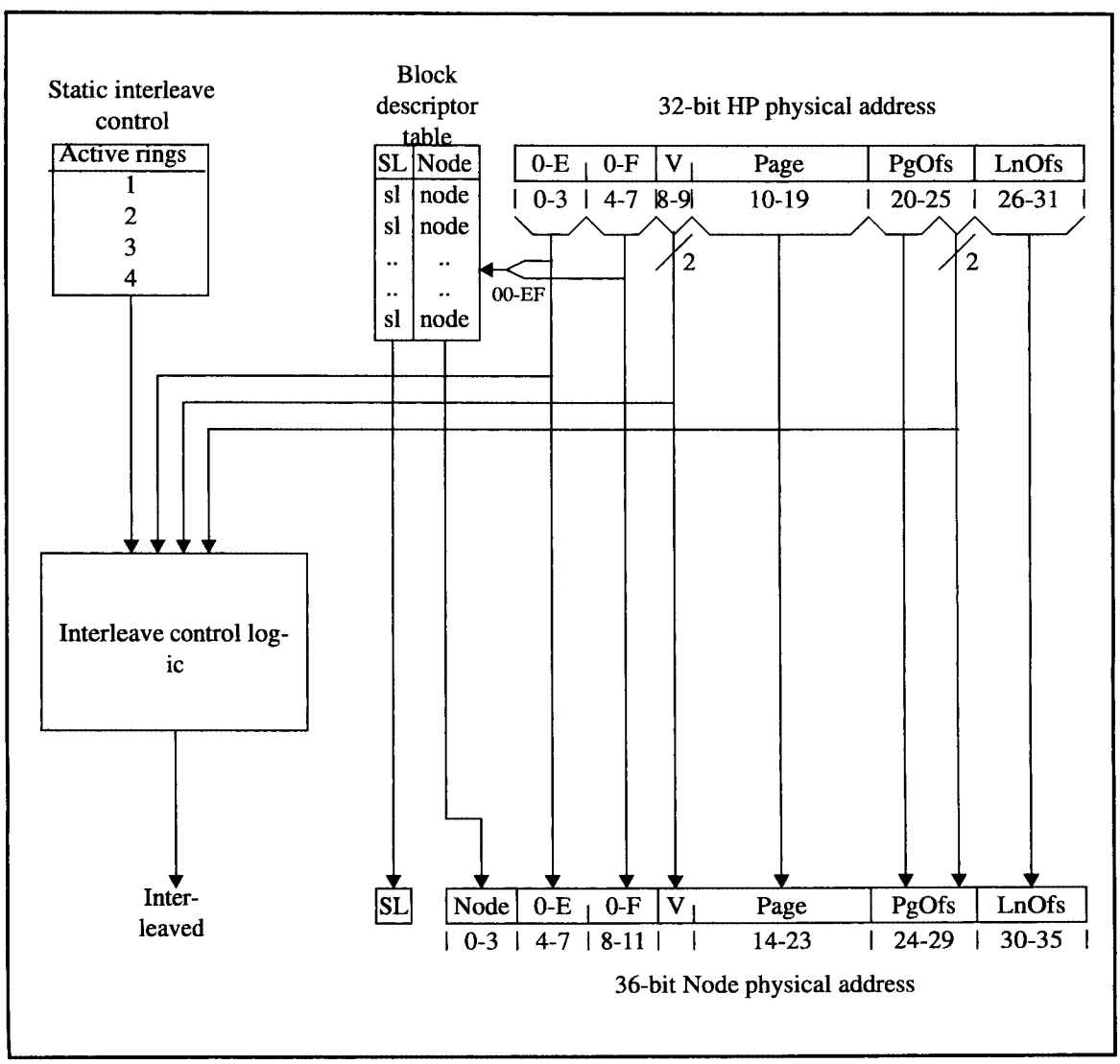

Figure 17 Coherent address space

The 32-bit HP physical address is partitioned into an 8-bit block descriptor table (BDT) index (bits 0-7), a 2-bit virtual ring (VR) corresponding to a ring of the interhypernode interconnect (bits 8-9), and the Normal Page, Page offset, and cache line offset fields.

The interleave control logic combines portions of the HP physical address with the selected number of active rings to produce an interleaved ring (IR) number. Table 5 shows the resulting IR number for one through four active rings for non-CD mode and Table 6 for CD mode.

Table 5 Virtual ring (VR) versus interleaved ring (IR) for non-CD mode

4 active rings			**3 active rings**			**2 active rings**			**1 active ring**		
VR	Ofs	IR	VR	Ofs	IR	VR	Ofs	IR	VR	Ofs	IR
00	00	00	00	00	00	00	00	00	00	00	00
00	01	01	00	01	01	00	01	01	00	01	00
00	10	10	00	10	00	00	10	00	00	10	00
00	11	11	00	11	01	00	11	01	00	11	00
01	00	01	01	00	01	01	00	01			
01	01	10	01	01	10	01	01	00			
01	10	11	01	10	01	01	10	01			
01	11	00	01	11	10	01	11	00			
10	00	10	10	00	10						
10	01	11	10	01	00						
10	10	00	10	10	10						
10	11	01	10	11	00						
11	00	11									
11	01	00									
11	10	01									
11	11	10									

Table 6 Virtual ring (VR) versus interleaved ring (IR) for CD mode

4 active rings			3 active rings			2 active rings			1 active ring		
VR	Ofs	IR	VR	Ofs	IR	VR	Ofs	IR	VR	Ofs	IR
00	00	00	00	00	00	00	00	00	00	00	00
00	01	01	00	01	01	00	01	01	00	01	00
00	10	10	00	10	00	00	10	00	00	10	00
00	11	11	00	11	01	00	11	01	00	11	00
01	00	01	01	00	01	01	00	01	01	00	01
01	01	10	01	01	10	01	01	00	01	01	01
01	10	11	01	10	01	01	10	01	01	10	01
01	11	00	01	11	10	01	11	00	01	11	01
10	00	10	10	00	10	10	00	10	10	00	10
10	01	11	10	01	00	10	01	11	10	01	10
10	10	00	10	10	10	10	10	10	10	10	10
10	11	01	10	11	00	10	11	11	10	11	10
11	00	11	11	00	11	11	00	11	11	00	11
11	01	00	11	01	11	11	01	10	11	01	11
11	10	01	11	10	11	11	10	11	11	10	11
11	11	10	11	11	11	11	11	10	11	11	11

The columns labeled VR and Ofs indicate the possible input combinations of virtual ring (VR) number and page offset (Ofs) for each possible number of active rings. The resulting interleaved ring (IR) number is read from the column labeled IR.

The interleave is performed on the virtual ring (VR) number, which always has a value between zero and the maximum number of active rings, regardless of the actual physical rings involved. The resulting interleaved ring (IR) is then translated by a four-entry static map to the appropriate physical ring number. In a non-CD configuration, the number of active memory boards in a node is the same as the number of active rings. For example, if there is only one active ring, there is only one interleave. In the CD configuration, the number of memory boards does not have to equal the number of active rings (CTI rings). For example, there may be only two active rings, but four memory boards in a node. In this case, IR0 and IR1 can be mapped to CTI rings and the

memory boards on those rings, and IR2 and IR3 would be mapped to the two remaining memory boards (there is no internode access on those two rings). Noncoherent accesses (BDT index = Fx) are not interleaved.

Exemplar coherent memory address augmentation

The 32-bit HP physical address is partitioned into an 8-bit block descriptor table (BDT) index (bits 0-7). The most significant 4 bits also specify whether access is coherent (values 0-E) or noncoherent (value of F). The BDT index selects a value for the hypernode ID and a 1 bit sharing level (SL) code from the BDT. The SL code, defined in Table 7, is used to qualify various performance monitor measurements.

Table 7 Exemplar memory types

SL value	Memory type
0	Hypernode-private
1	Global

The block descriptor table contains 240 entries for coherent memory access. Each of these entries specifies the actual physical hypernode ID for a 16-Mbyte block of coherent memory, for a total of 3,840 Mbytes of addressable coherent memory per hypernode. An Exemplar system will implement one of four hypernode memory sizes; as shown in Table 8. The redundant BDT entries may then be used to alias the implemented physical memory, providing simultaneous access to the same block index from multiple hypernodes.

Table 8 Exemplar coherent memory options

Option	DRAM size	Hypernode memory size
1	4 Mbits	256 Mbytes
2	4 Mbits	512 Mbytes
3	16 Mbits	1024 Mbytes
4	16 Mbits	2048 Mbytes
5	64 Mbits	4096 Mbytes

SPP1000 BDT

From a software perspective, the BDT entries are accessed as two separately packed register files: one for the hypernode ID (NID) and one for the sharing level (SL). These register files are illustrated in Figure 18. Accesses to these register files are through CSR address space using word-aligned word loads and stores only.

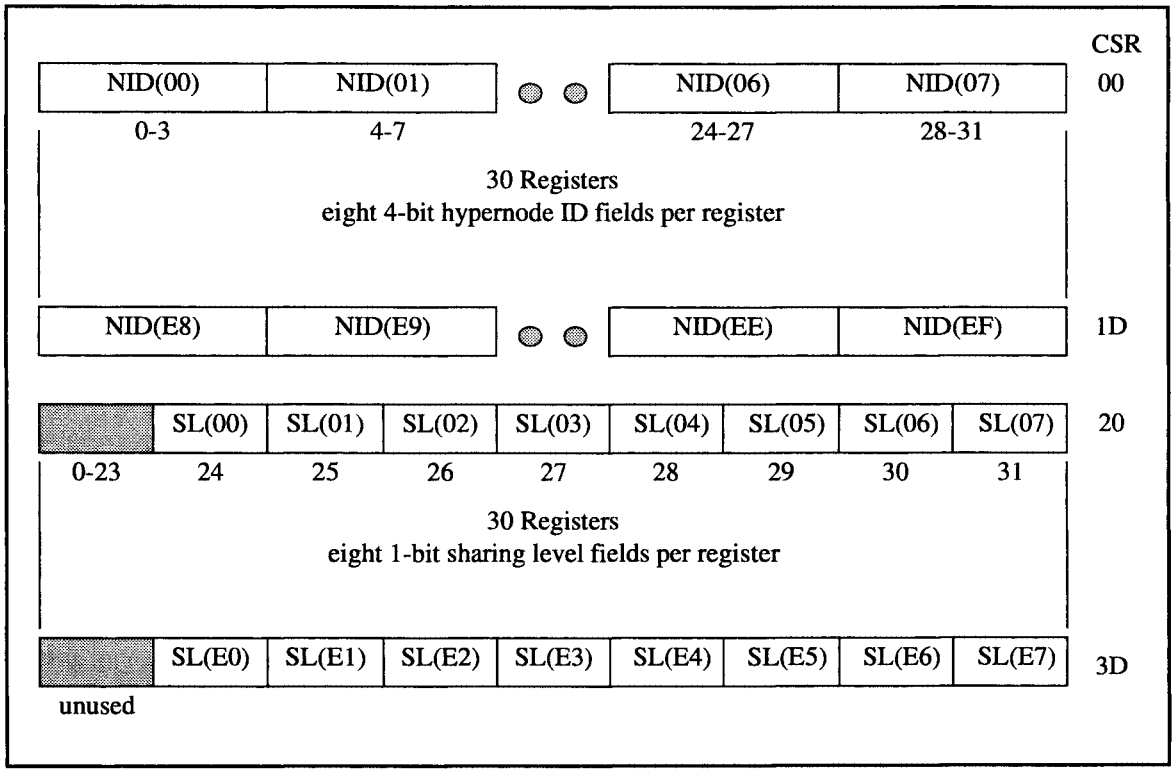

Figure 18 Software view of BDT registers for SPP1000

SPP1200 and SPP1600 BDT

The SPP1200 and SPP1600 use only one sharing-level bit for all memory blocks. From a software perspective, the BDT entries are accessed as one packed register file containing the hypernode ID (NID) and one single register for the sharing level (SL). These register files are illustrated in Figure 19. Accesses to these register files are through CSR address space using word-aligned word loads and stores only.

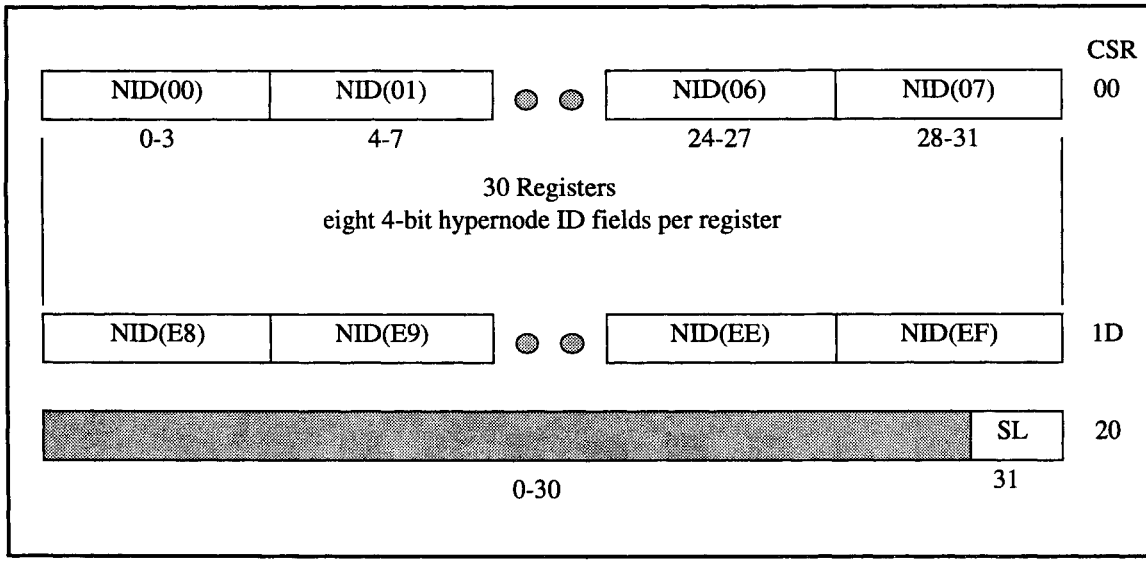

Figure 19 Software view of BDT registers for SPP1200 and SPP1600

BDT aliasing

Implemented memory is distributed across the 240 BDT entries and the CTI cache. Memory is installed starting at address 0x7FFFFFFF and grows toward address 0x00000000. The CTI cache is always allocated starting at address 0x7FFFFFFF and consists of a contiguous block of 2**(n+24) bytes for n=0,1,2,....

Figure 20 illustrates allocation of memory blocks through the BDT and possible aliasing.

Each column represents 256 Mbytes of memory. The last column (0xF0000000) is noncoherent memory space and unavailable for mapping. Each horizontal row represents the possible BDT aliases for the physical memory implemented in that row. The arrow in each example points from the beginning to the end of the configured CTI cache. In the figure, the 256-Mbyte example has 32 Mbytes of CTI cache; all the others have 64 Mbytes of CTI cache. BDT entries that alias to the CTI cache in the addressed hypernode are not usable.

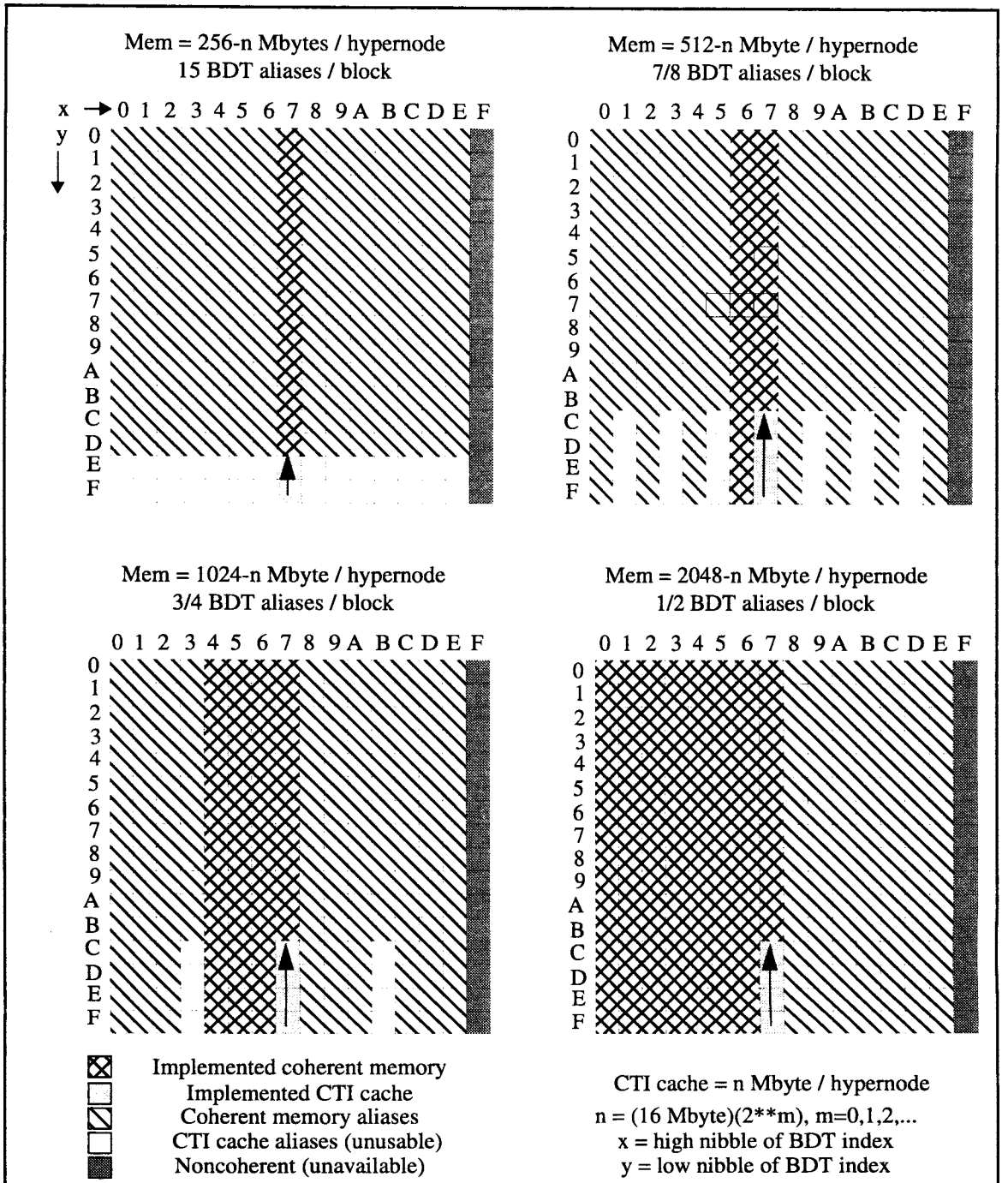

Figure 20 Allocation of BDT(xy) for all Exemplar memory configurations

Figure 21 illustrates the address aliasing performed by a memory module of each possible size. The module ignores the bits corresponding to possible address aliasing. The operating system then allocates the BDT entries based on knowledge of the implemented size of each referenced hypernode, as well as the amount of memory allocated to the referenced hypernode CTI cache.

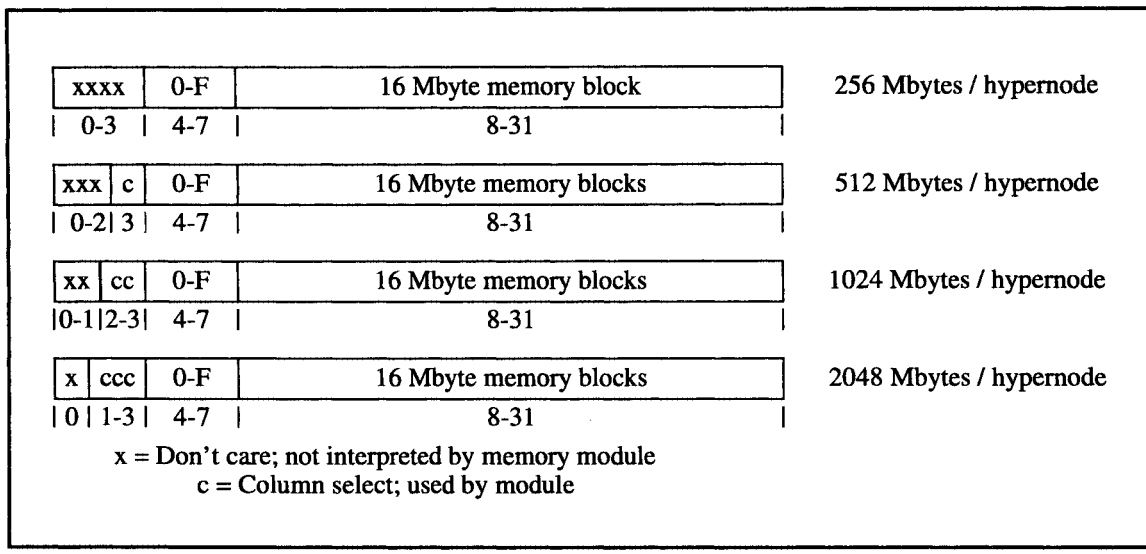

Figure 21 Memory module address aliasing versus hypernode memory size

All caches record the complete cache line address, including the alias bits, in their various tags to differentiate between identical blocks (same index within hypernode) accelerated from different hypernodes. Software must always use the same physical address alias to reference any given cache line to maintain correct cache coherency.

Exemplar noncoherent address space

This section describes mapping of the physical addresses generated by the HP processor for noncoherent space to the 36-bit hypernode physical addresses, and the 64-bit interhypernode physical addresses. Figure 22 shows the noncoherent address space map.

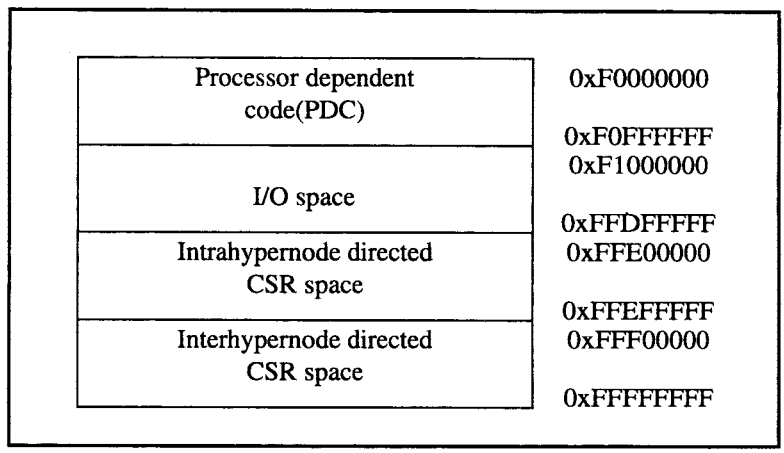

Figure 22 Noncoherent address space

Processor dependent code (PDC) space

Processor dependent code (PDC) consists of 16 Mbytes of space for boot and diagnostic code, as well as any special hardware shared by all processors on a hypernode for diagnostics or console communication. Accesses to PDC space are routed to the Service Hypernode interface subsystem. The address range corresponding to PDC space is from 0xF0000000 through 0xF0FFFFFF (see Figure 23).

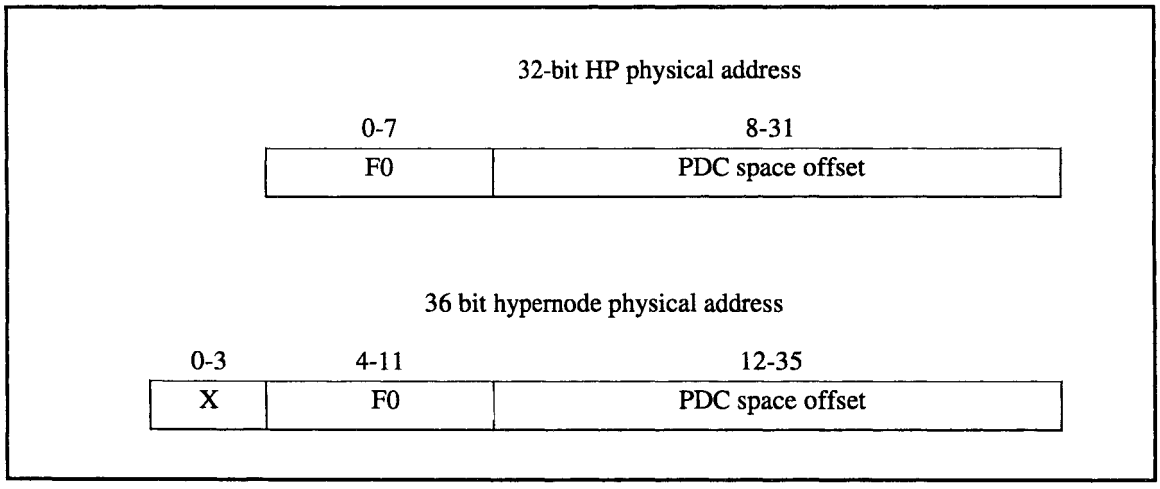

Figure 23 PDC space

The most significant 4 bits of the 36-bit hypernode physical address are not used by the Service Hypernode interface subsystem and may default to any value.

I/O space

All accesses to I/O space are routed to one of two I/O crossbar destinations. The I/O address space starts at address 0xF1000000 and continues until 0xFFDFFFFF. The partitioning of the address ranges to the two I/O controllers is determined by bit 7 of the 32-bit processor physical address. When bit 7 has the value 0, I/O controller 0 receives the packet. Similarly, if the value is 1, I/O controller 1 receives the packet. Figure 24 shows the translation map for the 32-bit HP physical address to the 36-bit hypernode physical address.

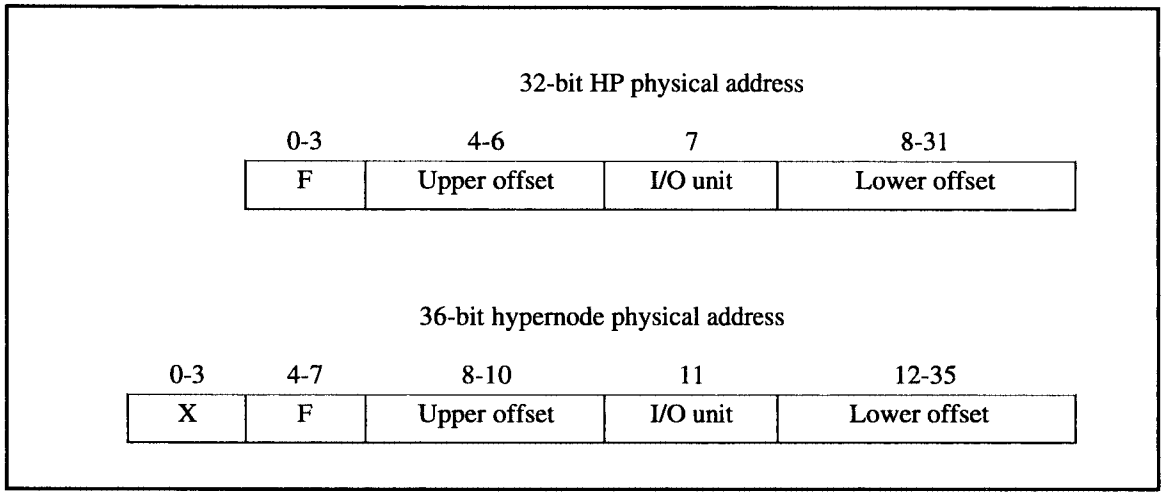

Figure 24 I/O space

Currently, only the address range of 0xF2000000 through 0xF7FFFFFF is used by the I/O controllers. The addresses below 0xF2000000 and above 0xF7FFFFFF are detected by the I/O controllers, and an unimplemented address error response will be returned to the requesting processor.

The most significant 4 bits of the 36-bit hypernode physical address are not used by the I/O controller and may contain any value.

Intrahypernode-directed CSR space

The 32-bit physical address generated by the HP processor used to access the intrahypernode-directed CSR space is shown in Figure 25.

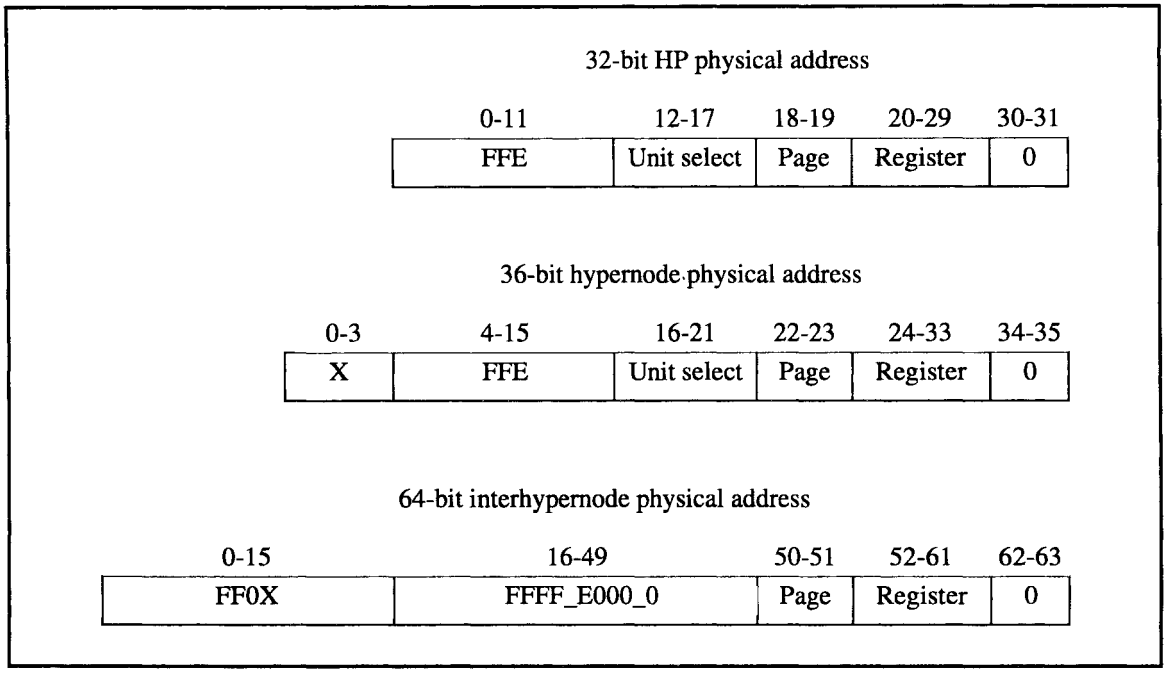

Figure 25 Intrahypernode directed CSR space

Unit select field

The *Unit select* field is 6 bits wide and is used by the system to route the request packet to the appropriate destination. The most significant 4 bits of the unit select field, also known as the *major unit select value*, are used by the CPU agent to construct the 16 bit CxBar route word.

The least significant 2 bits of the unit select field, also known as the *minor select value*, are passed from the 32 bit HP physical address to the 36-bit CxBar hypernode-physical address (to allow further packet routing at the CxBar destination).

A unit select value of 0xF represents a routing destination that is the same as the requesting processor, that is, the destination is a CSR of the "local" processor.

Table 9 shows the defined values for the unit select field.

Table 9 Unit select field values

Major unit select value	Minor unit select value	Routing destination
0	0	Processor 0
1	0	Processor 1
2	0	Processor 2
3	0	Processor 3
4	0	Processor 4
5	0	Processor 5
6	0	Processor 6
7	0	Processor 7
8	0	CMC 0
8	1	CTI 0
9	0	CMC1
9	1	CTI 1
A	0	CMC 2
A	1	CTI 2
B	0	CMC 3
B	1	CTI 3
C	0	I/O 0
D	0	I/O 1
E	0	Utility board
F	0	"local" Processor

Page field

The *Page* field is 2 bits wide, and is used by the destination to qualify the CSR location accessed by the appropriate access restrictions.

Table 10 shows access restrictions for each of the four available pages.

Table 10 Page field values

Page field value	Legal access modes	Intended usage
0	R/W	Kernel access All CSRs mapped
1	Read Only	User access User readable CSRs
2	R/W	User access User R/W CSRs
3	None	Reserved

Page zero has all CSRs mapped and is used by the operating system kernel to access any CSR. Page one is mapped into user space to allow access to read-only CSRs. Only the CSRs to be read by a user process should be aliased to page one. Page two is also intended to be mapped into user space. Read and write access is allowed to the CSRs aliased to this page. Page three is reserved; no CSRs are accessible through page three.

Internal CSRs for SPP1200 and SPP1600

When a CPU in the SPP1200 or SPP1600 systems (PA7200) accesses its internal CSRs, it does so by addressing Major_unit=E, Minor_unit=3, page=0. Accesses to that page are not routed through the system (ei.the MU cannot access the registers on that page). When software accesses that page, it has access to its own PA7200 CSRs. The system ignores the request. The same PA7200 registers are mapped into the Processor pages (Major_unit=(0-7), Minor_unit=0) so the other processors and the MU can access the registers. Major_unit=F can also access the PA7200 CSRs.

Access via Major_unit=E, Minor_unit=3, page=0 is the highest-performance way to access the PA7200 local CSRs, but its only a local access directly on the CPU (no MU access).

Note

Accessing the CTI devices through intrahypernode-directed CSR space will result in accesses to the private addresses of the CTI hypernode chip (FFFF_E000_0XXX).

The IO_EIR addresses for the CPU agents should be FFE0_0000 for processor 0, through FFE7_0000 for processor 7.

Interhypernode directed CSR space

The 32-bit physical address generated by the HP processor used to access the interhypernode directed CSR space is shown in Figure 26.

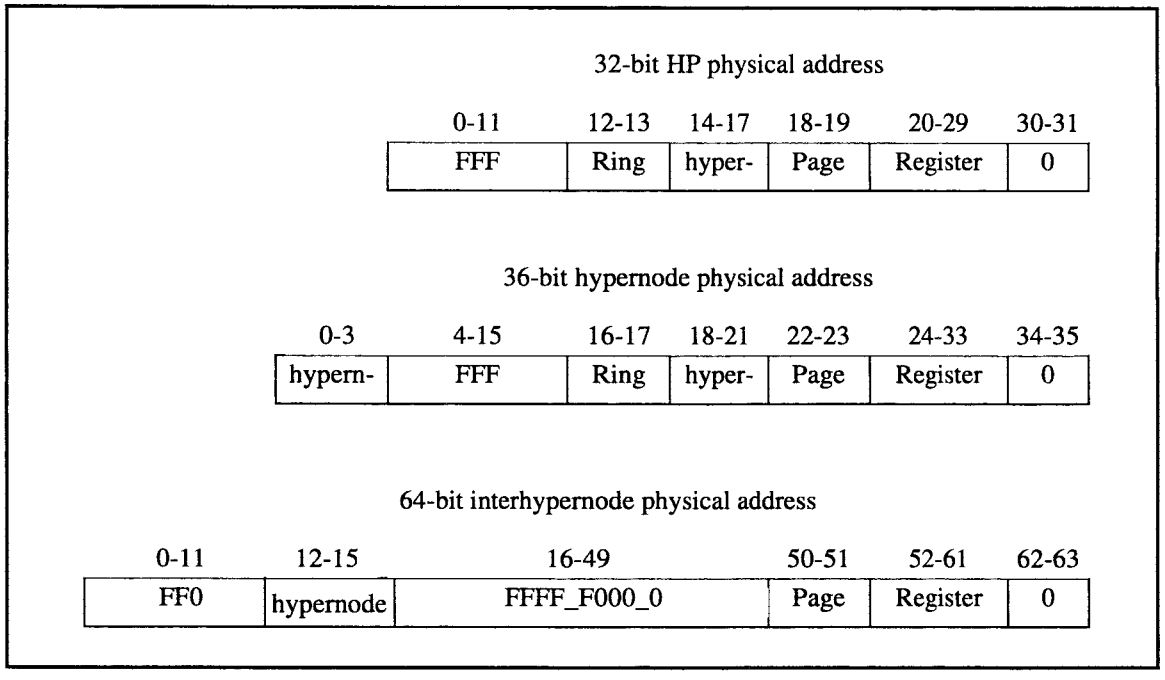

Figure 26 Interhypernode directed CSR

The ring field of the 32-bit HP physical address is used in generating the 16-bit route mask for the crossbar. The ring bits are carried through to the 36-bit hypernode physical address.

The Page field is used to select one of the four available pages per destination hypernode to be accessed. The four pages are used for access restriction checking.

Virtual memory 4

Overview

This chapter describes the virtual memory system for the Exemplar, including address translation, memory protection, and virtual address space allocation.

Virtual memory architecture

The Exemplar is constructed using a hierarchical memory architecture, meaning several types of memory are available with different sharing and latency characteristics. The following types of memory are provided, listed in order of increasing memory latency:

- **CPU-private** memory is provided for data that is accessed within a single CPU only. Since this memory is typically implemented on the same board with the CPU, it has the lowest latency.
- **Hypernode-private** memory is provided for data that is shared only by CPUs within a single hypernode. The hypernode-private memory of one hypernode may not be accessed from a different hypernode. Hypernode-private memory may be implemented as an interleaving of all CPU-private memory on the hypernode.
- **Near-shared** memory is globally accessible from all hypernodes, but has affinity for its *home* hypernode. Accessing near-shared memory from any hypernode other than the home hypernode results in a latency penalty.
- **Far-shared** memory is globally accessible from all hypernodes and is accessed with equal latency from any hypernode participating in the application. Far-shared memory may be implemented as an interleaving of all the near-shared memories of the hypernodes participating in the computation.

SPP implementations are not required to physically implement all four types of memory, provided that each memory type can be logically provided to software. This gives the hardware

implementation considerable latitude; for example, CPU-private memory could be implemented as an operating system partitioning of near-shared memory, as is the case in Exemplar.

Note

It may be possible to access CPU-private memory on another hypernode. The operating system would use this facility for I/O or other system management functions. Internode references to CPU-private or hypernode-private memory are not exported to user applications.

SPP is architecturally defined to have 64-bit virtual addressing capabilities. Exemplar products are compliant with PA-RISC Version 1.1, Level 1. This implementation provides a 4-Gbyte address space to a single thread of computation.

PA-RISC virtual memory architecture

The PA-RISC 1.1 architecture structures virtual memory as a set of virtual *spaces*, each containing 2^{32} bytes (4 Gbytes). Spaces are identified by a *space id* that is obtained from a space id register. An entire virtual address is formed by the concatenation of the space id and a 32-bit *offset* within the space.

For memory management purposes, each space is divided into pages of 4 Kbytes each. Additionally, a space is logically broken into four 1-Gbyte regions called *quadrants*.

Different levels of the PA-RISC architecture implement varying numbers of spaces: Level 1 implements 2^{16} spaces, level 1.5 implements 2^{24} spaces and level 2.0 implements 2^{32} spaces.

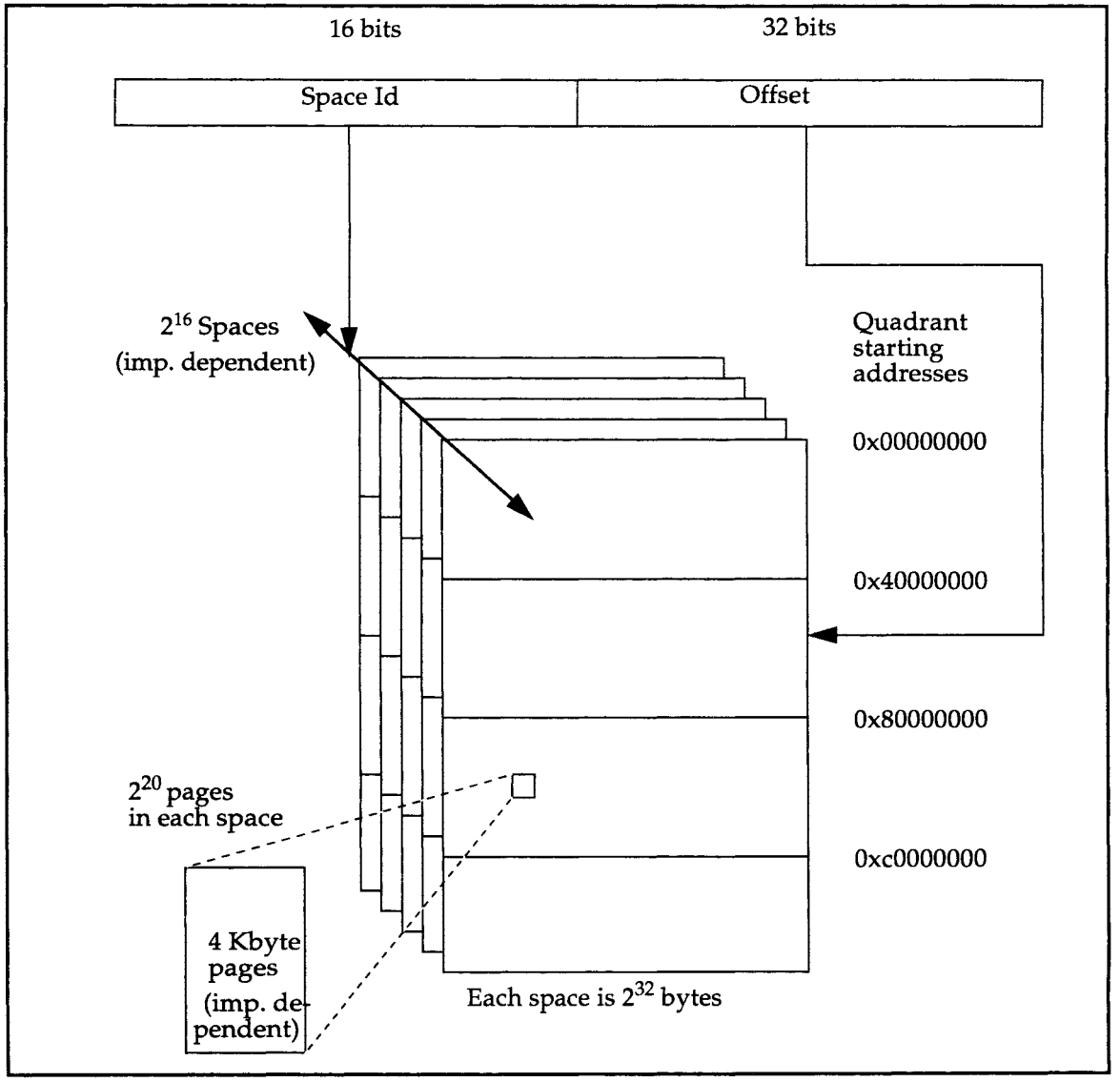

Figure 27 Structure of spaces, offsets and pages in Exemplar

Pointers and PA-RISC address generation

Pointers in the Exemplar are defined to be 32 bits long. The Convex compilers allocate 4 bytes of storage on 4-byte boundaries to hold pointer and address container variables. Relying on the size of pointers (converting pointers to other data types, for example) should be avoided. C-language constructs for pointer manipulation are guaranteed to function properly on all Convex SPP platforms.

Instruction addresses consist of a space identifier and a 30-bit word offset, which come from the instruction address space queue (IASQ) and the instruction address offset queue (IAOQ). Instructions are always word-aligned. The bottom two bits of the IAOQ are interpreted to be the *privilege level* (PL) of the corresponding instruction (0=most privileged). Each instruction address queue is two-deep, corresponding to the current instruction and the next instruction to be executed.The IASQ and IAOQ queues serve the same purpose as the program counter (PC) in conventional architectures.

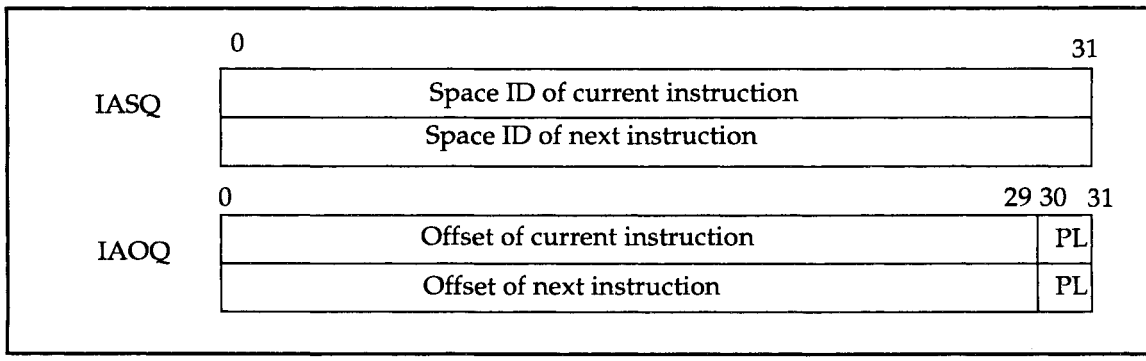

Figure 28 IASQ and IAOQ format

Data addresses consist of a space ID provided by one of eight space registers (SR[0]..SR[7]) and an offset. The offset is the sum of the base register, index register and sign-extended displacement specified in the instruction making the data reference. The space register is specified by a two or three bit s-field in the instruction (the width is instruction-dependent). If the s-field is three bits wide, it specifies the space register to be used directly. If the s-field is two bits wide and nonzero, it specifies SR[1], SR[2], or SR[3]. If the s-field is two bits wide and zero, the space register is determined by adding four to the most significant bits of the base register used by the instruction.

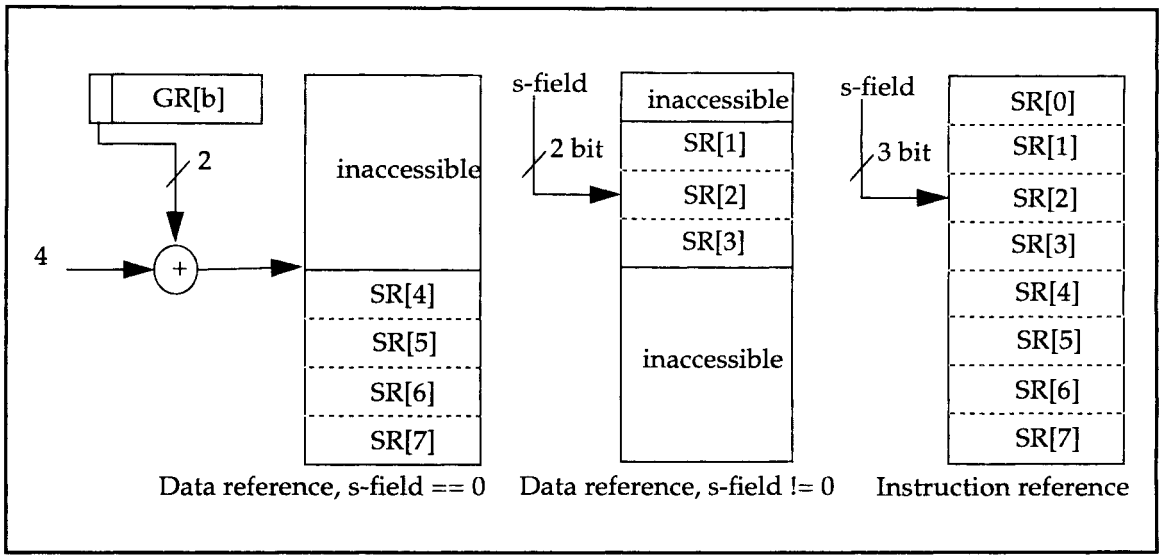

Figure 29 Space register selection for data and instruction references

Data references with the two bit s-field equal to zero permit addressing of four distinct spaces selected by program data. This is called short pointer addressing because a 32-bit value from a base register selects a space register and provides a 30-bit offset to the space. Only one quadrant of each space is directly addressable by the base register. The quadrant is selected by the uppermost 2 bits in the base register.

Virtual-to-physical address translation

The PA-RISC architecture uses a translation look-aside buffer (TLB) to translate virtual-to-physical addresses. The TLB acts as a cache of active translations. Some PA-RISC CPUs depend on the operating system to fetch nonresident TLB translations, while others provide a hardware-miss handling facility.

CPUs using hardware TLB refill maintain a directory of virtual-to-physical translations for all memory resident pages in a structure called the physical page directory (PDIR). In the event of a TLB miss, the CPU chip fetches entries from the PDIR to update the TLB. The format of the PDIR is CPU implementation-dependent. Systems using software TLB refill may use a PDIR or another software data structure to locate information needed to refill the TLB.

There are two types of TLB entries: page entries and block entries. Page entries each translate one physical page, while block entries translate contiguous virtual pages to a contiguous region of physical address space. The number of each type of entry and the size of block translations is implementation-dependent.

Note

Block TLB entries should be used whenever possible, because they allow a larger range of virtual address translations to remain in the TLB without misses. Block TLB entries are especially appropriate for mapping pieces of the operating system and within an application, for mapping 16-Mbyte blocks that can be paged in or out together. Block TLBs may not be purged or maintained with the regular TLB instructions (PDTLB, PITLB, PDTLBE, PITLBE, IDTLBA, IITLBA, IDTLBP, or IITLBP). Refer to the "Exemplar implementation specific information" for information on maintaining block TLB entries.

Each TLB page entry contains the following fields: virtual page number, physical page number, access rights, access ID, and four single bit flags. The sizes for Exemplar are shown in Figure 30.

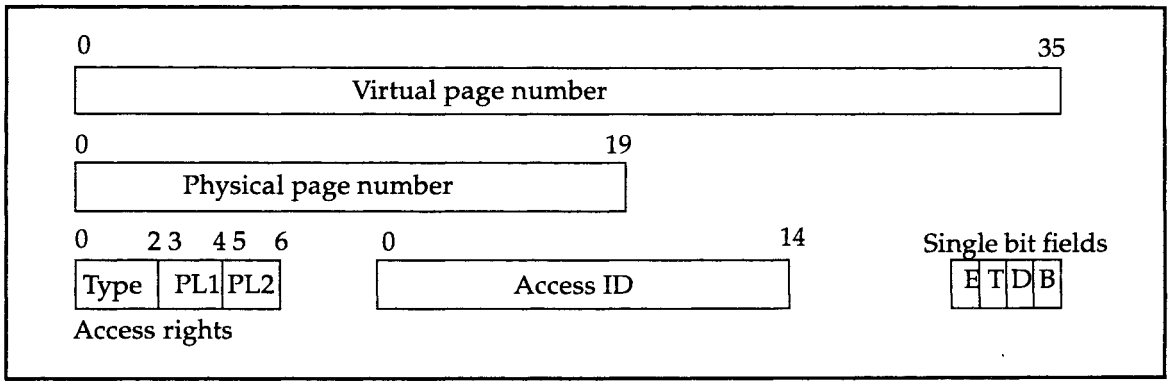

Figure 30 Format of a TLB page entry in Exemplar.

The access rights field is 7-bits, and the access ID field is 15 bits. The access rights field is subdivided into a 3- bit *Type* subfield and two 2- bit privilege level subfields, *PL1 and PL2*.

The four single-bit fields are:

- **E**—Entry valid. When 1, the translation is valid.
- **T**—Page reference trap. When 1, data references using this translation cause a page reference trap. This bit is used for debugging.
- **D**—Dirty. When 0, store and semaphore instructions cause a TLB dirty bit trap on systems with software TLB-miss handling. When 0, store and semaphore instructions set the D bit in the DTLB entry to 1 on systems with hardware TLB-miss handling. When 1, no trap or update occurs. The D bit is used to determine which pages have been modified.
- **B**—Break. When 1, instructions that could modify data using this translation cause a data memory break trap interruption, if enabled. Store instructions, the purge data cache (PDC) instruction and semaphore instructions are the only instructions that could modify data. This bit is used for debugging.

The PA-RISC architecture allows separate TLBs to be used for instruction and data references; these are referred to as ITLBs and DTLBs, respectively. A combined TLB can be used alternately, in which case it is referenced as the DTLB.

There are also instructions for purging the TLB entries: PITLB, PITLBE, PDTLB and PDTLBE. The PITLB and PDTLB use

broadcast transactions to ensure that all CPUs within a hypernode purge the corresponding TLB entries; the PITLBE and PDTLBE instructions affect only the CPU executing the instruction. In no case are the broadcast TLB instructions visible across hypernode boundaries.

Note

Software is responsible for all TLB consistency issues across multiple hypernodes.

The block TLB entries and the hardware TLB refill mechanism are CPU implementation-dependent and are discussed in the "Exemplar implementation specific information" section on page 87.

Memory access control

An access is validated if both the check of the access rights and the protection identifiers succeed. If the access is validated, the instruction or data reference is completed. If the access is not validated, the instruction is terminated with a protection trap. Instruction access violations are reported with instruction memory protection traps. Data read or write access violations are reported with data memory protection or unaligned data reference traps.

Each process has a set of process attributes that are used in access control validation. Process attributes include:

- **Privilege level (PL)**—Every instruction is fetched and executed at one of four privilege levels (0-3, with level 0 being most-privileged). The privilege level is kept in bits 30 and 31 of the address register of the current instruction (the front element of IAOQ). All accesses except probe instructions use the PL of the current instruction. Probe instructions specify the PL explicitly.
- **Protection IDs**—The four control registers: CR8, CR9, CR12 and CR13, contain protection identifiers associated with the current process. These registers are used to allow several different protection groups to be accessed. Each register contains a protection identifier (protection ID) and a write disable (WD) bit as shown in Figure 31.

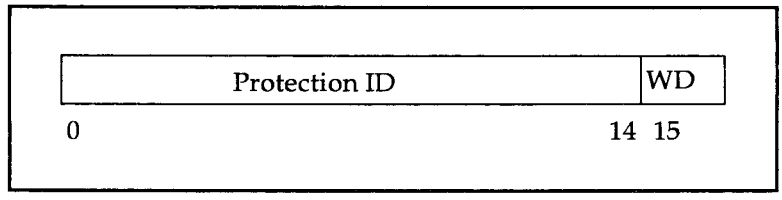

Figure 31 Format of protection IDs.

- **Program status word (PSW)**—Three bits in the PSW enable or disable address translation and the protection identifier check. The protection identifier check is enabled if the P bit is set. A zero value disables this check. Address translation and access rights checking for data references is enabled if the D bit is 1.0 disables translation and access rights checking for data references. Address translation and access rights checking for instruction (execute) is enabled if the C bit is set. A zero value disables the translation/check.

The access rights check is accomplished according to the following table. The type field from the TLB access rights field selects the type of accesses allowed. This is designated by rows in Table 11. The requested access is validated by the rules shown in the "Privilege check" column of the table by comparing the privilege level of the currently executing instruction (PL) to the PL1 and PL2 subfields in the TLB access rights field.

Table 11 Access rights interpretation

Type field from TLB	Allowed access types	Privilege check
0	Read-only; data page	read: PL <= PL1 write: not allowed execute: not allowed
1	Read/Write; dynamic data page	read: PL <= PL1 write: PL <= PL2 execute: not allowed
2	Read/Execute; normal code page	read: PL <= PL1 write: not allowed execute: PL2 <= PL <= PL1
3	Read/Write/Execute; dynamic code page	read: PL <= PL1 write: PL <= PL2 execute: PL2 <= PL <= PL1
4	Execute: gateway promote to privilege level 0	read, write: not allowed execute: PL2 <= PL <= PL1
5	Execute: gateway promote to privilege level 1	read, write: not allowed execute: PL2 <= PL <= PL1
6	Execute: gateway promote to privilege level 2	read, write: not allowed execute: PL2 <= PL <= PL1
7	Execute; gateway promote to privilege level 3	read, write: not allowed execute: PL2 <= PL <= PL1

The protection identifier check is accomplished by comparing the TLB access ID with the four protection IDs in the process attributes. The check is validated if any of the protection IDs are equal to the access ID. In the case of write-access checks, the WD bit of at least one matching protection ID must be zero to allow access. If the TLB access ID field is zero, a protection identifier check is not done; zero is used to designate public pages.

CPU caches

The PA-RISC architecture defines separate instruction and data caches. These caches are virtually indexed and physically tagged. The size and indexing function of the caches are CPU implementation-dependent. Some CPU implementations may use a combined data/instruction cache. The data cache is normally implemented to have a write-back policy for updating memory.

The operation of moving information from memory into a cache is referred to as a move-in. Only instructions and data referenced by executed instructions may be moved in. Additionally, no data reference may cause either an instruction cache or data cache move-in. This means only the data cache must be flushed to guarantee cache lines referenced as data are removed from the cache system. Similarly, only the instruction cache must be flushed to guarantee that cache lines referenced as instructions are removed from the cache system.

A move-in may cause more than a single cache line of data to be imported into the caches. All of the cache lines of the referenced page may be imported for data references. For instruction references, all of the cache lines in the referencing page and the following page (virtual or physical) may be imported. A flush cache, purge cache, or purge TLB instruction stops any subsequent move-in operations to that page until another reference is made.

SPP1200 and SPP1600 on-chip cache

The PA7200 implements a 64-line, fully-associative, prefetch-miss (PM) on-chip data cache that is accessed in parallel with the main off-chip cache. This PM cache helps hide memory latency and minimize cache miss rates. In addition, the PA7200 uses an efficient prefetch algorithm for instruction and data caches that reduces the stall cycles associated with memory latency.

Cache flushes

The caches may be flushed or purged with the PDC, FDC, FIC, FDCE, or FICE instructions. These instructions are not privileged. The PDC, FDC and FIC instructions are broadcast to other CPUs within a hypernode that may be sharing the same cache lines. These instructions, therefore, have global effects within a hypernode.

Note

Cache flush instructions must be followed by a sync instruction to ensure that all flushes have made it to memory.

Address aliasing

Normally, a virtual address does not translate to two different physical addresses. The operating system must ensure this does not happen through correct management of TLBs and PDIRs.

Several virtual addresses, or a virtual address and a physical address, often map to the same physical memory. Such mappings present a consistent view of memory if the current read operation produces the same data that was stored last.

The CPU caches permit a physical memory location to be accessed by both a physical address and a virtual address where the two are identical. Such a virtual address is said to be equivalently mapped. Equivalently mapped virtual and physical references present a consistent view of memory.

Two or more distinct virtual addresses mapped to the same physical page are said to be virtual aliases. Virtual aliases may or may not provide a consistent view of memory, depending on the difference in their virtual addresses (both the offset and space portions). In order for two mappings to present a consistent view, they must index the same cache line. The actual indexing function is CPU implementation-specific.

To provide a consistent mapping of virtual aliases, only bits 0, 1, 2, 3, 8, 9, 10, 11, 16, 17, 18, and 19 and be used for space handling.

Note

The *PA-RISC 1.1 Architecture Manual* lists a hardware requirement to ensure consistent views of virtual aliases that have only offset bits 12-31 equal, regardless of the space bits. The operating system assumes that the data cache will grow. As a result, the operating system should be designed allow aliasing only when the offset differs by multiples of 16 Mbytes or more.

Caches are not required to provide a consistent view of memory for virtual aliases that differ in bits from those listed above. Virtual and physical mappings to the same physical page that are not equivalently mapped are not required to provide a consistent view of the memory.

The operating system flushes the cache lines when:

- The address mapping in the TLBs is changed.
- A physical access is made to a location that might reside in the caches as a result of a virtual access that was not equivalently mapped.
- A virtual access is made to a location that might reside in the caches as a result of a physical access that was not equivalently mapped.

- A virtual access is made to a location that might reside in the caches as a result of another virtual access that differs in bits from those listed above.

Cacheability

Memory references eligible for cache move-in are said to be cacheable references. Some memory references may not legally cause a move-in; these references are referred to as uncacheable.

I/O space references (with a CPU physical address of 0xf0000000 through 0xffffffff) are uncacheable (also referred to as *noncoherent* references). The I/O address space may be referenced in words, halfwords and bytes using normal load and store instructions. Floating point loads and stores to I/O space are undefined.

Cache coherence with I/O

I/O controllers reference data directly in memory; this data may be inconsistent with data in the processor caches. The operating system is required to:

- Flush cache lines involved in I/O before an I/O output.
- Flush or purge cache lines involved in I/O before an I/O input.
- Refrain from making data references to the page containing I/O buffers until the I/O operation has completed. This a consequence of the fact that a data reference may cause all cache lines on the referenced page to be imported into the cache.

This restriction implies I/O buffers should be aligned with the cache line; simultaneous use of the same cache lines by CPUs and I/O can result in an inconsistent view of memory.

Cache coherence between processors

Purges and flushes do not cause TLB faults on other processors. In the cache-coherent part of a multiprocessor system, all data references must be satisfied by data obtained using cache coherence checks. This data must have remained coherent since it was moved in. Implementations with write buffers must also check buffer contents on cache coherence checks in order to ensure proper ordering of storage accesses.

Store ordering

SPPs have the option of using weak-store ordering, that is, stores and cache flushes may complete out of order. When weak ordering is used, the sync instruction must be implemented to

ensure that pending stores and cache flushes have been completed.

SPP implementations that use weak-store ordering must provide a mode bit to force strong ordering, that is, constraining stores and cache flushes to complete in order. Strong ordering is required to fully support the PA-RISC application binary interface, since current PA-RISC machines enforce strong ordering. The mode bit allows a mix of strong- and weak-ordered applications to run on the same CPU under operating system control.

The CTI cache

Each hypernode maintains a cache of memory references sent from the CTI to other hypernodes. This is referred to as the CTI cache. Any CTI data that has been moved into a CPU cache on the same hypernode and is still resident in the CPU cache is also "encached" in the CTI cache. Consequently, the CTI cache directory information can be used to locate any global data currently encached by the hypernode.

The CTI cache is physically indexed and tagged with the global physical address. Since the cache is physically indexed, there are no aliasing requirements.

The SPP system ensures cache coherence between multiple hypernodes (two or more hypernodes that map the same global address will get a consistent view). This is done by maintaining a linked sharing list that contains a list of all the hypernodes sharing each cache line, or the hypernode that exclusively owns the cache line. Within every hypernode a record is kept of which CPUs have encached each line in the CTI cache so that CTI coherency requests can be forwarded to the appropriate CPUs.

The CTI cache line size is 64 bytes. This may be a multiple of the CPU cache line size, depending on CPU implementation.

Cache flushing and purging

The flush and purge cache instructions have different results, depending on whether the cache line was moved in from CPU-private, hypernode-private, or near- or far-shared memory. These instructions are: FDC, FDCE, FIC, FICE, and PDC. These instructions are not privileged.

FDCE and FICE flush entries from the executing processor only. If the cache line in the data cache is dirty, it will be written back to local memory or the CTI cache.

FDC and PDC flush entries from all processors on the hypernode and from the CTI cache as well. FDC will write data from dirty cache lines back to memory; PDC will discard data in dirty cache lines.

The SPP1200 and SPP1600 CPUs do not support PDC purges at the privileged-user (OS) level. Nonprivileged PDC instructions are translated to FDC instructions.

FICE and FIC flush the instruction caches of one or all CPUs within the hypernode, but these operations are ignored by the CTI cache.

Hardware interface routines are provided to implement a global cache flush (`cache_flush`) cache line prefetch into the CTI cache (`read_prefetch` and `write_prefetch`). These operations are provided as subroutine calls to ensure forward compatibility across all SPP implementations. The implementation strategy for Exemplar is given in the "Exemplar implementation specific information" section on page 87.

The following table gives the result of executing the various flush, purge and prefetch primitives for differing initial conditions.

Table 12 Cache flush and purge instruction effects

Instruction	Current state	Mem type	Resulting CPU cache state	Resulting CTI cache state	Other hypernodes state
FDCE, FDC, FIC, FICE, PDC	clean, in data or instruction cache	local	no longer encached	not applicable	not applicable
FDC, FDCE	dirty, in data cache	local	data written to memory, no longer encached	not applicable	not applicable
PDC	dirty, in data cache	local	data lost; no longer encached	not applicable	not applicable
FDC, FIC, PDC	not encached in CPU	local	no longer encached on any CPU in hypernode	not applicable	not applicable
FDCE, FICE, FIC	clean, in CPU data or instruction cache	global	data no longer encached	unchanged, data may still be encached	may be encached
FDCE	dirty, in CPU data cache	global	data written to CTI cache; no longer encached	dirty; encached	not encached (current hypernode is exclusive owner)
FDC, PDC	clean, in CPU data cache	global	data no longer encached	data no longer encached	may be encached
FDC	dirty, in CPU data cache	global	data written to memory; no longer encached	no longer encached	memory consistent; no longer encached

Table 12 Cache flush and purge instruction effects—**(continued)**

Instruction	Current state	Mem type	Resulting CPU cache state	Resulting CTI cache state	Other hypernodes state
PDC	dirty, in CPU data cache	global	data lost; no longer encached	no longer encached, not required to be written to memory	no longer encached
FDC, PDC	data not in CPU cache, but present in CTI cache	global	unchanged (not encached)	written back if dirty; no longer encached	may be encached if clean
FDCE, FICE	data not in CPU cache, but present in CTI cache	global	unchanged (not encached)	unchanged	unchanged
cache_flush	any state	global	written to memory if dirty; no longer encached	no longer encached, written to memory if dirty	no longer encached, written to memory if dirty
read_prefetch	not encached in hypernode	global	not encached	encached, shared	may be encached
write_prefetch	not encached in hypernode	global	not encached	encached, exclusive	not encached

The definition of the hardware interface routines is as follows:

```
void cache_flush(virt_addr);
void *virt_addr;

void read_prefetch(virt_addr);
void *virt_addr

void write_prefetch(virt_addr);
void *virt_addr;
```

`Cache_flush` is a CTI-wide cache line flush; after its execution the referenced cache line is not encached anywhere in the system until a subsequent reference (to the line or another line on the same page). `Read_prefetch` accelerates a cache line into the CTI cache; it can be shared with other hypernodes. `Write_prefetch` accelerates a cache line into the CTI cache; it will be exclusively owned.

Note

`Read_prefetch` and `write_prefetch` may flush/purge the data from the issuing CPU and all other CPUs within the same hypernode if it is present in a CPU cache. This action is implementation-dependent. Issuing a prefetch on data already within the CPU cache may result in a performance loss.

Virtual memory software models

This section discusses the virtual memory model provided to applications by the operating system. In reality, two models are provided: a model that runs PA-RISC ABI- (application binary interface) compliant programs and a Convex extended model that provides larger virtual addressing capabilities.

PA-RISC ABI compliant programs

Each PA-RISC ABI compliant application has its own short address space, consisting of four quadrants totalling 2^{32} bytes. Short pointers are used exclusively within applications.

The address space is laid out as shown in Figure 32.

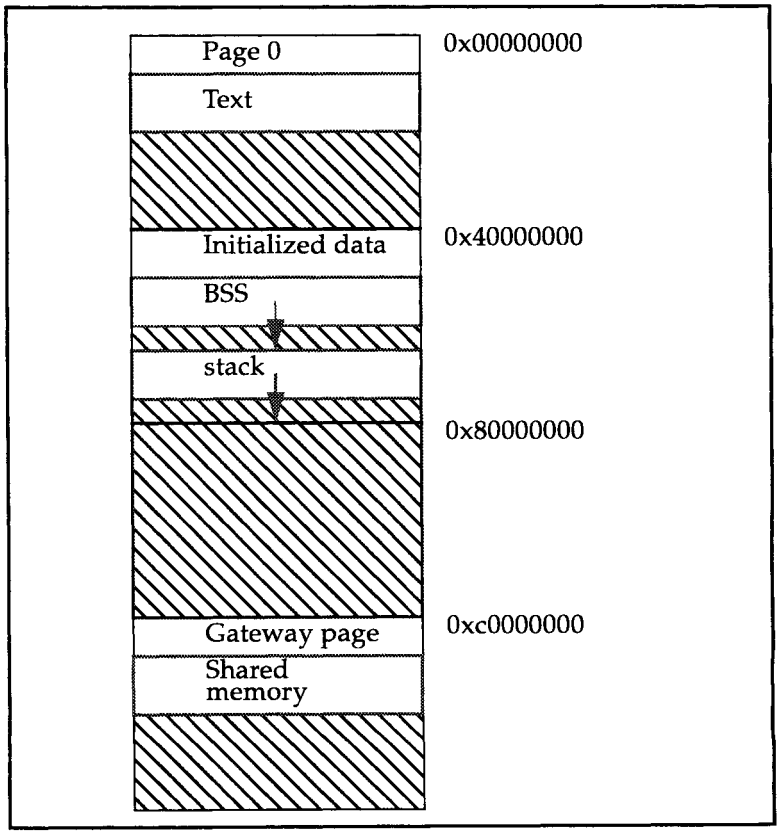

Figure 32 Virtual memory of a PA-RISC ABI-compliant application

Each quadrant in the virtual address space potentially points to a different space. By using a different space for text vs. data and bss, the text region can be shared among multiple processes without

replicating the text data in the instruction cache. The process private data is restricted to the second quadrant; this points to a space uniquely allocated for each process. Shared memory is allocated in the fourth quadrant because implementation is based on System V shared memory semantics that typically have pools of shared memory buffers in kernel address space.

Note

The stack and bss regions both grow toward higher-numbered addresses.

The PA-RISC ABI does not currently support parallel programs. Thus, all memory used by the application may be thought of as CPU- or hypernode-private, but the operating system may substitute other near- or far-shared memory if required for load balancing, performance, or resource limitation reasons.

This approach has a limitation in that an application can only use 1 Gbyte of dynamic data, including its initialized data, bss and stack regions. Additionally, there is no way to specify memory allocation from each of the SPP memory hierarchy levels. These limitations lead to the development of a Convex SPP-specific memory model that allows greater exploitation of the memory hierarchy.

SPP applications

SPP application virtual memory model allows exploitation of the memory hierarchy provided by the SPP. This model is based on a different set of assumptions from those of the PA-RISC ABI.

- It is unlikely that text will be shared between multiple processes running on the same CPU. Even if it is shared, the scheduling granularity is large enough that reloading the instruction cache is insignificant when rescheduling different processes.
- It is highly desirable to allow programs with > 1 Gbyte of dynamic storage.
- It is desirable to provide access to CPU-private, hypernode-private, near-shared and far-shared memory models.

The proposed memory layout, based on the above requirements, is shown below. The virtual address space is divided into 16-Mbyte segments; the virtual space within a segment is used only for a single type of memory allocation, such as CPU-private, hypernode-private, near-shared, or far-shared. This allows the operating system to employ block TLB descriptors and

interleaving for various 16-Mbyte regions within the program at its discretion.

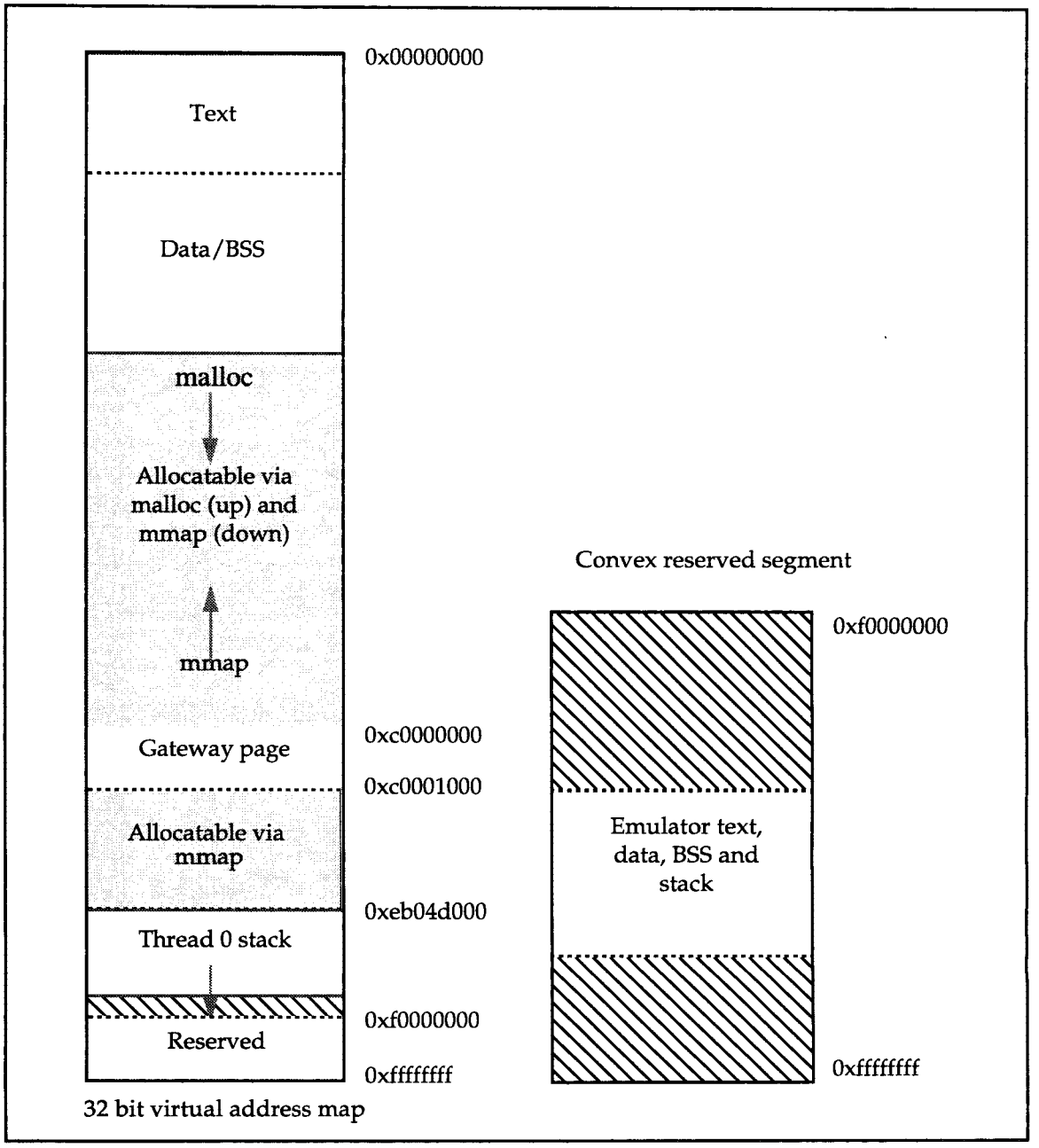

Figure 33 Virtual memory of a Convex SPP application

The first page of the 32-bit virtual address space is reserved for page 0 (which is usually unmapped to trap pointer problems), but may be used for text if desired.

Text begins at 0x00001000 and may extend for an arbitrary length. At least one unused page must be present between text and any subsequent memory allocation since text may accelerate the page that follows the page containing the current instruction into the instruction cache. (Note: page size is CPU implementation-dependent.)

The system gateway page (used for system call traps) is placed at virtual address 0xc0000000 for compatibility with HP/UX library code, which assumes that location. This creates a hard boundary on the amount of space that can be acquired through `malloc()` at 3 Gbytes, minus the size of text + data + BSS. Files mapped in via `mmap()` will be mapped starting below the thread stack region and will extend downward, consuming space below the gateway page, if necessary.

The last 256-Mbyte segment of the 3-bit virtual address (starting at 0xf0000000) is architecturally reserved, with the only identified use being the operating system emulator. The stack starts approximately 83.5 Mbytes below that boundary (for compatibility with HP/UX) and grows upward (via page faults). The stack may not cross into the reserved area. Any attempt to access a page in the reserved area may cause a SEGMENTATION FAULT or other unspecified program behavior.

Note

Programs should not have hard-coded (static) references to their stack origin.

The area between the text and gateway page is allocated by the operating system to any type of memory in the hierarchy: CPU-private, hypernode-private, near-shared, or far-shared.

The various types of memory can be spread throughout the address space of an application with page-level granularity, but the physical memory must be allocated with 16-Mbyte granularity.

Although the operating system maintains a segmented address space, the notion of segments is not exported to the compiler or application programmer. The programmer views the virtual address space as a flat 4-Gbyte range, using short pointers.

It is expected that more detailed software conventions will be defined regarding placement of different memory allocations within the virtual space. The operating system should make this configurable; the architecture allows an arbitrary mix of

hypernode-private *vs.* global memory segments to facilitate different programming models. A strict message-passing programming model would employ all local memory, whereas a shared memory program might use a minimal amount of hypernode-private memory and large amounts of near and/or far-shared memory.

Note

The allocation segments provided to a Convex SPP application are not related to the spaces defined in the PA-RISC architecture.

Expanding the address space past four Gbytes

Exemplar will support only those applications with a 32-bit virtual address map per thread or CPU. This restriction occurs because only 32-bit pointers are implemented.

Applications that are dependent upon a virtual address space larger than 4 Gbytes will be referred to as large applications. The general format of their address layout is shown in Figure 34.

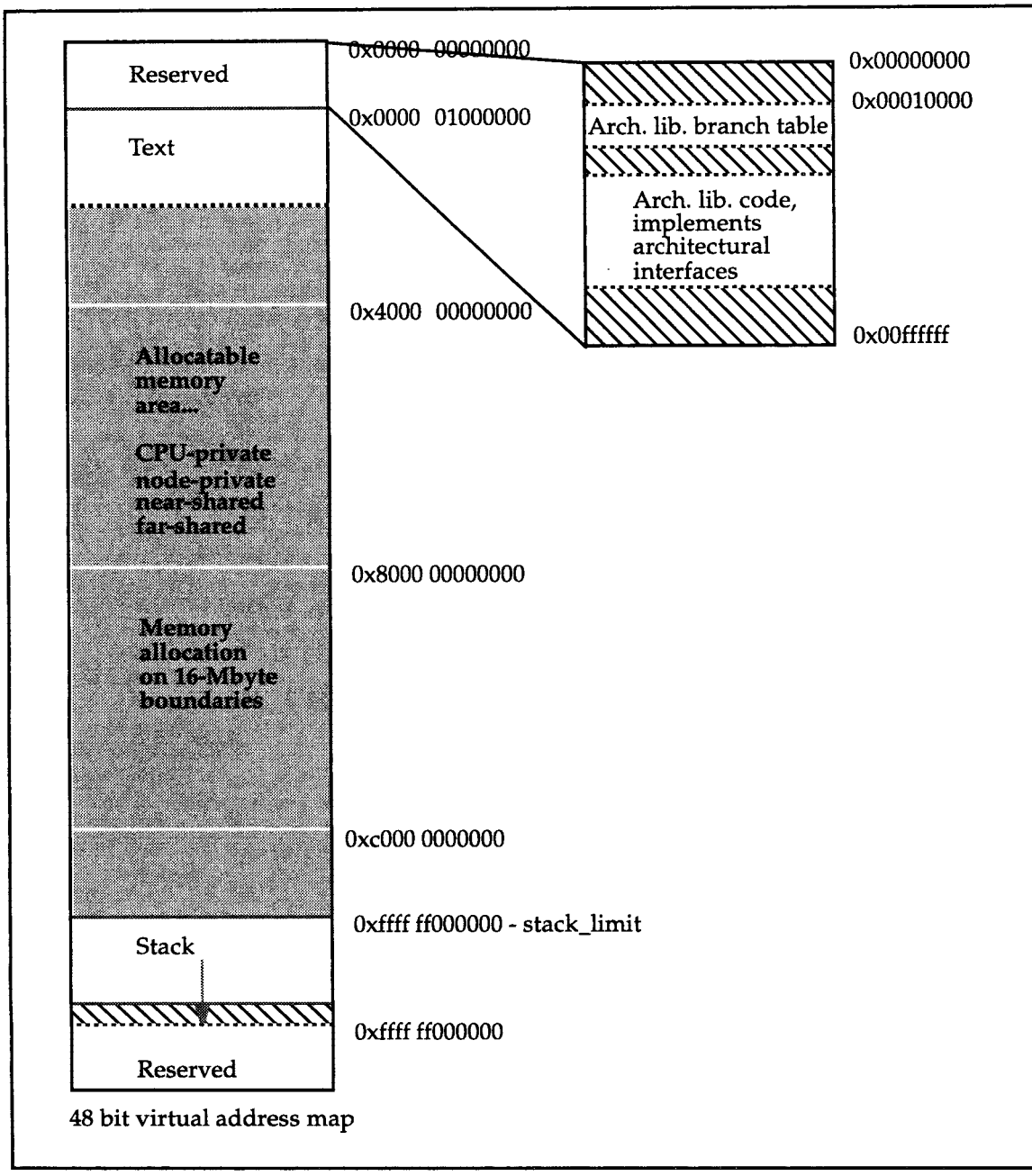

Figure 34 Virtual memory of a Convex SPP large application

This diagram illustrates the layout for a 48-bit virtual address space, but the architecture does not specifically define the number of bits of virtual address that must be accessible by an application.

The memory model for large applications is similar to the model for regular SPP applications. The library branch table and text origin are at the same address in both models. Also note that the stack and upper-reserved area have been moved to larger addresses, allowing for more allocatable memory in the area between text and stack. The 16-Mbyte allocation granularity for various types of memory is retained, as explained above.

Architectural interface library (AIL)

The architectural interface library (AIL) is a set of subroutines that manipulate the hardware on behalf of a user program. Operations are typically implemented in the library rather than directly by in-line instructions because the instruction sequence used to implement the operation is implementation-dependent.

The table below lists all the functions in the architectural interface library. Each function is described in Chapter 6. All functions follow the standard C entry, exit, and argument-passing conventions.

Table 13 Architectural interface library entry points

Entry point	Description
int fetch_and_clear32 (int *var)	Fetch and clear 32-bit value
int fetch_and_inc32 (int *var)	Fetch and increment 32-bit value
int fetch_and_dec32 (int *var)	Fetch and decrement 32-bit value
int fetch32 (int *var)	Fetch (noncoherently) 32-bit value
void barrier_sync (u_int *semaphore, u_int sema_init, u_int *release, u_int release_value)	Barrier synchronization
void cache_flush(void *addr)	Internode cache flush
void read_prefetch(void *addr)	Read prefetch into CTI cache
void write_prefetch(void *addr)	Write prefetch into CTI cache

Exemplar implementation specific information

This section discusses information specific to the Exemplar implementation.

SPP1000 CPU specific parameters and constants

Table 14 Exemplar implementation-specific parameters

Parameter	Value
PA RISC architecture	Level 1 (48-bit virtual address. Virtual addresses consist of a 16-bit space ID and 32-bit offset.)
Physical memory	32 bits (up to 4 Gbytes) are exported from the PA7100 CPU. Up to 4 Gbytes of physical memory and address space may be addressed by any single hypernode.
Data cache	1-Mbyte, 64-bit access, direct-mapped, virtually indexed by a hashed address, 32-byte cache line size.
Instruction cache	1-Mbyte, 64-bit access, direct-mapped, virtually indexed by a hashed address, 32-byte cache line size.
Page size	4096 bytes.
TLB	Unified instruction/data TLB, consisting of 120 entries, each mapping one page, plus 16 block entries each mapping 512 Kbytes - 64 Mbytes. Hardware TLB refill for page entries.
Memory size	Up to 4 Gbytes of physical memory may be provided per hypernode. This memory includes CPU-private, hypernode-private, global (shared) memory and the CTI cache. The size of memory devoted to the CTI cache is runtime configurable.

SPP1200 CPU specific parameters and constants

Table 15 Exemplar implementation-specific parameters

Parameter	Value
PA RISC architecture	Level 1 (48-bit virtual address. Virtual addresses consist of a 16-bit space ID and 32-bit offset.)
Physical memory	32 bits (up to 4 Gbytes) are exported from the PA7200 CPU. These address lines are expanded to 40 bits, but only 32 bits are used in the SPP1200. Up to 4 Gbytes of physical memory and address space may be addressed by any single hypernode.
Data cache	256-Kbyte, 64-bit access, direct-mapped, virtually indexed by a hashed address, 32-byte cache line size.
On-chip data cache	Sixty-four line, fully-associative "prefetch-miss" (PM) data cache.
Instruction cache	256-Kbyte, 64-bit access, direct-mapped, virtually indexed by a hashed address, 32-byte cache line size.
Page size	4096 bytes.
TLB	Unified instruction/data TLB, consisting of 120 entries, each mapping one page, plus 16 block entries each mapping 512 Kbytes - 64 Mbytes. Hardware TLB refill for page entries.
Memory size	Up to 4 Gbytes of physical memory may be provided per hypernode. This memory includes CPU-private, hypernode-private, global (shared) memory and the CTI cache. The size of memory devoted to the CTI cache is runtime configurable.

SPP1600 CPU specific parameters and constants

Table 16 Exemplar implementation-specific parameters

Parameter	Value
PA RISC architecture	Level 1 (48-bit virtual address. Virtual addresses consist of a 16-bit space ID and 32-bit offset.)
Physical memory	32 bits (up to 4 Gbytes) are exported from the PA7200 CPU. These address lines are expanded to 40 bits, but only 32 bits are used in the SPP1600. Up to 4 Gbytes of physical memory and address space may be addressed by any single hypernode.
Data cache	1-Mbyte, 64-bit access, direct-mapped, virtually indexed by a hashed address, 32-byte cache line size.
On-chip data cache	Sixty-four line, fully-associative "prefetch-miss" (PM) data cache.
Instruction cache	1-Mbyte, 64-bit access, direct-mapped, virtually indexed by a hashed address, 32-byte cache line size.
Page size	4096 bytes.
TLB	Unified instruction/data TLB, consisting of 120 entries, each mapping one page, plus 16 block entries each mapping 512 Kbytes - 64 Mbytes. Hardware TLB refill for page entries.
Memory size	Up to 4 Gbytes of physical memory may be provided per hypernode. This memory includes CPU-private, hypernode-private, global (shared) memory and the CTI cache. The size of memory devoted to the CTI cache is runtime configurable.

TLB management

The Exemplar TLB consists of 120 page entries and block entries. The page and block entries are initialized and replaced via the different mechanisms described below. Additionally, a hardware refill mechanism is provided for the page entries.

Each page entry stores and compares a 36-bit virtual page number (VPN) that identifies a single 4-Kbyte page. For SPP1000, each TLB contains the 20-bit physical page number, a 15-bit access ID, a 7-bit access rights field and the E, T, D and B flag bits. For SPP1200 and SPP1600, each TLB contains the 20-bit physical page number, an 18-bit access ID, a 7-bit access rights field and the E, T, D and B flag bits. Each entry can be individually locked out (that is, its hit comparator is disabled) or locked in (that is, excluded from replacement) by using diagnose instructions.

Insertion of page TLB entries can be accomplished with the IITLBA, IDTLBA, IITLBP, and IDTLBP instructions. Page TLB entries may also be inserted by the hardware refill mechanism described below. Block TLB entries are inserted via diagnose instructions, also described below.

The target replacement page entry for the IITLBA and IDTLBA instructions are selected by a hardware algorithm. Highest priority is given to any entry whose VPN field matches the VPN being inserted, in order to avoid multiple mappings of the same virtual page. If no matching VPN is found, the lowest-numbered invalid entry (E=0) is selected. If no entry with a matching VPN is found and there are no invalid entries, a pseudo random entry is chosen.

It is possible to circumvent the hardware algorithm selecting the replacement page entry and specify directly the entry to be replaced during the next insertion. This function is known as diagnostic insertion.

In addition to the architected bits, each TLB entry contains a lock-in bit. If the lock-in bit is set, the entry will be selected for replacement only if its VPN matches the VPN being inserted, or if the entry is selected for replacement by diagnostic insertion.

A diagnose control register is defined for the TLB (DR8.) This is a 16-bit register loaded by the MOVE TO DIAGNOSE instruction. This register is write-only; reading it returns undefined data. The bits in the TLB diagnose register are defined in Table 17.

Table 17 TLB diagnose register (DR8[16:31])

Bit	Description	Disabled value
16	Force VPN Mismatch. If this bit and Inhibit replacement (bit 17) are both set, the TLB entry is locked out (effectively disabled.)	0
17	Inhibit replacement; sets the lock-in bit of the entry. Set this to one for Lock-in or Lock-out.	0
18-24	Replacement (page) entry pointer; in the range 0:119, indexes the target for diagnostic inserts	>= 120
25	Replacement override bit; when set all IxTLBA instructions are interpreted as diagnostic inserts	0
26	Reserved; this bit must be zero.	0
27-30	Block entry select; indexes the block entry targeted for subsequent block diagnostic insertions.	not used
31	Block override bit; directs a subsequent diagnostic insert to the block entry specified by bits 27-30.	0

In general, the TLB diagnose register affects subsequent diagnostic insertions. This means that the register is set with the mode desired (inhibit replacement, for example), one or more diagnostic insertions are made that have the lock-in bit set as a consequence of the register setting and the register is restored to normal operating mode. Subsequent insertions are normal.

At system power-on, all 120 page entries and all 16 block entries are locked out (that is, they have forced VPN mismatch and the lock-in bit is set). Initialization code is required to clear all of the lock-out bits and load a unique VPN in each entry.

A diagnostic insertion may be accomplished by explicitly designating an entry in the replacement entry pointer, setting the

replacement override bit (with the block override bit zero) and performing the insertion. To turn off diagnostic insertion, the replacement override bit should be set to zero and the replacement entry pointer set to a value greater than 119.

Block TLB entries

Block TLB insertion uses a combination of diagnose write functions and architected insert instructions. The block entries do not respond to the architected insert address and protection instructions, nor do they respond to purge, purge entry, or broadcast purge, unless DR8 has been previously loaded with a certain set of values.

Use the following algorithm to insert block entry N (N=0..15):

1. Load the replacement pointer with a value in the range of 120..127.
2. Set the replacement override bit to 1.
3. Set the block override bit to 1.
4. Set the block entry select field to point to the entry slot N to be inserted.
5. Assemble a virtual address for any page within the block to be mapped by concatenating the space and most significant 20 bits of offset.
6. Modify that virtual address by overwriting the seven least significant bits of the offset [13:19] with the block size specifier given by Table 18.
7. Assemble the physical page number in a general register in bits [7:26].
8. Modify that physical page number by overwriting the seven least significant bits of the offset [13:19] with the block size specifier given by Table 18.
9. Execute an insert address instruction using the virtual address and physical page number assembled (as modified) above.
10. Execute an insert protection instruction.
11. Reset the diagnose register to its original (disabled) state.

Table 18 Block TLB size specifier

Block size specifier mask (binary)	Vpn bits compared	Physical page bits returned	Block TLB size (pages)
0000000	0-5	0-5	16,384

Table 18 Block TLB size specifier—**(continued)**

Block size specifier mask (binary)	Vpn bits compared	Physical page bits returned	Block TLB size (pages)
1000000	0-6	0-6	8,192
1100000	0-7	0-7	4,096
1110000	0-8	0-8	2,048
1111000	0-9	0-9	1,024
1111100	0-10	0-10	512
1111110	0-11	0-11	256
1111111	0-12	0-12	128

Note

Block entries do not respond to PxTLB or PxTLBE instructions. This also implies that block TLBs are unaffected by PxTLB instructions on other processors.

A block entry may be purged in a similar manner. When both the block and replacement override bits are set, the block entry selected in DR8 becomes the target for subsequent purge instructions. Block entries must have both protection and address inserted using the block entries selected in DR8. Block entries do not respond to purge entry instructions.

Probe and LPA instructions are essentially TLB-translation accesses and thus function in the same manner for both block and page entries.

Hardware TLB refill and the PDIR table

The hardware TLB refill handler may be invoked for either instruction or data translations not present in the on-chip TLB. The handler computes the address of a PDIR (Physical page Directory) entry based on the missing space and vpn. It then accesses the PDIR entry and checks three things:

1. The PDIR must have a valid tag (V bit must be 1),
2. The PDIR must have a matching tag (one that matches the missing space and offset vpn) and
3. The reference bit (R) in the PDIR must be 1.

If these checks pass, the RPN (physical page number) and protection of the PDIR are inserted into the TLB and the original access is retranslated. If any of the checks fail, the handler will not insert the PDIR entry and the instruction will trap to software.

The address of the PDIR entry is generated by computing a hashed offset from the referenced space and vpn, masking the offset and adding the masked offset to a base address. The starting base address of the PDIR table is stored in diagnose register DR24. The mask in diagnose register DR25 is masked with the offset to determine the size of the table. The table must start on a boundary that is a multiple of its size. The address computation is performed with the following algorithm:

```
int dr24;          /* diagnose register 24 -
contains base */
int dr25;          /* contains mask bits based
 on table size */
int tmp1, tmp2;    /* temporaries */
int result;        /* result used to
access PDIR entry */

tmp2 = (off >> 8) & 0xfffff0;/* vpn in 8:27*/
tmp1 = (spc << 9);/* position space
 at 7:22 */
tmp2 = tmp1 ^ tmp2;/* perform xor hash */
tmp2 &= dr25 | 0xfff;/* mask bits 0:19 */

result = (dr24 & 0xfffff000) | tmp2;/* or in
base 0:19 */
```

For an inverted page table, the PDIR table contains the first level entries of the inverted page table. For a forward-mapped page table, the PDIR table contains a cache of entries that are distinct from the actual page tables.

Each PDIR entry takes 4 words (16 bytes). A cache line can hold two PDIR entries. The format of each entry for the SPP1000 is shown in Figure 35 and in Figure 36 for SPP1200 and SPP1600.

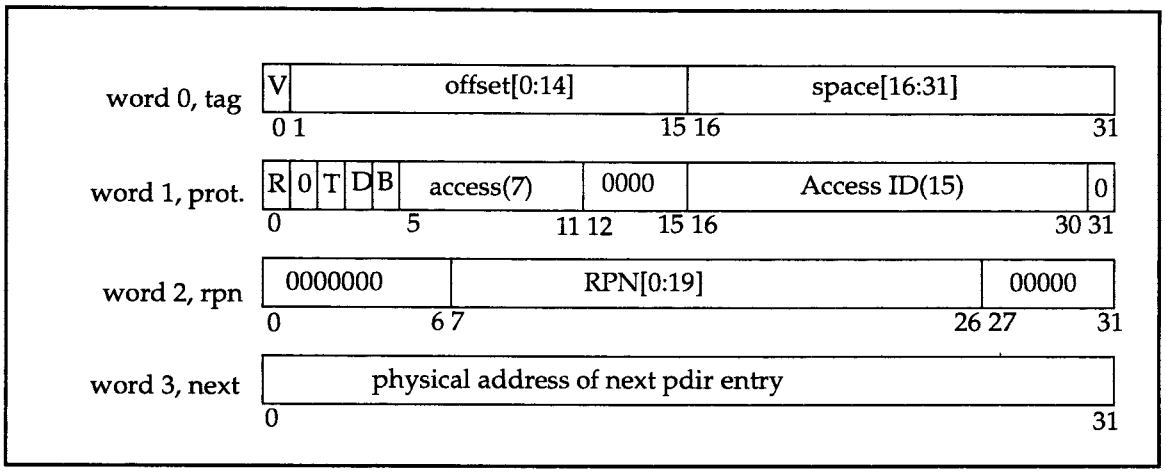

Figure 35 SPP1000 PDIR structure

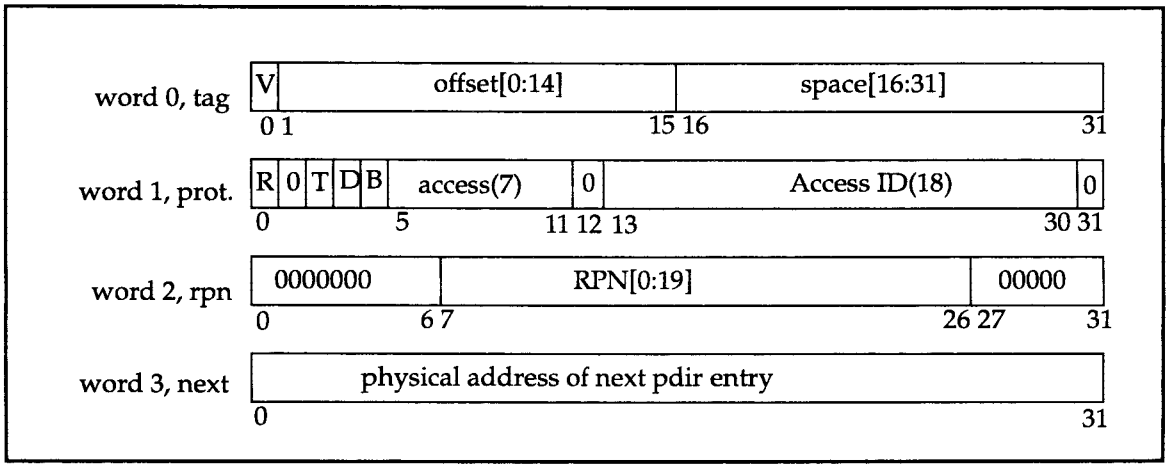

Figure 36 SPP1200 and SPP1600 PDIR structure

The bit fields are defined as follows: V indicates the tag is valid and must be 1 for a valid entry. R indicates the entry is referenced and must be 1 for a valid entry. T, D and B are used to initialize the corresponding bits in the TLB entry.

The field marked access(7) above initializes the access rights field (7) in the TLB. The other fields initialize the corresponding field in the TLB. The physical address of the next PDIR entry field is

passed, under certain circumstances, to a software TLB handler in CR28 if the hardware TLB refill mechanism could not service the miss.

Offset bits 15-19 are implicit in the address of the PDIR entry and are not stored in the tag.

After the insert to the on-chip TLB is done, the access with the TLB miss is retranslated and any TLB resulting traps that occur will cause a software DTLB_MISS_FAULT. In this case, CR28 is used to pass information to the trap handler about the address of the PDIR. The contents of CR28 are set according to the following algorithm, depending on bit 29 of diagnose register 25:

```
int tag_match();/* returns true if pdir tag
                          matches reference */

if (!pdir.R || !pdir.V) CR28 = &pdir;
else if (!tag_match() & DR25[29] == 0)
CR28=&pdir;
else if (!tag_match() & DR25[29] == 1)

CR28=pdir.next_pdir_entry;
```

The diagnose register formats for the hardware TLB miss mechanism are shown in Figure 37.

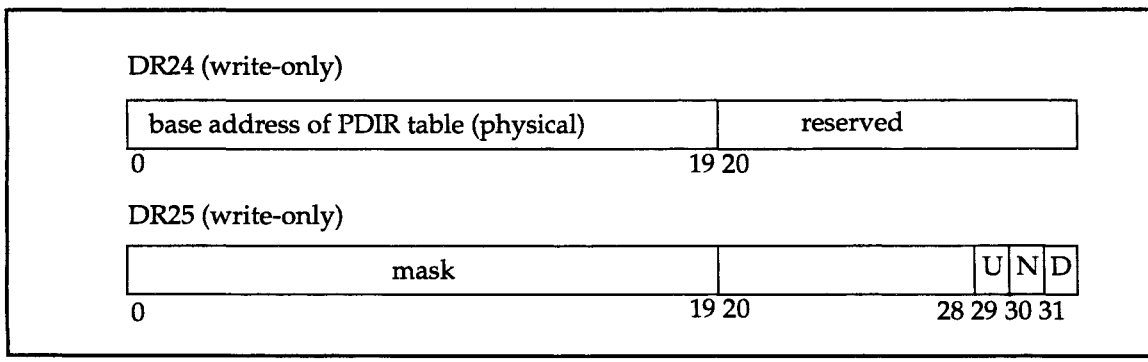

Figure 37 Diagnose registers for hardware TLB refill (DR24 & DR25)

The base address of the PDIR table must be aligned to a boundary that is a multiple of the size of the table. This allows the base address bits to be inclusive in the calculation, rather than requiring an addition.

The mask sets the size of the hardware visible table. Each zero bit in the mask selects the corresponding base address bit to be used in the address computation. Each one bit in the mask selects the

corresponding bit from the hash computation of VPN and space to be used in the address computation.

If the D (disable) bit in DR25 is one, the hardware TLB refill mechanism is disabled. If zero, the mechanism is enabled. Its state is undetermined at power-up.

The U (update CR28) bit in DR25 controls the updating of CR28 with the next pointer from the PDIR in the event of a tag mismatch (see algorithm on the previous page). A one selects the pdir.next_pdir_entry field; zero selects the address of the initial pdir entry.

The N bit should be set to zero.

The mask should be set according to the table below. The PDIR table size can be any power of two between 4096 bytes (256 entries) and 4 Gbytes (256M entries.)

Table 19 DR25 mask values for various PDIR table sizes

Table size	No. entries	DR25 mask (0:19) (binary)
4 Kbytes	256	0000 0000 0000 0000 0000
8 Kbytes	512	0000 0000 0000 0000 0001
16 Kbytes	1024	0000 0000 0000 0000 0011
32 Kbytes	2048	0000 0000 0000 0000 0111
64 Kbytes	4096	0000 0000 0000 0000 1111
4 Gbytes	268,435,456	1111 1111 1111 1111 1111

Fast TLB insert instructions

The Exemplar products implement four new TLB insert instructions. These instructions insert to the virtual address in the ISR/IOR (interruption space register and interruption offset register) for DTLB inserts and the IIASQ/IIAOQ for ITLB inserts. The instructions incur fewer penalty cycles than the normal TLB insert instructions.

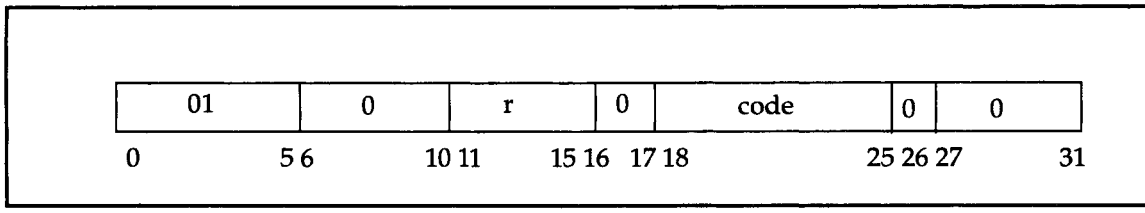

Figure 38 Fast TLB insert instructions

In the above format, code 10 is a fast-insert ITLB protection instruction (`iitlbpf`), code 11 a fast-insert ITLB address instruction (`iitlbaf`), code 50 a fast-insert DTLB protection instruction (`idtlbpf`) and code 51 a fast-insert DTLB address instruction (`idtlbaf`).

The protection/Real address is specified by the r field for insert protection/address, respectively. This instruction is privileged.

Caution

Do not use these instructions without adhering to the following restrictions:

Software must avoid any data accesses (loads, stores, or other memory management instructions) for two instructions following an idtlbpf.

The instruction immediately following an idtlbpf must not be an RFI.

Every idtlbpf should be preceded by an idtlbaf. There must be no traps taken, or other TLB inserts, between the idtlbaf/idtlbpf pair.

Every iitlbpf should be preceded by an iitlbaf. There must be no traps taken, or other TLB inserts, between the iitlbaf/iitlbpf pair.

Processor cache management

The Convex Exemplar architecture is based on globally shared memory, meaning that memory is physically distributed among the hypernodes of the system complex. User applications can access memory in any hypernode, but system performance would suffer were it not for Exemplar's unique instruction and data cache hardware implementation.

As with most RISC processor based systems, the Exemplar communicates with memory by only two means: a load instruction and a store instruction. Before a processor can use a data object in memory, the object must first be encached.

The Exemplar GSM is fully cache coherent, achieved totally by hardware; there is no software involved in cache maintenance, except for the cache synchronization instruction for weak-ordered mode.

Cache coherence implies that all processor loads and stores are seen by all other processors in the system. Exemplar uses a distributed cache directory to guarantee this. This directory exists in the cache tag bits associated with each cache line. Included in the tag bits are several bits that indicate the state of the cache line. The SPP1000 and SPP1200 use a three-state cache coherency mechanism, and the SPP1600 a four-state mechanism that greatly improves the performance of the system.

Three-state cache coherency mechanism

The SPP1000 and the SPP1200 use a three-state coherency mechanism to maintain hardware cache coherency. As stated above, the PA-RISC processor has only two ways to move instructions and data to and from memory: the load and store instructions.

Before a processor can use a data object (assuming the object is not currently encached), it must copy it from memory into the cache using an load instruction. If the address is not resident in the cache, it is called a cache miss. If a processor issues a store instruction to an address that is not encached, it must first load it from memory into the cache. Only after the address is cache resident can the actual store take place. Since the memory of the system takes longer to access than the cache, cache misses for both loads and stores should be minimized.

The three cache states used in the SPP1000 and SPP1200 cache coherency scheme are:

- ***Invalid***— The cache line location is *empty*. The data for this line in the cache is invalid.

- *Shared*—The cache line is valid for read access only. This cache line may or may not be shared by other processors in the system.
- *Private Dirty*—The cache line is valid for both read and write access. This line is not shared by any other processor; only one processor can have a cache line in the Private state. A Private Dirty line is one that has been modified by the processor and differs from the line in memory. It is the only up-to-date copy of the line in the system. With the Private Dirty state, the processor is responsible for writing the line back to memory if the line is flushed or castout by the processor.

At system boot time, the processor initializes all of its cache lines to the Invalid state. As the processor performs loads and stores to memory, it copies cache lines into its cache in either the Shared or Private Dirty state. If the processor executes a load operation, it tags the cache line as Shared. Likewise, a store operation encaches the line as Private Dirty. A cache hit event is independent of the cache line state.

A cache miss can occur to a line that is in the Shared or Private Dirty state when the desired line is not encached and another line, mapping to the same cache entry, is encached. In this case, the processor loads the desired line from memory, and one of two things occurs to the cache line currently occupying the data cache entry:

- If the current line is in the Shared state, the processor invalidates it, because it is not possible that the line could have been altered since Shared is a read-only state.
- If the current line is in the Private Dirty state, the processor must write the line back to memory.

  Updating memory in this manner is commonly called a copyout, writeback or castout event.

A cache miss event can occur, even if the desired line is currently encached, when the processor executes a store operation to a line that is encached in the Shared state. The processor does not have write permission to the line and must issue a new request to memory in order to obtain write permission. When this happens the line is recopied into the cache and marked Private Dirty. Clearly, the requisite operations for this cache miss event requires two memory transactions in order to complete the store operation.

Four-state cache coherency mechanism

The SPP1600 uses a four-state cohrency mechanism. This implementation adds a fourth state, *Private Clean*, to the cache line tag bits.

The Private Clean state logically fits between the Shared and Private Dirty states. Like the Private Dirty state, a cache line in the Private Clean state is valid for both read and write accesses, and like the Shared state, the processor has *not* modified it. Since a processor has not modified the cache line in the Private Clean state, it does *not* have the responsibility of writing the line back to memory.

If a cache line is tagged with the Private Clean state and the processor executes a store operation to that line, a cache hit event occurs where the state changes to Private Dirty without having to go to memory.

Like the Private Dirty state, only one processor in the system can encache a line in the Private Clean state at a time. Otherwise, there would not be a consistent view of memory.

There are two ways a cache line may be tagged as Private Clean. The first occurs when a processor issues a load operation to a cache line that is not encached by any other processor in the system.

If a line is already encached shared, it is returned to the processor in the Shared state. If the line is encached Private Dirty, the snoop returns data Private CLean. If a CTI transaction is required to service a memory request, the line is always tagged as Shared, regardless of whether the line is encached by other processors or not, because the CTI coherency scheme is based on the three-state protocol. Four-state coherency applies only within a physical hypernode, regardless of the subcomplex.

The second way a cache line may be tagged as Private Clean is by a store prefetch operation where a processor automatically begins loading or storing the data before it knows if it is necessary.

The three-state protocol allows for load prefetching only; the four-state design allows for load and *store* prefetching.

For most cases, a store prefetch looks just like a store miss to the rest of the system, except to the processor involved. The processor marks a cache line as Private Clean on a store prefetch operation instead of Private Dirty, as in the store miss operation. All store prefetch data cache lines return to the processor as Private Clean, even when a CTI transaction is required to service the request, because the rest of the system treats the store prefetch as a regular store miss operation. Only the local coherency domain is

aware of the Private Clean state. The CTI coherency domain does not recognize the Private Clean state, because it operates with the three-state protocol.

Performance benefits

The four-state mechanism improves performance by reducing the number of data cache misses. Two common cases where a four-state cache improves performance over a three-state cache are read-modify-write code sequences and codes or applications with high store content.

The four-state cache can possibly increase performance through its indirect effect on instruction scheduling. Since storing to a line that is in the Private Clean state is not a store miss event, the instruction pipeline will not stall and performance is increased.

Cache flushing, purging, and prefetch

This section discusses the Exemplar specific implementation of the AIL functions `cache_flush()`, `read_prefetch()` and `write_prefetch()` for Exemplar.

These functions are implemented using instructions intended for graphics operations. These instructions are *not* part of the PA-RISC architecture and are not privileged instructions.

The instructions are known as graphics flush write, graphics flush read and graphics flush read/write.

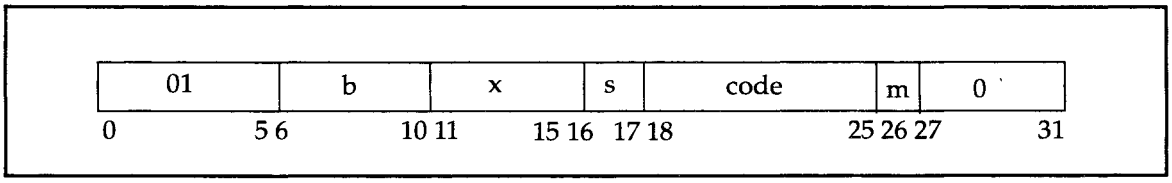

Figure 39 Graphics instruction format

Functionally, the instructions are similar to the flush data cache (FDC) instruction, and the code determines which instruction. FDC is code 4a, graphics flush write is code 5a, graphics flush read is code 6a and graphics flush read/write is code 7a. The operation of these instructions is defined as follows:

```
if (s==0)space = SR[GR[b]{0..1} + 4];
elsespace = SR[s];
if (m == 1) {
offset = GR[b];
GR[b] = GR[b] + GR[x];
}
switch(code) {
case 5a:check_TLB_write_access(); break;
case 6a:check_TLB_read_access(); break;
case 7a:check_TLB_read/write_access();
 break;
}
flush_data_cache(space,offset);
```

Possible exceptions are:

- Data TLB miss fault/data page fault
- Data memory protection trap
- Page reference trap
- Data memory break trap (for codes 5a and 7a only)
- TLB Dirty bit trap (for codes 5a and 7a only)

The graphics instructions are useful in implementing the AIL functions because they perform a virtual-to-physical address translation and memory protection check, and generate a bus transaction with an identifiable opcode that the hardware can decode and use to implement operations.

`Cacheflush()` is implemented with graphics read/write. This operation causes a CTI-wide cache flush to occur. Both read and write access permission are required.

`Readprefetch()` uses the graphics read instruction and causes a read prefetch of the specified cache line to occur to the CTI cache. Note that if the line is already in a CPU cache on the same hypernode, it will be flushed back to the hypernode cache. Read access permission is required.

`Writeprefetch()` uses the graphics write instruction and causes a write prefetch (exclusively owned) in the CTI cache. Note that if the line is already in a CPU on the same hypernode, it will be flushed back to the hypernode cache. Write access permission is required.

Noncoherent read prefetch

A noncoherent prefetch operation causes data to be accelerated into the CTI cache noncoherently. Its primary use is to move a block of data across the CTI. The destination processor for the move will noncoherently prefetch the data into a local CTI cache, copy the data from the CTI cache to local memory and then purge the data from the CTI cache. The noncoherent operation is used for block data moves to eliminate the cache coherency overhead.

The noncoherent prefetch is issued by setting a bit (NON_COHERENT_PREFETCH) in the exception context register (EXC_CONTEXT), then issuing a Flush data cache (FDC) instruction with hint bits set for a graphics read.

Copying of the data is performed using load and store instructions. The data prefetched into the CTI cache may be forced out of the cache by other CTI cache activity. If the data is not in the CTI cache when the data copy is performed, the data is re-accessed coherently across the CTI.

Once the data has been copied, the prefetched data within the caches must be purged. The Purge data cache instruction (PDC) is used to purge the data from the cache of the issuing processor and the CTI cache.

Note

The architecture does not preclude further overloading of the graphics instructions for other uses. This might be accomplished with additional mode bits or other state control per CPU. For Exemplar, the architecture must ensure a software-defined machine state in that the instructions behave as described in this section.

CTI cache maintenance

A flush of the entire CTI cache is accomplished by executing an implementation-dependent loop containing FDC instructions that reference the physical addresses implementing the cache. A specially designated address must be used so that the CTI cache control logic will flush the line, regardless of whether the physical page number and virtual index tags match the reference made by the CPU.

This implies that the physical addresses devoted to the CTI cache must be addressable by the CPU, even though in normal circumstances load or store instructions are not issued on these addresses.

Store ordering

Exemplar provides a mode bit (STRONG_ORDER) in the exception context register (EXC_CONTEXT) to force strong ordering.

Availability implications

The availability implications of the virtual memory architecture for Exemplar are:

- The four protection levels provided by the architecture should be employed to implement a hierarchical level of reliability, with the microkernel running at the highest protection level, other operating system servers running at intermediate protection levels and user code running at the lowest protection level, for example. Using the protection scheme in this way should provide isolation between disjointed pieces of the operating system.
- The use of global memory (near- or far-shared) for operating system data structures is generally ill-advised. It is anticipated that the operating system data structures for managing global memory will be in global memory (in memory free lists, for example).

In particular, any interleaved memory is subject to failure if any of the hypernodes of which it is composed fails. Ideally, interleaved memory should be used only for applications that can be restarted from recoverable checkpoints or I/O buffers. Performance implications may negate this suggestion under other circumstances.

Messaging 5

Overview

This chapter describes the messaging mechanisms provided by the SPP, including the low-level programming interface and the underlying hardware support for those mechanisms within the Exemplar implementation.

SPP architecture

The SPP architecture supports message passing from one thread within a process to another within the same or a different process. Messages are sent using dedicated hardware and shared memory copying.

Exemplar implementation

This section describes the compliance of the Exemplar implementation to the SPP architecture. The hardware interface for messaging and block data movement operations are also described.

Exemplar compliance

Exemplar supports messaging in hardware for messages of a fixed, 64-byte length. A message of greater than 64 bytes can be sent either as multiple 64-byte messages, or as a single 64-byte message containing the address of a shared memory region which contains the remainder of the message.

Exemplar messaging hardware overview

With Exemplar, a message can be sent from any processor in the system to any other processor. Figure 40 shows a high-level diagram of a four hypernode Exemplar system. A hypernode consists of four pairs of processors connected to a five-ported cross bar switch (four ports are used for processors). Hypernodes are interconnected using four CTI rings.

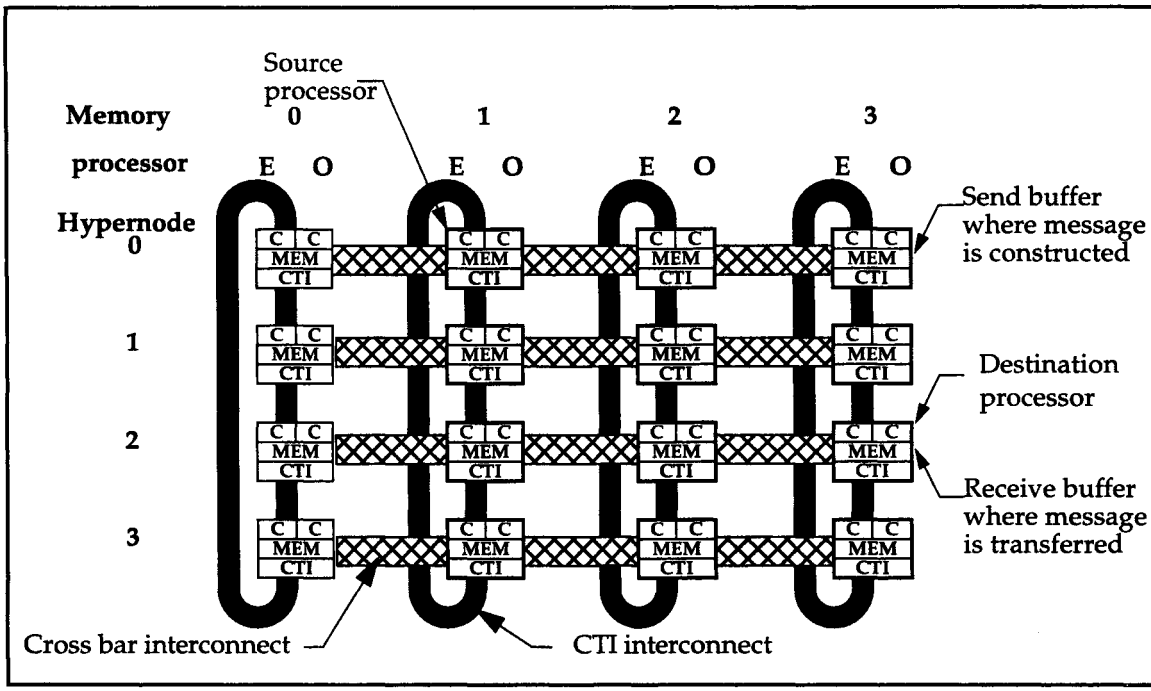

Figure 40 Exemplar system diagram (4 hypernodes)

Message buffers are allocated in each hypernode memory for the construction of messages sent and received. All messages sent by hardware are 64-bytes long. The processor sending the message constructs it within the hypernode in the memory located at the same CTI ring as the message destination processor.

The following example describes the process of sending a 64-byte message from hypernode 0, processor 1E to hypernode 2, processor 3O.

1. The source processor constructs the 64-byte message in a send message memory buffer in the memory of hypernode 0, memory 3. The message is constructed by using coherent stores (leaving the 64 bytes in the source processor cache), then flushing the message to the memory.
2. The message is then sent by issuing a send message command from the source processor to the source CTI controller at hypernode 0, memory 3.
3. The source CTI controller reads the 64-byte message and writes the 64 bytes noncoherently across the CTI ring to the message mail box of destination processor at hypernode 2, memory 3. The destination CTI controller is responsible for managing the message receive queue and issuing an interrupt to the destination processor.
4. The destination processor accesses the message from the receive queue using coherent load instructions. Once the destination processor is finished using the received message queue data, it is flushed from the processor cache, and the queue read pointer is updated.

Hardware structures to support messaging

Exemplar products support two messaging systems: standard and enhanced. The enhanced system functions exactly as the standard, except that it has more receive-message buffer memory. This section describes both systems.

The mechanisms for both systems include:

- Sending and receiving messaging memory buffers
- Constructing a message in a send-message buffer
- Sending a message
- CSR addressing of messages
- Receiving a message
- Receiving the message interrupt

Messaging memory buffers

Messages are constructed in send-message buffers and transferred to receive-message buffers. All message buffers are 64-bytes long. The send and receive messaging buffers for the standard system are organized as shown in Figure 41 and the enhanced in Figure 42. This memory structure is replicated once per cross bar port.

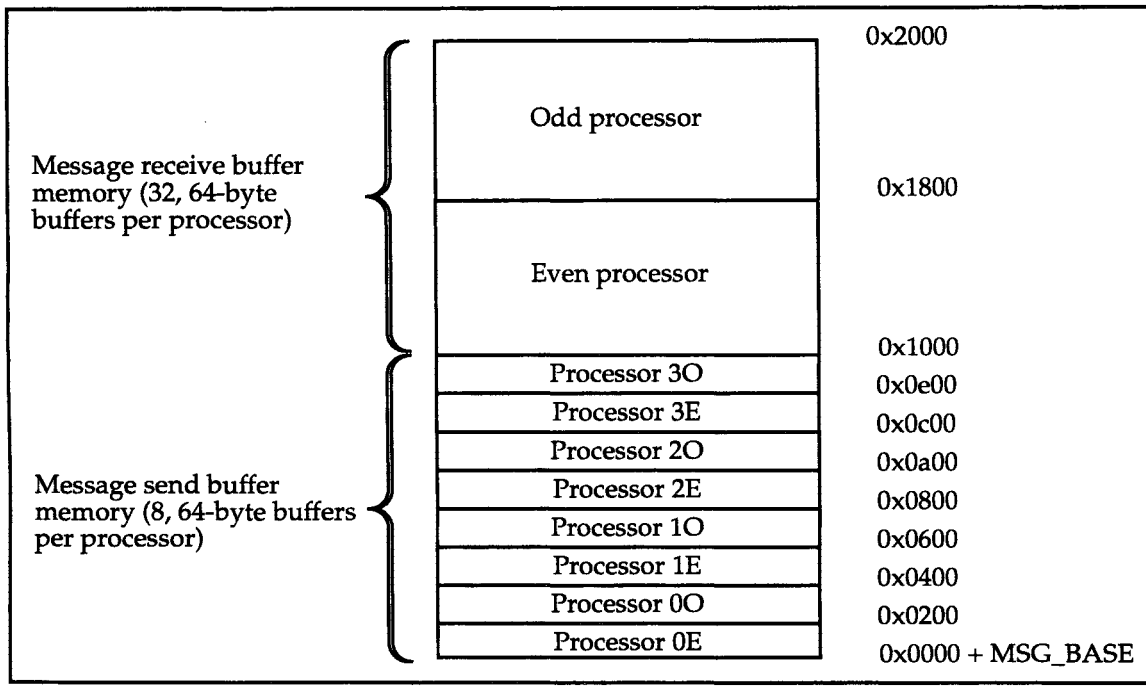

Figure 41 Standard messaging memory

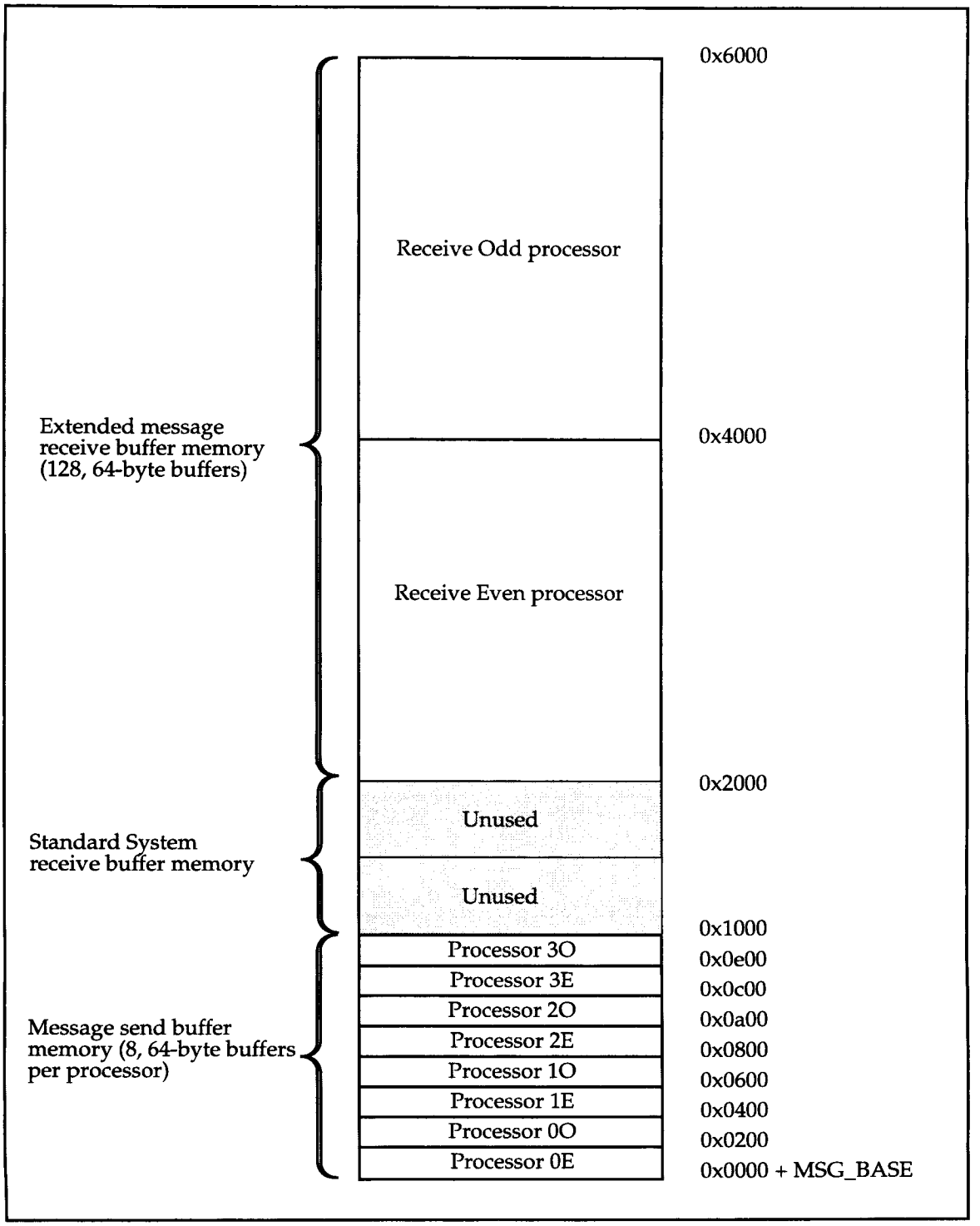

Figure 42 Enhanced messaging memory

Each processor in a hypernode has 8 send-message buffers per cross bar port memory. The processor constructs messages within these buffers. The send-message command uses a 3-bit index to specify a message buffer.

Both processors at a cross bar port have a queue of receive-message buffers. Each receive queue has space for 32 messages (of 64-bytes) for the standard system and 128 messages (of 64 bytes) for the enhanced. A message is sent to a CSR address at the destination CTI controller. Each CTI controller has two CSR addresses defined for receiving messages: one for the even processor and one for the odd.

Each memory controller contains a base address register (MSG_BASE) for the message buffer memory. The format of the standard system register is shown in Figure 43 and the enhanced in Figure 44. Bit 30 (EQE) enables the message-receive queue for the even processor. Bit 31 (OQE) enables the message-receive queue for the odd processor.

Note

If a processor receive queue enable bit is not set, sending a message to the processor causes the sending processor to take a high-priority machine check.

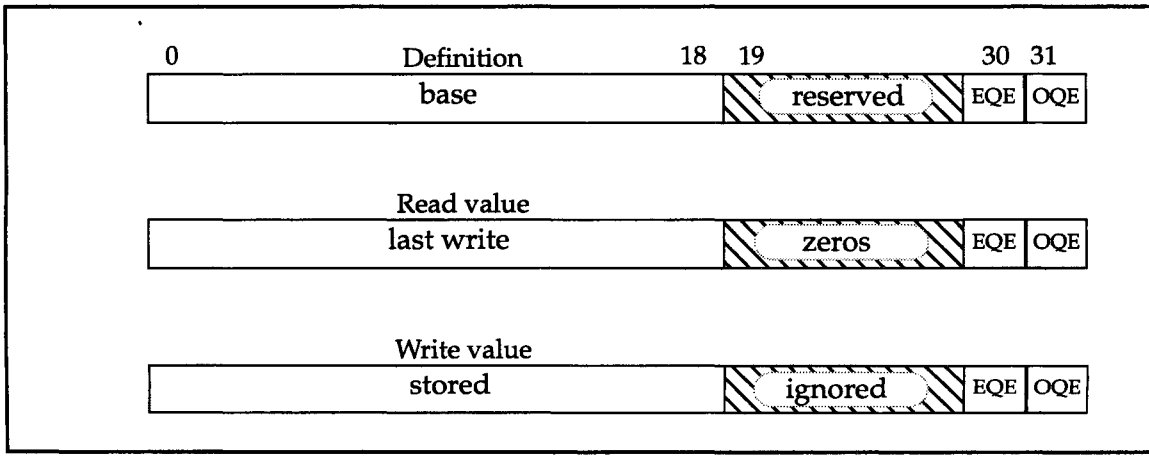

Figure 43 MSG_BASE register format for standard messaging system

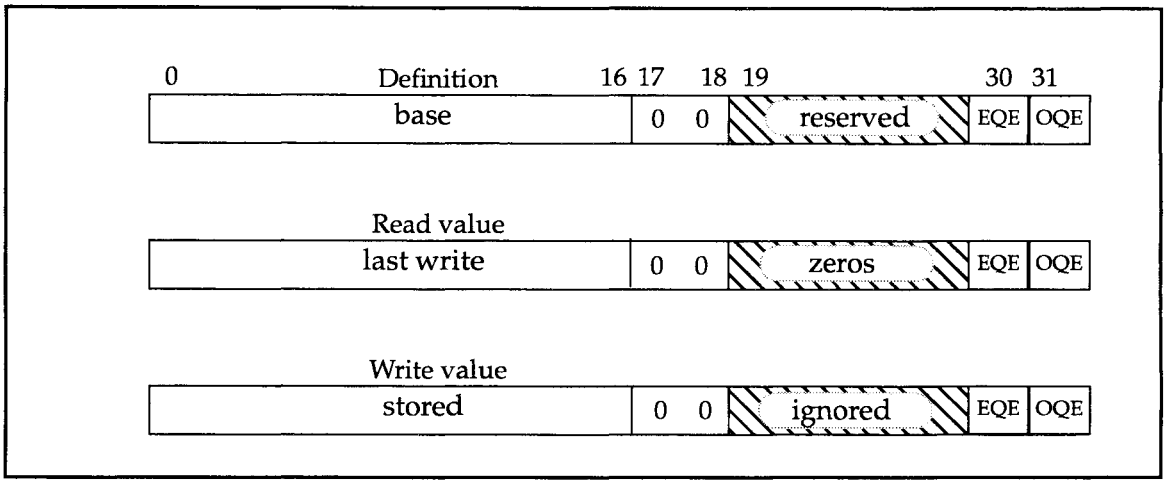

Figure 44 MSG_BASE register format for enhanced messaging system

The MSG_BASE register allows the message memory to start at any 8-Kbyte boundary for standard and any 32-kbyte boundary for enhanced, within a maximum memory size of 1 Gbyte per CTI memory controller.

Writing to a send-message buffer

Message buffer memory is coherent and is encached within a processor. A message is constructed within the processor cache using store instructions, then flushed to memory when the message is complete. Since the cache line size is 32-bytes, two cache lines must be flushed to write the message to memory. The FDC instruction is used to flush the cache.

Send message mechanism

The `send message` command is implemented by preconditioning the CPU agent to intercept a flush data cache instruction, FDC (with hint bits equal zero), and force a different operation to be performed. After executing the preconditioned FDC instruction, the mechanism resets, and all subsequent FDC instruction are interpreted normally. The address specified by the FDC instruction is used as the message destination address. The FDC instruction address determines which source CTI memory controller and ring to send the message.

An FDC instruction is preconditioned by writing to the MSG_SEND register. There is one MSG_SEND register per processor. Figure 45 shows the format of the MSG_SEND register.

Bits 23 through 25 specify the index within the message send buffer memory of the send message.

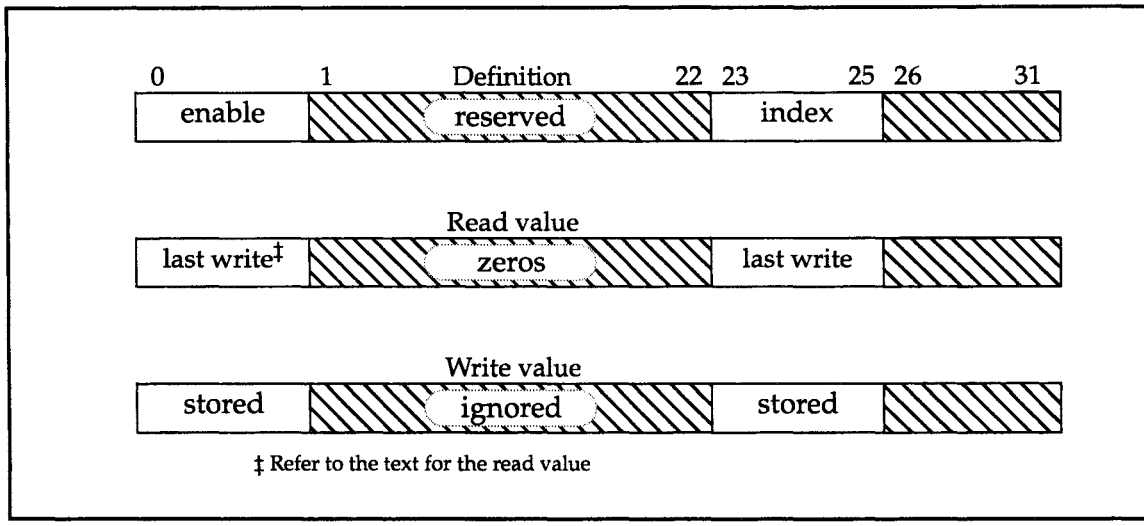

Figure 45 MSG_SEND register format

The enable field determines whether an FDC instruction (with hint bits equal to zero) is to be treated as a normal data cache flush or as a `send message` command. The field is written with the value of data bit zero when a store is performed to the MSG_SEND register. The value is reset automatically when an FDC instruction with hint bits set to zero is executed.

MSG_SEND register context

An interrupt may occur after the enable field of the MSG_SEND register is set, but before the FDC instruction sends the message. For this reason, the message send register is part of the process context. To reduce the interrupt context switch time, two MSG_SEND registers are used. The EXC_CONTEXT register selects the active register. A base-level interrupt switches to the alternate MSG_SEND register. All interrupts above the base level must save the MSG_SEND register context prior to using the send message mechanism.

Receive message mechanism

A message destination address is associated with each processor. Since a pair of processors share a port to the cross bar, each CTI memory controller recognizes two message destination addresses (MSG_ERCV and MSG_ORCV). These addresses are located in

the CSR (control space register) portion of the CTI address space. Refer to Table 20 for the address values.

Each message destination address has a message-receive queue, as well as two hardware registers used to manage the reading and writing of the queue. Figure 41 and Figure 42 show the messaging memory structure for the standard and enhanced messaging systems, respectively.

In the standard messaging system, the receive queue base-memory address is located at MSG_BASE+0x1000 for the even processor, and MSG_BASE+0x1800 for the odd. Each queue holds a maximum of 32 messages.

In the enhanced messaging system, the receive queue base-memory address is located at MSG_BASE+0x2000 for the even processor, and MSG_BASE+0x4000 for the odd. Each queue holds a maximum of 128 messages

The registers used to manage the reading and writing of the queues are: MSG_EQRD, MSG_EQWR, MSG_OQRD, and MSG_OQWR. These registers reside in CSR space and are not encached by the processor. All four registers have the same format, with the index field used for either read or write indexing. Figure 46 shows the format for these registers for the standard system and Figure 47 for the enhanced.

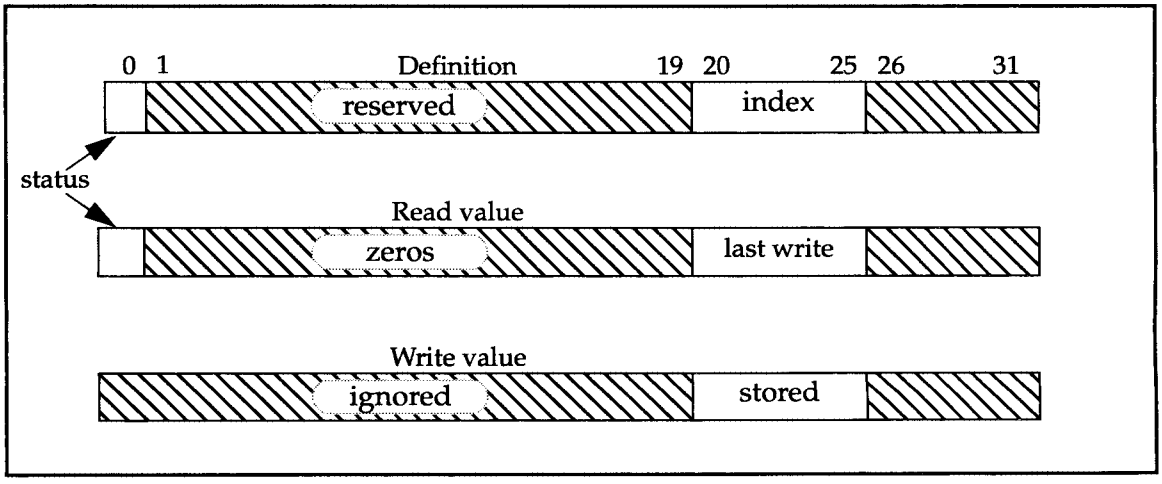

Figure 46 Queue maintenance register format

Bit zero is read-only. A 0 indicates the standard system is employed and a 1 the enhanced.

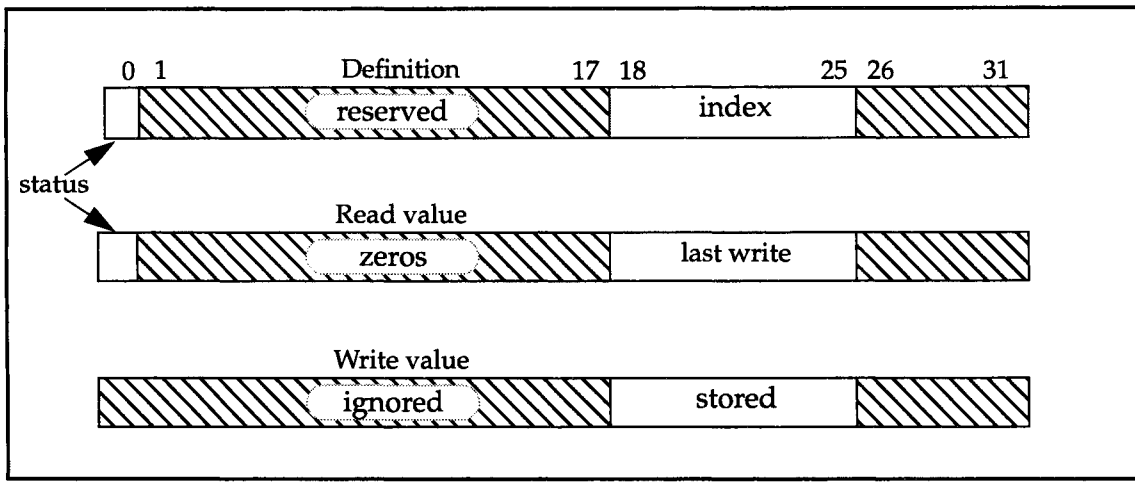

Figure 47 Queue maintenance register format

The processor loads from and stores to all queue maintenance registers. During normal usage, only the CTI memory controller stores to the queue write registers. The software is given store access to the queue write registers for initialization and test purposes.

Note

Software storing to a queue write register while hardware is receiving a message produces undefined results.

The hardware increments the index field of the queue write register by one after each message is written to the queue. Incrementing an index field when the value is all ones produces a result of all zeros.

Software is responsible for incrementing the index field of the queue read register after each message is read from the queue. Note that the receiving processor must flush or purge the receive-queue memory associated with the received message prior to incrementing the queue read register.

In the standard messaging system, the index field is six-bits wide, but only the least significant five bits are used to access receive-queue memory. In the enhanced system, the index field is eight-bits wide, but only the least significant seven bits are used to access receive-queue memory. The most significant index bit is used to differentiate between the queue full and queue empty conditions. The hardware interprets the condition in which all read and write index bits are equal except the most significant bit

to indicate queue full. Likewise, software detects a queue empty condition as all bits of the read and write indices being equal.

Receive message interrupt

Part of the process of receiving a message is the generation of an interrupt to the processor owning the receive-message queue. The interrupt is issued after the incoming data has been stored in the buffer and the queue write register has been incremented. The level of the interrupt is obtained from the MSG_EINTR or MSG_OINTR registers shown in Figure 48. The interrupt level field is 5 bits wide and specifies which one of the 32 interrupt levels should be issued for the received message.

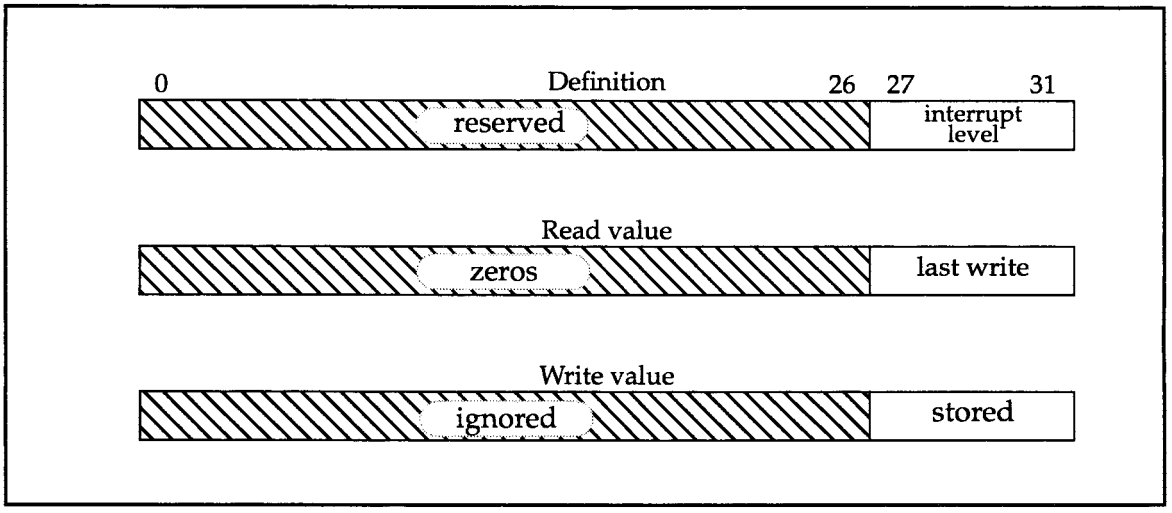

Figure 48 MSG_EINTR/MSG_OINTR register format

The interrupt is sent to the even processor or odd processor associated with the given CTIring. The interrupt level resides in bits 27-31.

Hardware register summary

Table 20 lists the hardware registers used for messaging with the hardware address, and a brief description of the functionality.

Table 20 Messaging hardware registers

Address	Name	Description
0xffef0900	EXC_CONTEXT	Process context info - message send register selection One register per processor
0xffef08c0	MSG_SEND	Send message precondition register One register per processor
0xffe*y*0060[1]	MSG_BASE	Base address of messaging memory One register per CTI memory controller
0xffe*y*0070[1]	MSG_EINTR	Even receive message interrupt register One register per CTI memory controller
0xffe*y*0078[1]	MSG_OIN	Odd receive message interrupt register One register per CTI memory controller
0xffe*y*0090[1]	MSG_EQRD	Even receive-queue read register One register per CTI memory controller
0xffe*y*0080[1]	MSG_EQWR	Even receive-queue Write register One register per CTI memory controller
0xffe*y*0098[1]	MSG_OQRD	Odd receive-queue read register One register per CTI memory controller
0xffe*y*0088[1]	MSG_OQWR	Odd receive-queue Write register One register per CTI memory controller
0xfff*yy*840[2]	MSG_ERCV	Even message receive address One register per CTI memory controller
0xfff*yy*880[2]	MSG_ORCV	Odd message receive address One register per CTI memory controller

1. The value of *y* is 10rr, where rr is the physical ring number (0-3) of the destination CPU.
2. The value of *yy* is rrnnnn00, where rr is the physical ring number (0-3) of the destination CPU and nnnn is the physical node number (0-15) of the destination CPU.

Synchronization 6

Overview

This chapter describes synchronization mechanisms provided by the Convex SPP series. It also discusses the low-level programming interface that accesses these mechanisms and the underlying hardware that supports them within the Exemplar implementation.

SPP architecture

The SPP architecture supports synchronization of multiple threads of a process through the use of semaphore and synchronization operators. These operators can be applied to any memory in the complex: CPU-private, node-private, near-shared, or far-shared.

Semaphore operators

SPP architecture supports three semaphore operators: fetch and clear, fetch and increment, and fetch and decrement. These perform their respective operations without encaching the semaphore variable. If the semaphore variable were encached prior to the semaphore operation, the variable is flushed from the caches before the operation is performed. One additional operator, fetch, loads a semaphore variable into a register without encaching it. The semaphore operations execute atomically on memory and support both 32-bit and 64-bit operands.

In addition, the architecture supports the fetch semaphore operators, the load and clear, defined in the HP PA-RISC instruction set are also supported.

Fetch semaphore operators

The fetch operators operate atomically on data in memory. The interface for the fetch operators is through calls to the architecture interface library. The semaphore variable address is an argument to the calls, and the original semaphore variable value is returned.

The definition of the library calls are specified in the "Low-level programming interface" section on page 123 of this chapter.

Uses for the fetch operators include the following:

- Nonencached binary semaphore locks
- Barrier synchronization
- Multiple reader/writer synchronization
- Synchronization of queues manipulated by multiple processors.

Shown below is the functionality of each fetch operator.

Fetch and clear:

```
new_semaphore_data = 0;
returned_value = orig_semaphore_data;
```

Fetch and increment:

```
new_semaphore_data = orig_semaphore_data + 1;
returned_value = orig_semaphore_data;
```

Fetch and decrement:

```
new_semaphore_data = orig_semaphore_data - 1;
returned_value = orig_semaphore_data;
```

Fetch:

```
returned_value = orig_semaphore_data;
```

PA-RISC load and clear semaphore operator

SPP supports the load and clear semaphore instructions, LDCWS and LDCWX, defined by the PA-RISC instruction set. The operator clears the semaphore variable and returns the original semaphore value. Note that this semaphore operator on data within the processor cache, provided the data is encached and is *dirty* from a previous write operation. The fetch operators flush the data from the cache prior to initiating the operation.

The load and clear semaphore operator coherently manipulates semaphore locks.

Note

The following restriction imposed by the PA-RISC instruction set on load and clear operations applies to both load and clear instructions as well as the fetch semaphore operations. The address of the semaphore variable must be 16-byte aligned. If the address is not aligned, the operation of the instruction is undefined.

Load and clear verses fetch and clear

Functionally, load and clear and fetch and clear are identical. The difference exists in the caching mechanisms. As an example, manipulation of binary semaphore locks with a load and clear semaphore operator is shown below.

Load and clear binary semaphore:

```
loop:LDCWSaddr,%1; Obtain lock
COMIBT,=0,%1,loop; Spin on failure
 •
 •; locked operation
 •
LDI1,%1
STW%1,addr; Release lock
```

Using a coherent store releases the lock, leaving the semaphore variable encached in the processor. If the same processor obtains the lock next, the load and clear will execute on data in the cache, reducing the execution time. If a different processor obtains the lock next, the normal coherency overhead to obtain ownership of the semaphore variable cache line extends the latency of the operation.

When fetch and clear is used to obtain a lock, the accompanying release lock operation should be a fetch and increment. This keeps the semaphore variable in memory, eliminating the cache coherence mechanism overhead.

The decision to use load and clear, coherent store or fetch and clear, or fetch and increment as the lock mechanisms should be based on whether the same processor or a different processor is expected to obtain the lock next.

Barrier synchronization

There is a single synchronization operator that is supported by the SPP architecture, `barrier_sync`. The operation is supported for 32-bit operands only, because the operator is performed on integer values and the architecture defines integers to be 32 bits in size.

The barrier synchronization operation is constructed with a fetch and decrement used to determine when the last thread of a process has entered the barrier synchronization, followed by a release mechanism to all threads to continue execution.

Below is the functionality of the barrier synchronization operation:

```
semaphore = semaphore - 1;
if (semaphore == 0) {
semaphore = semaphore_init;
release = release_value;
} else
while (release != release_value);
```

The interface for the barrier synchronization operation is a call to the architectural interface library. The semaphore variable, semaphore initialization value, release variable, and the release value are inputs to the call. The definition of the library call is specified in the "Low-level programming interface" section on page 123 of this chapter.

The barrier synchronization synchronizes all threads of a process after a section of code completes and before the next section of code starts.

Low-level programming interface

This section presents the software interfaces to the architectural interface library for the semaphore and synchronization operators.

Semaphore operators

The SPP architecture supports two types of semaphore operators: those supported directly by the PA-RISC instruction set and those accessed by library calls. This section describes the interfaces for the library calls. Refer to the *PA-RISC Assembly Language Reference Manual* for the syntax and operation of the semaphore operators, LDCWS and LDCWX, supported directly by the PA-RISC processor.

Fetch semaphore interfaces

There are two versions of the *fetch* operators, one operating on 32-bit quantities and one operating on 64-bit ones. For all routines, the argument is the address of the semaphore variable. The value returned is the original value of the semaphore variable before the operation is performed.

Fetch and clear:

```
int fetch_and_clear32 ( int *variable );
long fetch_and_clear64 ( long *variable );
```

The operation of the fetch and clear routines is functionally equivalent to the following routine, with the operation performed atomically in memory.

```
original = *variable;
*variable = 0;
return original;
```

Fetch and increment:

```
int fetch_and_inc32 ( int *variable );
long fetch_and_inc64 ( long *variable );
```

The operation of the fetch and increment routines is functionally equivalent to the following routine, with the operation performed atomically in memory.

```
original = *variable;
*variable = original + 1;
return original;
```

Fetch and decrement:

```
int fetch_and_dec32 ( int *variable );
long fetch_and_dec64 ( long *variable );
```

The operation of the fetch and decrement routines is functionally equivalent to the following routine, with the operation performed atomically in memory.

```
original = *variable;
*variable = original - 1;
return original;
```

Fetch:

```
int fetch32 ( int *variable );
long fetch64 ( long *variable );
```

The operation of the fetch routines is functionally equivalent to the following routine, with the operation performed atomically in memory.

```
original = *variable;
return original;
```

Note

All semaphore variables must be 16-byte aligned. Semaphore operations to nonaligned variables produce undefined results. The SPP architecture defines the 32-bit and 64-bit interfaces to the fetch semaphore library routines, however, only the 32-bit routines are supported for Exemplar.

Barrier synchronization

The barrier synchronization operation is a 32-bit operation. The first argument is the address of the semaphore variable. The second argument is the value to reinitialize the semaphore variable, once the barrier is reached. The third argument is the address of the release variable, and the final argument is the value assigned to the release variable.

```
barrier_sync ( unsigned int *semaphore,
unsigned int semaphore_init,
unsigned int *release,
unsigned int release_value);
```

Note that the arguments for `barrier_sync()` are 32 bits in size.

The operation of the `barrier_sync()` routine is functionally equivalent to the following routine, with the operation performed atomically in memory.

```
*semaphore = *semaphore - 1;
if (*semaphore == 0) {
*semaphore = semaphore_init;
*release = release_value;
} else
while (*release != release_value);
```

There is no return value for the `barrier_sync()` routine.

Note

For best performance, the semaphore variable and release variables should be in different cache lines.

Exemplar implementation specific information

This section describes the compliance of the Exemplar implementation to the SPP architecture. In addition, the hardware interface for semaphore operations are presented.

Exemplar compliance

Exemplar does not support the 64-bit versions of the semaphore operators: `fetch_and_clear64()`, `fetch_and_inc64()`, `fetch_and_dec64()`, or `fetch64()`.

Exemplar hardware interface

The Exemplar products support the load and clear semaphore operator. The additional semaphore operators required to support the SPP architecture are implemented by preconditioning the agent to intercept the load and clear operation and force a different operation. Once the preconditioned load and clear instruction executes, the agent resets to treat load and clear operations normally (as defined by the PA-RISC architecture).

SPP1000 semaphore

The normal load and clear instructions allow the operand to be encached. When the encached line is *dirty,* the load and clear instruction executes using the encached line; it does not go to memory. This feature reduces the time required to execute a normal load and clear instruction.

This preconditioned load and clear mechanism fails when the semaphore variable is encached within the cache of the issuing CPU. If the preconditioned load and clear instruction executes on encached data, the data is cleared in the cache. Load and clears that operate in the cache are not seen by the agent, and thus do not force the intended semaphore operation.

Note

To ensure correct operation of preconditioned load and clear operations, the semaphore variable must not be in the processor cache.

Agent hardware registers

This section presents the hardware registers contained within the agent used to support the SPP1000 semaphore operations.

Table 21 Semaphore hardware addresses for SPP1000

Address	Name	Description
0xffe# 2840[1]	SEMA_CLEAR	Precondition for fetch and clear semaphore Writable from user space
0xffe# 2844[1]	SEMA_INC	Precondition for fetch and increment semaphore Writable from user space
0xffe# 2848[1]	SEMA_DEC	Precondition for fetch and decrement semaphore Writable from user space
0xffe# 284c[1]	SEMA_FETCH	Precondition for fetch (noncoherent read) Writable from user space
0xffe# 2880[1]	SEMA_TYPE	Semaphore type Read/write from user space

1. The "#" sign specifies the physical processor number within a node. If a processor issues a request with # = 0xf, the address will be modified to the requesting processor.

SEMA_TYPE Register

The SEMA_TYPE register specifies the type of semaphore operator to be performed when the next load and clear instruction is executed. Figure 49 shows the format of the fields within the register.

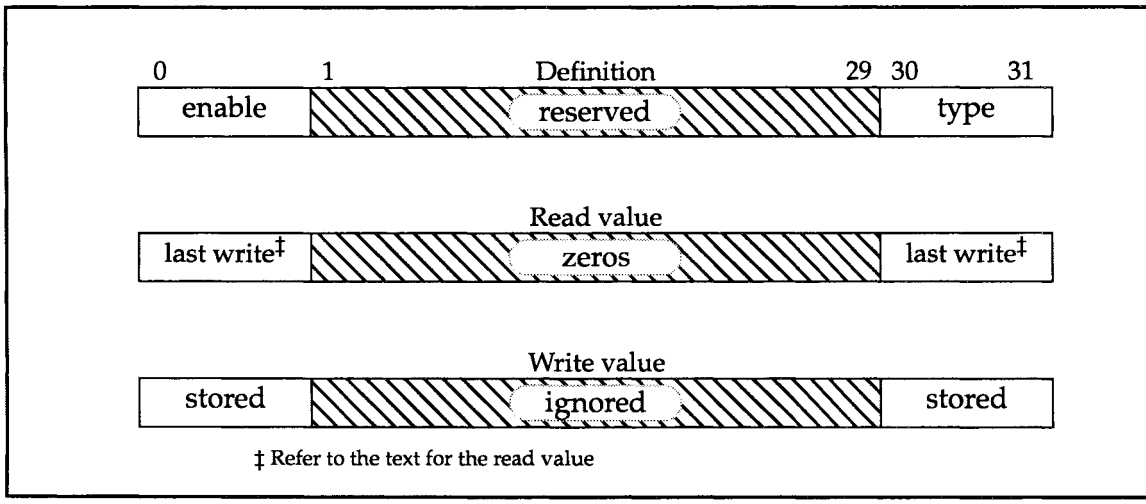

Figure 49 SEMA_TYPE register format for SPP1000

The meaning of the value in the type field is shown in Table 22.

Table 22 SEMA_TYPE [30:31] register values for SPP1000

Register value	Value meaning
0	Fetch and clear semaphore
1	Fetch and increment semaphore
2	Fetch and decrement semaphore
3	Fetch

The register is written in one of three ways. First, the register can be written by issuing a store to the address of the register. The value written is the value of the store data.

Second, the register is written by storing to one of the following addresses or *pseudo registers*: SEMA_CLEAR, SEMA_INC, SEMA_DEC, or SEMA_FETCH. Writing to one of these addresses writes to the SEMA_TYPE register with bits 28 and 29 of the address. The value written is used to specify the alternate type of

semaphore operation to perform when a load and clear instruction is executed and the enable bit is set.

Third, the register is written when a load and clear instruction is executed. The enable field of the SEMA_TYPE register is cleared after each load and clear instruction. The value written into the SEMA_TYPE register for each of the previously listed ways is shown in Table 23.

Table 23 Values written into SEMA_TYPE register for SPP1000

Method register is written	Value written	Usage description
Store to SEMA_TYPE Register	Store Data	Restore context
Execute load and clear instruction	Enable = 0	Perform preconditioned operation and reset register for normal load and clear operation
Store to SEMA_CLEAR address	Enable = 1 Type = 0	Precondition for fetch and clear
Store to SEMA_INC address	Enable = 1 Type = 1	Precondition for fetch and increment
Store to SEMA_DEC address	Enable = 1 Type = 2	Precondition for fetch and decrement
Store to SEMA_FETCH address	Enable = 1 Type = 3	Precondition for fetch (noncoherent read)

Reading the register will return the value in the register, and is normally only performed to save context. Once a preconditioned semaphore operation has been initiated by writing to the SEMA_TYPE register, it will be cleared by either performing the load and clear instruction, or storing the value zero to the enable field of the register.

SEMA_CLEAR, SEMA_INC, SEMA_DEC, and SEMA_FETCH registers

These addresses simplify the stores required to setup a preconditioned semaphore operation. Stores to these addresses actually write to the SEMA_TYPE register. The value written is used to specify the type of semaphore operation which will be performed. The value written to the type field of the SEMA_TYPE register is bits 28 and 29 of the store address as shown in Table 28. The enable field of the SEMA_TYPE register is set to a value of 1.

The value is normally only written to setup a preconditioned semaphore operation. The result of a read to these addresses is undefined.

SEMA_TYPE register context

The semaphore type register is process context because an interrupt may be received after the SEMA_TYPE register enable field is set, but before the LOAD AND CLEAR instruction. To reduce the interrupt context switch time, two SEMA_TYPE registers are implemented. The EXC_CONTEXT register selects active register. A base-level interrupt need only switch to the alternate SEMA_TYPE register. All interrupts above the base level must save the SEMA_TYPE register context prior to using the preconditioned load and clear mechanism. As stated in a previous section, processors executing preconditioned load and clear semaphore operations must not have the semaphore variable encached in the processor cache.

Assembly instruction sequences

The following assembly instruction sequences should be used to manipulate the semaphore registers to ensure proper operation.

Semaphore setup

The following sequence sets up a preconditioned semaphore. The example shown is for *fetch and increment*.

```
STW%0,SEMA_INC; initialize SEMA_TYPE
LDCWSsema_semaphore,%1; Perform Fetch and Inc
```

`Fetch and clear`, `fetch and decrement`, and `fetch` have similar instruction sequences.

Context save and restore

Two sets of semaphore registers are provided to reduce the required frequency of semaphore context saves and restores. The two sets are expected to be used in a similar fashion to the way the shadow registers are used to reduce context saves and restores for interrupt processing on the PA-RISC processor. The EXC_CONTEXT register is used to switch the active semaphore type register.

The sequence to save context for the semaphore mechanism is shown below.

```
LDWSEMA_TYPE,%1; Save TYPE register
STW%0,SEMA_TYPE; Clear the enable field
```

The restore context sequence simply reloads the original context as shown below.

```
STW%1,SEMA_TYPE; Restore TYPE register
```

Hardware error conditions

Errors detected while executing a preconditioned load and clear instruction are reported by setting the appropriate bit in the EXC_CAUSE register and issuing a machine check to the processor. Refer to Chapter 7 for more information.

SPP1200 and SPP1600 semaphore

This section presents differences in semaphore operations for the SPP1200 and SPP1600.

SPP1200 and SPP1600 agent hardware registers

The SPP1200 and SPP1600 agent is primed in the same manner as the SPP1000 agent for semaphore operation. After priming, a graphics flush is performed. The address of the flush and the node number are stored in the semaphore address register as the full address to be used for the semaphore. Table 24 details the SPP1200 and SPP1600 semaphore operation, and Table 25 shows their semaphore CSR addresses.

Table 24 SPP1200 and SPP1600 semaphore operation

Semaphore operation	Description
write to SEMA_TYPE register	defines the type of semaphore operation
graphics flush to semaphore address	loads the SEMA_ADDR register
read SEMA_START register	starts the semaphore operation using the address in the SEMA_ADDR register and the type in the SEMA_TYPE register

Table 25 Semaphore CSR addresses for SPP1200 and SPP1600

Address	Name	Description
0xffe# 2840[1]	SEMA_CLEAR	Precondition for fetch and clear semaphore Writable from user space
0xffe# 2844[1]	SEMA_INC	Precondition for fetch and increment semaphore Writable from user space
0xffe# 2848[1]	SEMA_DEC	Precondition for fetch and decrement semaphore Writable from user space
0xffe# 284c[1]	SEMA_FETCH	Precondition for fetch (noncoherent read) Writable from user space
0xffe# 0880[1]	PAGE 0 SEMA_TYPE	Context restore address
0xffe# 0884[1]	SEMA_ADDR	Semaphore address. Thirty-two bits define a 16-byte aligned address

Table 25 Semaphore CSR addresses for SPP1200 and SPP1600—**(continued)**

Address	Name	Description
0xffe# 2880[1]	SEMA_TYPE	Semaphore type
0xffe# 2890[1]	SEMA_START	Semaphore start magic address

1. The "#" sign specifies the physical processor number within a node. If a processor issues a request with # = 0xf, the address will be modified to the requesting processor.

SEMA_TYPE Register

The SEMA_TYPE register specifies the type of semaphore operator to be performed when there is a read of the SEMA_START register and after the SEMA_ADDR is primed.

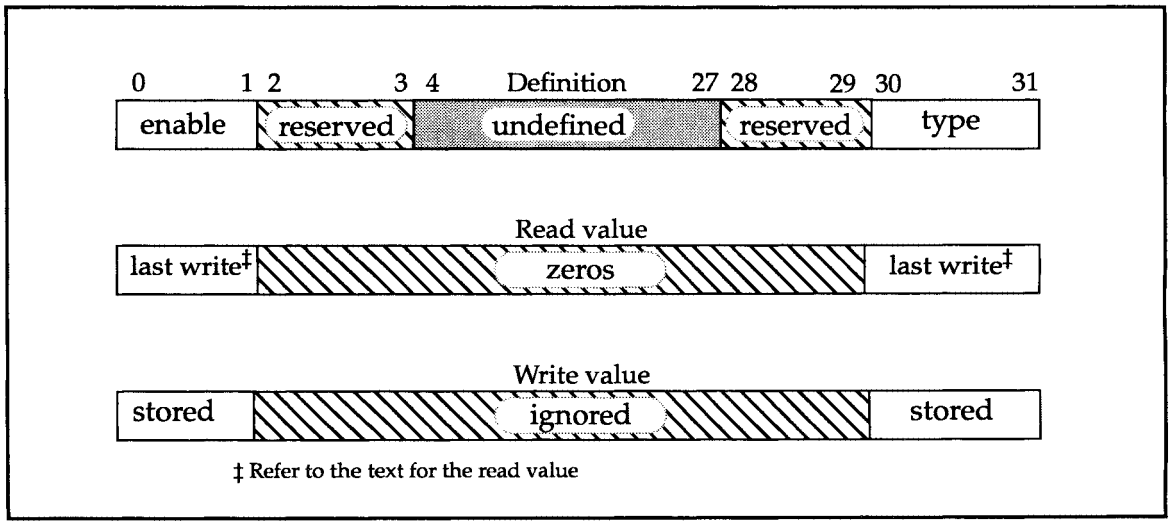

Figure 50 SEMA_TYPE register format for SPP1200 and SPP1600

Notice that the SPP1200 and SPP1600 have two enable bits. These bits are written to in the same manner as those in the SPP1000 SEMA_TYPE register with the addition of the new Page 0 context restore, but are cleared differently. The SPP1200 and SPP1600 agent clears the enable bits when the semaphore is returned to the agent. This difference is transparent to a user program, since they are cleared before the data is returned to the processor.

The enable bits are written with each step of the semaphore operation and are defined in Table 26.

Table 26 SPP1200 and SPP1600 SEMA_TYPE [0:1} enable bits

Value	Definition
00	Normal operation, no semaphore behavior
10	SEMA_TYPE has been primed
11	SEMA_TYPE and GR_FLUSH have been primed
01	Semaphore operation has been issued and is pending

The meaning of the value in the type field is shown inTable 27.

Table 27 SEMA_TYPE [30:31] values for SPP1200 and SPP1600

Register value	Value meaning
0	Fetch and clear semaphore
1	Fetch and increment semaphore
2	Fetch and decrement semaphore
3	Fetch

The SEMA_TYPE register is written in five ways:

- Issuing a store to the address of the SEMA_TYPE register itself
- Reading from the SEMA_START register

  The enable field of the SEMA_TYPE register is cleared after read from the SEMA_START register.

Note

User code should only use the SEMA_TYPE address. The operating system code uses the PG0_SEMA_TYPE address.

- Storing to one of the following addresses or *pseudo registers*: SEMA_CLEAR, SEMA_INC, SEMA_DEC, or SEMA_FETCH

  Writing to one of these addresses writes to the SEMA_TYPE register with bits 28 and 29 of the address. The value written specifies the alternate type of semaphore operation to perform when the SEMA_START register is read and both enable bits are set.
- Performing a graphics flush with the enable bits=10
- Resetting the enable bits (enable=00)

The value written into the SEMA_TYPE register for each of the previously listed ways is shown in Table 28.

Table 28 Values written into SEMA_TYPE register for SPP1200 and SPP1600

Method register is written	Value written	Usage description
Store to SEMA_TYPE Register	Store Data	Restore context
Read SEMA_START with enable = 11	Enable = 01	Perform preconditioned operation
Store to SEMA_CLEAR address	Enable = 10 Type = 0	Precondition for fetch and clear
Store to SEMA_INC address	Enable = 10 Type = 1	Precondition for fetch and increment
Store to SEMA_DEC address	Enable = 10 Type = 2	Precondition for fetch and decrement
Store to SEMA_FETCH address	Enable = 10 Type = 3	Precondition for fetch (noncoherent read)
Graphics flush with enable = 10	Enable = 11	Store address for semaphore operation
Semaphore data returns to CPU	Enable = 00	Semaphore operation complete

Reading the register will return the value in the register, and is normally only performed to save context. Once a preconditioned semaphore operation has been initiated by writing to the SEMA_TYPE register, it will be cleared by either performing a read from the SEMA_START register, or storing the value zero to the enable field of the register.

SEMA_ADDR register

The SEMA_ADDR bits are written in two ways only. The first is the write to Page 0 address during context restores. The second is when the enable bits=10 and a graphics flush with the hint bits set to 11 is performed. The SEMA_ADDR is loaded with the virtual address and node translation provided by the graphics flush.

SEMA_CLEAR, SEMA_INC, SEMA_DEC, and SEMA_FETCH registers

These addresses have been defined to simplify the stores required to setup a preconditioned semaphore operation. Stores to these addresses actually write to the SEMA_TYPE register. The value written is used to specify the type of semaphore operation which will be performed. The value written to the type field of the

SEMA_TYPE register is bits 28 and 29 of the store address as shown in Table 28. Additionally, the enable field of the SEMA_TYPE register is set to the value 01.

The value is normally only written to set up a preconditioned semaphore operation. The result of a read to these addresses is undefined.

SEMA_TYPE register context

The semaphore type register is process context because an interrupt may be received between setting the SEMA_TYPE and the graphics flush, or between the graphics flush and reading the SEMA_START. To reduce the interrupt context switch time, two SEMA_TYPE registers are implemented. The selection of which of the two registers is active is made by the EXC_CONTEXT register. A base-level interrupt need only switch to the alternate SEMA_TYPE register. All interrupts above the base level must save the Page 0 SEMA_TYPE register context prior to using the semaphore mechanism.

For context saves, the SEMA_TYPE and the SEMA_ADDR registers are read and stored to memory. The SEMA_TYPE register is duplicated for shadow contexts A and B. There is, however, only one SEMA_ADDR register for each CPU. When switching the shadow context, the SEMA_ADDR register must also be saved and restored to memory.

When restoring the SEMA_TYPE register, a new Page 0 CSR address is used to write the address bits as well as the SEMA_TYPE enable bits. This prevents a user application from writing the address bits without TLB checks. Only writes to the Page 0 SEMA_TYPE address change the type and enable bits directly. The SEMA_ADDR register must also be restored during shadow context switches.

Hardware error conditions

Errors detected while executing a preconditioned read from the SEMA_START register are reported by setting the appropriate bit in the EXC_CAUSE register and issuing a machine check to the processor. For SPP1200 and SPP1600, if the SEMA_ENABLE bits are not sequenced properly, the error conditions in Table 29 result.

Table 29 SEMA_ENABLE bits error conditions

SEMA_ENABLE	Error condition behavior
00	CSR load to SEMA_START returns undefined data
10	CSR load to SEMA_START returns undefined data

Exceptions and interrupts 7

Overview

This chapter describes the exception and interrupt mechanisms provided by the Convex Exemplar.

PA-RISC architecture

The PA-RISC architecture specifies most of the interrupt and exception handling mechanisms.

PA-RISC architected registers

Several processor status word (PSW) bits are relevant to interrupt or exception processing. Table 30 shows the PSW bits and their significance.

Table 30 PWS bits used for interrupt processing

PSW Bit	Field	Significance
15	M	High-priority machine check mask. When 1, high-priority machine checks (HPMCs) are not allowed. Normally 0, this bit is set to 1 after an HPMC and set to 0 after all other interruptions.
27	R	Recovery counter enable. When 1, recovery counter traps occur if bit 0 of the recovery counter is a 1. The R bit also enables decrementing of the recovery counter.
28	Q	Interrupt state collection enable. When 1, interruption state is collected. This state is recorded in the interruption instruction address queue (IIAQ,) the interruption instruction register (IIR,) the interruption space register (ISR,) and the interruption offset register (IOR.)
31	I	External interrupt, power failure interrupt, and low-priority machine check interruption unmask. When 1, these interruptions are unmasked and can cause an interruption. When 0, the interruptions are held pending.

Many of the control registers (CRs) are relevant to interrupt and exception processing. These are summarized in Table 31.

Table 31 Control registers used for interrupts and exceptions

Register	Name of register	Use
CR0	Recovery counter	The recovery counter is used to provide software recovery of hardware faults in a fault-tolerant system. CR0 counts down by 1 during the execution of each non-nullified instruction for which the PSW R-bit is 1. When the left-most bit of CR0 is 1, a recovery counter trap occurs.
CR14	Interrupt vector address (IVA)	Contains the absolute address of the base of an array of service procedures assigned to the interruption classes. The array is indexed by interrupt/exception number. This address must be a multiple of 2048.
CR15	External interrupt enable mask (EIEM)	A 32-bit register containing a bit for each of the 32 external interrupts. When 0, bits in the EIEM mask interruptions pending the external interrupts corresponding to those bit positions.
CR16	Interval timer	The interval timer consists of two registers: a read-only count register which is continually incrementing, and a write-only comparison register. When the count register equals the comparison register, bit 0 of the External interrupt request register (EIR) is set to 1. This will cause interruption if the interrupt is not masked.
CR17	Interruption instruction address space queue (IIASQ)	A two-deep queue containing the space of the current and next instruction. These registers are updated when the PSW Q-bit is 1. Reading IIASQ when the Q-bit is 0 returns the front element; writing IIASQ when the Q-bit is 0 copies the back element to the front, and replaces the back element.
CR18	Interruption instruction address offset queue (IIAOQ)	A two-deep queue containing the offset (and protection level) of the current and next instruction. These registers are updated when the PSW Q-bit is 1. Reading IIAOQ when the Q-bit is 0 returns the front element; writing IIAOQ when the Q-bit is 0 copies the back element to the front, and replaces the back element.
CR19	Interruption instruction register (IIR)	Contains the instruction being executed at the time of certain interruptions or exceptions. This register is only updated when the Q-bit is 1and may be read-only when the Q-bit is 0.

Table 31 Control registers used for interrupts and exceptions—**(continued)**

Register	Name of register	Use
CR20	Interruption space register (ISR)	Contains the space of a virtual address reference for certain exceptions. This register is updated when the Q-bit is 1, and may be read-only when the Q-bit is 0.
CR21	Interruption offset register (IOR)	Contains the offset of a virtual address reference for certain exceptions. This register is updated when the Q-bit is 1, and may be read-only when the Q-bit is 0.
CR22	Interruption Processor Status Word (IPSW)	Contains the PSW at the point of interruption, regardless of whether the PSW Q-bit was 1 or not. IPSW can be read or written reliably only when the Q-bit is 0.
CR23	External interrupt request register (EIR)	The EIR is a 32-bit register containing a bit corresponding to each external interrupt. When 1, the bit designates that in interruption is pending for the corresponding interrupt. Both the PSW I-bit and the corresponding bit in the External interrupt enable mask (CR15) must be 1 for an interrupt to occur. A MTCTL instruction with EIR (CR23) as its target bit-wise ANDs the complement of the contents of the source register with the previous contents of CR23, and places this result in CR23. Thus, the processor can only set the EIR register bits to 0.
CR24-CR31	Temporary registers	CR24, 25, and 28-31 are usable as temporary register by OS interrupt/exception handlers.

PA-RISC Interrupts

Interrupts cause the flow control to be passed to an interruption handling routine. Upon completion of interruption processing, an RFI or RFIR instruction is executed, which restores the saved processor state, and the execution proceeds with the interrupted instruction.

From the viewpoint of response to interruptions, the processor behaves as if it were not pipelined. That is, it behaves as if a single instruction is fetched and executed. Any interruption conditions raised by that instruction are handled at that time. If there are none, the next instruction is fetched, and so on.

Faults, traps, interrupts, and checks are different classes of interrupts.

A fault occurs when an instruction requests a legitimate action that cannot be carried out due to a system problem, such as the absence of a page from main memory. After the problem has been corrected, the instruction causing the fault executes normally. Faults are synchronous with respect to the instruction stream.

A trap occurs when a function requested by the current instruction cannot or should not be carried out (attempting to access a page for which user has insufficient privilege, for example). Another example is when system intervention is desired by the user before or after the instruction is executed, such as page reference traps used for debugging. Traps are synchronous with respect to the instruction stream.

An interrupt occurs when an external device requires CPU attention. This is done by setting a bit in the external interrupt request register. Interrupts are asynchronous with respect to the instruction stream.

A check occurs when the processor detects a malfunction. The malfunction may or may not be correctable. Checks can be either synchronous or asynchronous with respect to the instruction stream.

Table 32 Interrupts

Group	Int. number	Interruption type
1	1	High-priority machine check
2	2	Power failure interrupt
2	3	Recovery counter trap
2	4	External interrupt
2	5	Low-priority machine check
3	6	Instruction TLB miss fault/instruction page fault
3	7	Instruction memory protection trap
3	8	Illegal instruction trap
3	9	Break instruction trap
3	10	Privileged operation trap
3	11	Privileged register trap
3	12	Overflow trap
3	13	Conditional trap

Table 32 Interrupts —**(continued)**

Group	Int. number	Interruption type
3	14	Assist exception trap
3	15	Data TLB miss fault/data page fault
3	16	Nonaccess instruction TLB miss fault
3	17	Nonaccess data TLB miss fault/no-access data page fault
3	18	Data memory protection trap/unaligned data reference trap
3	19	Data memory break trap
3	20	TLB dirty bit trap
3	21	Page reference trap
3	22	Assist emulation trap
4	23	Higher privilege transfer trap
4	24	Lower privilege transfer trap
4	25	Taken branch trap
3	26	Data memory access rights trap
3	27	Data memory protection ID trap
3	28	Unaligned data reference trap

The interrupt numbers listed in the second column of Table 32 index the array of interrupt service procedures pointed to by CR14.

PA-RISC interrupt handling

The following actions take place on an interruption:

- The PSW in effect at the time of the interruption is saved in the IPSW.
- The defined bits in the PSW are set to 0 if the interrupt is not a high-priority machine check (interrupts are masked, absolute addresses enabled). If the interruption is a high-priority machine check, all defined bits in the PSW are set to 0 except the Mbit, which is set to 1.
- Instruction address information in the IIASQ and IIAOQ queues is frozen as a result of setting the PSW Q-bit to 0.
- The current privilege level is set to level zero.
- Information about the interrupting instruction is saved in the IIR, ISR, and IOR if it is potentially useful in processing the interruption. The value saved in the ISR is undefined if data address translation is not enabled.
- General registers 1, 8, 9, 16, 17, 24, and 25 are copied to the shadow registers.
- Execution begins at the address given by:
  - Interruption vector address + (32 * interrupt Number)
  - High-priority machine checks always vector to physical address 0xF0000000

Exemplar exceptions

The Exemplar hardware may detect abnormal conditions. These conditions generate low- or high- priority machine checks, power fail interrupts, or other exceptions.

Exception cause register (EXC_CAUSE)

Exemplar hardware provides an exception cause register (EXC_CAUSE) for each CPU. Each bit in the EXC_CAUSE corresponds to some type of exception detected by the hardware. Exceptions include parity error, bus error, and nonexistent memory. The hardware sets the corresponding bit for the cause of each exception in the EXC_CAUSE register. The definition of the exception conditions represented by each bit is implementation dependent.

The EXC_CAUSE register of any given processor need not be accessible to any other processor. EXC_CAUSE access requires CPU-privilege level zero.

Exception context register (EXC_CONTEXT)

Each CPU is associated with an exception context register, EXC_CONTEXT. Software exception and handlers use this 32-bit register to select interrupt or base-level context. The bits in this register are a collection of several different fields used in various parts of the SPP architecture. In general, the EXC_CONTEXT register is updated on entry to, and exit from, each exception or interruption handler. The EXC_CONTEXT register may be read and written only at CPU-privilege level 0.

Interrupts

The interrupt registers are accessible only at CPU-privilege level zero.

INT_IO_EIR register (post interrupt to designated CPU)

Each CPU has an INT_IO_EIR (I/O external interrupt register) that is used for posting interrupts to that CPU. The INT_IO_EIR is 32 bits wide. Each bit in the INT_IO_EIR corresponds to a bit in the external interrupt register (EIR) of the CPU.

A store to the INT_IO_EIR with any bits set to one results in the same bits being set in the corresponding CPU EIR (CR23) register. Bits set to zero in the INT_IO_EIR have no effect. Bits are cleared in the INT_IO_EIR (by hardware) when the corresponding EIR bit has been set. Software does not normally read the INT_IO_EIR.

The INT_IO_EIR is addressable by both intra-hypernode and interhypernode CPUs within the SPP complex. By addressing the correct INT_IO_EIR, an interrupt may be directed at any CPU within the SPP1000.

IO_EIR register (SPP1200 and SPP1600 only)

In the SPP1200 and SPP1600, if there is more than one bit written to INT_IO_EIR, the processor agent translates a single write to the INT_IO_EIR into multiple back-to-back writes to the PA7200 IO_EIR. The value written to this register is a 5-bit, encoded interrupt level instead of a bit vector in the intranode space as is the INT_IO_EIR register in the SPP1000. The INT_IO_EIR register functions in the SPP1200 and SPP1600 as it does in the SPP1000 for backward compatibility.

Interrupt target/mask registers

Each hypernode provides one interrupt target register (INT_TARGET) and interrupt mask register (INT_MASK). The INT_TARGET register provides an address for casting interrupts to all the CPUs on the hypernode. A write to this target address is ANDed bit-wise with the INT_MASK register, then distributed to all units on the hypernode.

INT_TARGET is a 32-bit wide register. Each bit in the INT_TARGET register corresponds to a bit in a CPU EIR. A store to the INT_TARGET register with any bits set to one results in the same bits being set in the EIR registers of all CPUs within the hypernode containing the INT_TARGET register (if the corresponding mask bit in the INT_MASK register is also set). Bits are cleared in the INT_TARGET register by hardware when they

have been delivered to the EIRs. Software does not normally read the INT_TARGET register.

The INT_MASK register is a 32-bit wide mask register used with the INT_TARGET register as described above. The INT_MASK register may be read and written by software.

The INT_TARGET register is addressable by both intra-hypernode and inter-hypernode CPUs. By addressing the correct INT_TARGET register, interrupts can be broadcast to all CPUs within the addressed hypernode.

The INT_MASK register is accessible only from the hypernode containing it.

Low-level programming interface

The software interface to the interrupt and exception processing facilities is considered implementation-dependent, and is not exported to the user.

Exemplar compliance

This section describes the Exemplar implementation specifics for exception and interrupt handling.

Table 33 shows the address for the EXC_CAUSE, EXC_CONTEXT, INT_IO_EIR, INT_TARGET, and INT_MASK registers.

Table 33 Exceptions and interrupts

Register	Address
Exception cause register (EXC_CAUSE)	0xffe*n*-0408 (*n*=[0...7,f] corresponding to the processor number in the hypernode
Exception context register (EXC_CONTEXT)	0xffe*n*-0900 (*n*=[0...7,f] corresponding to the processor number in the hypernode
IO external interrupt register (INT_IO_EIR)	0xffe*n*-0400 (*n*=[0...7,f] corresponding to the processor number in the hypernode
Interrupt Target register (INT_TARGET)	0xffee0160
Interrupt mask register (INT_MASK)	0xffee0900

Exemplar exception registers

This section describes the exception registers: EXC_CAUSE and EXC_CONTEXT.

EXC_CAUSE register for SPP1000

Table 34 gives each of the possible exception causes (indicated in the EXC_CAUSE register), the recommended recovery procedure for the exception, and possible availability implications of the exception for the SPP1000. Reading the register clears it.

Table 34 EXC_CAUSE register for each SPP1000 CPU

Bit	Field	R/W	Description
0	Reserved	clr/set	
1-4	Reserved	clr/set	
5	hu_deadlock_err	clr/set	Deadlock timeout
6	aps_mes_length_err	clr/set	Invalid message transaction length
7	aps_dque_wr_par_err	clr/set	Write parity error on APS data buffer
8	aps_dque_rd_par_err	clr/set	Read parity error on APS data buffer
9	csr_access_err	clr/set	Error reading agent CSR
10	hu_inv_resp_err	clr/set	Received an invalid response
11	hu_inv_mjmn_err	clr/set	Received an invalid MAJOR/MINOR response pair
12	hu_empty_gone_err	clr/set	Received GONE with a null present vector
13	hu_invalid_err	clr/set	Received a response to an invalid TID
14	hu_err_cycle	clr/set	Received error cycle response
15	hu_err_resp	clr/set	Received MJ_ERROR response
16	Software LPMCH	clr/set	Reserved for software use
17-20	Reserved	clr/set	
21	apq_mjmn_req_err	clr/set	Invalid request transaction
22	apq_mes_length_err	clr/set	Invalid request transaction length
23	apq_non_exist_err	clr/set	Request to unavailable processor
24	csr_access_err	clr/set	Error reading agent CSR

Table 34 EXC_CAUSE register for each SPP1000 CPU—**(continued)**

Bit	Field	R/W	Description
25	roi_mes_length_err	clr/set	Invalid request transaction length
26	roi_access_size_err	clr/set	Nonword aligned/size reference
27	pas_dq_rpe	clr/set	PAS RAM read parity error
28	pas_dq_wpe	clr/set	PAS RAM Write parity error
29	pas_inv_s_err	clr/set	Issued an INV.SHARED_INV to a processor that hits with the line in the "private" state
30	pas_inv_d_err	clr/set	Issued an INV.DIRTY_INV to a processor that hits with the line in the "shared" state
31	pas_rdi_s_err	clr/set	Issued a RD_IND.* to a processor that hits with the line in the "shared" state

EXC_CAUSE registers for SPP1200 and SPP1600

Table 35 gives each of the possible exception causes (indicated in the EXC_CAUSE register), the recommended recovery procedure for the exception, and possible availability implications of the exception for the SPP1200 and SPP1600.

The bits in the EXC_CAUSE registers in the SPP1200 and SPP1600 are read and written differently from the SPP1000, as follows:

- Reading the register does not change the state of its bits (that is, reading it does not clear them as in the SPP1000).
- Writing a 1 only *toggles* the state of the bits, it does not set them to a 1.
- Writing a 0 does not change the state of the bits at all.

This scheme allows a software handling routine to read the register without resetting the bits. To reset the register, the routine would write back to the EXC_CAUSE register the same data it read (making all bits that had equaled a 0 remain a 0 and all bits that equaled a 1 be set to a 0). Another consequence of this scheme is, if additional bits are set in the register (as a result of additional errors), they are not reset by the write operation.

Table 35 EXC_CAUSE register for each SPP1200 and SPP1600 CPU

Bit	Field	Description
0	pas_csri_sprefsnp_err	A snoop (RD_INV or INV) came into APQ for line for which a processor has a st_prefetch outstanding.
1	pas_csri_invshpr_err	An INV.SHARED_INV came into APQ for the line that the processor has private.
2	pas_csri_lram_err	PAS LATCH RAM parity error
3	pas_csri_coh_uflow_err	A processor issued a coherency response when there was no transaction issued for which to respond.
4	c_illegal_trans	The CPU issued an illegal Runway Transaction. Probably DPURGE, READ_BS, or WRITE_SHORT_DONE. Could also be caused by parity error in the transaction type of an address cycle.
5-6	rbi_csri_rwy_aperr[0:1]	Runway Parity Error on address/data bus. Bit [0] covers address/data bits 0:31. Bit [1] covers address/data bits 32:63.
7	rbi_csri_rwy_cperr	Runway Control Parity Error. Parity covers Master-ID and Transaction-ID[3:5].
8	paq_sbdt_adr_err	Incorrect access of SBDT. Each CPU can only access the SBDT address range assigned to it.
9	nram_rd_bperr	BDT Node LRAM byte parity error for read data. Only checked when the node LRAM is accessed.
10	reserved	
11	paq_vring_err	Virtual ring addressing error caused by accessing a ring that is not in the system configuration.
12	c_sema_start_err1	Semaphore start error caused by reading the SEMA_START register when the semaphore logic is incorrectly primed.
13	csri_addr_err	Access by a CPU to a nonexistent CSR in APA.
14	csri_size_err	Non 32-bit word sized access to a CSR in APA.
15	apq_mes_length_err	Incoming XBAR request message length error.
16	apq_mjmn_req_err	Incoming XBAR request major/minor type error.
17	apq_access_size_err	Incoming XBAR request accessing a CSR with a non-word sized transaction.

Table 35 EXC_CAUSE register for each SPP1200 and SPP1600 CPU—**(continued)**

Bit	Field	Description
18	apq_csr_addr_err	Incoming XBAR request accessing a non-existent CSR.
19	reserved	
20	reserved	
21	reserved	
22	hu_csri_empty_gone	Gone Response with no bit in the Present Vector set.
23	hu_csri_rsp_err	Error Response which does not return data was received. This also excludes also message send errors when the hpmch bit is not set.
24	hu_csri_err_tail	Error Tail Received on a data return.
25	aps_csri_hpmc_perr	APS Buffer Parity Error during Unlock_WR. This will cause HPMC to CPU since the CPU may not have detected this error.
26	reserved	
27	hu_csri_rtn_err	Log-Only. Error response instead of data response was received. Machine check has already been flagged to the CPU by setting ADDR_VALID and DATA_VALID at the same time that the data was returned, so this is a log-only function.
28	rbi_csri_broad_error	Log-Only. CPU issued BROAD_ERROR and is entering Machine-Check. Do not send anymore HPMCs.
29	aps_rbi_perr	Log-Only. APS Buffer Parity Error while returning data to the CPU. It is expected that the CPU will detect this error and machine check itself.
30	hu_rbi_msg_err (no soft-to-hard effect)	Log-Only. Error response received to a Message Send Transaction. CPU should be flagged via an interrupt.
31	reserved	

When the hardware detects a machine check condition, the appropriate bit is set in the EXC_CAUSE register. If enabled, the machine check is reported.

EXC_CAUSE register

The fields in EXC_CONTEXT are defined in Table 36.

Table 36 EXC_CONTEXT register

Bit	Field Width	Field use and designation
0	1	Enables collection of performance monitor data when bit 0 is 1. When bit 0 is zero, performance monitor data is not collected. (PMON_ENABLE)
1	1	Forces stores to be strongly ordered. (STRONG_ORDER)
2	1	The interpretation of the graphics read instruction is changed to be a noncoherent prefetch if this bit is set. If this bit is clear, the graphics read instruction is interpreted as a CTI cache (coherent) prefetch. (NON_COHERENT_PREFETCH)
3-26	25	Reserved. return zeros when read, ignored when written.
27	1	This bit selects the register set used for message send operations. (MSG_SEND_REGSET)
28-30	3	Reserved. return zeros when read, ignored when written.
31	1	This bit selects the register set used for semaphore operations. (SEMA_TYPE)

Exemplar interrupt registers

The Exemplar interrupt structure supports 32 interrupts per processor. Each CPU has an INT_IO_EIR (I/O external interrupt register) that posts interrupts to that CPU. The INT_IO_EIR is 32-bits wide. Each bit in the INT_IO_EIR corresponds to a bit in the external interrupt register (EIR) of the CPU.

A store to the INT_IO_EIR with any bits set to one results in the same bits being set in the corresponding CPU EIR (CR23) register. Bits set to zero in the INT_IO_EIR have no effect. Bits are cleared in the INT_IO_EIR (by hardware) when the corresponding EIR bit has been set. Software does not normally read the INT_IO_EIR.

The INT_IO_EIR is addressable by both intrahypernode and interhypernode CPUs. By addressing the correct INT_IO_EIR, an interrupt may be directed at any CPU within the SPP.

INT_IO_EIR register

Interrupts can be set by writing to the INT_IO_EIR register, with the specific interrupt bit being nonzero. When the INT_IO_EIR is nonzero, an interrupt is sent to the processor. The current interrupts are cleared when the interrupt is sent to the processor. Table 37 describes the INT_IO_EIR register.

Table 37 INT_IO_EIR register

Bit	Field	r,w	Description
0-29	General purpose	r/set	Software definable interrupts
30	Pmon_Interrupt	r/set	Performance monitor interrupt
31	General purpose	r/set	Software definable interrupt

I/O subsystem

8

Overview

This chapter defines the Exemplar I/O subsystem, the low-level programming interface that accesses the I/O subsystem, and the underlying hardware.

I/O system overview

The Exemplar I/O subsystem serves as a bridge between the SPP system and peripheral devices. The major architectural hardware component of the SPP I/O system is an adapter board that provides memory-mapped access from CPU(s) to I/O controller register space and allows external devices to transfer data in and out of system memory. The adapter hardware implements a programmable direct memory access (DMA) engine capable of executing complex I/O data transfer operations, including address translation and read-ahead/write-back buffering. DMA transfers are associated with I/O channels. These are logical constructs which define and control the movement of data through the SPP I/O system. The I/O adapter supports multiple active channels at the same time.

I/O operations are configured and initiated by system software. A memory-mapped, register-level programming interface is implemented within the adapter for this purpose.

Currently, there are three versions of the adapter board: the SIOP1, SIOP2, and SIOP3. All are generally the same in terms of the programming model, the differences are discussed in appropriate sections of this chapter.

I/O adapter block diagram

The major hardware components of the SPP I/O system are shown in Figure 51.

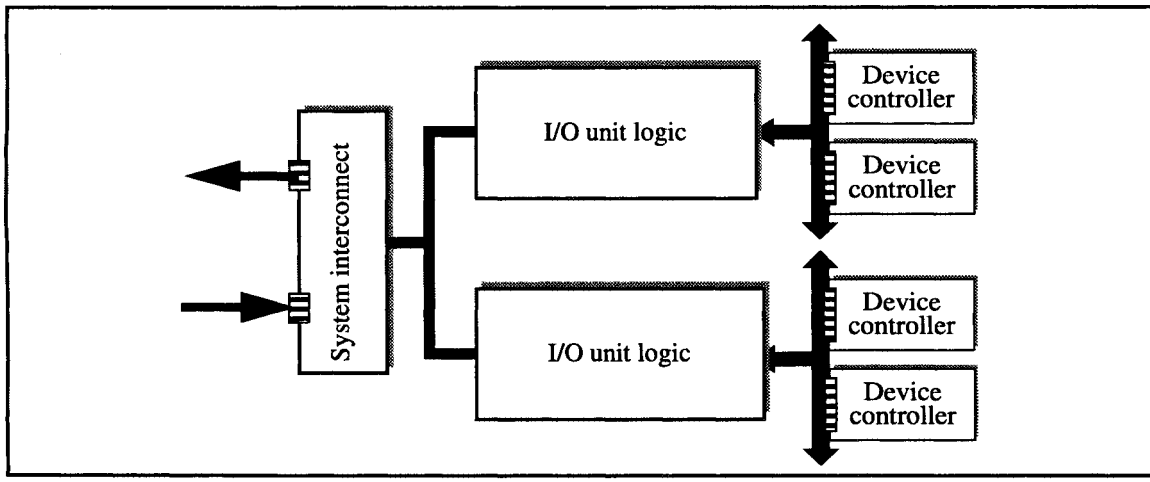

Figure 51 SPP I/O system hardware

The architectural features of the Exemplar I/O system are primarily associated with the I/O unit logic. A more detailed view of the functional components of an I/O unit is shown in Figure 52. The remainder of this chapter describes the features and functionality of an SPP I/O unit.

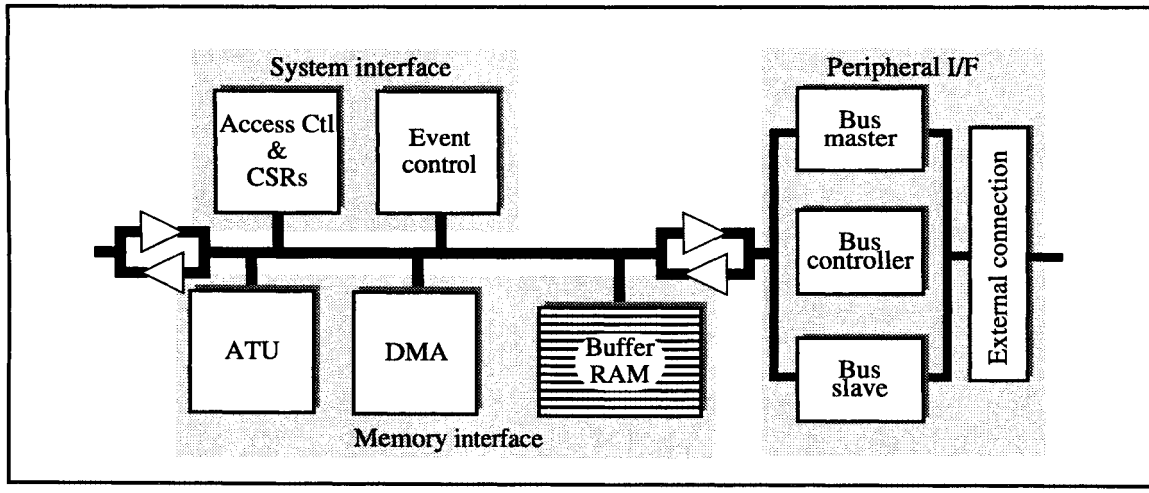

Figure 52 I/O unit logic components

Hardware-software interface

The SPP architecture does not assume an embedded control processor within the I/O subsystem. The I/O hardware must be initialized and managed by processors external to the I/O subsystem (CPU or Utilities Card, for example). This requires that all control structures and operational parameters are accessible within the SPP system address space. All I/O registers, configuration RAMs, and data buffers are mapped to the noncoherent address range (I/O space) of the processor physical memory space. These are accessed as bytes, halfwords, or words via LOAD/STORE commands.

Processor accesses to device controller registers are routed through the SPP I/O adapter and mapped to a peripheral address space. The address map is defined in a set of programmable registers in the I/O CSR set.

The programming model for I/O hardware has been defined with maximum flexibility in mind. The basic structure of the subsystem is modular and potentially scalable. Variations of the architecture are not required to implement the full functionality.

Low-level programming interface

The Exemplar I/O unit is configured and managed through control and status registers (CSRs) and logical data structures. These registers and structures constitute the I/O hardware-programming model. The manner in which they are programmed and/or interpreted by software and influenced or affected by the hardware is the heart of the SPP I/O architecture. All CSRs and control structures mapped into the system address space are read by SPP processors. They are either written by software, modified by hardware, fetched from system memory, or some combination of these methods.

The SPP I/O programming interface is loosely organized as a set of functional interfaces and their associated control structures. The following sections present an operational description of the three major interfaces: the system interface (SIF), the memory interface (MIF), and the peripheral interface (PIF).

System interface

The system interface enables the I/O system to interact with the SPP processors. The system interface has two major functions: it responds to read/write accesses directed to the I/O adapter or peripheral controllers and it processes locally generated events such as interrupts or exception conditions. The I/O system interface accesses are not restricted to architectural control and status structures. All storage locations within the I/O system are accessible via LOAD/STORE operations from system processors. The buffer RAM of the I/O unit and possibly the peripheral address segments, are used as general-purpose shared memory regions accessible by SPP processors and I/O peripherals. Read and write requests are routed to the I/O hardware through the system interconnect where they are further decoded and mapped as needed. Only aligned word, halfword, or byte accesses are supported. The I/O hardware does not require or support lock operations (fetch and add, compare and swap, for example). The I/O units do not participate in the cache coherency protocol.

Memory map

SPP I/O units are mapped into noncoherent system memory space (refer to Chapter 3). Each I/O unit responds to accesses within a fixed 4-Kbyte address range known as the unit's hard physical address (HPA). In addition, several extended address segments may be defined for a unit within soft allocated I/O space.

HPA segment

The I/O hard physical address (HPA) space consists of 64 sets of sixteen 32-bit registers. These contain I/O configuration control and status registers (CSRs), unit ID PROMs and other architecturally defined control structures. Some of the registers are defined within the *HP I/O Architecture Specification* and others are Convex SPP I/O-specific. The location, format, and use of the individual HPA register sets is shown in Figure 53 and described below.

Start Address*	Allocation	Size
0xFFEn0000	I/O unit SRS	64 B
0xFFEn0040	I/O unit ARS	64 B
0xFFEn0080	Reserved	3712 B
0xFFEn0F00	I/O event control Reg sets 0-3	64 B
0xFFEn0F40	I/O event control Reg sets 4-7	64 B
0xFFEn0F80	I/O event control Reg sets 8-B	64 B
0xFFEn0FC0	I/O event control Reg sets C-F	64 B

* n = 'C' for I/O unit 0 & 'D' for unit 1

Figure 53 I/O unit hard physical space

- I/O unit supervisor register set (SRS)—A set of 16 architecturally defined registers associated with the unit as a whole. Their exact form and function are implementation-dependent and are defined in the adapter hardware functional specification.
- I/O unit auxiliary register set—A set of sixteen SPP I/O-specific registers defining the location and size of the extended address spaces associated with the I/O unit. The initialization and buffer segments are direct-mapped into the soft allocated I/O space. The mapped peripheral segment is indirectly mapped at a page granularity. The mapped peripheral segment register (offset 6) allocates a portion of system memory to the segment; the peripheral map segment register (offset 7) defines the location of the mapping registers. All segments are restricted to sizes equal to a power

of two number of pages ((2**[size])*4096 bytes) must be aligned on boundaries that are multiples of their size and contained entirely within the I/O soft allocated memory space. The ARS registers currently defined are shown below. Details regarding the structure and use of the segments they specify are given in subsequent sections.

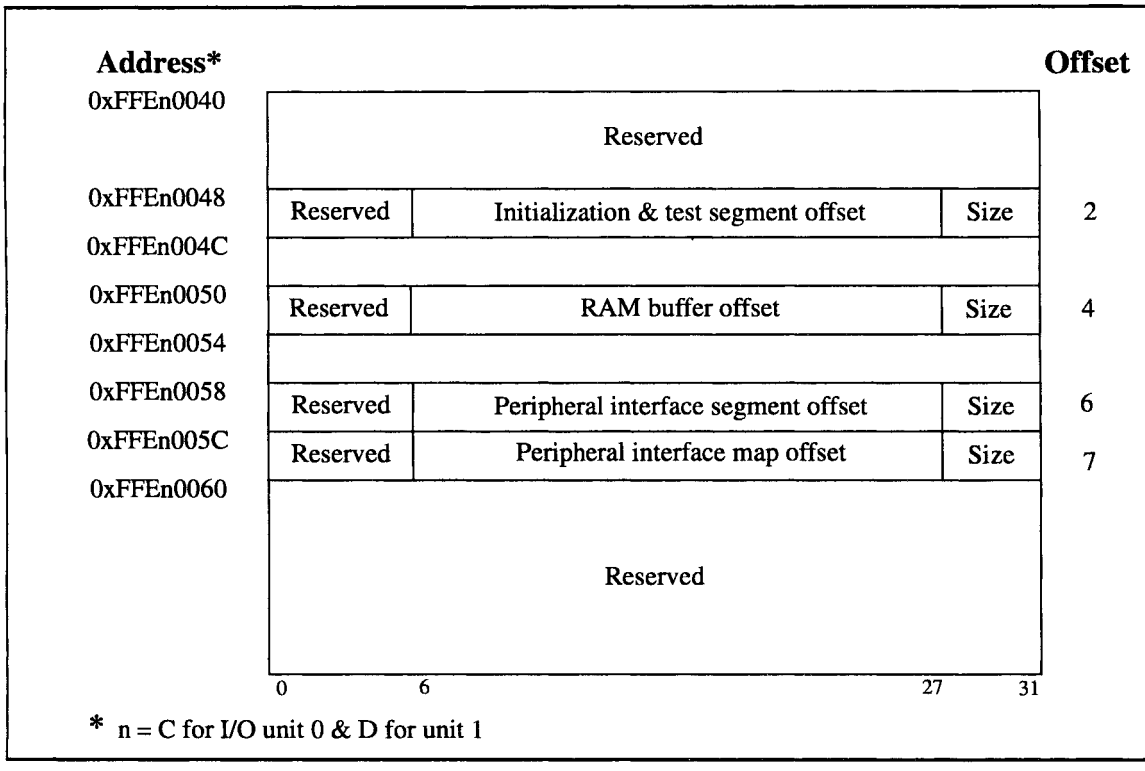

Figure 54 I/O unit ARS register set

- I/O event control registers—The occurrence of an I/O event is recorded and reported by means of 16-byte control structures referred to as event control registers. There are sixteen event-control registers, each of which is associated with fifteen individual events (yielding a total of 240 events). The first three words of the register set are read/writable by software and never modified by the hardware. The final halfword of the structure, containing event bits 1-15, is individually set by the hardware when the associated event has occurred. The event bits may be read by software with no effect. Writing to this location will cause the hardware to perform a bit-wise comparison operation between the write value and the current register contents. If both equal one, the

bit is cleared; if not, the bit is not modified. This mechanism acknowledges an event occurrence. All reserved bits may be written and read by software and are not interpreted or modified by hardware. Further details regarding I/O events and associated functionality are given in the "I/O events" section of this chapter.

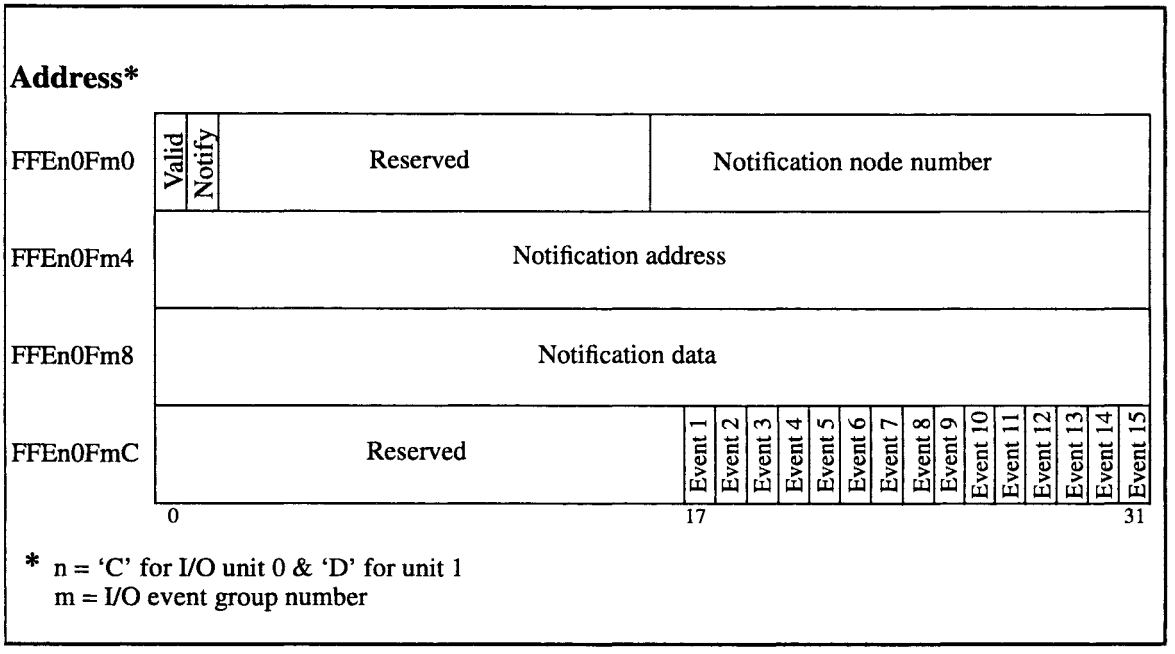

Figure 55 I/O event control registers

Initialization & test segment

An optional extended address segment used for I/O unit initialization and test purposes exists within the system's I/O soft allocation space. The location and size of the segment is specified by value of ARS register offset 2 in the I/O unit HPA. If enabled, the content of this segment is implementation specific.

I/O buffer RAM segment

The SPP I/O units contain a RAM array used to store data and control information associated with I/O channel operations. It can be up to 16-Mbyte in size and is directly mapped—as an extended I/O address segment—into the system's noncoherent address space. Its location and size are specified within the unit's HPA (ARS register offset 4). It is read or written by software and accessed and updated by hardware in a variety of ways. Detailed

descriptions of the use of the buffer RAM are presented in the "I/O channels" section on page 164.

Peripheral interface address segment

The I/O unit peripheral interface exports a 64-Gbyte, 36-bit address space for communication between SPP system components and external devices and controllers. Segments of this space are dynamically allocated to I/O devices for DMA channel operations (see subsequent sections for details). SPP processors access this space via an extended I/O unit address segment known as the peripheral interface address segment. The size and location within system physical address space is defined within the HPA (ARS register offset 6). This segment of the noncoherent memory range is mapped on a 4-Kbyte page granularity into the 64-Mbyte peripheral address space. The mapping is performed "on the fly" by the I/O hardware using the peripheral segment map. The location of the segment map is specified by ARS register offset 7, it contains an entry for each 4-Kbyte page of the peripheral interface segment map (described below).

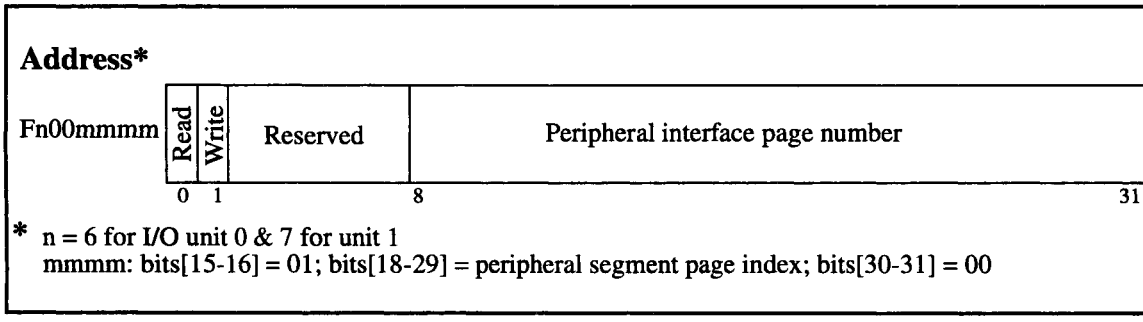

Figure 56 Peripheral interface segment map

I/O events

I/O events are either generated by the I/O hardware in response to some internal condition or received from external devices through the peripheral (or system) interface. DMA completion, device controller interrupts, and exception conditions are examples of events recognized by the SPP I/O subsystem. Events are organized into sixteen groups. There are fifteen events within a group. These are numbered 1 through 15, with zero being defined as no event. Each group has an associated control register set that records the occurrence of individual events in a bit vector. The individual bits of the vector are set upon the occurrence of the corresponding event within the group and cleared when written with a value of 1 by software.

If the *notify* bit in the first word of the group control structure is set, a write operation to system memory (or CSRs) will be executed by the event-control hardware. The SPP node number, node address, and data word used in the write operation are supplied from the event group registers.

The hardware does not throttle or queue events. A notification write occurs each time an event occurs, regardless of the state of the event bit. Also, a set bit is cleared when written with a one, even if multiple events have occurred since the bit was initially set.

I/O channel exceptions and transfer completion events are mapped to specific events groups and numbers via programmable parameters within the channel state tables. The mechanism for mapping external device/controller interrupts to events is implementation specific.

Note

The event notification operation is commonly used to assert a processor interrupt by writing to a node or CPU interrupt register, however, it may be used for other purposes. For example, event notification writes targeted to I/O channel or device controller registers can effectively "chain" I/O operations.

Memory interface

I/O units transfer data to and from SPP main memory. The type, size, and location of these transfers is determined by software-programmable parameters and external device controller requests. The functional logic element of the I/O unit responsible for memory access is referred to as the memory interface. Once programmed and enabled, the memory interface logic is capable of managing and executing complex operations without further intervention from software.

The DMA hardware supports multiple concurrent transfer operations. These are multiplexed and scheduled to ensure optimal throughput. The address Translation unit (ATU) translates the logical address space exported to the peripheral interface to SPP physical memory locations at a 4-Kbyte page granularity. The memory interface logic also cooperates with the peripheral interface hardware to manage a segmented read-ahead and write-back buffer. All memory interface operations are governed by the initial parameters and current status values associated with logical entities known as I/O channels.

I/O channels

The central concept of the memory interface is the logical I/O channel. All I/O transfer operations are associated with a logical channel. An I/O unit may have many channels active simultaneously. The exact number is implementation specific, up to an architectural limit of 4096. A channel is defined by a control structure as the channel state table. The contents of the channel state table are used by the hardware to determine transfer size and type, manage access to data buffers and other I/O resources, perform address translations, and record operational status.

Channel state tables reside in the lowest address region of the I/O unit buffer RAM. They are read or written by software through the buffer RAM segment of system noncoherent address space. The exact location of a particular channel table within the RAM is determined by multiplying the channel number by 128 (the table size). The channel 0 table begins at buffer byte 0, channel 1 at 128, channel 2 at 256, and so on. All of the defined fields are read out of the buffer RAM by various hardware elements. Many are modified and written back as well. It is the responsibility of software to ensure the coherency of channel tables when writing fields associated with active channels.

The format of a channel state table is shown in Figure 57. The individual elements of the state table are defined in the sections pertaining to their specific function. Reserved fields have no effect

on channel operations and may be used by software as read/write registers if desired.

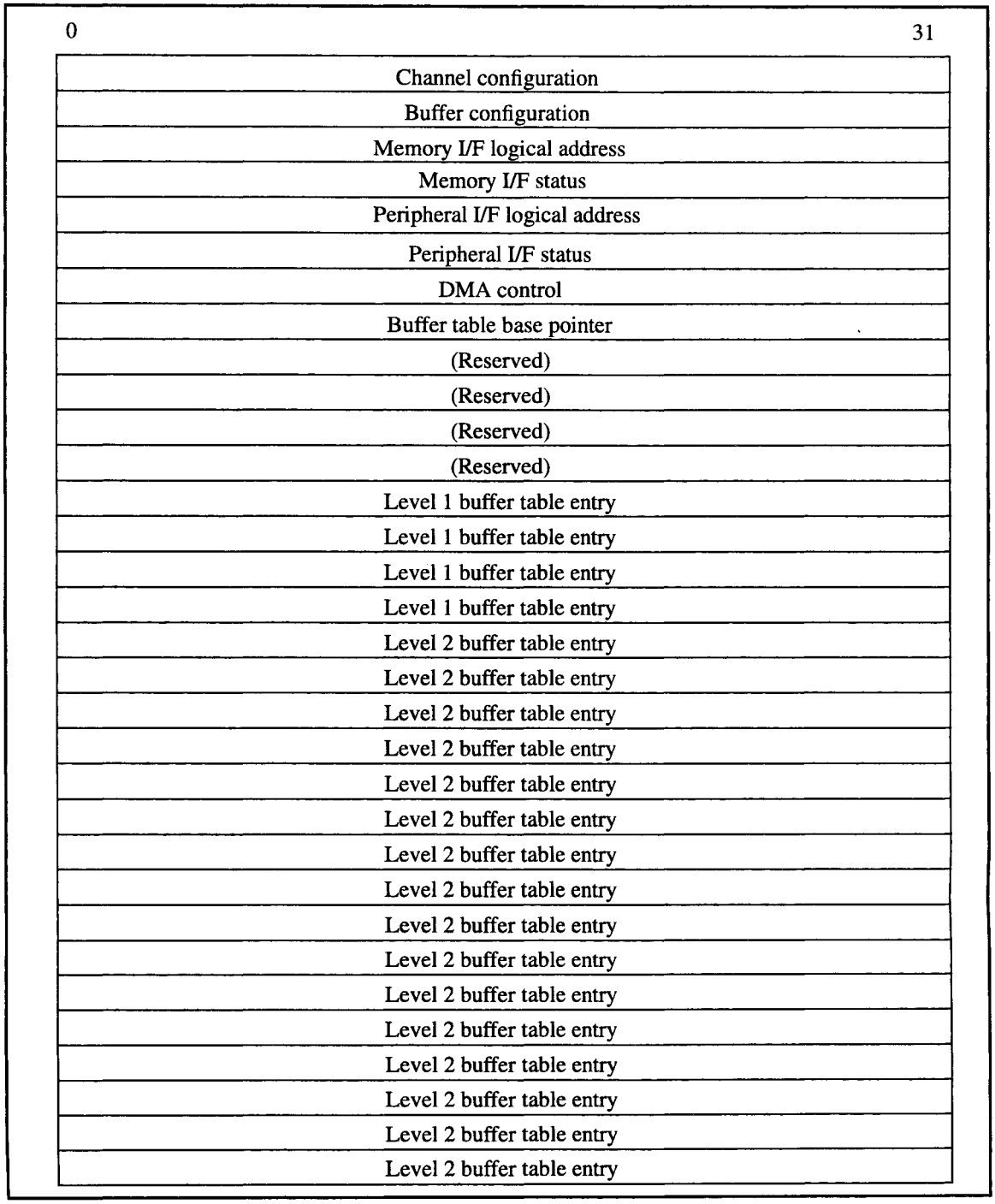

Figure 57 Channel state table format

Channel configuration register

The channel configuration register allocates I/O channels. It is written by system software only and referenced by the hardware to define and validate channel operations. The various fields are shown and described in Figure 58.

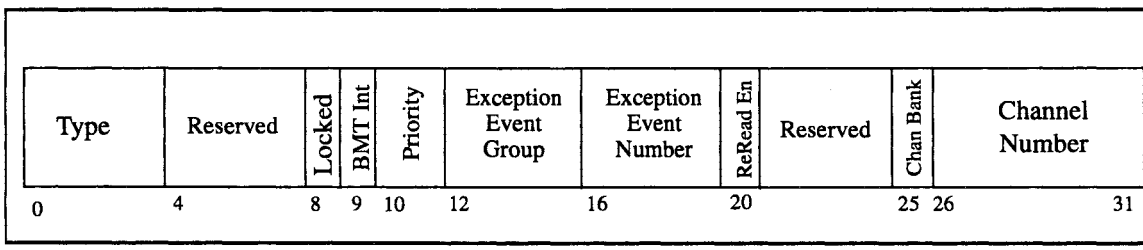

Figure 58 Channel configuration register

- Type—This pseudo-encoded 4-bit field enables an I/O channel and specifies allowed modes and functions. The channel types are listed below:
  - 0—Invalid—The Logical channel has not be initialized.
  - 1-7—Reserved—Not currently defined and reserved for future enhancements in I/O functionality.
  - 8—Shared memory—No prefetch or write back. The I/O buffer RAM is statically mapped to system memory and peripheral space, and may be randomly accessed by CPUs and device controllers.
  - 9—Interlocked Input—Not implemented.
  - A—Interlocked Output—Not Implemented.
  - B—Interlocked I/O—Not Implemented.
  - C—Buffered Random Access—Not Implemented.
  - D—Buffered Input—Data is written into channel buffer space by device controllers through the peripheral interface and asynchronously transferred to main memory by the DMA hardware. Buffer address mapping will "roll over" automatically to support arbitrarily long transfers. Consecutive writes to discontiguous buffer addresses are allowed but will cause the buffer to be flushed to memory before the second buffer write occurs. Read accesses from device controllers are exceptions.
  - E—Buffered Output—Data is prefetched from main memory into the channel buffer by the DMA hardware and read out by the peripheral interface. Buffer address mapping will "roll over" automatically to support arbitrarily long transfers. Discontiguous accesses from

the peripheral interface are allowed but may result in the current buffer contents being invalidated. Write accesses from device controllers are exceptions.

- F—Buffered I/O—Not Implemented.

- Locked—If set this bit will inhibit further channel operations and the memory and/or peripheral interface logic will set their respective "locked" bits in the status registers when a channel operation is scheduled or attempted.
- BMT Int—Buffer Empty Interrupt enable. If set, the MIF will assert a DMA event whenever the channel buffer segment is empty. This is intended to be used to notify system software when an I/0 input operation has completed to memory (i.e. the buffer data write back has finished).
- Priority—This 2-bit field is used by the hardware to select a "scheduling" strategy for DMA operations. Two of the four possible values are currently defined; '00' will result in the DMA channel being scheduled by the PIF following every controller access to the buffer RAM, '11' will cause DMA to be schedule only when 4-Kbyte address boundaries are reached by peripheral controllers.
- Exception Event Group—A 4-bit encoded field that specifies one of the sixteen I/O event groups. This field is used in conjunction with an event number to report the occurrence channel exception conditions.
- Exception Event Number—A 4-bit encoded field which selects the specific event within an event group used to record the occurrence of a channel exception condition. A value of zero is defined to be "no exception".
- ReRead Enable—(SIOP2 and SIOP3 Only)—Enables "ReRead Mode" which allows peripherals to access previously read data within a 512- byte window without causing a channel buffer "flush and refetch".
- Channel Bank—(SIOP2 and SIOP3 Only)—The most significant bit of the channel number in Extended Channel Mode.
- Channel Number—A 6-bit value which serves as a unique channel identifier. Channels are indexed and selected according to channel number.

Buffer Configuration Register

The Buffer Configuration Register specifies the segment of the I/O buffer RAM assigned for use by a particular channel. This register is initialized by system software and never modified by the hardware. Its format and field definitions are given below.

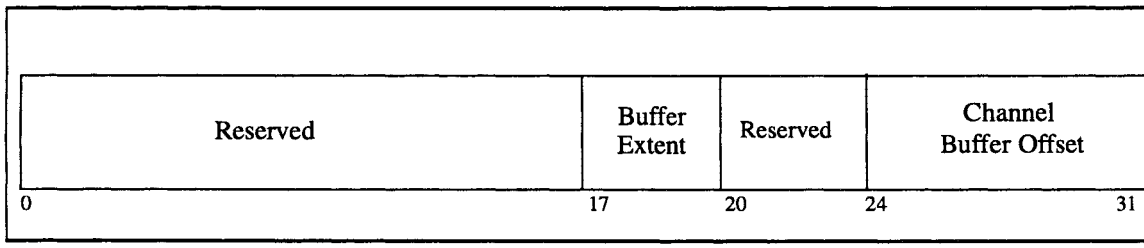

Figure 59 Buffer Configuration Register

- Buffer Extent—A 3-bit field containing a power of two value which determines the size of the channel buffer in 4k byte increments. Channel buffer sizes are restricted to 4-Kbyte, 8-Kbyte, 16-Kbyte,... 512-Kbyte. The maximum channel buffer is 512-Kbyte.
- Channel Buffer Offset—This 8-bit field defines the location of the channel buffer within the I/O buffer RAM. The value of the field is the number of 4-Kbyte blocks from the top (i.e. address zero) of the RAM where the channel buffer begins. The channel buffers be aligned within the Buffer RAM modulo their size. For example 8-Kbyte channel buffers must begin on 8-Kbyte boundaries, 64-Kbyte buffers on 64-Kbyte boundaries etc.

Memory interface logical address

The memory interface logical address is the next sequential logical address accessed by the DMA hardware. In connection with the peripheral interface logical address, this is used to control the buffer mapping to memory and peripheral space, track valid data and available space within the buffer, and detect access discontinuities. The memory interface logical address may be read

or written by software and is updated by hardware as DMA operations are executed.

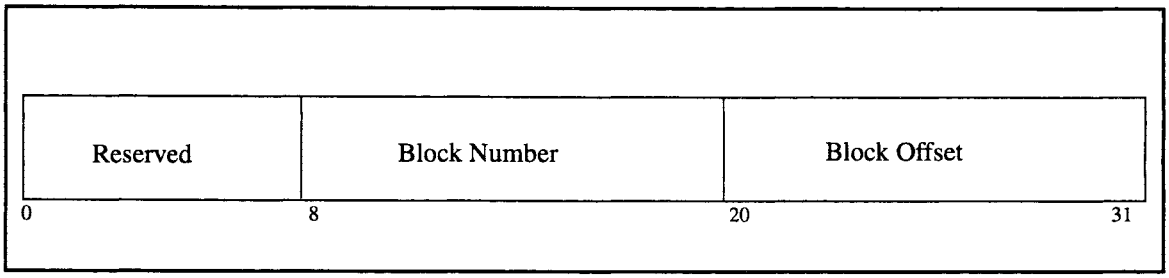

Figure 60 Memory interface logical address

Memory interface status register

The memory interface status register reflects the current state of the memory interface hardware. It is interpreted and updated by the I/O hardware in order to manage and record the progress of data transfer operations. It is then read by software to obtain completion status and exception information.

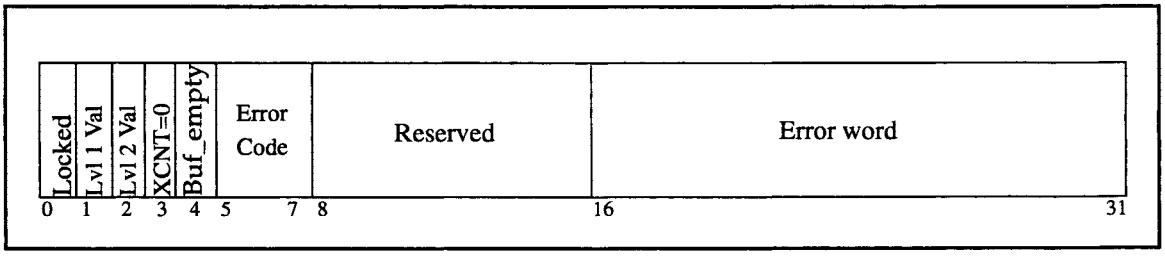

Figure 61 Memory interface status register

- Locked—Inhibits further channel operations (DMA transfers) when set. It is set by hardware in response to certain exceptions, or if the "locked" bit in the channel configuration register is set. It may also be set and/or cleared directly by software.
- Level 1 Valid—Indicates that the level 1 buffer table entries in the channel state table are valid and relate to the current memory interface logical address. It is set and cleared by the ATU hardware and may be set or cleared by software.
- Level 2 Valid—Indicates that the level 2 buffer table entries in the channel state table are valid and relate to the current memory interface logical address. It is set and cleared by the ATU hardware and may be set or cleared by software.

- XCNT=0—Indicates that the current value of the DMA prefetch counter is zero.
- Buf_empty—Indicates that the MIF and PIF logical addresses are identical.
- Error code—3-bit field indicating MIF channel status encode as:
    * 000—Nominal operating mode, no error.
    * 001—Invalid channel type
    * 010—DMA response error
    * 011—Invalid BTE
    * 100—Reserved
    * 101—Reserved
    * 110—DMA time-out
    * 111—Reserved

Error word—The word (E0) captured from an error response packet.

Peripheral interface logical address

The peripheral interface logical address is the next sequential logical address accessed by the peripheral interface hardware. This value—in connection with the memory interface logical address—is used to control the buffer mapping to memory and peripheral space, track valid data and available space within the buffer and detect access discontinuities. The memory interface logical address may be read or written by software and is updated by hardware as data transfers are executed by I/O controllers.

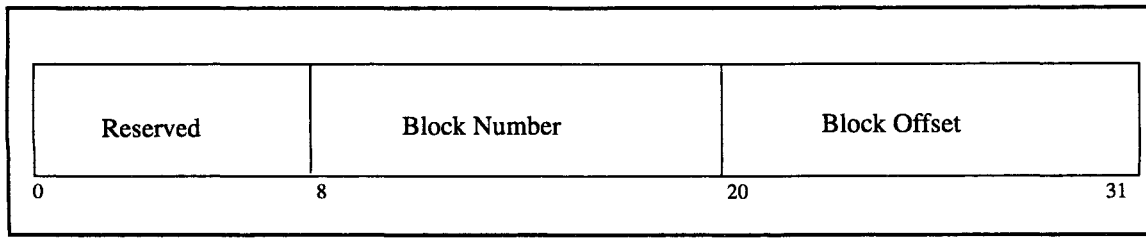

Figure 62 Peripheral interface logical address

Peripheral interface status register

The peripheral interface status register reflects the current state of the peripheral interface hardware. It is interpreted and updated by the I/O hardware to manage and record the progress of data transfer operations. It is read by software to obtain completion status and exception information and may be written by software.

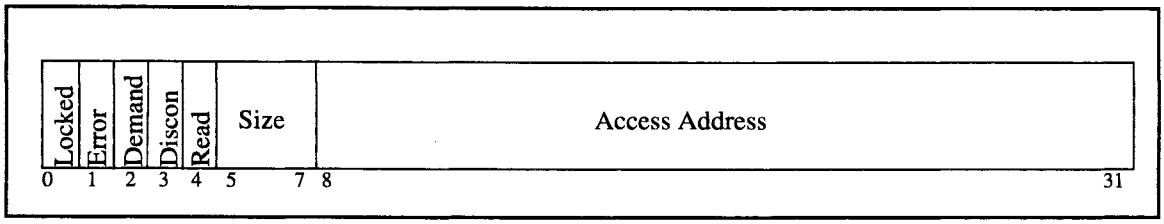

Figure 63 Peripheral interface status register

- Locked—Inhibits further channel operations (buffer read/writes) from the peripheral interface when set. It is set by hardware in response to certain exception conditions, or if the "Locked" bit in the channel configuration register is set when a peripheral access is attempted. It may also be set and/or cleared directly by software.
- Error—Indicates that an exception condition associated with a peripheral controller-initiated channel operation has been detected.
- Demand Fetch—Indicates for an output channel that a peripheral made an access outside the range of prefetched data.
- Discontiguous access—Indicates that a peripheral made a nonsequential access.
- Read—Records the Type field of the last peripheral Mbus transfer. In the case of an error, this value is preserved for diagnostic use.
- Size—Records the Size field of the last peripheral Mbus transfer. In the event of an error, this value is preserved for diagnostic use.
- Access Address—Records the 24 least significant bit s of the last peripheral Mbus transfer address.

DMA control register

The DMA control register tracks the amount of data transferred via the channel DMA hardware. It may be programmed to assert an I/O event when the transfer count reaches zero. The register is initialized by software and may be read or written at any time. The transfer count field is updated by the DMA hardware as transfer operations progress.

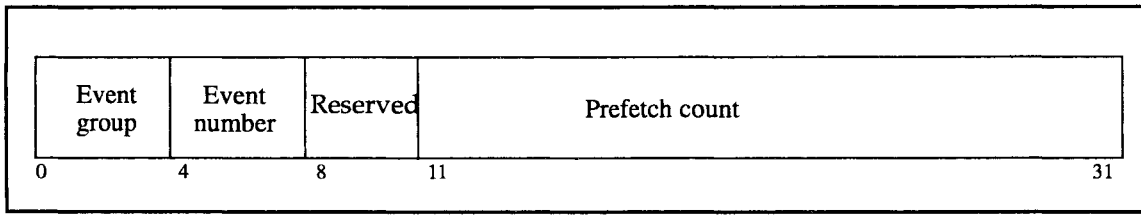

Figure 64 DMA control register

- Event group—Specifies the event group used when a DMA event occurs. DMA events include the prefetch count going to zero and the channel buffer segment "going empty."
- Event number—Specifies the event number to be used for a channel DMA event. Zero indicates event notification is not enabled.
- Prefetch count—Decrements from its initial value whenever the DMA hardware executes a read data transfer from main memory. The count value is in 16-byte blocks. When the count reaches zero, no further transfers are performed. Write operations have no effect on the counter.

Buffer table base pointer

This register contains the data structure base physical address used in logical-to-physical address translation. The translation tree begins at a 4-Kbyte boundary, although the actual location of valid entries is determined by an index field in the logical address (refer to "Address translation unit (ATU)" section on page 176). This register is loaded by system software. It may be read or written at any time and is not modified by the I/O hardware.

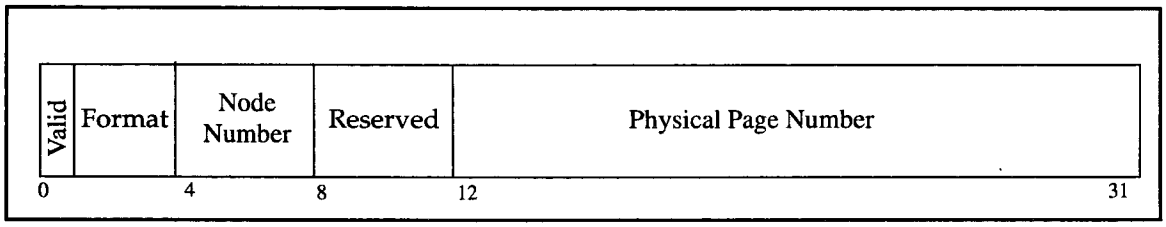

Figure 65 Buffer table base pointer

- Valid—Indicates that the buffer table base pointer has been initialized and contains a pointer to a valid buffer entry tree.
- Format—Specifies one of eight possible buffer table formats.
- Physical page number—contains the physical address of the level 1 entry block of a buffer translation table. The actual format of the physical page number (node bits and offset) is implementation-dependent and is determined by the format field.

Level 1 buffer table entry

The channel state table contains four level 1 buffer table entries. Each points to a block of level 2 entries that must be aligned on a 4-Kbyte boundary. They may be read or written by software and fetched from main memory by the address translation hardware as needed (refer to "Address translation unit (ATU)" section on page 176).

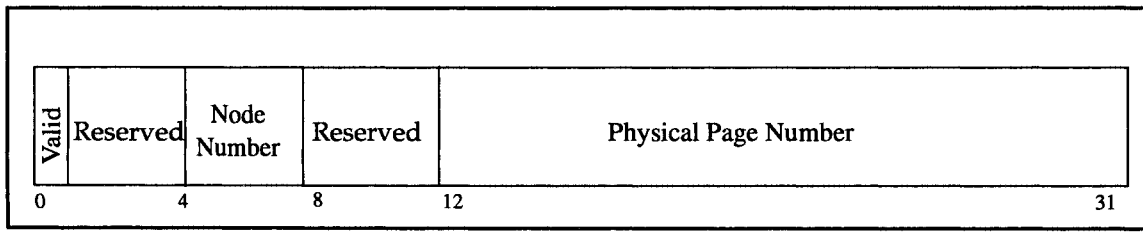

Figure 66 Level 1 buffer table entry

- Valid—Indicates the level 1 buffer table entry has been initialized and contains a pointer to a valid block of level 2 entries.
- Node Number—A 4-bit field that identifies the physical node within the system containing the buffer page.
- Physical page number—Contains the physical address of a level 2 entry block of a buffer translation table. The actual format of the physical page number (node bits and offset) is implementation-dependent and is specified by the format field in the buffer table base pointer.

Level 2 buffer table entry

The channel state table contains sixteen level 2 buffer table entries. Each points to a 4-Kbyte block of physical memory to be used in I/O data transfer operations. They may be read or written by software and are fetched from main memory by the address translation hardware as needed (refer to "Address translation unit (ATU)" section on page 176 for details of how these entries are used).

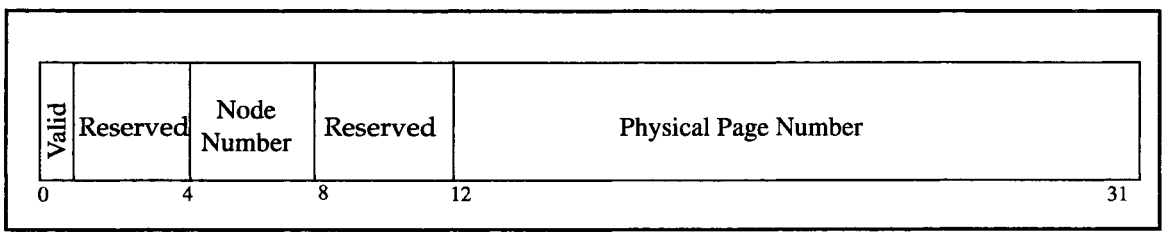

Figure 67 Level 2 buffer table entry

- Valid—Indicates the level 2 buffer table entry has been initialized and contains a pointer to a valid I/O buffer block in main system memory.
- Node Number—A 4-bit field which identifies the physical node within the system containing the buffer page.
- Physical page number—Contains the physical address of the first entry of a 4-Kbyte block of system memory used in channel DMA transfers. The actual format of the physical page number (node bits and offset) is implementation-dependent and specified by the format field in the buffer table base pointer.

Direct memory access (DMA)

The I/O unit DMA logic serves as a data transfer agent for other system elements. DMA transfers are generally component parts of higher level I/O transactions involving system software and external devices/interconnects. The DMA hardware, however, transfers data blocks of a specified size and address between main memory and the I/O unit RAM buffer. The DMA hardware does not interpret, modify, realign, or conditionally redirect the transferred data.

The transfer type is defined by the channel configuration register and data transfer requests from the peripheral interface. If buffered mode is enabled (channel types C-F), the DMA element will transfer data automatically, up to the limit of the transfer count or space/valid data available in the buffer, beginning with the current (or initial) memory interface logical address value. If the channel type is interlocked, DMA transfers execute only in response to data read/write requests received from the peripheral interface. Interlocked transfers are restricted to the size and address of the peripheral access request.

DMA operations are *scheduled* (queued for execution) when the associated channel configuration register is updated. Software uses this feature to initiate channel activity; the peripheral interface does so when it requires DMA services. An active channel encountering an empty or full channel buffer is rescheduled when the peripheral interface updates the channel state table (as a result of subsequent buffer reads or writes). Active DMA operations may also be multiplexed to maximize overall throughput. The specific method and criteria used to multiplex channel operations is implementation-dependent.

Address translation unit (ATU)

I/O DMA transactions are defined and managed within a contiguous logical address space. The actual system memory access, however, occurs within the SPP physical address space, which is interleaved across single nodes on 64-byte boundaries. These may be randomly scattered throughout the complex at a 4-Kbyte page granularity.

The SPP I/O units contain a hardware-based address translation unit that performs required mapping. The address map is a two-level tree structure that describes the physical memory areas to be used by a particular I/O channel. The channel state tables contain a pointer to the root of the map structure and *encached* entries from both levels.

Level 1 and level 2 entries can be loaded directly into the state table by software, or they are fetched by the ATU logic as needed to support channel DMA transfers. The table entries are always fetched as an aligned block (four at a time for level 1 entries and sixteen at a time for level 2). For example, if a translation miss occurs for level 2 entry number 53 (as selected by the level 2 index field of the logical DMA address), entries 48 through 63 are fetched and loaded into the channel state table. Whether the state table has encached the entry block containing the correct entry for the memory interface logical address is indicated by the level 1 and level 2 valid bits of the memory interface status register.

Individual entries within a a block may or may not be valid, as indicated by the BTE "valid" bit. The base pointer and the level 1 entries reference 4-Kbyte blocks of lower-level entries. Only those entries indexed by the DMA logical address must be valid. If a selected entry (one required for DMA execution) is invalid, a channel exception is recorded.

The exact format of the physical address, the translation tree entries and the I/O DMA logical address may vary across implementations and/or product generations. A 3-bit field in the buffer table base pointer allows up to eight different address and table formats to be defined. Specifics regarding currently defined formats can be found in the implementation sections of this document.

I/O buffer

The I/O unit buffer is a physically undifferentiated RAM accessible to the I/O hardware elements, as well as system software. It provides general storage for I/O control structures and data. The buffer is logically partitioned at a 4-Kbyte block (page) granularity. Software is responsible for global allocation and management of the buffer RAM.

Channel state tables

The first buffer page(s)—beginning with zero—is reserved for channel state table storage. The maximum number of channels defined per unit (in blocks of 32 channels) determines how many pages are allocated for this purpose.

Channel data buffer

A variable-sized segment of the total buffer RAM may be assigned to each I/O channel. A channel buffer segment functions as a cache of the logical DMA address space. At any one time, the segment is mapped to a single contiguous region of the address space. Head and tail pointers in the channel state table define the valid data area within the mapped segment.

Shared memory

Shared memory is similar to channel data buffers in that position and extent are defined by a channel state table. No DMA to or from main memory occurs, however. The buffer segment serves as shared memory region between SPP processors and external devices. It is accessed as noncoherent memory by the CPUs and as peripheral interface address space by devices/controllers.

Peripheral interface

The I/O unit peripheral interface serves as the connection point for external devices and communication links for the SPP system. The inboard side of the interface—channel selection, buffer access, mapping to system address space, event assertion—is architecturally defined. The specific features and functions supported at the outboard side are dependent upon the peripheral connection and device type(s). These may vary according to I/O adapter configuration and/or version.

The heart of the peripheral interface is a 64-bit synchronous address and data bus. The bus protocol defines a packet transfer transaction set. Access is centrally arbitrated and parity is optional. The bus has a 36-bit address mapped to system memory for transactions initiated by both SPP processors and external devices. The function set and mapping methods employed by the three major components of the interface logic are discussed below. Except for address mapping, the low-level operation of the peripheral bus is invisible to the I/O programming interface.

Master interface

Outward accesses (load/store requests targeted to storage elements residing within external devices or controllers) are executed by the peripheral bus master interface logic. These accesses are initiated by SPP processors and mapped through the peripheral address segment of noncoherent memory space using the map entries programmed in I/O unit CSR space (refer to the "Memory map" section of this chapter). They are restricted to single aligned word, halfword, and byte accesses and are not queued.

Slave interface

The peripheral bus slave interface of the I/O unit responds to access requests from attached I/O device controllers, then transforms them into channel operations. Channel selection is done using the upper (most significant) twelve bits of the starting address for a transfer request. This results in a 24-bit or 16-Mbyte range available for individual channel operations. Slave interface transfers may be of variable length and are mapped and routed to/from the I/O buffer according to channel state parameters. A single bus transaction cannot cross a 16-Mbyte boundary. If this occurs, an exception condition is identified. Logically sequential transfer requests spanning a 16-Mbyte boundary select separate channels. The slave interface logic also coordinates buffer management and DMA operation in conjunction with the memory interface logic.

Bus contriler

The peripheral interface bus controller is the central arbiter for the peripheral bus. It services access requests from bus masters according to a general arbitration strategy that ensures fair-share access for all masters. The specific request-grant algorithm is implementation-dependent.

The bus control logic is also responsible for error detection. It monitors all transactions, illegal conditions, time-outs, and data errors. Parity is optional for peripheral bus data masters and slaves. The bus controller determines whether parity is being provided by the data source of a transaction. If it is, the controller checks the data and asserts an I/O exception if an error is detected. If the data was not supplied with parity, the bus controller will generate proper parity and drive it on the bus as the data is forwarded to the I/O buffer or external connections.

External connection

The SPP I/O architecture deliberately avoids specifying the functional characteristics and programming interface for a particular class (or classes) of external connection. The generic nature of the peripheral interface bus and address space is intended to provide maximum flexibility with regard to SPP I/O configuration requirements. The interface exported to the external world by the SPP I/O units is sufficiently versatile and scalable to support all current or anticipated I/O platforms. Both direct connections from the peripheral bus and indirect attachment via industry standard board level interfaces (SBus, VME I/O, or PCI controllers, for example) can be incorporated into Convex SPP systems. The specifics of I/O connection management are therefore implementation-dependent. For example, additional mapping between the peripheral address space and an external address space defined by the I/O connection may be required; the programming interface is obviously dependent upon the device/controller type.

Exemplar compliance

The Exemplar I/O hardware is implemented within a single system circuit board. There are currently three models of the I/O circuit board: the SIOP1, the SIOP2, and the SIOP3.

Many of the fundamental characteristics of the Exemplar I/O subsystem are not architectural in nature. The I/O hardware design has been heavily influenced by practical considerations such as third party product availability and physical packaging constraints. This is particularly true in the area of external connections to the I/O unit peripheral interface.

Exemplar I/O configuration limits, unsupported feature and functions, an overview of the device connectivity strategy, and the differences in the two I/O adapter designs are briefly presented in this section.

Finally, a general discussion of availability issues and how they relate to Exemplar I/O is included in this section.

Implementation restrictions and limits

All three designs implement a true subset of the architectural features presented in this document. A summary of the areas limited on these designs is presented below.

Physical addressing

In general, the space allotted within I/O control structures for physical memory addresses is larger than that required for Exemplar. This has been done to facilitate ease of migration to later-generation products, which may have larger physical memory addresses. As a result, many of the address-related fields could extend into reserved fields.

I/O channels

The number of channels on the SIOP1 design is 64, as compared with the architectural limit of 4096. The number of channels on the SIOP2 and SIOP3 designs is 128.

Only three of the defined channel types are supported: shared memory (type 8), buffered input (type D), and buffered output (type E).

DMA logic

While an arbitrary number of DMA channels may be "active" simultaneously, the hardware can only support outstanding requests for a single channel. All pending transfers must be completed before another channel may access system memory.

I/O unit buffer

The total size of the Exemplar I/O unit RAM buffer is 1 Mbyte.

Peripheral interface

The SIOP1 board has a total of four IEEE 1496 Standard SBus connectors for external device/interface controllers. The controllers are positioned as *daughter* cards that extend to a bulkhead opening in the Exemplar chassis.

SBus transactions are transformed into peripheral bus operations by the I/O unit hardware. While not an integral part of SPP architecture, the incorporation of SBus has a pervasive impact on Exemplar system design. The hardware, software, and electromechanical aspects of SBus connections are fully documented in the Exemplar I/O functional specification.

The SIOP2 supplements the SBus connectors with four 100-pin microstrip connectors on the unit's internal peripheral bus. These provide a higher bandwidth connection for other standard or custom controllers, and they may also be used to add SBus slots.

The SIOP3 is similar to the SIOP2, but includes enhancements that allow the use of PCI Mezzanine Card (PMC) I/O adapters in any of the four expansion slots. The SIOP3 can support any mix of PMC, SBus, or custom controllers in the expansion slots in addition to the four main SBus slots.

Availability

The general philosophy is one of fault isolation rather than fault tolerance. The system comprises sets of hierarchically connected functional units or modules. At every level of the hierarchy there typically are multiple instances of the same module type. Every node has an I/O adapter with two I/O units, each of which connect to multiple controllers. I/O modules are not inherently redundant, but could be configured for redundancy at added cost. They are, however, distributed and functionally independent. In a multinode SPP system, there should be no single point of catastrophic failure for the I/O system. Individual components are still susceptible to failure, but the system can continue to operate at reduced capacity.

Failure prediction and time to repair are also important aspects of availability. Both are enhanced by the use of thorough diagnostic tests. The Exemplar I/O hardware is designed with testability as a functional requirement. A comprehensive test suite has been developed to support SIOP and its attached controllers and devices. Hardware features such as JTAG boundary scan at the component level, software access to hardware state and test capabilities such as DMA transfer "chaining and copyback" have been incorporated into the design.

The modular nature of the I/O subsystem provides good fault identification. Functional failures can be traced to a particular branch of the connection hierarchy. If all modules below a certain level become inoperable, but peer modules at that level continue to function, the location of the failure is usually obvious.

Performance monitors

9

Overview

This chapter describes the Exemplar performance-monitoring mechanisms, the low-level programming interface that accesses and implements those mechanisms, and the underlying hardware supporting them within the Exemplar.

Performance factors

Application performance in an Exemplar is dependent upon several factors. Exemplar provides ways to measure each of the principal factors. The measurements indicate the overall results and provide data that enables programmers to identify the algorithmic changes providing the improvement in overall results. Some of the important factors, and ideas for their measurement, are:

- Cache-hit rate—Algorithms must optimize cache use to perform well. Consequently the Exemplar provides data on overall hit ratios, hit ratios per CPU, hit ratios over time, cache miss trace data, and data that tells which program statements are causing the most misses.
- Communication costs—Communication costs includes communication within and across hypernodes. Concerns in communication include the ratio of local-to-global memory use, the memory access patterns of particular code sections, the topology of how the application is laid out across many hypernodes, and the communication costs between threads.
- Efficient parallel algorithms—Parallel algorithms pose both validity and performance problems for the programmer, and often prove more difficult to debug than single-threaded applications. Useful tools include trace data correlated between threads, deadlock detection, synchronization statistics, measurement of effective parallelism, granularity measurements of parallel regions, and lock order enforcement.

- Programming model— Exemplar is designed to accommodate both shared-memory and message-passing programming models efficiently.
- I/O performance—The largest I/O factor is data transfer rates in the disk subsystem. Of particular interest are peak measurements, as most I/O is done in a burst mode. Also, information about disk access patterns should be available.

The Exemplar architecture provides means of measuring the key parameters in each of the above areas. The process of measuring performance is an intrusive process and can impact the system performance.

SPP architecture

This section describes the architecturally defined hardware facilities used in performance measurement.

Cache measurements

Particular attention has been given to obtain accurate measurements of cache effectiveness.

The CPU does not inform external hardware of cache hits, so it is not possible to count them directly in hardware. All types of cache misses are counted (individually or as a group). The average latency incurred by the processor for various types of memory accesses is directly measurable with PMON_EVENT (SPP1000 only), PMON_LATENCY (the latency counter is internal to the CPU in the SPP1200 and SPP1600, but this difference is transparent to the user if the system-provided performance-monitor software calls are used), and PMON_CONTROL (SPP1000 only). These counters can also be used to gather statistics on coherency-request frequency and latency.

The CTI cache-hit rate is computed by programming the multiple runs of performance monitoring registers to record CTI cache-hits and misses, respectively. The latency incurred for hits and misses is also available.

As all of these facilities are enabled or disabled, measurements over specific intervals (such as routines) can be made under software control. The operating system maps the per CPU facilities into the application address space. This enables low-overhead reads and writes by performance monitoring code. The per hypernode facilities are privileged since they are under the simultaneous control of multiple CPUs.

The minimal number of implemented bits in the event and latency counters were calculated to allow sampling only once per second, with events happening at the maximum rate without counter overflow. Software accumulates the results of these counters in memory at least once per second to avoid overflow.

Communication cost measurements

The ratio of local-to-global memory accesses is measured in a single pass using two runs with different PMON_EVENT counter configuration.

Obtaining quantitative data about which hypernodes are communicating with other hypernodes is difficult. If programs use a message-passing paradigm, it is easy to keep histograms of the count and cumulative length of messages sent between each pair of hypernodes.

An interrupt-based facility similar to that described above can be used to gather statistics about the distribution of a CPU memory accesses over hypernodes and CTI paths. The hypernode and memory unit within the hypernode are provided in the PMON_INT_STATUS register.

Parallel programming measurements

The most important facilities provided for parallel program development include the very large memory and TIME_TOC register. Used together, these can implement low-overhead, time-stamped trace data stored in hypernode private memory. The time-stamped trace data from each hypernode can be merged in a post processing step to provide an accurate global picture, with event sequences in the 5-to-10 microsecond range.

The TIME_TOC is also used to time-stamp transmitted messages. The receiver can then determine the transmission time by subtracting time-stamp from its current time.

The high-resolution CPU timer is used to measure events of short duration that need not be correlated across hypernodes (execution time of basic blocks or code fragments and remote procedure calls, in particular).

Hardware is provided for process virtual-time measurement. Maintenance of process virtual time is expensive in software since the number of executing threads is adjusted frequently (as a result of each system call, for example), and without hardware support, binary semaphore locking is required. Accurate process virtual time can be compared with the total thread CPU time to get an approximation of parallel speedup.

Deadlock detection, synchronization statistics, lock order enforcement, and other aspects of semaphore performance and validity analysis are left to software techniques.

Programming model-specific measurements

The messaging software produces (optionally) a histogram of message count, and bytes transferred by sender and receiver CPU number. Statistics on the amount of traffic vs. the interconnect path can also be computed. Messaging overhead can be computed by time-stamping at the sender and receiver.

The predominate overhead in the shared memory model is cache management. Other useful measurements are:

- Synchronization overhead (especially barrier synchronization overhead)
- Cost of a fork/join pair
- Time to fill a new CPU cache

These can all be measured with the CPU timer.

I/O measurements

I/O measurements can be made in software, if given the accurate timers to do time-stamps. Trace data can be used to record latency through different parts of the I/O path. Recording the average and peak I/O bandwidth consumed is also an easy operation. Software must also consider measuring the distribution of the I/O load across the SPP.

Histograms

PMON_CONTROL provides the ability to interrupt the processor after the logging of each cache miss. This facility enables *histogramming* of cache misses by program counter (IAOQ) address as follows:

- Upon receipt of each interrupt, the interrupt handler uses the PC in the IIAOQ to index a table in main memory. Each entry in the table contains a count and latency field for a particular range of PC addresses.
- The IIAOQ may not contain the address that actually missed the caches, as other instructions may have completed before the interrupt was taken, and the CPU may be executing several instructions simultaneously.
- The PMON_EVENT and PMON_LATENCY registers are then cleared in preparation for the next miss, and the interrupt handler returns to the application.

This technique slows the application substantially, but should provide an accurate picture of which statements in the program are causing cache misses. With correct software, interrupt latency are relatively low in this implementation.

Low-level programming interface

The Exemplar architecture provides application-level access to all the performance registers described in this chapter through the Architectural Interface Library (AIL). The AIL is a set of subroutines that manipulate the hardware on behalf of a user program. All functions follow the standard C entry/exit argument-passing convention.

Table 38 Architectural interface library entry points

Entry point	Description
unsigned int toc_read()	Returns the current 64-bit value of the time of century counter.
cnx_toc_t toc_per_sec_read()	Returns the number of time of century counter ticks per second.
int pvt_enable()	Allocates a process virtual timer to the calling process and initializes it to zero.
int pvt_disable()	Turns off and deallocates the process virtual timer for the calling process.
cnx_pvt_t pvt_read()	Returns the current 64-bit value of the process virtual timer.
unsigned int pvt_per_sec_read()	Returns the number of process virtual timer ticks per second.
int ttr_enable()	Enables the thread timer for the calling thread and initializes the timer to zero.
int ttr_disable()	Turns off the thread timer of the calling thread.
cnx_ttr_t ttr_read()	Returns the 64-bit value of the timer of the calling thread.
cnx_ttr_t ttr_write()	Adjust the current value of the timer of the calling thread.
cnx_ttr_t ttr_incr()	Adjust the current value of the timer of the calling thread.
unsigned int ttr_per_sec_read()	Returns the number of thread timer ticks per second.
int pmon_enable	Allocates a pmon register of the specified type and initializes it to zero.
int pmon_disable()	Deallocates the specified pmon register.
int pmon_readv()	Reads the value of the specified pmon registers.

Table 38 Architectural interface library entry points —**(continued)**

Entry point	Description
int pmon_incrv()	Adjusts the current values of the specified pmon registers
int pmon_writev()	Adjusts the current values of the specified pmon registers
unsigned int pmon_per_sec_read()	Returns the number of ticks per second for pmon registers that measure latencies.

Application profiling

CXpa provides performance monitoring with no programming effort. CXpa profiles the following performance factors on a per-thread basis over user-selectable regions of application code without source modification:

- On all SPP Series architectures:
  - Wall clock time
  - CPU time
  - Concurrency factor (CPU/Wall clock time)
  - Iteration/execution counts
- On SPP1000 systems:
  - Locally resolved (CTI not used) cache-miss counts and latency
  - Remotely resolved (CTI used) cache-miss counts and latency
  - Locally and remotely resolved cache-miss counts and latency
  - Average cache miss latency
- On SPP1200 systems:
  - Data cache misses, accesses, and latency
  - Data cache hit rates
  - Average data cache miss latency
  - Instruction cache misses and latency
  - Average instruction cache miss latency
  - Instructions completed
  - Clock cycle
  - Average clock cycles per instruction
- On SPP1600 systems:
  - Locally resolved (CTI not used) cache-miss counts and latency
  - Remotely resolved (CTI used) cache-miss counts and latency
  - Locally and remotely resolved cache-miss counts and latency
  - Combinations of locally versus remotely resolved cache-miss counts for reads, writes, and both.
  - Data cache misses, accesses, and latency
  - Data cache hit rates
  - Average data cache miss latency
  - Instruction cache misses and latency
  - Average instruction cache miss latency

- Instructions completed
- Clock cycle
- Average clock cycles per instruction

After profiling, CXpa supports visualization of profiling results in 2D and 3D graphs. Refer to the CXpa Reference (DSW-253) or the cxpa man page for more information.

Exemplar performance monitor hardware

Key registers are used to record events in the system and enable performance measurement. The registers include:

- A time of century clock (TIME_TOC)
- A CPU-interval timer (CR16)
- A performance-monitor set for each CPU

The following section describes the structure and use of these registers.

Time of century counter (TIME_TOC)

Each CPU must have access to the TIME_TOC counter. It is not necessary for each CPU to implement its own counter; one will be implemented per hypernode. The access time for any CPU that would normally use that counter is less than 1 microsecond. A CPU within a hypernode can read the counter within that hypernode in 1 microsecond, assuming no contention.

TIME_TOC characteristics are as follows:

- Frequency does not change regardless of clock margin conditions.
- TIME_TOC period is 1 microsecond.
- TIME_TOC counters within an SPP are synchronized so that no more than the least significant bit varies between any two counters read at the same instant.
- TIME_TOC is 64 bits wide.
- TIME_TOC is readable, but not writable, at CPU-privilege level 3 (user level).

The service processor, or the CPUs (but not necessarily both), must be able to initialize the TIME_TOC registers on each hypernode in such a way that they become synchronized as required above. This implies that software can be used to synchronize the clocks, provided the accuracy requirements can be met.

The time of century match register (TIME_TOCMR) is provided for each TIME_TOC counter. It enables generation of an interrupt when the TIME_TOC reaches a predetermined value. Each TIME_TOCMR is continuously compared with its corresponding TIME_TOC; when they are equal, an interrupt is sent to all the CPUs serviced by that TIME_TOC by setting bit 31 in each CPU external interrupt register (EIR).

The TIME_TOCMR is readable and writable by any CPU within the hypernode only at CPU privilege level 0.

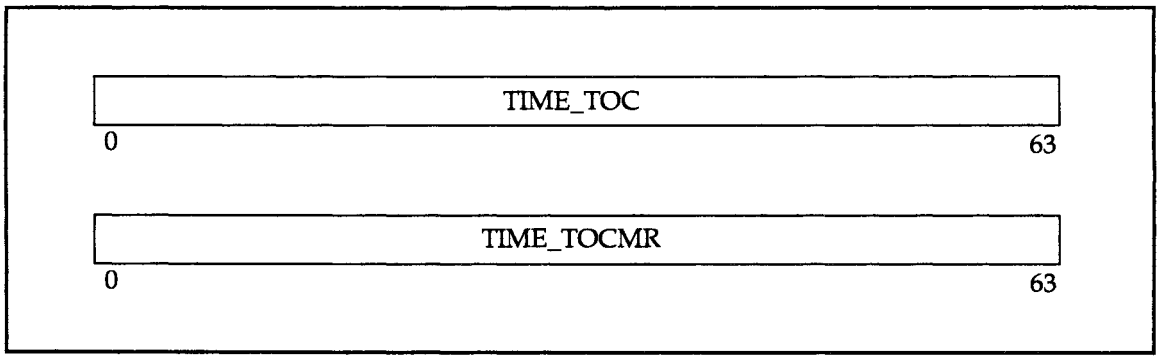

Figure 68 TIME_TOC and TIME_TOCMR registers

CPU interval timer (CR16)

The PA-RISC architecture provides a 32-bit timer in CR16 of each CPU. These timers count at a frequency between twice the peak instruction rate and half the peak instruction rate. These timers are not synchronized, nor can they be loaded by software. The timers may optionally generate an interrupt when the count reaches a certain value.

These timers are used for:

- Generation of periodic clock interrupts to the CPUs for scheduling purposes.
- Measurement of fine granularity time intervals within a CPU that do not have to be correlated to time of the other CPUs. An example of such a measurement would be the length of time necessary to execute a code segment.
- Implementation of a thread timer register (TTR) in software. The implementation strategy is described below.

CR16 is readable at CPU-privilege level 3 if the S bit in the PSW is zero. The operating system allows user-read access to this timer in order to use the TTR.

Performance-monitor set

The performance-monitor sets are different between the SPP1000 and the other Exemplar platforms. In the SPP1000, the set is external to the processor and is contained in the agent. The PA7200 processor, however, has built-in performance monitor registers located in its I/O space. While the internal performance-monitor set improves overall processor performance measurement, external registers are still required to measure those events external to the CPU. Performance monitoring tools are written to make these differences transparent to the user. The following sections describe those performance-monitor sets that are external to the CPU.

SPP1000 performance-monitor set

In the SPP1000, the performance-monitor set provided for each CPU is shown in Figure 69 and consists the following registers:

- PMON_LATENCY—a 32-bit latency counter
- PMON_EVENT—a 24-bit event counter
- PMON_CONTROL—the performance control register
- PMON_INT_STATUS—the interrupt status register

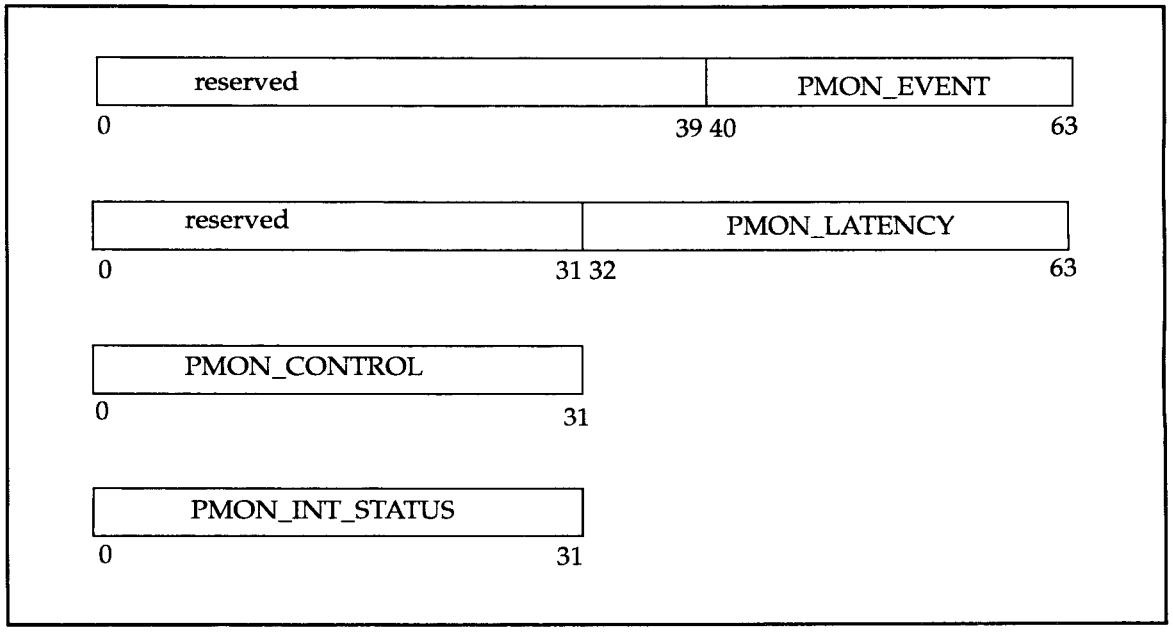

Figure 69 SPP1000 performance monitor register set per CPU

The latency counter records the time a processor waits for a cache miss to be served. The latency counter is clocked by a 15ns clock. If the values in the latency counter is 0x003f, then 945ns have been spent processing cache misses.

The 24-bit event counter counts the number of cache misses and miscellaneous events. The control registers determine which events increment it. The event counter does not roll over. Once the MS0 bit is set, it stops counting.

The control register governs the behavior of the event counter. Setting bits in the control register determine the kind of performance measurement.

The four major kinds of performance monitoring are:

- Cache-miss latency and event counting
- Message-send counting
- Received coherency-request counting
- Sent coherency-request counting

A performance-monitor set can only be used to measure one kind of performance at a time. If undefined combinations are set in the control register, the integrity of the event counter is lost.

The latency counter and event counter are architecturally defined to be 64-bits wide, but Exemplar implementations of these counters is less than 64 bits. When the MSB of each of the counters is set, the counters stop counting. The MSB of the latency counter is bit 32, and the MSB of the event counter is bit 40. This does not mean that the largest value for the Latency counter is 0x80000000. The event counter increments on each cache miss, increasing by the number of cycles that block the processor during the cache miss.

PMON_CONTROL register

The PMON_CONTROL register determines when the latency and event counters increment. Not every control register setting is sensible, but all are allowed. The functionality of each control bit is described in Table 39.

Table 39 SPP1000 performance monitor control register

Bit	Description
0	Enables instrumentation of cache miss to private memory
1	Reserved
2	Enables instrumentation of cache miss to shared memory
3	Reserved
4	Coherency request received, read indirect or invalidate received
5	Coherency request sent, read indirect or invalidate sent
6	Messages send to crossbar message is sent)
7	Used CTI Bit. CTI was used to fetch the data for the line given to the processor. For accesses to far-shared blocks, this is a CTI cache miss bit.
8	Didn't use CTI. CTI was not needed to fetch the data given to the processor. For far- shared blocks, this is a CTI cache-hit bit.
9-25	Reserved
26	Write request, write miss or load and clear
27	Read requests, read miss or instruction cache miss
28	Reserved
29	Interrupt sent but not finished being serviced
30	Reserved
31	Interrupt enable for a cache miss that increments the event counter

Figure 70 shows the bits used in setting the performance monitor control register to monitor cache miss latency and event counting.

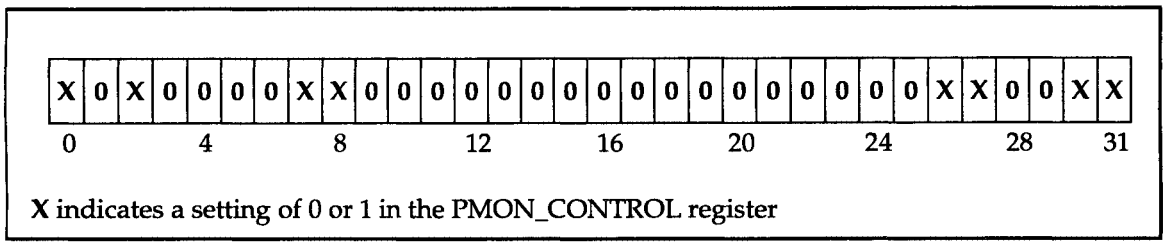

Figure 70 SPP1000 cache miss latency and event counting settings

To set the PMON_CONTROL register to count only send messages, set bit 6 as shown in Figure 71. This counts the number of message send packets a processor transmits to memory.

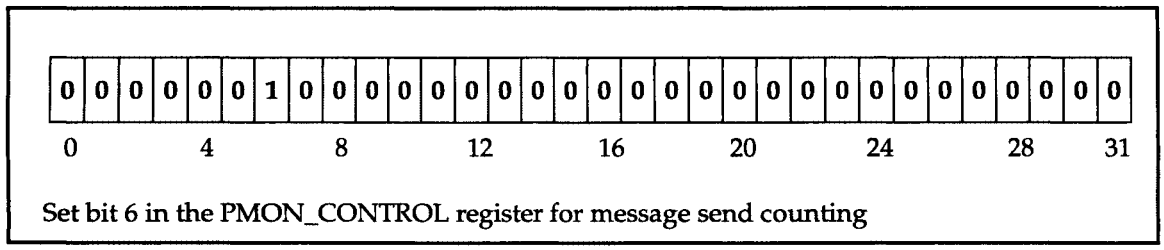

Figure 71 SPP1000 message send counting

To set the PMON_CONTROL register to count received coherency requests, set bit 4 as shown in Figure 72.

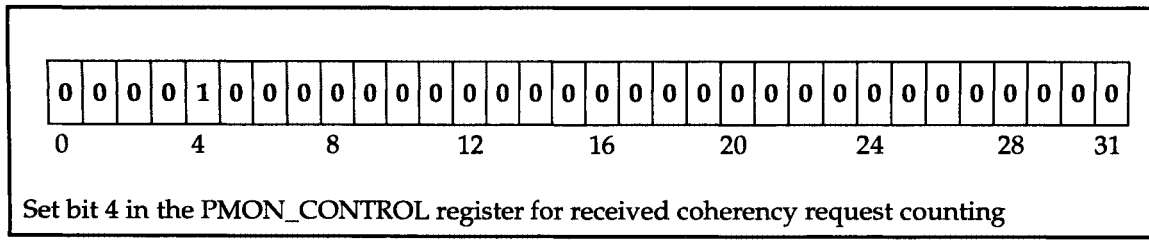

Figure 72 SPP1000 received coherency request counting

To set the PMON_CONTROL register to count sent coherency requests, set bit 5 as shown in Figure 73.

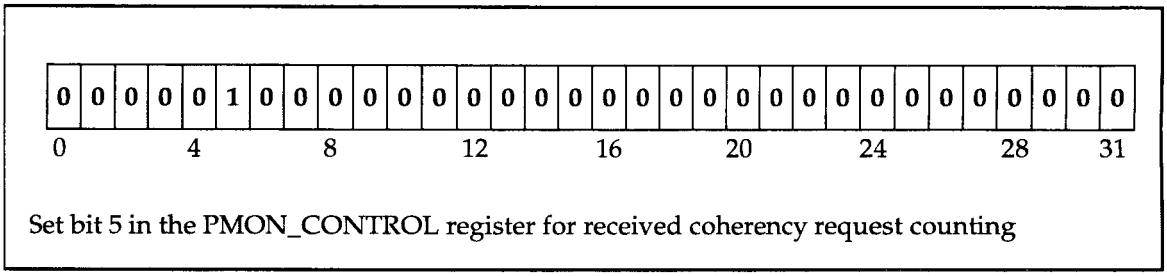

Figure 73 SPP1000 sent coherency request counting

PMON_INT_STATUS register

PMON_INT_STATUS provides information on the last cache-miss to the CPU (when bit 31 of PMON_CONTROL is set). Software may use these fields to trace the frequency of memory references to various hypernodes vs. local.

The hypernode field of PMON_INT_STATUS contains the target hypernode of the reference causing the interrupt. The unit field of PMON_INT_STATUS contains the target memory unit within the hypernode of the reference causing the interrupt. The type field contains the type of reference causing the interrupt, as shown in Table 40.

Table 40 SPP1000 interrupt status register

Bits	Description
0-11	Reserved
12-15	Physical node number of the HOME node
16-21	Reserved
22-23	Physical ring number of the HOME ring
24-28	Reserved
29	Set if CTI was used to fetch the data given to the processor
30	Sharing level of the reference 0 -Private, 1- Shared
31	Reserved

The PMON_EVENT, PMON_LATENCY, PMON_CONTROL, and PMON_ADDRESS registers are readable and writable.

PMON_INT_STATUS is a read-only register, updated by hardware, and is available to the user for read operations. Bits not used in these registers return zeros, and are ignored on writes. It is at the discretion of the operating system whether these registers are only accessible at CPU-privilege level 0, or also at other privilege levels.

Accessing 64-bit registers

Although most of the registers defined for performance monitoring have been specified as 64 bits wide, Exemplar only accesses these registers using 32-bit, noncached loads, stores, or load and clears. Registers having implemented bits in both the most significant and least significant 32 bits must use two loads or stores to access them. In this case, the hardware places no restrictions on the software as to which half of the register is accessed first.

Special attention should be given when reading a timer or counter that is wider than 32 bits. An algorithm similar to that in the example shown below should be used to ensure both parts read separately reflect the count accurately:

```
for (x=0,z=1; x != z; ) {
x = counter.upper;
y = counter.lower;
z = counter.upper;
}
```

SPP1200 performance-monitor set

Performance measurement for the SPP1200 is accomplished with registers both external and internal to the PA7200 CPU.

External registers

The external registers function similarly to those in the SPP1000, except that latency counting is accomplished by the internal performance monitor registers. See the "Internal registers" section on page 206.

The external performance-monitor set for each CPU is shown in Figure 74 and consists the following registers:

- PMON_CONTROL—a read/write register that controls interrupts.
- PMON_INT_STATUS—a read only status register used to track the destinations of misses.

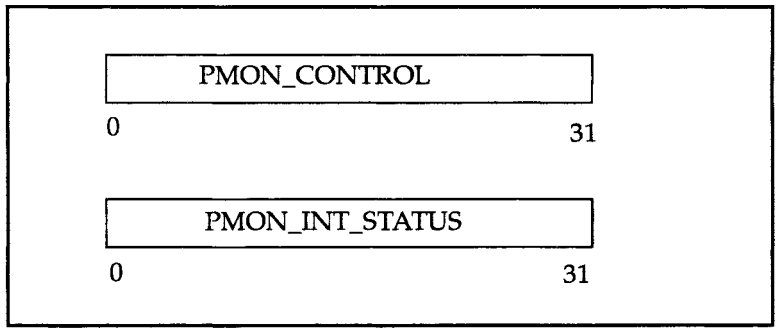

Figure 74 SPP1200 performance monitor external register set per CPU

PMON_CONTROL register

The functionality of each control bit in the PMON_CONTROL register is described in Table 41.

Table 41 SPP1200 PMON_CONTROL register

Bit	Description
1-6	Reserved
7	Enable—Data return event which used CTI
8	Enable—Data return event which did not use CTI
11-25	Reserved
26	Enable—Write request (DRD_PR) event
27	Enable—Data read request (DRD_SH) event
28	Enable—Instruction read (IRD_SH) event
29	Interrupt sent status (sent=1)
30	Reserved
31	Interrupt enable for a cache miss that increments the event counter

If an enabled data-miss event occurs and the interrupt enable (bit 31 in the PMON_CONTROL register) is set, the CPU is sent a level 30 interrupt. The *interrupt sent* status bit (29) is set when that interrupt is sent. It locks the PMON_INT_STATUS register from logging more information and prevents any more interrupts. Prefetch events do not cause interrupts since the PMON_INT_STATUS register only logs misses.

PMON_INT_STATUS register

PMON_INT_STATUS provides information on the last cache-miss to the CPU (when bit 31 of PMON_CONTROL is set). Software may use these fields to trace the frequency of memory references to various hypernodes vs. local references.

The PMON_INT_STATUS register contains the home node and ring numbers of the enabled event. It locks on the first enabled data-miss event until the *interrupt-sent* bit is cleared. It also contains a bit which indicates whether or not CTI was used to return the data.

Table 42 SPP1200 iInterrupt status register

Bits	Description
0-11	Reserved
12-15	Physical node number of the HOME node
16-21	Reserved
22-23	Physical ring number of the HOME ring
24-28	Reserved
29	CTI used for data return (CTI_used=1)
30-31	Reserved

Enabled instruction-read events can also be used to trigger interrupts. Normally, data and instruction events are not enabled at the same time. For instruction-read events, the *CTI used* log bit is not applicable since the instruction returns are not monitored. Logged instruction events can not be determined reliably as to whether they are miss or a prefetch.

The PMON_CONTROL register is readable and writable. The PMON_INT_STATUS register is a read-only register, updated by hardware, and is available to the user for read operations. Bits not used in these registers return zeros, and are ignored on writes. It is at the discretion of the operating system whether these registers are only accessible at CPU-privilege level 0, or also at other privilege levels.

Internal registers

The PA7200 CPU has four, 32-bit performance monitor registers used to access the Performance Monitor Coprocessor.

- Performance Control register (PCR)—determines how each performance counter is used, either an event counter or a latency counter (do not confuse this register with the PMON_CONTROL register in "PMON_CONTROL register" section on page 204).
- Performance Counter 0—(PC0) used as either an event counter as a latency counter. Only Performance Counter 0 can be used as a latency counter.
- Performance Counters 1-2—(PC1 and PC2) used as event counters.

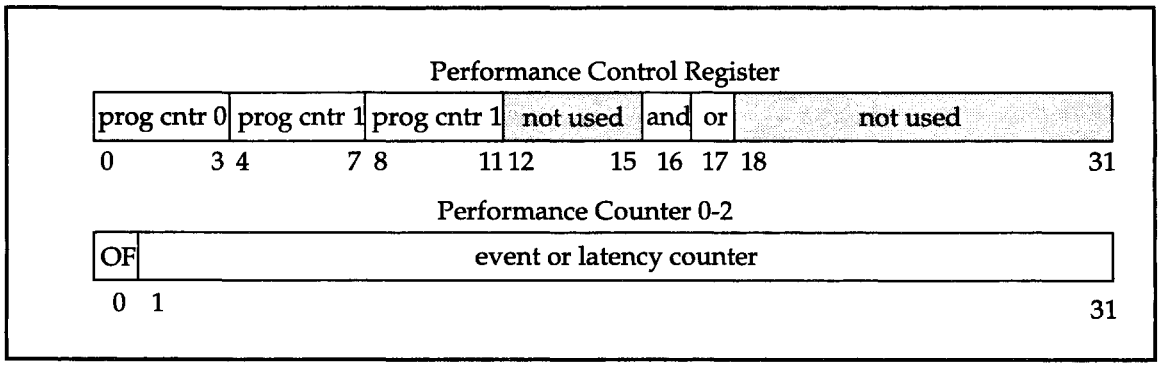

Figure 75 SPP1200 internal CPU registers

The three performance counter registers (PC0-2) use 31 bits for counting and bit 0 as a sticky overflow bit. The counters are cleared by writing zeros to the register. The counter can also be set to any nonzero value.

Performance Control register

The PCR has three fields that determine the event or latency to count. Counters are enabled and disabled with either coprocessor arm and disarm instructions or by writes directly to the PCR. Simultaneous start of the three performance counters on all CPUs is accomplished with a single broadcast write.

Table 43 SPP1200 performance Control register format

Bit	Description
0-3	Performance Counter 0 programming bits
4-7	Performance Counter 1 programming bits
8-11	Performance Counter 2 programming bits
12-15	Reserved
16	AND'ed with CPU arm/disarm: count enable override
17	OR'ed with CPU arm/disarm: alternative to CPU arm/disarm
18-31	Reserved

Enabling the counters with coprocessor instructions is accomplished by setting bit 16 of the PCR and starting or stopping the counters with the arm and disarm instructions, respectively.

Enabling the counters with I/O instructions is accomplished by setting both bit 16 and 17 of the PCR to a 1. Clearing bit 16 stops the counters.

The following equation specifies count enable control:

```
count enable = (ARM/DISARM + PCR[17] * PCR[16])
```

Event Counters

PC0-2 registers count once each time the corresponding event signal occurs. The event to count is programmed into the appropriate field of the PCR. Table 44 shows the number and corresponding count event.

Table 44 SPP1200 count events

Number	Event
0	Software handled DTLB misses
1	Software handled ITLB misses
2	Data cache misses
4	Instruction cache misses
5	Completed instructions
6	Clocks CPU cycles
7	Data cache accesses

Latency Counter

Only PC0 can be programmed to count one of the four possible latencies shown in Table 45. The latency counter has four lstart count signals and one stop. Only one start signal can be active at any time as programmed by the PMR. When the start signal occurs, the counter starts and continues to count until the stop signal occurs. The counter counts the number of stall cycles in the CPU pipeline.

Table 45 SPP1200 latency count events

Number	Latency event
12	Software handled DTLB misses
13	Software handled ITLB misses
14	Data cache misses (includes prefetch hit-outstanding hits and I/O accesses)
15	Instruction cache misses (includes I/O accesses)

Note

Events 12 and 13 do not give accurate results. The number of cycles from the assertion of the start signal until the first DMISS of the TLB trap handler is counted.

SPP1600 performance-monitor set

Like that in the SPP1200, performance measurement for the SPP1600 is accomplished with registers both internal and external to the PA7200 CPU.

The external registers function similarly to those in the SPP1000. The main difference is the latency counter is not in the agent, but in the PA7200 CPU itself. Only the external registers are described in this document. Refer to the Hewlett-Packard PA7200 Processor Module Family Technical Reference Manual for a detailed description of the internal CPU registers.

The external performance-monitor set for each CPU is shown in Figure 76 and consists the following registers:

- PMON_EVENT—a 24-bit event counter. This is a write-to-clear register that the operating system accumulates in a virtual 64-bit counter.
- PMON_CONTROL—a read/write register that controls events and interrupts.
- PMON_INT_STATUS—a read only status register used to track the destinations of misses.

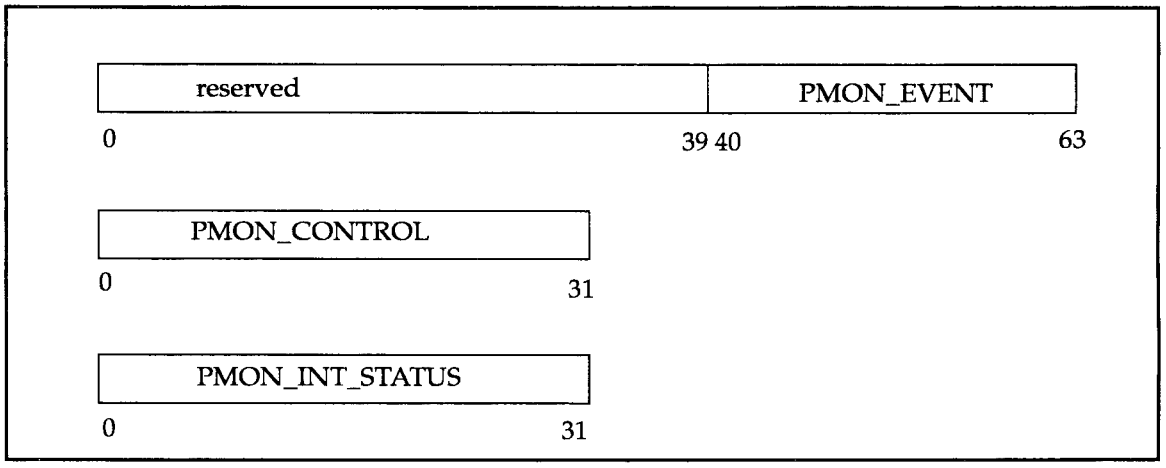

Figure 76 SPP1600 performance monitor external register set per CPU

PMON_EVENT counter

The 24-bit event counter counts the number of events enabled by the PMON_CONTROL register. The event counter does not roll over. Once the MS0 bit (bit 40) is set, it stops counting.

PMON_CONTROL register

The PMON_CONTROL register determines which events increment the PMON_EVENT counter. Not every control register setting is sensible, but all are allowed. The functionality of each control bit is described in Table 46 and the valid combinations of the enable bits are shown in Table 47.

Table 46 SPP1600 performance monitor control register

Bit	Description
0	Enables instrumentation of cache miss to private memory
1	Reserved
2	Enables instrumentation of cache miss to shared memory
3	Reserved
4	Coherency request received, read indirect or invalidate received
5	Coherency request sent, read indirect or invalidate sent
6	Messages sent to crossbar
7	Enable—Data return event which used SCI
8	Enable—Data return event which did not use SCI
9	Enable—Prefetch enable
10	Enable—Non-prefetch enable
11-25	Reserved
26	Enable—Write request (DRD_PR) event
27	Enable—Data read request (DRD_SH) event
28	Enable—Instruction read (IRD_SH) event
29	Interrupt sent status (sent=1)
30	Reserved
31	Interrupt enable for a cache miss that increments the event counter

Table 47 SPP1600 PMON_CONTROL register enable bit event definitions

S	N	R	W	M	P	Event definition
0	0	x	x	x	x	No events monitored
x	x	0	0	x	x	No events monitored
x	x	x	x	0	0	No events monitored
1	1	1	1	1	1	All data requests (misses and prefetches)
1	1	1	0	1	1	Data requests caused by loads
1	1	0	1	1	1	Data requests caused by stores
1	1	1	1	1	0	All misses
1	1	1	0	1	0	Misses caused by loads
1	1	0	1	1	0	Misses caused by stores
1	1	1	1	0	1	All prefetches
1	1	1	0	0	1	Prefetches, caused by loads
1	1	0	1	0	1	Prefetches, caused by stores
0	1	1	1	1	1	Data requests resolved locally
0	1	1	0	1	1	Data requests resolved locally, caused by loads
0	1	0	1	1	1	Data requests resolved locally, caused by stores
0	1	1	1	1	0	Misses resolved locally
0	1	1	0	1	0	Misses resolved locally, caused by loads
0	1	0	1	1	0	Misses resolved locally, caused by stores
0	1	1	1	0	1	Prefetches resolved locally
0	1	1	0	0	1	Prefetches resolved locally, caused by loads
0	1	0	1	0	1	Prefetches resolved locally, caused by stores
1	0	1	1	1	1	Data requests resolved remotely
1	0	1	0	1	1	Data requests resolved remotely, caused by loads
1	0	0	1	1	1	Data requests resolved remotely, caused by stores
1	0	1	1	1	0	Misses resolved remotely
1	0	1	0	1	0	Misses resolved remotely, caused by loads
1	0	0	1	1	0	Misses resolved remotely, caused by stores
1	0	1	1	0	1	Prefetches resolved remotely
1	0	1	0	0	1	Prefetches resolved remotely, caused by loads
1	0	0	1	0	1	Prefetches resolved remotely, caused by stores

S = SCI used
N = SCI not used
R = Read (load)
W = Write (store)
M = Miss (non-prefetch)
P = Prefetch
0 = not enabled
1 = enabled
x = don't care

The instruction-read-event enable in the PMON_CONTROL register functions separately from the other six data-event enables. When enabled, all memory fetches caused by instruction misses and prefetches are counted, because the hardware can not distinguish instruction misses from instruction prefetches.

If an enabled data-miss event occurs and the interrupt enable (bit 31 in the PMON_CONTROL register) is set, the CPU is sent a level 30 interrupt (write_short addr=fffffc0000 data=0000001e). The *interrupt sent* status bit (29) is set when that interrupt is sent. It locks the PMON_INT_STATUS register from logging more information and prevents any more interrupts. Prefetch events do not cause interrupts since the PMON_INT_STATUS register only logs misses. The PMON_EVENT counter can be checked to see if the interrupt handler is missing events since the event counter will continue to count events.

PMON_INT_STATUS register

PMON_INT_STATUS provides information on the last cache-miss to the CPU (when bit 31 of PMON_CONTROL is set). Software may use these fields to trace the frequency of memory references to various hypernodes vs. local references.

The PMON_INT_STATUS register contains the home node and ring numbers of the enabled event. It locks on the first enabled data-miss event until the *interrupt-sent* bit is cleared. It also contains a bit which indicates whether or not SCI was used to return the data.

Table 48 SPP1600 interrupt status register

Bits	Description
0-11	Reserved
12-15	Physical node number of the HOME node
16-21	Reserved
22-23	Physical ring number of the HOME ring
24-28	Reserved
29	SCI used for data return (SCI_used=1)
30-31	Reserved

Enabled instruction-read events can also be used to trigger interrupts. Normally, data and instruction events are not enabled at the same time. For instruction-read events, the *SCI used* log bit is not applicable since the instruction returns are not monitored.

Logged instruction events can not be determined reliably as to whether they are miss or a prefetch.

PMON_EVENT, PMON_CONTROL, and PMON_ADDRESS registers are readable and writable. PMON_INT_STATUS is a read-only register, updated by hardware, and is available to the user for read operations. Bits not used in these registers return zeros, and are ignored on writes. It is at the operating system's discretion whether these registers are only accessible at CPU-privilege level 0, or also at other privilege levels.

CPU agent registers

Table 49 and Table 50 show the CPU-local physical address of the *per CPU* registers for the SPP1000 and the SPP1200 and SPP1600 respectively.

Table 49 CPU agent performance monitoring register for SPP1000

Register	CPU address (hex)
PMON_EVENT	0xffef1c00
PMON_LATENCY	0xffef1c08
PMON_CONTROL	0xffef1c10
PMON_INT_STATUS	0xffef1c14

Table 50 CPU agent performance monitoring register for SPP1200 and SPP1600

Register	CPU address (hex)
Performance Control (PA7200 internal)	0xffef0040
Performance Counter 0 (PA7200 internal)	0xffef0048
Performance Counter 1 (PA7200 internal)	0xffef0050
Performance Counter 2 (PA7200 internal)	0xffef0058
PMON_CONTROL	0xffef1c10
PMON_INT_STATUS	0xffef1c14

TIME registers

Table 51 shows the hypernode physical address of the TIME performance monitoring registers.

Table 51 TIME performance monitoring registers

Register	CPU address of most significant 32 bits (hex)	CPU address of least significant 32 bits (hex)
TIME_TOC	0xffee2000	0xffee2010
TIME_TOCMR	0xffee0140	0xffee0150
TIME_PVT	0xffee1010	0xffee1020
TIME_TC	unimplemented	0xffee1000

Software implementation of performance measurements

This section discusses software algorithms that are appropriate for making performance measurements and software interaction with the hardware.

Thread-timer register

Hardware is not provided for a thread-timer register (TTR). The operating system may implement a user-readable/writable TTR as described in this section.

Software implementation of the thread-timer register

For each thread, the operating system exports a CPU-private page to the application. Each thread in a process accesses a different page. The thread timer variables need not be in a dedicated page. They could, for example, be at one end of the thread stack so that part of the page is dedicated to thread timer variables and the remainder of the page is dedicated to general stack use.

Two variables must be in the thread-timer page:

- TTR_RESCHED—the value of the thread-timer register at the last reschedule (when the thread began execution). TTR_RESCHED is a 64-bit, unsigned integer value.
- CR16_RESCHED—the value of the CPU interval timer (CR16) at the last reschedule (the value of CR16 that corresponds to the thread time in TTR_RESCHED). TTR_RESCHED is a 32-bit, unsigned integer.

When the thread is running, its current thread time is given by:

```
TTR = CR16 - CR16_RESCHED + TTR_RESCHED;
```

The operating system must update the thread timer variables when entering and leaving the kernel. This is done as follows:

1. When entering the kernel on interrupt or system call, for example, the kernel must compute the value of the TTR according to the equation above and store it in the thread context.
2. When leaving the kernel of interrupt, system call or context switch, for example, the kernel must restore the thread time to TTR_RESCHED and copy CR16 into CR16_RESCHED. These actions must be done without interruption for maximum accuracy.

Note that user level code can update the thread-timer register by writing to TTR_RESCHED; deltas applied to TTR_RESCHED are carried forward in all future calculations of the TTR.

Using the software scheme assumes that TTR_RESCHED and CR_RESCHED are in the same cache line. Since the operating system writes these values when scheduling a thread, the CPU is likely to have the line encached in its on-chip cache. In this case, the following instruction sequence could be used to compute TTR:

```
LDIL    l%TTR_RESCHED,r3
LDO     r%TTR_RESCHED,r3 address of
        TTR_RESCHED-> r3
L1:
LDWS    0(r3), r4; top half of TTR_RESCHED
LDWS    4(r3), r5; bottom half TTR_RESCHED
LDWS    8(r3), r7; CR16_RESCHED
MFCTL   CR16,r6;   CR16
SUB     r6,r7,r6;  CR16 - CR16_RESCHED
ADD     r5,r6,r29; lower TTR_RESCHED + DELTA
ADDC    r4,r0,r28; upper TTR_RESCHED + DELTA
LDWS    0(r3),r6;  top half TTR_RESCHED
LDWS    4(r3),r7;  bottom half TTR_RESCHED
COMB,<> r4,r6,L1;  if top half changed
COMB,<> r5,r7,L1;  if bottom half changed
```

In their normal cache, the three loads hit the CPU cache and the branches are not taken. This results in 110 nanoseconds for computing TTR. The worst side effect is found in using seven registers. Since all values are loaded from the same cache line, only the first load can miss, resulting in a worst-case additional penalty of 470 nanoseconds. The branches are taken only when the operating system reschedules the process during the TTR calculation.

Service processor 10

Overview

This chapter defines the Exemplar service processor functions, including the low-level programming interfaces that access those functions and the underlying hardware supporting those functions.

Service processor architecture

The service processor functionality is distributed among the hypernodes in the Exemplar system. In the Exemplar architecture, each hypernode contains its own boot, diagnostic, and environment-monitoring hardware. The services related to these functions are provided by the hypernodes themselves. Some higher-level services, such as system consoles and auxiliary disk storage, are not distributed, but provided by a single *service hypernode.*

There are two aspects of the service processor functionality: the service hypernode interface to those functions resident within the hypernodes, and the PA-RISC processor interface to all service processor functions. Both are described in this section as they relate to Exemplar.

Overview

From the service processor perspective, an Exemplar system is a collection of hypernodes interconnected by the diagnostic and remote testing bus (DaRT), as illustrated in Figure 77. These hypernodes can be either standard Exemplar hypernodes (*system hypernodes*) or special *service hypernodes.* The implementation of the service hypernode evolves through the succession of Exemplar generations, and could consist of software running on a PC, a workstation, or another system hypernode.

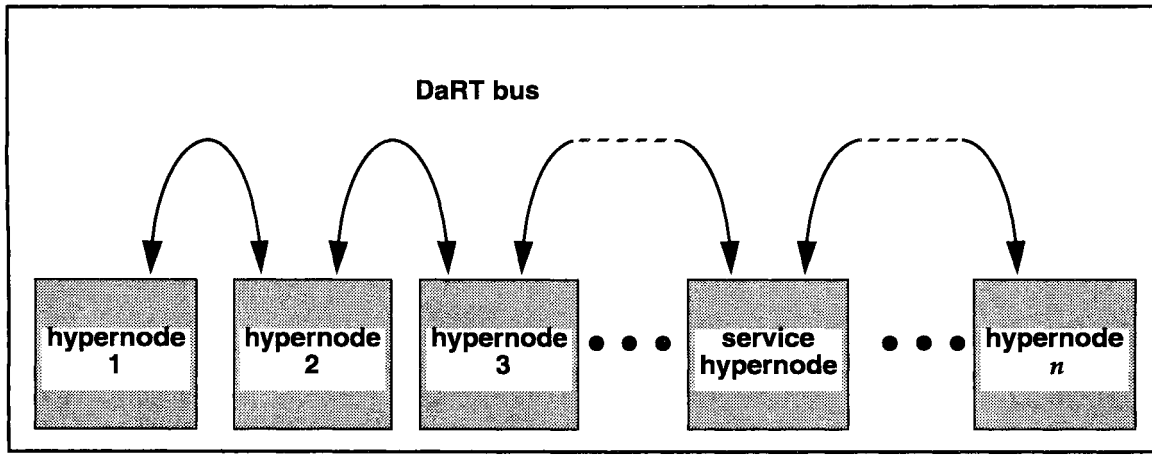

Figure 77 Exemplar service processor interface

The DaRT bus is a bidirectional, high performance data path between all hypernodes in a Exemplar system. It allows communication between the service hypernode and the system hypernodes for both service processor functions and diagnostic operations. Because the DaRT bus is independent from the main system internode CTI (CxRING), it provides reliable communication without consuming system bandwidth.

Service hypernode interface

This section describes the service processor interface between an Exemplar system service hypernode and its system hypernodes. This is not necessarily the interface that is available to the PA-RISC processors, however. In an Exemplar system, for example, this is the interface that a system hypernod utility processor uses to communicate with the service hypernode.

DaRT bus access

The service hypernode interface is based on the capability of the service hypernode to access the address space of every system hypernode via the DaRT bus. In some cases, the service hypernode directly accesses memory or CSRs in the target hypernode. In other cases, these hypernodes communicate by passing messages on the DaRT bus.

Communication on the DaRT bus requires each hypernode in the Exemplar system to have a unique hypernode ID. These hypernode IDs are determined at boot time using an implementation-specific algorithm. As part of this algorithm, the system hypernodes discover the ID of the service hypernode, and vice versa.

For direct access to system hypernode resources, the service hypernode uses the DaRT bus protocol to specify the addresses of the system hypernode memory and CSRs. These are 64-bit addresses consisting of a 16-bit hypernode ID (providing up to 64-Kbyte hypernodes) and a 48-bit node offset (providing 256 Tbytes of hypernode address space).

For message-based communication, each message is written to a mailbox location within the target node. Messages contain the following information:

- Operation to be performed
- Data for the operation
- Interrupt mode, which specifies whether an interrupt should be sent to the message sender when the operation is complete
- Interrupt address, which is the offset address within the message sender node to which an interrupt should be sent (if enabled)
- Interrupt level, which specifies the level of the interrupt to be sent (if enabled)

Hypernode reset control

An Exemplar system hypernode performs various reset functions at the request of the service hypernode. These functions implement hardware resets that can be performed on an entire hypernode, or on individual components within a hypernode. Typically, a reset of an entire hypernode is performed prior to booting the hypernode, as described in Chapter 11.

Reset control is available for the major functional components within a system hypernode. These components include:

- PA-RISC processors
- Memory controller hardware
- I/O channels
- I/O controllers

These reset functions provide the following capabilities:

- Control of a component while testing that component
- Control of a component while testing another component in the node
- Isolation of a faulty component when it is possible to resume normal hypernode operation with the faulty component held in its reset state

Each hypernode has two CSRs used for reset control: NODE_RESET and NODE_RUN. Both contain one bit for each component within the hypernode. Writing a "1" to a component bit in the NODE_RESET register places the component in a reset state. The component remains in the reset state until a "1" is written to its bit in the NODE_RUN register.

NODE_RESET

Figure 78 shows the general format of the NODE_RESET CSR. The only field in this register, *reset_array,* is a bit field where each bit corresponds to a component within the hypernode. The actual assignment of bits in this field is implementation-specific and is the same as that in the NODE_RUN CSR.

Writing a "1" to a bit in *reset_array* places the corresponding component into the reset state. Writing a "0" leaves it unchanged. To allow a component to run, use the NODE_RUN register, described below.

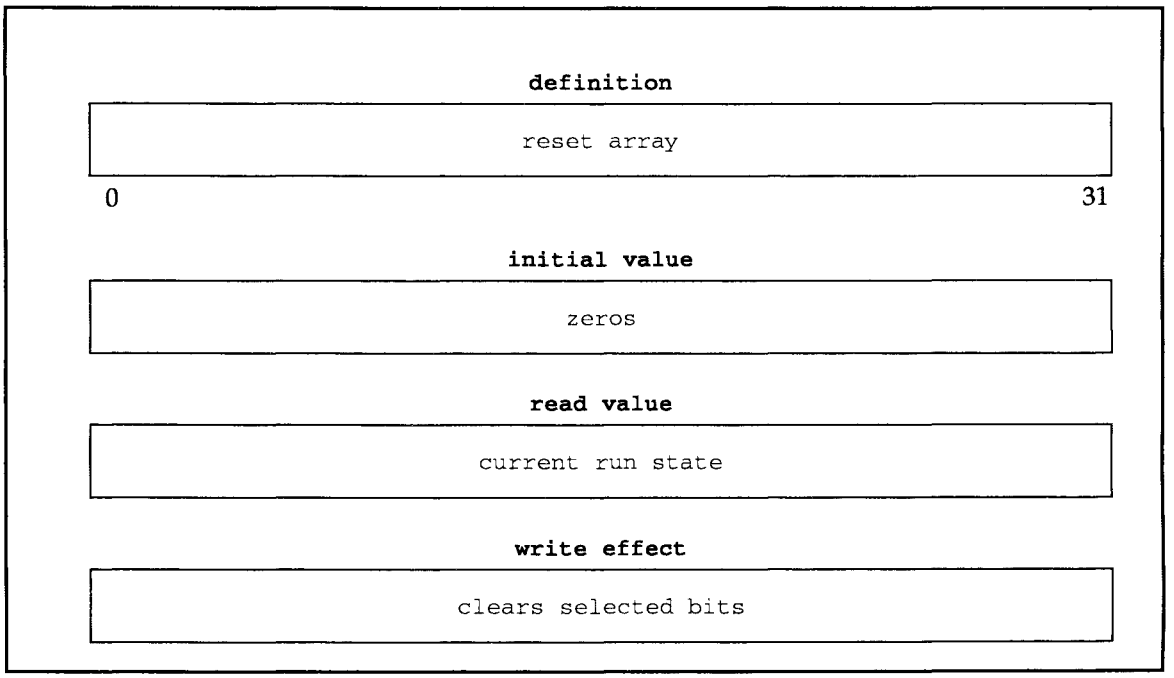

Figure 78 NODE_RESET format

Note

When reading this register, a value of "1" signifies that the component is NOT reset (that is, it is running).

NODE_RUN

Figure 79 shows the general format of the NODE_RUN CSR. The only field in this register, *run_array*, is a bit field where each bit corresponds to a component within the hypernode. The actual assignment of bits in this field is implementation-specific and is the same as that in the NODE_RESET CSR.

Writing a "1" to a bit in *run_array* allows the corresponding component to run. Writing a "0" leaves it unchanged. To reset a component, use the NODE_RESET register, described above.

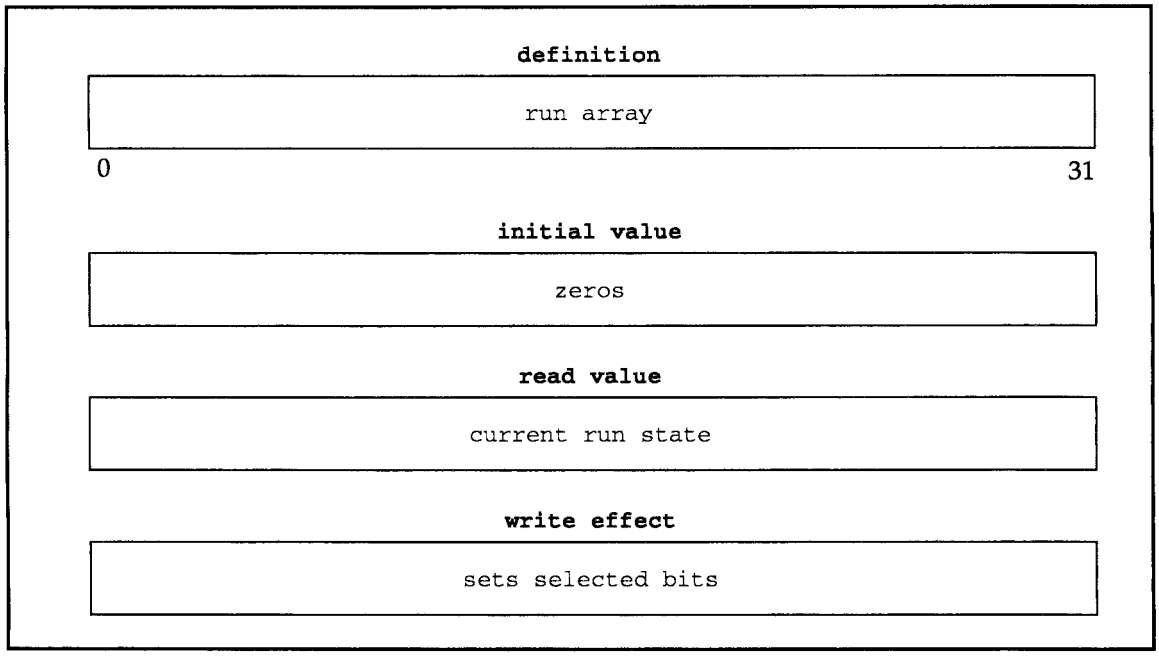

Figure 79 NODE_RUN format

Scan interface

Each Exemplar system hypernode provides access to its scan-based test facilities, which are based on the IEEE 1149.1 standard (also known as JTAG). This is a message-based interface where each system hypernode translates scan command messages from the service hypernode into hardware scan operations. These scan command messages include the operation to be performed (read, write, compare) and the data necessary for the operation.

Console interface

System *consoles* reside on the service hypernode. A console is a character-oriented display device at which an operator can read messages and enter commands. A system can have many different consoles; some are shared by multiple system hypernodes, and some are used by a single system hypernode.

System nodes access these consoles by sending console interface messages to the service hypernode. Each message contains a console ID, a console command, and console data. The console ID identifies which console is being referenced, and the command identifies the type of reference (read, write, status, or request).

Boot console interface

At system boot time, before the system I/O configuration is established, and before I/O devices have been initialized, system hypernode boot software uses a special *boot* console to communicate boot progress and error messages to the operator. This console is implemented as any other console in the Exemplar system, except that it is automatically created by the service hypernode at boot time. To determine the ID of the boot console, the system hypernode requests it using a console interface message.

The service hypernode can establish a single boot console to be shared among all system hypernodes, or it can partition a system so that one or more hypernodes share a boot console. When multiple hypernodes share a boot console, the hypernodes retain responsibility for coordination of their console activities.

Diagnostic event logger interface

Exemplar diagnostic event logging is performed with event logging messages on the DaRT bus, where the bus provides a one-way path from each system hypernode to the service hypernode. Any software running on a hypernode, including the OS, servers, application software, and stand-alone diagnostic software, may send event logging information.

The Exemplar diagnostic system uses event logging information to expedite fault detection, fault isolation, and fault recovery. The event log data structures include generic information common to all events such as date, time, event ID, event category, event source, and severity. Event-specific information is also included in the event data structure. Examples of this information include register contents, controller status information, error counter values, peripheral error/status information, diagnostic software test results, and hypernode configuration information.

Processor interface

In general, there are two types of PA-RISC interface: register-based and message-based. The PA-RISC uses the register-based interfaces by directly accessing the specified registers. For the message-based interfaces, the PA-RISC sends a message to the service processor. The details of the message-passing mechanism between the PA-RISC and the service processor are different for each Exemplar generation.

Console interface

The console interface for the PA-RISC processors is register based. To perform console operations such as establishing a new console, changing an existing parameters, and handling console traffic, the processors directly access console interface registers. There are four registers for this purpose: CONSOLE_READ, CONSOLE_WRITE, CONSOLE_CONTROL and CONSOLE_STATUS. These registers present a UART type of interface, which is consistent with traditional methods of dealing with system consoles.

In the console-interface CSRs, the *console_ID* value 0x001 is reserved for the boot console. The PA-RISC processors access this console in much the same manner as used for other consoles. The processors do not need to open the boot console, however. If multiple hypernodes access the boot console, operator keystrokes are sent to the hypernode that sent characters to this console last. The PA-RISC processors must coordinate among themselves to provide an orderly handling of boot-console traffic.

CONSOLE_READ

The PA-RISC uses the CONSOLE_READ register to read characters that have been entered by the operator on one of the consoles. This register fields are defined as follows (see Figure 80):

- *console_ID*—the ID of the console in which the *char* character was entered.
- *char*—the value of the last character entered in the console specified by *console_ID*.

A system hypernode receives only characters from consoles which it has created, and from the boot console, as described above. If a system hypernode has more than one open console, it is the responsibility of the hypernode software to de-multiplex the incoming stream of characters.

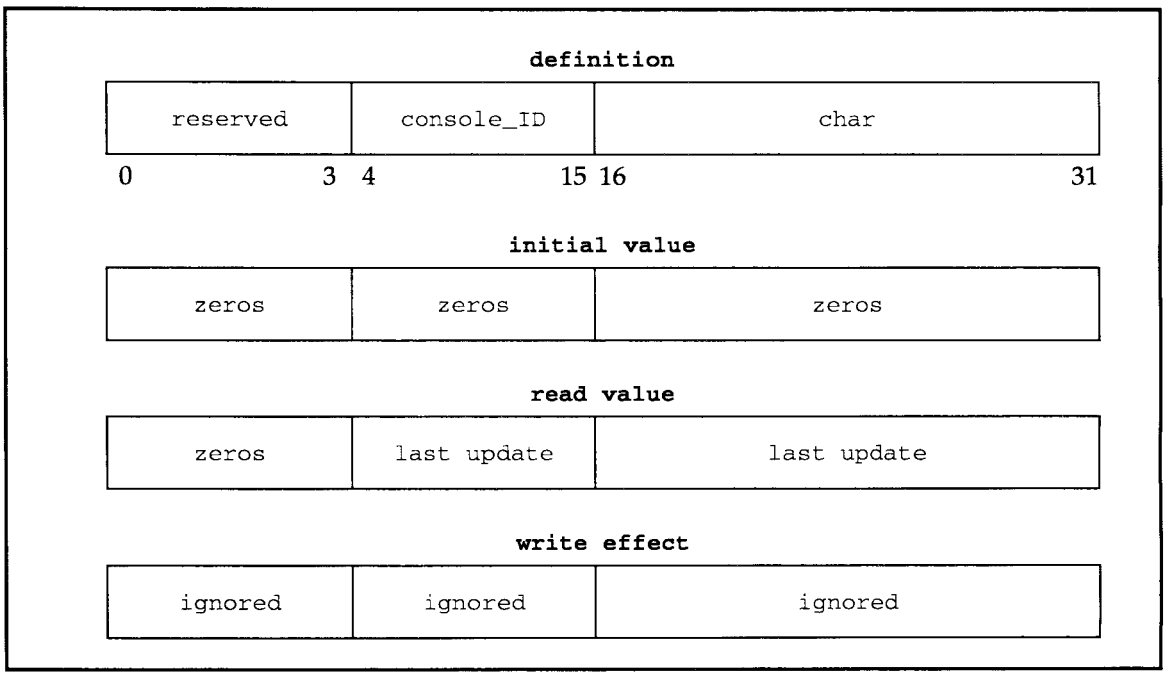

Figure 80 CONSOLE_READ format

CONSOLE_WRITE

The PA_RISC uses the CONSOLE_WRITE register to send characters to a console. The format of this register is shown in Figure 81. Its fields are defined as follows:

- *console_ID*— the ID of the console to which the *char* character is to be written.
- *char*—the character to be written to the console specified in *console_ID*.

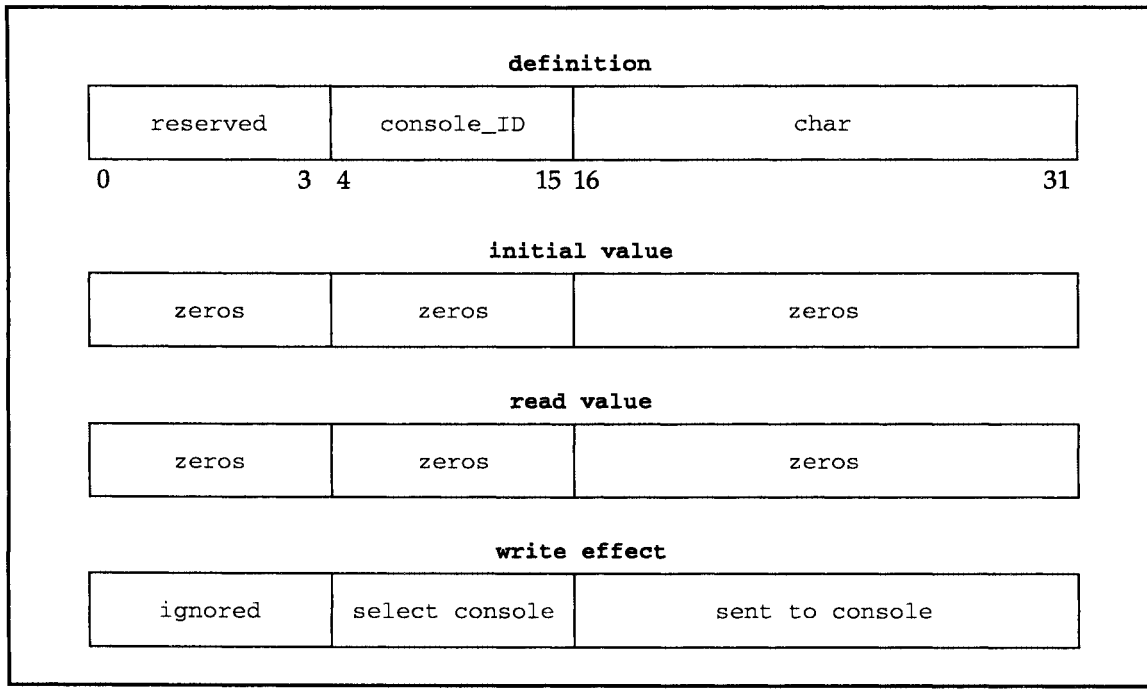

Figure 81 CONSOLE_WRITE format

CONSOLE_CONTROL

The PA_RISC uses the CONSOLE_CONTROL register to open, close, and modify the parameters of consoles. The fields of this register are defined as follows (see Figure 82):

- *console_ID*—the ID of the console for which this operation should be performed. This field is ignored for the *open* operation.
- *target (tgt)*—the identifier for the PA-RISC processor that should be interrupted when the next character from the *console_ID* console arrives in the CONSOLE_READ register (if interrupts are enabled via *mode*).
- *level*—the level of the interrupt that should be generated if interrupts are enabled via the *mode* field.
- *mode (m)*—the interrupt mode for this console, which specifies whether the PA-RISC should be interrupted when a character arrives in the CONSOLE_READ register. Writing a "1" to this field specifies that the PA-RISC should be interrupted. Writing a "0" specifies that no interrupt should be generated.
- *operation (op)*—the operation to be performed with this CONSOLE_CONTROL register access.

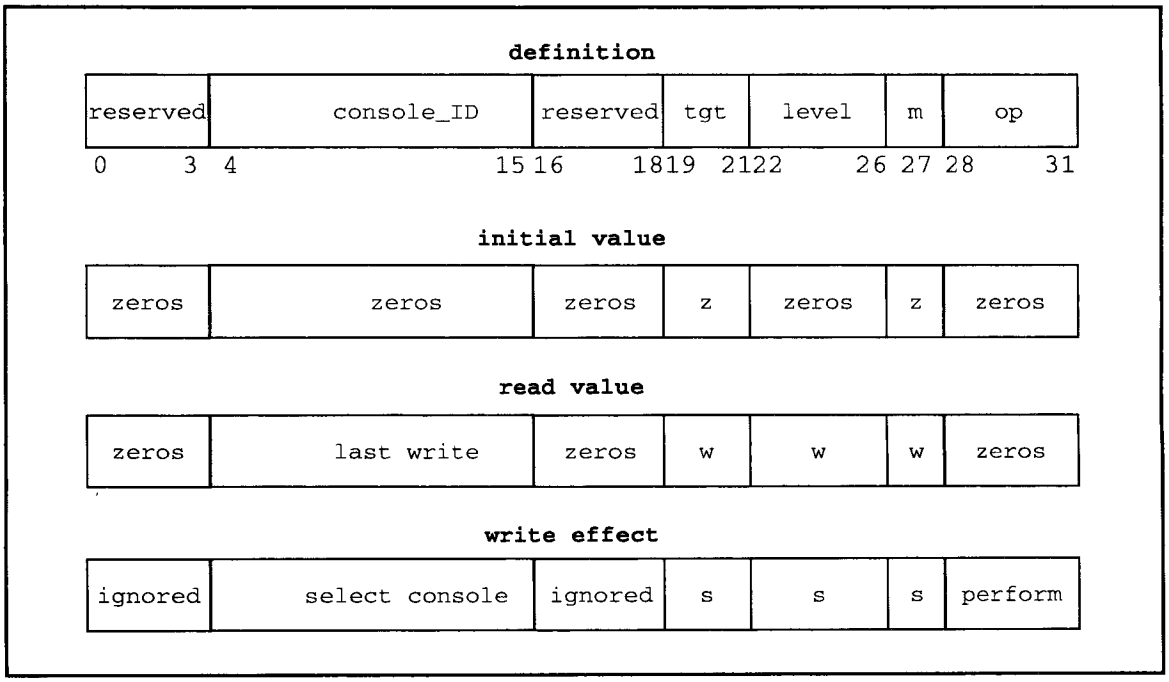

Figure 82 CONSOLE_CONTROL register

CONSOLE_STATUS

The PA_RISC uses the CONSOLE_STATUS register to determine the status of the CONSOLE_READ and CONSOLE_WRITE CSRs. This register fields are defined as follows (see Figure 83):

- *console_ID*—the ID of the console to which the status field value applies.
- *Status*—status of the most recent attempt to open or close a console. This field also indicates any errors that have occurred while writing to a console.
- *write_rdy (wr)*—set to "1" whenever the CONSOLE_WRITE register is ready to accept another character. Set to "0" when the CONSOLE_WRITE register is written.
- *read_rdy (rr)*—set to "1" whenever a valid, unread character is available in the CONSOLE_READ register. Cleared to "0" when the CONSOLE_READ register is read.

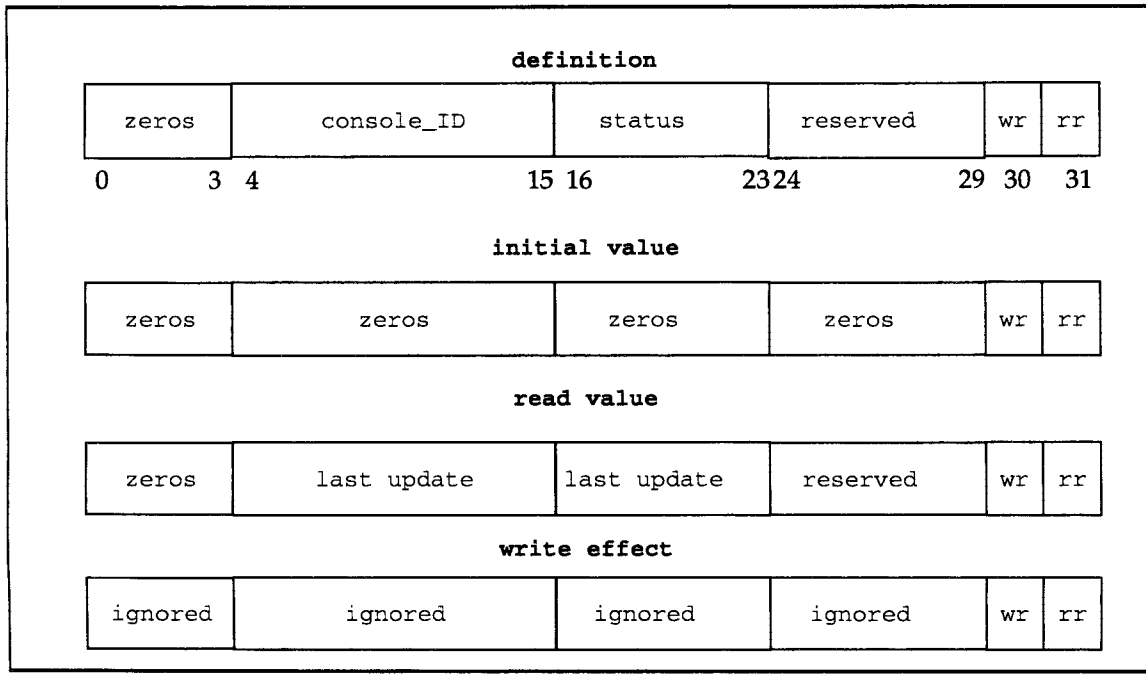

Figure 83 CONSOLE_STATUS format

Event logger interface

The PA-RISC event logger interface is a message-based interface. To log events, the PA-RISC chips send messages to the service processor. These messages include the same information as that mentioned in the service hypernode Interface section of this chapter.

Exemplar compliance

This section describes the Exemplar service processor functionality as it differs from that of the SPP system in general.

Service hypernode interface

In the Exemplar system, the service hypernode interface describes the interaction between the system hypernodes utility processors and the service hypernode. The PA-RISC processors do not use this interface. Instead, the utility processors present their own interface to the PA-RISC processors and relay this information to the service hypernode.

Service hypernode

Service hypernode facilities will be provided by system hypernodes in an SPP system, but for Exemplar, they are provided by a PA-RISC workstation. An independent workstation is used here because the Exemplar system is not robust enough to provide service processor functions when system faults occur. Successful service processor operations on a standard system hypernode require that the system continue some level of processing when a system fault occurs. This could be accomplished with multiple system partitions where a failure in one partition is diagnosed by a diagnostic process running in another partition.

DaRT bus access

The Exemplar DaRT bus interface is based on the Ethernet. The service hypernode implementation incorporates an Ethernet controller board in the HP workstation. The Exemplar system hypernode DaRT bus interface is provided by the hypernode utility unit.

Hypernode reset control

The service hypernode reset control interface is not implemented for Exemplar.

Scan interface

The Exemplar service hypernode scan interface is provided by the utility unit at each hypernode. The test station drives scan operations through DaRT bus transactions, and the utility unit controls hypernode scan operations with a JTAG test bus controller (TBC) under microprocessor control.

The Exemplar hardware does not require scan initialization for normal hypernode initialization and boot. Hypernode scan

initialization may be required at boot time, however, if a hypernode must be configured for operation with certain failed components disabled.

Console interface

The Exemplar service hypernode console interface conforms to the Exemplar model described in the previous section. Through this interface, any Exemplar hypernode can access consoles on the test station.

Diagnostic event logger interface

The Exemplar service hypernode diagnostic event logger interface conforms to the Exemplar model described in the previous section. For Exemplar, hypernode-generated events are transmitted over DaRT bus to the test station, where they are placed into a disk file for later analysis. Diagnostic event analysis is performed manually by a diagnostic expert and is performed automatically by the CXTS expert system.

Processor interface

In the Exemplar system, each PA-RISC processor accesses the service processor functions indirectly through its utility processor. It is the responsibility of the utility processor to either perform the requested functions, or use the service hypernode interface to request that the service hypernode perform the functions.

In general, the Exemplar system conforms to the processor interfaces described in the SPP architecture section of this chapter. The only item of note is the manner in which messages are passed between a PA-RISC and its utility processor.

Messages

A PA-RISC processor sends messages to its utility processor by first building the message in a buffer in its own memory. It then writes the address of the message to the utility processor mailbox. The utility processor then copies the message from the PA-RISC memory into its own memory before responding to the PA-RISC write access (the one that sent the message address to the utility processor).

Availability

In order to increase the system availability, the service processor architecture provides distributed operation, parallel test access to hypernodes, and independent operation.

Parallel testing

The Exemplar architecture enables simultaneous testing to be performed on multiple hypernodes under control of the service hypernode. Parallel testing dramatically reduces the amount of system downtime required to detect and isolate hardware failures.

DaRT bus

DaRT bus provides an independent communication path between each system hypernode and the service hypernode. This means that an Exemplar system has redundant data paths to the service hypernode for error reporting. During normal system operation, the service hypernode periodically probes each hypernode over DaRT bus and verifies a proper response from every hypernode. In this manner, the service hypernode quickly detects and responds to hypernode-access problems.

Exemplar test station

Exemplar availability is enhanced through the use of an independent service hypernode. The independent service hypernode provides full system test capability without reliance on any basic system hardware or software functionality.

Initialization and booting 11

Overview

This chapter defines the procedures and mechanisms provided by the Convex Exemplar series to support general system initialization and OS booting. Booting refers to the sequence of events used to load and execute OS code. This sequence, or *boot procedure*, typically begins at power-on with the machine in an unknown state, and ends when it begins executing the OS.

Various aspects of the boot mechanisms described in this chapter conform to IEEE architecture standards.

SPP boot architecture

The initialization and booting mechanisms for an SPP machine are highly dependent on its implementation. A few aspects of these procedures, however, are common to all implementations.

The boot procedure

The SPP boot procedure is a distributed task in which all hypernodes participate concurrently. Each hypernode boots itself in an autonomous manner at power-on. Additionally, an outside entity (the human operator or another hypernode) can force the hypernode to reboot. Interaction between hypernodes is minimized so that a hypernode can successfully boot without assistance from, or knowledge of other hypernodes in the system, as long as it has access to a boot device. In practice, however, each system will probably have only one or two hypernodes with direct access to a boot device. All other hypernodes will access these boot devices via the interhypernode CTIs.

The boot procedure for an SPP hypernode consists of the following phases:

- Hypernode self test
- Hypernode hardware initialization
- Memory initialization
- Interhypernode CTI initialization
- I/O device configuration
- OS boot

Each hypernode executes these in the order shown. There may, however, be implementations of the SPP architecture that require a slightly different ordering.

Hypernode self-test

The boot procedure begins with a self-test of the hypernode hardware. This test provides a confidence check that the hardware operates well enough to continue booting and perform normal operational tasks. It does not provide extensive diagnostic capabilities. If the self-test fails, additional, externally controlled software must be used to isolate and troubleshoot the problem.

Each unit within the hypernode executes its own self-test code. By physically locating the code with the hardware that it tests, the problems of tracking hardware revisions are minimized. The exact mechanisms by which this code is accessed and executed is implementation-dependent.

Self-test failures are reported to the SPP operator on the hypernode diagnostic display devices. There are two such devices within each hypernode: an array of individual LEDs and a multidigit alphanumeric LED display. These devices are limited in the amount of information they convey to the operator. Unlike an operator console, however, they require only a small portion of the hypernode hardware to be functional. Access to these devices is through the hypernode LED CSRs. (See the "IEEE 1212 CSRs" section on page 238 for more information.)

The array of individual LEDs functions as a unit-activity indicator. It provides a visual indication that the hypernode has successfully completed its self-test. When power is initially applied to the hypernode, or the hypernode is commanded to reboot, these LEDs are "blinking." Once the hypernode completes testing of a unit, it turns on the LED associated with that unit so that it is no longer blinking. With this scheme, a total failure of the hypernode will still result in a "fail" indication, that is, the LEDs will be blinking, or, in the case of a power failure, they will be off.

The multidigit, alphanumeric LED display provides a mechanism for the hypernode to communicate more detailed information about self-test failures. This can aid the operator in diagnosing failures.

Hypernode hardware initialization

Once the hypernode successfully completes the self-test, it places its hardware in an *initial state*. This ensures that the hypernode software can begin normal operation in a known environment.

The initialization process is unique to each implementation of the SPP series. Even within a given implementation, different revisions of hardware require different methods of initialization. For this reason, each unit within a hypernode provides its own initialization code. As is the case for the self-test code, the initialization code is located in nonvolatile memory within each unit.

Memory initialization

The next task in the boot procedure is initializing the hypernode memory system. This includes memory that is private to the hypernode, the global memory resident on the hypernode, and the caches that support the CPUs. Initializing these memories involves setting parity, ECC, and tag fields within the RAM arrays. It ensures that the memory system is in a known, "good" state before it is used for normal operating purposes.

Interhypernode CTI initialization

At this point in the boot procedure, the hypernode initializes the CTI hardware that it uses to communicate with other hypernodes in the SPP machine. As is the case for hypernode hardware, the CTI initialization process is unique to each type of CTI.

I/O device configuration

The ultimate goal of the boot procedure is to load the OS code from a boot I/O device so that the hypernode can execute it. Toward this end, the next step in the boot procedure is locating and identifying all I/O devices attached to the hypernode. In so doing, the hypernode also determines if a boot device is present.

The mechanism for identifying I/O devices is based on the Open-Boot PROM interface standard. This standard defines a vendor-independent method of identifying and communicating with I/O devices. There is a brief discussion of Open-Boot PROM later in this chapter, but a more complete description is found in the *IEEE Draft Std P1275/D3 Standard for Boot Firmware* document mentioned in the introduction to this manual.

OS boot

The final step in the boot procedure is to load OS code from the boot device. If the hypernode has found an appropriate boot device attached locally, it will load the OS boot code from it. The Open Boot PROM interface architecture defines the procedures for reading boot code from an I/O device. If no such boot device is available, the hypernode will attempt to access a boot device on another hypernode via the interhypernode CTI.

This is the first time in the boot procedure that a hypernode may depend on another hypernode for help in booting. If the hypernode of the boot device fails to respond to CTI requests, either because of a hypernode or CTI failure, the boot procedure will stop at this point. Otherwise, the OS code is loaded from the boot device and executed by the CPUs.

IEEE 1212 CSRs

Each SPP hypernode adheres to the IEEE 1212 Control and Status Register (CSR) architecture standard. This standard defines address-space maps, bus transactions, CSRs, and ROM formats for multihypernode systems. Some of the registers defined by this standard deal with boot and initialization functions. Described below are the boot aspects of these CSRs, along with some SPP-specific CSRs. The CSRs that are relevant to SPP boot and initialization are shown in Table 52.

Table 52 CSR location

Name	Description
STATE_CLEAR	Hypernode state and control information
NODE_IDS	Hypernode ID
RESET_START	Hypernode reset/reboot
LED_BLINK	LED array state and control
LED_STEADY	LED array state and control
LED_ALPHA	Alphanumeric LED state and control

In the register descriptions below, the following attributes are defined:

- DEFINITION—the register format, including field names.
- INITIAL VALUE—the register value immediately after power-on or command-reset.
- READ VALUE—the register value when read.
- WRITE VALUE—the effect on the register when written.

The values for these attributes are defined as follows:

- RESERVED or R—this field is not currently used, but is reserved for future use.
- LAST_WRITE or W—when read, this is the value of this field most recently written.
- LAST_UPDATE or U—when read, the value of this field is that to which it was most recently updated by the hypernode hardware.
- EFFECT or E—the value written to this field affects the hypernode state, but the effect is not immediately visible to reads of the same register.

STATE_CLEAR

This IEEE 1212 register is a collection of fields used to control and observe the state of the hypernode. A description of these fields is shown in Figure 84. For booting purposes, only the *lost* and *state* fields are of interest. The other fields in this register are not used during the boot procedure.

The lost field is set to 1 at power-on, at command-reset, and when the hypernode enters the dead state (described later). This signifies to other hypernodes that the initial units space and initial memory space of this hypernode are not accessible. Any attempt to access these portions of the hypernode results in an error response. The lost field is cleared to 0 by the OS code once it has informed its I/O device drivers of the reset or fatal event. To clear this field, the OS code writes a 1 to the lost field.

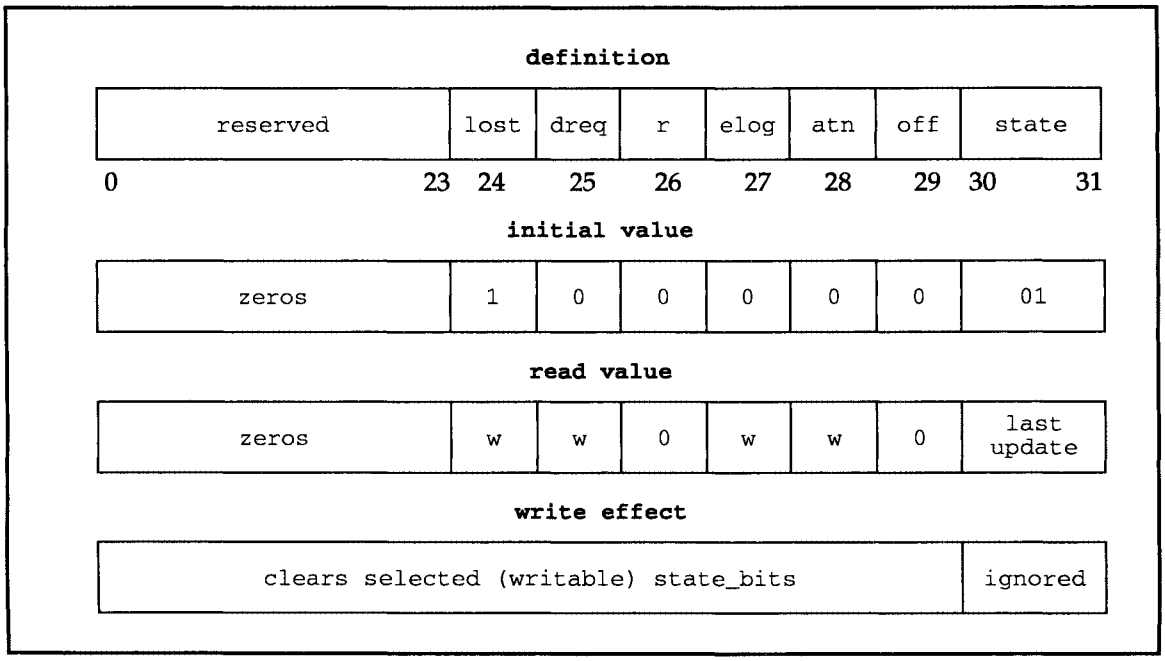

Figure 84 STATE_CLEAR format

The state field reflects the current state of the hypernode. It can assume one of four values: *running* (0), *initializing* (1), *testing* (2), or *dead* (3). When booting, the hypernode sets this field to the initializing state. This signifies that the hypernode is still executing the boot procedure. When the boot procedure is complete, the hypernode sets the state field to either the dead or running state. If a problem was encountered during the boot

procedure, the dead state is entered. If the boot completes without errors, the OS code changes this field to running. The testing state is not used during the boot procedure.

NODE_IDS

This IEEE 1212 register contains the hypernode physical ID within the SPP machine. The value that is stored here is determined during the interhypernode CTI initialization phase of the boot procedure. It can, however, be changed by the OS code at any time during normal operation. Unlike other 1212 CSRs, the contents of this CSR are unchanged by a reset command. This allows a hypernode to be reset without changing the address of its initial hypernode space.

Figure 85 provides a description of this register. The 16-bit hypernode ID is stored in the upper portion of this 32-bit register. If the hypernode is attached to multiple CTIs, its NODE_IDS registers must contain the same value for each CTI. This ensures that the CSR and memory resources within a hypernode share the same address space across all CTIs.

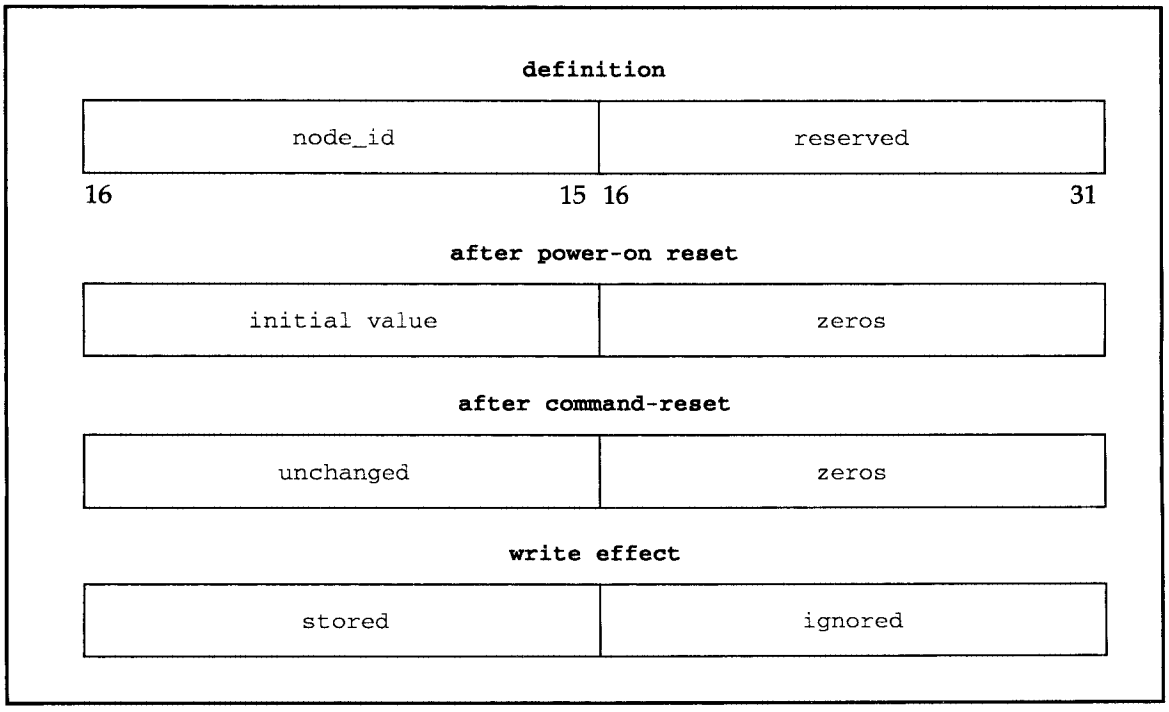

Figure 85 NODE_IDS format

RESET_START

This IEEE 1212 register provides a mechanism to reboot a hypernode after it has completed its power-on boot procedure. Writing to this CSR forces the hypernode to perform a command reset, which reboots the hypernode. The hypernode enters the initializing state, and then returns to either the running or dead state when the reboot is complete (see STATE_CLEAR, above).

Figure 86 describes the RESET_START register. Reading this register will always return a value of 0. As a consequence, writing to this register does not affect its contents. It does, however, affect the STATE_CLEAR register. The value that is written to this CSR is not important, only the fact that it is written.

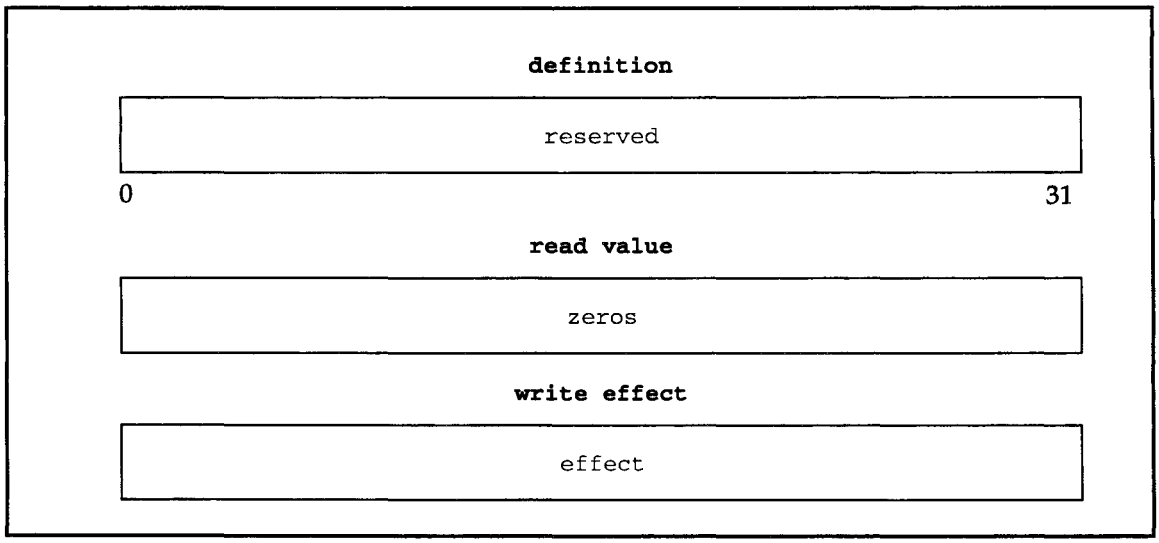

Figure 86 RESET_START format

LED_BLINK

This SPP-specific register allows the hypernode to control and observe the LED array diagnostic display. The only field in this register, LED_array, is a bit field, where each bit corresponds to an individual LED (see Figure 87). Writing a 1 to a bit will cause its LED to blink; writing a 0 will leave it unchanged. To turn on an LED so that it does not blink, use the LED_STEADY register described below:

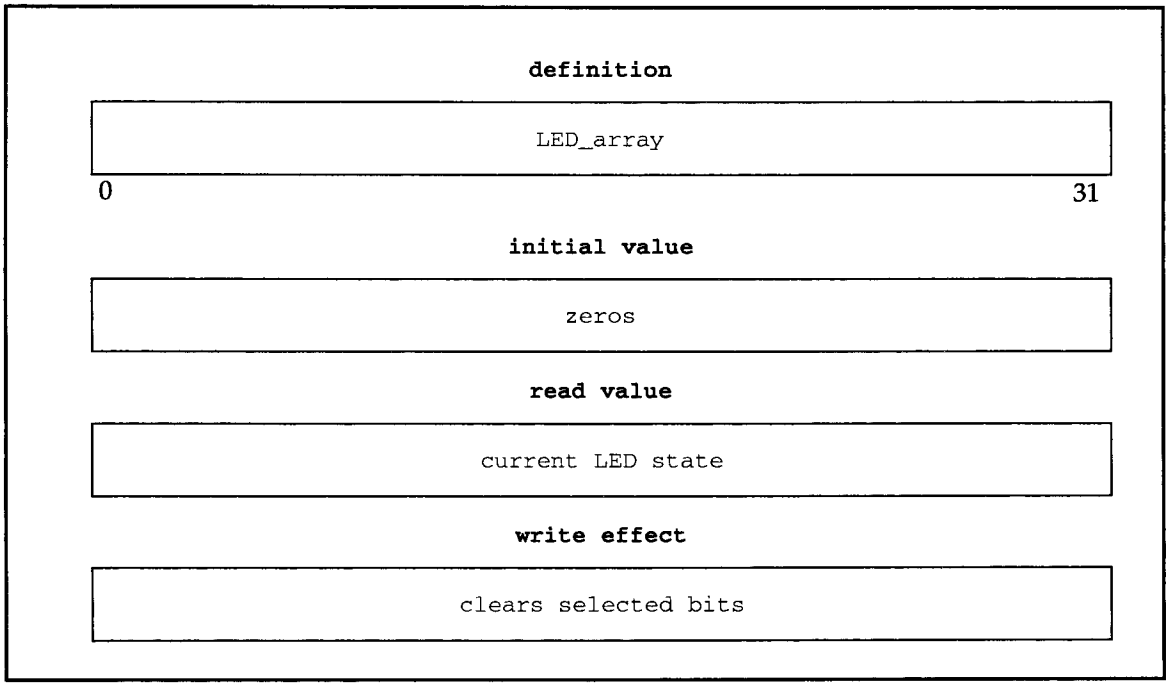

Figure 87 LED_BLINK format

LED_STEADY

This SPP-specific register allows the hypernode to control and observe the LED array diagnostic display. The only field in this register, *LED_array*, is a bit field, where each bit corresponds to an individual LED (see Figure 88). Writing a 1 to a bit will turn on its LED so that it does not blink; writing a 0 will leave it unchanged. To force an LED to blink, use the LED_BLINK register, described above.

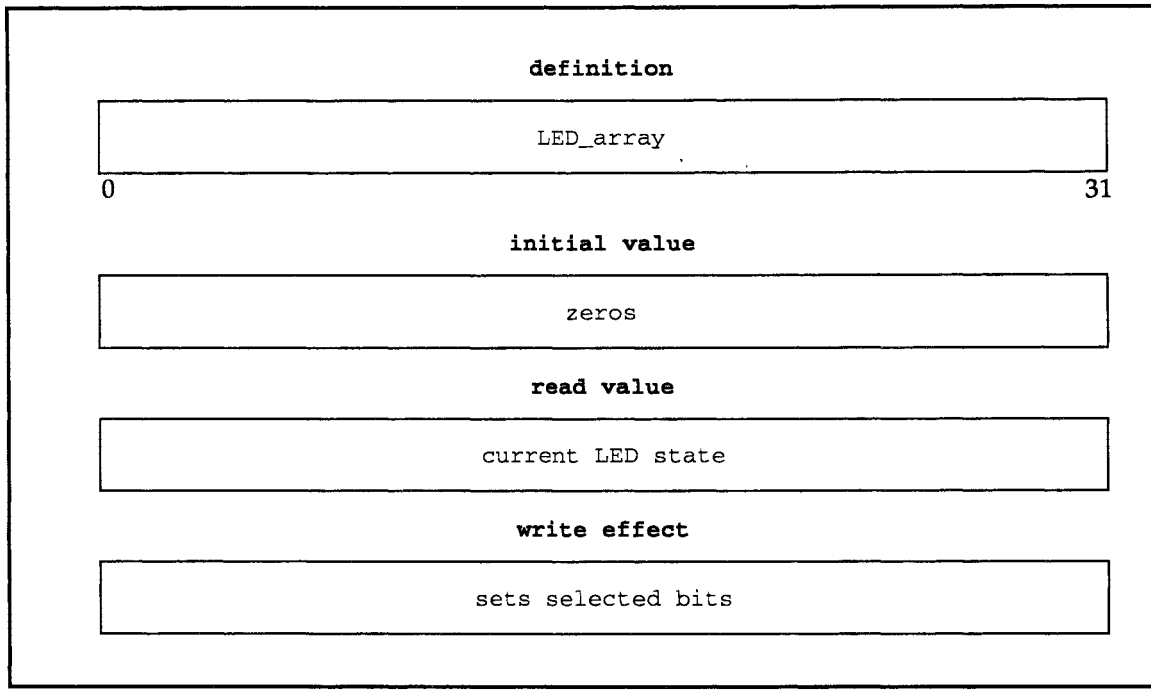

Figure 88 LED_STEADY format

LED_ALPHA

This SPP-specific register allows the hypernode to control and observe the alphanumeric LED diagnostic display. The only field in this register, alpha_char, is used to place characters on the display (see Figure 89). When a valid ASCII-printable character value is written to this field, all currently displayed characters are shifted left by one position, and the new character is placed in the right-most position. The display also responds to a subset of the ASCII control characters.

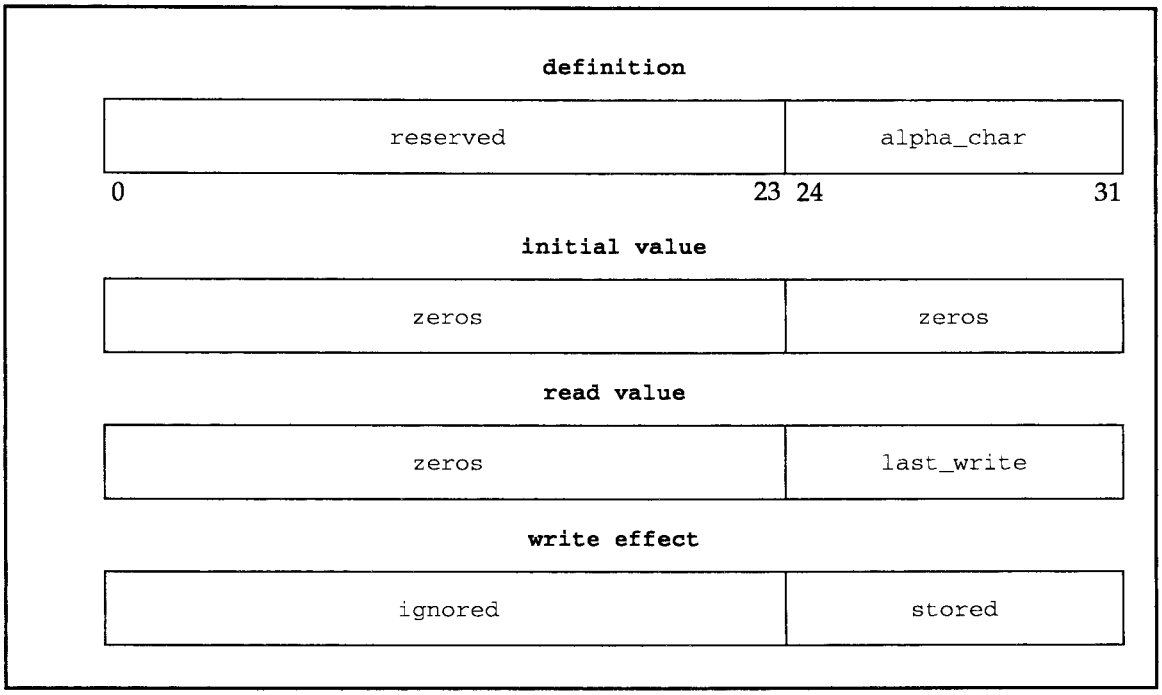

Figure 89 LED_ALPHA format

Table 53 Valid LED character values

Character Value	Effect on Display
0x0C	Display is cleared
0x20 - 0x7E	Corresponding ASCII character is displayed
All Others	Display is not changed

The boot console

One of the key elements in the boot procedure is the boot console. During boot, this is the operator primary interface to the SPP machine. The operator can use the boot console to monitor machine progress and interrupt the boot procedure to run diagnostics or change boot parameters. Communication between the SPP hypernodes and the operator conforms to the Open Boot PROM interface architecture.

The boot console is a character-oriented display device shared by all hypernodes in a system. There may, however, be more than one boot console, in which case a subset of the hypernodes may be assigned to each console. The boot code must ensure that only one hypernode communicates with the boot console at a time.

The boot procedure does not depend on the presence of a boot console. The absence of a boot console simply means that the operator does not have visibility into the progress of the boot procedure. Even though this situation is allowed, it is not recommended.

Open-Boot PROM

The Open-Boot PROM interface architecture (OBP) defines standard mechanisms for booting a system. These mechanisms include tasks such as user interaction, I/O device configuration, and debugging support. Because OBP is standardized, it allows new boot devices to be used in the SPP system without changing the boot code.

Exemplar compliance

This section describes the SPP boot and initialization issues as they relate to the Exemplar implementation.

To facilitate the booting process, system administrators and field engineers should refer to the *Exemplar Open Boot Quick Reference,* order number DSW-854. This reference booklet contains a summary of the following Exemplar Open Boot commands and related information including the following:

- Boot commands
- Configuration parameters
- Device tree browsing commands
- Commands for creating and examining device aliases
- Help commands
- File loading commands
- Diagnostic test commands
- FCode (Forth language) interface
- PA RISC register commands
- Assembler and disassembler commands
- Breakpoint commands

Boot procedure

The Exemplar boot procedure conforms to the SPP boot procedure described in the previous section. The early phases of the procedure are performed by the hypernode utility processor, while the later phases are performed by the CPUs in the hypernode.

Hypernode self-test

At power-on, all units, except for the utility unit, are held in a reset condition. This allows the utility processor to control the testing and initialization of the hypernode.

The first task for the utility processor is to test the utility hardware. This is a crawl-out type of test that verifies the unit EEPROM, RAM, internal functions, and external interfaces.

Hypernode hardware initialization

Once the utility processor is confident that the utility hardware is functioning properly, it places this hardware into a known, initial state. It then initializes the hardware in the other units within the hypernode. The utility processor uses the JTAG scan rings to access the hypernode hardware not within the utility unit.

During this phase of the boot procedure, all of the CPUs within the hypernode begin executing their reset code. The hardware in the hypernode maps this address to nonvolatile memory within the utility unit.

The first piece of code executed by the CPUs is the primary loader. Its purpose is to initialize the CPUs and perform a confidence check on the private and cache memories located within the CPU units.

The next piece of code executed by the CPUs is the secondary loader. The purpose is to continue the boot procedure for the hypernode. The secondary loader is maintained by the OS group.

Memory initialization

In this phase of the boot procedure, the CPUs initialize the hypernode memory. This includes the share of the global memory and any private memory that was not initialized during the CPU start-up phase.

Interhypernode CTI initialization

The Exemplar machine uses two types of interhypernode CTIs: DaRT and CxRing. Both of these require some level of initialization.

The diagnostic and remote testing bus (DaRT) is the CTI that connects all Exemplar hypernodes within a system to the Exemplar test station. It allows the utility processor to determine the existence and location of a boot console and/or a test station. It also allows the utility processor to inquire about the availability of test station services such as disk file access and diagnostic support.

In the Exemplar boot procedure, DaRT is initialized as soon as the utility processor has tested enough of the utility hardware to gain access to its DaRT interface hardware. This is done early so that the test station, if present, has the opportunity to take control of the boot procedure as early as possible. The details of this process are implementation-dependent.

The other interhypernode CTI within the Exemplar is the CxRing. This is the primary hypernode-to-hypernode communication path during normal system operation. Here again, the details of this CTI initialization are implementation-dependent.

In general, the DaRT and CxRing CTIs require little assistance from the hypernode for initialization.

I/O device configuration

At this point in the boot procedure, the CPUs use the mechanisms defined by the Open-Boot PROM standard to locate and identify the I/O devices that are attached to the hypernode. This information is also shared with other hypernodes in the machine so that a machine-wide I/O configuration table can be created.

OS boot

Once the I/O devices have been identified, the CPUs determine if they have access to an appropriate boot device. If one is found, the CPUs load and execute the OS boot code from the device. If a boot device is not accessible, the hypernode will hang, waiting for one to become available. After a sufficient wait, the hypernode displays an error message on the diagnostic display and boot console (if present) if a boot device is not found.

IEEE 1212 CSRs

The Exemplar implementation does not conform to the IEEE 1212 CSR standard features defined for the SPP architecture.

The boot console

The boot console for an Exemplar machine is implemented as a window on the system test station. The implication of this configuration is that without a test station, the operator has no visibility to the progress of the boot procedure. As stated before, this is not a recommended configuration.

For more details on the Exemplar boot console, refer to the Boot console interface section in Chapter 10.

Open-Boot PROM

The Exemplar machine supports the Open-Boot PROM interface standard described in the SPP boot architecture section of this chapter. The OBP code runs on the CPUs within each hypernode in the system. It communicates with the operator via the boot console.

Availability

The SPP boot procedure enhances system availability in several ways. It is fault-tolerant, which allows the SPP system to boot in the presence of hardware failures. It is also distributed, which allows the machine to boot quickly, thus reducing downtime.

The SPP boot procedure is designed so that each hypernode boots without knowledge or assistance from either the boot console or other hypernodes in the system. This means that an SPP machine will boot even if some of its hypernodes fail to boot. The only interhypernode dependencies in the boot procedure are that without a boot device, a hypernode cannot load the operating system code, and without a boot console, operator interaction is not possible (for error messages and status messages, for example).

Fault tolerance also exists within a hypernode. The boot procedure is designed so that if a CPU fails, it does not affect the other CPUs within a hypernode. This allows the hypernode to boot successfully in the presence of CPU failures.

A third fault-tolerant feature of the boot procedure is the provision for multiple boot devices within a machine. If a primary boot device fails, the system can boot from a secondary (tertiary, and so on) boot device. This concept can also be used to reduce the time-to-boot by distributing the access to the boot devices, thus allowing more than one device to be used at a time. Care must be taken here, though, to ensure that the boot devices are consistent.

Because all hypernodes boot independently and in parallel, the time-to-boot is reduced. This enhances availability by reducing the time required to recover from a system failure. It also means that a single hypernode can fail, be repaired or replaced, and reboot without affecting the availability of the rest of the machine.

Glossary

A

ABI
Application binary interface. A software implementation that allows executable programs to run on computer systems with different hardware architectures. Convex SPP systems are implemented with an ABI that allows compatibility with other computer systems that use the Hewlett-Packard PA-RISC processors.

absolute address
An address that does not undergo virtual-to-physical address translation when used to reference memory or the I/O register area.

access mode
Any of the five processor access modes in which software executes. On the Convex system, processor access modes are (in order from most to least privileged and protected): kernel (mode 0), executive (mode 1), supervisor (mode 2), agent (mode 3), and user (mode 4). The operating system uses access modes to define protection levels for software executing in the context of a process.

address
A number used by the operating system to identify a storage location. Also a unique number or character string that identifies a particular network node. Also called a *network address*, *host address*, and *internet address*.

address space
Address space, either physical or virtual, available to a process.

agent
The gate array on Convex SPP systems that interfaces to pairs of PA-RISC processors.

alias
An alternative name for some object, especially an alternative

variable name that refers to a memory location. Aliases can cause recurrences, which prevent the compiler from vectorizing or parallelizing parts of a program.

alignment
The suitability of particular memory addresses for accessing particular types of data. For example, some processors require 16-bit data items to be stored beginning at even-numbered addresses.

ALU
See *arithmetic logic unit*

Amdahl law
A statement that the ultimate performance of a computer system is limited by the slowest component. In the context of SPP systems this is interpreted to mean that the sequential component of the application code will restrict the maximum speed-up that is achievable.

American National Standards Institute (ANSI)
A repository and coordinating agency for standards implemented in the U.S. Its activities include the production of Federal Information Processing (FIPS) standards for the Department of Defense (DoD).

analysis
The first phase of the `build` process, during which the Application Compiler inspects each procedure separately. This phase generates the intermediate-code representation and derives interprocedural information needed for program optimization.

ANSI
See *American National Standards Institute*

API
See *application program interface (API)*

apparent recurrence
A condition or construct that fails to provide the compiler with sufficient information to determine whether or not a recurrence exists. Also called a *potential recurrence.*

application program interface (API)
A set of system calls and library routines that provide programmers with access to network services.

A means of communication between programs. An API gives one program transparent access to another. APIs serve various computing purposes. In networking, an API offers software applications (such as a database manager) transparent access to files,

devices, or interprocess communications. Some network APIs let application programmers develop sophisticated distributed software program for networks.

architecture
The physical structure of a computer's internal operations, including its registers, memory, instruction set, input/output structure, and so on.

argument
In FORTRAN, either a variable declared in the argument list of a procedure (function or subroutine) that receives a value when the procedure is called (*dummy argument*) or the variable or constant that is passed by a call to a procedure (*actual argument*). C conventions refer to arguments as *parameters.*

argument pointer (AP)
An address value specifically dedicated to point to an operand.

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.

ASCII
American Standard Code for Information Interchange.

assembler
A program that converts assembly language programs into executable machine code.

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

assembly language
A programming language that relates directly to the native instruction set of a particular computer system.

asynchronous mode
Execution mode in which control returns to a user immediately following a library function call even if the corresponding asynchronous event has not occurred.

atomic access
A read or write operation, usually to a device register, that cannot

be subdivided into a sequence of smaller operations.

autoconfiguration
The process of determining, without human intervention, the set of devices that comprise a computer system and the characteristics of those devices, and of adjusting the firmware and software to behave appropriately with that set of devices.

B

backplane
The circuitry and mechanical elements used to connect the boards of a system. Also called *motherboard.*

balancing
See *tree-height reduction.*

bandwidth
A measure of the rate at which data can be moved through a device or circuit. Bandwidth is usually measured in Millions of bytes per second (Mbytes/sec) or millions of bits per second (Mbits/sec).

barrier synchronization
A control mechanism used in parallel programming that ensures all CPUs have completed the prior operation before continuing with the next operation.

base-level interrupt
An interrupt 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.

basic block
A linear sequence of program statements with a single entry and a single exit.

block
A group of data containing a fixed number of bytes.

block size
An integer representing the number of bytes in a block.

block TLB
A type of TLB entry that translates many contiguous virtual pages to an equal number of contiguous physical pages.

boot
The procedure by which a program is initiated the first time. Typically, a bootstrap is performed when power is first applied to the processor.

branch
A class of instructions, specifically relative to the program counter, used to transfer control of a program.

breakpoint
A predefined event point that you can place before source units, routines, lines of source code, or machine instructions. When the executing process reaches the breakpoint, execution stops and control returns to you.

buffer
A temporary storage area. Several types of buffers are used in computer systems, in both hardware and software. The most common types of buffers are those maintained by a computer operating system to mediate between processes and I/O devices.

buffer identifier
An integer handle for a message buffer used in library functions.

bus
A data path shared by several components within a computer system. A typical example is an I/O bus, which can connect processors, memory, and I/O controllers.

byte (b)
A group of contiguous bits starting on an addressable boundary. In Convex machines, a byte is 8 bits in length.

C

cache
See *cache memory*.

cache memory
A small, high-speed buffer memory used in modern computer systems to hold temporarily those 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 caches, physical caches, and instruction caches.

cache purge
The act of invalidating or removing entries in a cache memory.

cacheable
Memory references which are eligible for cache *move in*s are said to be cacheable references.

card cage
A rack inside the computer housing that holds the printed circuit boards. Also called *chassis*

CCMC
The Convex Coherent Memory Controller gate array.

cell
The primary unit of information in a Forth or FCode language system.

central processing unit (CPU)
The central processing unit (CPU) is that portion of a Convex computer that recognizes and executes the instruction set.

chassis
The physical box where the computer is housed, such as Multibus drawer, VMEbus chassis, and CPU chassis.

check
A type of *interruption* caused by the detection of an internal hardware detected malfunction.

child node
A node in any hierarchical system that has a parent (immediately higher) node. A child node is said to *descend* from its parent node.

cloning
The creation of duplicate procedures (clones) that enable the Application Compiler to perform different optimizations for different calls to the same procedure.

code
A computer program, either in source form or in the form of an executable image on a machine.

coherency
A term frequently applied to caches. If a data item is referenced by a particular processor on a multi-processor system, the data is copied into that processor cache and is updated there if the processor modifies the data. If another processor references the data while a copy is still in the first processor cache, a mechanism is needed to ensure that the second processor does not use an outdated copy of the data from memory. The state that is achieved when both processors' caches always have the latest value for the data is called cache coherency.

communication, interprocessor
The process of moving or sharing data, and synchronizing operations between processors on a multiprocessor system.

communication line
A physical communication path, such as coaxial or fiber-optic cable.

communication link
A logical communication path consisting of the hardware and software needed to establish a connection and transfer data between nodes. Also called a *logical link* or *virtual circuit*.

communication register
A high-speed register used for communication among the threads of a process. Threads communicate by sending and receiving data through the communication registers. A hardware-maintained lock bit is associated with each communication register. The lock bit guarantees mutually exclusive access to the register.

compiler
A computer program that translates computer code in a high-level language, such as FORTRAN, into its machine language.

complex
The complete set of processor and memory resources available on a Convex SPP system.

complex manager
A utility that allows Convex SPP system administrators to maintain a database of sub-complex definitions, load sub-complex definitions onto the system, and modify sub-complexes currently running on the system.

computational complexity
A measure of the amount of work normally required to implement an algorithm. Values are frequently expressed in terms of order of magnitude. For example, the LU factorization of a matrix of order n requires $(2/3)\,n^3 - (1/3)\,n^2 - (1/6)\,n$ floating-point operations (neglecting pivoting). This calculation is said to have a computational complexity of $O(n^3)$ (note the absence of a constant when stating computational complexity). The order of magnitude estimates are generally more accurate as n gets large.

computer network
A system of interconnected computers that enables machines and their users to exchange information and share resources.

concurrent
In parallel processing, threads that can execute at the same time are called concurrent threads.

conditional induction variable
A loop induction variable that is not incremented on every iteration.

configuration variable
A named parameter, whose value is stored in nonvolatile memo-

ry, that controls some aspect of the firmware behavior.

constant folding
Replacement of an operation on a constant with the result of the operation.

constant propagation
The replacement of variable references, by the compiler, with a constant value previously assigned to that variable, performed within a single procedure by conventional compliers and between procedures by the Application Compiler.

context (processor)
The entire, current state of the machine associated with the executing process.

control parallel programming
A type of parallel programming in which different functional sections of code are assigned to different processors for simultaneous execution. See also *data parallel programming.*

conventional compiler
A compiler that cannot perform inter-procedural optimization. This term encompasses all Convex compilers except the Applications Compiler and virtually all compilers available from other vendors.

coroutine module
A module consisting of a main program and a description function with optional initialization and destruction functions. Each executable file can contain only one module. The description function can have any name.

CPU
See *central processing unit (CPU).*

CPU-private memory
Memory that is accessed from a single CPU on a Convex SPP system. This type of memory has the lowest latency to a CPU. Compare with *node-private memory, near-shared memory,* and *far-shared memory.*

CPU time
The amount of time the CPU requires to execute a program. Because programs share access to a CPU, the wall-clock time of a program may not be the same as its CPU time. If a program can use multiple processors, the CPU time may be greater than the wall-clock time. (See *wall-clock time.*)

crossbar
A switching device used in multiprocessor, shared-memory computer systems that connects CPUs to the various banks of memory in the system.

CSB
Convex System Board. The board that contains all of the gate arrays and connectors for the daughter boards. Also called the *motherboard.*

CSR
Control/Status Register. A CSR is a software-addressable hardware register used to hold control information or state.

CTI cache
A cache within a node which contains coherent memory data fetched from other nodes in a Convex SPP system.

CTIRing
The ring interconnect that connects all the nodes of a multinode Convex SPP system together in a ring fashion. This is Convex implementation of SCI.

D

DaRT
Diagnostic and Remote Test. The DaRT Bus is the Ethernet-like connection between the nodes of a Convex SPP system that is used for diagnostic purposes. The nodes are connected together in a bus fashion.

data dependency
A relationship between two statements, such that one statement must precede the other to produce the intended result. (See also *loop-carried dependency* and *loop-independent dependency.*)

data redundancy
The duplication of data to ensure its protection against hardware, software, or other types of system failures.

data parallel programming
A type of parallel programming in which the data upon which an operation is to be performed is divided among several threads, which execute simultaneously in separate processors.

data type
The way in which bits are grouped and interpreted. For processor instructions, the data type identifies the size of the operand and the significance of the bits in the operand.

decompiler
A software tool that generates source code from compiled code. A decompiler is part of Forth language packages.

default processor set
Each node of a Convex SPP system has a default processor set. All processors in a node are initially assigned to the default set when the node is booted. A processor remains in the default set until it is explicitly assigned to a sub-complex. When a processor is removed from a sub-complex, it is returned to the default set.

def-use chain
A data structure created and used by optimizing compilers to track every site within a program where a particular variable or array subsection could possibly be used, starting from the point where the item is defined. See also *use-def chain*.

destination
The location where an operand will be stored at the end of an operation.

device alias
A shorthand representation of a *device path*.

device arguments
The component of a *device node name* that provides device-specific information.

device driver
The software responsible for managing low-level I/O operations for a particular hardware device of set of devices. A device driver contains all the device-specific code necessary communicate with the device and provides a standard interface to the rest of the system. Device drivers can be implemented either in firmware or in the operating system.

device interface
One of the standard interfaces specified in the IEEE P1275 Boot Firmware standard, allowing devices to be identified, characterized, and used to assist other firmware functions such as booting.

device node
An entry in a *device tree*, usually describing a single device or bus. A device node may have multiple child nodes and has exactly one parent node.

device node name
A text string of the form *driver-name@unit-address:device-arguments*, identifying a *device node* within the address space of its parent.

device path
A textual name identifying a *device node* by showing its position in the *device tree.*

device specifier
A *device path* or a *device alias,* or a hybrid path that begins with a *device alias* and ends with a *device path.*

device tree
A data structure that describes the set of devices attached to a computer system, including both permanently installed devices and plug-in devices. The IEEE P1275 Boot Firmware standard specifies a structure for the device tree; this structure is used in Convex SPP systems.

device type
Identifies a set of properties and package classes that a *device node* is expected to implement.

diagnostic insertion
A method used on Convex SPP systems for explicitly specifying the *TLB* entry to be used for the next TLB insertion using diagnostic instructions.

direct memory access (DMA)
A procedure or method defined for gaining direct access to main storage and achieving data transfers without involving the Central Processing Unit.

disassembled code
The assembly language code that is equivalent to the machine language instructions for your program. CXdb generates the disassembled code from the executable file for your program.

disassembler
A program that translates machine code into an equivalent human-readable assembly-language representation.

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. 16-bit displacement values are sign extended to 32 bits.

distributed memory
A memory architecture used in multi-CPU systems, in which the system memory is physically divided among the processors. In most distributed memory architectures, distributed memory is accessible from only a single processor; Convex GSM is an exception. See also *globally shared memory (GSM).*

distributed part
A loop generated by the compiler in the process of loop distribution.

DMA
See *direct memory access.*

domain decomposition
An optimization technique used in multi-CPU systems, in which a problem is divided into pieces for execution on separate processors. Special effort is required to put the results of each computational piece together to arrive at the final solution. Domain decomposition is frequently used in the solution of partial differential equations, especially when the finite element method (FEM) is used.

doublet
A unit of computer data consisting of sixteen bits.

DRAM
Dynamic random-access memory; memory that needs periodic refreshing. DRAM is usually slower than static random-access memory (SRAM).

drive
A mechanical device on which storage media, such as tapes or disks, are placed. A drive contains the physical devices used to position, read from, and write to the storage media.

E

ECMA
See *European Computer Manufacturers Association.*

EIA
See *Electronics Industry Association.*

efficiency
The ratio of the amount of a resource that is used in practice vs. the amount of the resource that could theoretically be used under ideal conditions. In SPP systems, the efficiency of the processors for a particular parallel programming application can be measured as the average percentage of time each processor assigned to the application is performing useful work on the application.

Electronics Industry Association (EIA)
A national trade association concerned primarily with the development of hardware-level standards. EIA standards are identified by the letters EIA, followed by a hyphen and a number, such as EIA-232, a standard used for connecting terminals to computers.

enabled
A condition or bit is said to be enabled when it is true or set to a one.

end user
The person or program that requests a service.

entity
An addressable unit of software or hardware that provides a service to or makes use of a service provided by another addressable unit.

environment variable
In UNIX operating systems, a shell variable that contains information about your environment.

EPROM
See *erasable programmable read-only memory.*

erasable programmable read-only memory (EPROM)
A read-only memory module that can be programmed repeatedly by first erasing the previous contents of the memory module. Reprogramming the memory modules usually involves removing them and using special hardware to burn a new pattern into them.

error code
The status returned by a function call.

error correction code (ECC)
Code used to decide which bit of a memory read operation is in error.

Ethernet
A popular local area network (LAN) technology developed by Xerox.

European Computer Manufacturers' Association (ECMA)
ECMA works closely with ISO and CCITT toward developing standards for data processing and data communication. ECMA includes all European computer manufacturers.

exception
A hardware-detected event that disrupts the running of a program, process, or system. See also *fault.*

execution stream
A series of instructions executed by a CPU.

expression
A combination of constants and symbolic names joined by operators.

far-shared memory
Memory that is globally accessible from all nodes in a Convex SPP system, and has equal access latency from all nodes. This type of memory has the highest latency. Compare with *node-private memory*, *near-shared memory*, and *CPU-private memory*.

fault
A type of *interruption* caused by an instruction which requests a legitimate action which cannot be carried out immediately due to a system problem.

FCode
A variant of the Forth programming language defined in the IEEE P1275 Boot Firmware standard. This language is used in the boot firmware for Convex SPP systems.

FCode driver
A *device driver*, written in *FCode*, intended for use in IEEE P1275 Boot Firmware and its client programs.

firmware
A computer program that controls the computer system from the time it is turned on until the time the operating system assumes control of the computer. Firmware typically resides in read-only memory.

flag
A status bit that is generally used to indicate the results of an operation. The results are in the form true or false.

floating point
A numerical representation. A floating point operand has a sign (positive or negative) port, an exponent port, and a fraction port. 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*.

FLOPS
Floating-point operations per second. A standard measure of computer processing power in the scientific community.

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

front end
A client application for presenting, inputting, and displaying data. It works with a back end or engine.

front-end network
The interconnection of workstations, PCs, facsimiles, terminals, and printers.

function
A procedure that returns a value. Any procedure in C or a procedure defined as a `FUNCTION` in FORTRAN.

functional unit (FU)
A part of a **CPU** that performs a set of operations on quantities stored in **registers**.

G

gallium arsenide (GaAs)
An alloy of gallium and arsenic that is used as the base material for integrated circuits, or chips. It is several times faster than silicon, currently the most common base material for making chips.

Gbyte
See *gigabyte*.

gigabyte
One billion bytes.

globally shared memory (GSM)
A memory architecture in which memory is physically distributed among the processors of a computer system, but all memory can be accessed by all processors in the system. This architecture can also support virtual memory. This type of memory is sometimes referred to as shared virtual memory or global virtual memory.

global optimization
A restructuring of program statements that is not confined to a single basic block. Global optimization, unlike inter-procedural optimization, is confined to a single procedure. Global optimization is done by all Convex compilers at optimization level `-O1` and above.

global memory
Memory that is accessible from several processors. See also *distributed memory, globally shared memory (GSM).*

global variable
A variable whose scope is the entire program. In C programs, a global variable is a variable that is defined outside of any one procedure. In FORTRAN, it is any variable declared in a `COMMON` block.

global virtual memory
See *globally shared memory (GSM).*

granularity
A measure of the relative size of objects where performance is a function of size. In higher-level language programs, possible sizes are routine, loop, block, statement, and expression. In I/O, granularity is often measured in terms of block size.

group
An ordered set of one or more tasks, named symbolically. Any task may be a member of zero or more groups.

GSDVM
See *globally shared memory (GSM)*.

guard bit
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 bit*.

H

h
Abbreviation for halfword.

halfword (h)
Two bytes (16 bits). See also *longword; word*.

hard error
An uncorrectable data error.

hardware address
The device-dependent physical address of a node attached to a communication line.

hierarchical model
An organizational model useful for object description that divides a system into levels of abstraction.

High Performance Computing and Communication (HPPC)
An initiative established and funded by the U.S. government whose goal is to develop the hardware, software, networks, and education to solve the *grand challenges* of computation by the end of the 1990s.

High-Performance Parallel Interface (HIPPI)
Currently the fastest industry standard for connecting high-performance computers. HIPPI hardware consists of HIPPI channel control unit (CCU) that supports dual simplex connections (one input, one output) to provide a data rate of 800 megabits per second over distances of up to 25 meters.

host
A computer system into which communications hardware and software have been installed, and which is accessed as a separate entity by other hosts.

hypercube
A topology used in some massively parallel processing systems. Each processor is connected to its binary neighbors. The number of processors in the system is always a power of two; that power is referred to as the dimension of the hypercube. For example, a 10-dimensional hypercube has 2^{10}, or 1,024 processors.

hypernode
In Convex SPP systems, a set of processors and physical memory organized as a symmetric multiprocessor (SMP) running a single image of the operating system microkernel. An SPP system consists of one or more nodes, with a high speed CTI connecting the nodes.

hypernode-private memory
Memory that is accessible by any CPU on a single node in a Convex SPP system. Compare with *CPU-private memory*, *near-shared memory*, and *far-shared memory*.

I

Icache
See *instruction cache.*

IEEE
See *Institute for Electrical and Electronic Engineers*

IEEE P1275 Boot Firmware standard
A software architecture for firmware that provides for *autoconfiguration* of a system and allows for customers to customize boot programs and *device drivers* without altering the firmware.

immediate (operand)
An operand that is contained within the instruction stream.

indexing
The process of adding a displacement value 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.

inlining
The replacement of a procedure (function or subroutine) call, within the source of a calling procedure, by a copy of the called

procedure code.

inode
A UNIX data structure containing information about a file, such as ownership, permissions, and the file location on disk. An inode exists for every file accessible to the operating system.

Institute for Electrical and Electronic Engineers (IEEE)
An international professional organization and a member of ANSI and ISO. IEEE created Project 802, the committee that developed a set of widely-used LAN standards known as the 802 standard.

instruction
One of the basic operations performed by a CPU.

instruction cache (Icache)
Accelerates the decoding of instructions to permit the simultaneous decoding on one instruction with the execution of another instruction.

instruction mnemonic
A symbolic name for a machine instruction.

interface
A logical path between any two modules or systems. (See *network interface.*)

interleaved memory
Memory that is divided into multiple banks to permit concurrent memory accesses. The number of separate memory banks is referred to as the memory stride. Programs can optimize memory accesses by using stride intervals so that each of the banks can be refreshed between memory accesses.

interprocedural optimization
Automatic analysis of relationships and interfaces between all subroutines and data structures within a program. Traditional compilers analyze only the relationships within the procedure being compiled. Convex C and Fortran compilers are capable of performing interprocedural optimization.

interprocessor communication
See *communication, interprocessor.*

interrupt
An occurrence that changes the normal flow of instruction execution. An interruption originates from hardware, such as an I/O device. See also *maskable interrupt.*

interval timer
An interval timer is used to generate an interrupt based on the passage of time.

I/O node
A node that contains a chassis filled with peripheral devices (e.g., disk drives) in place of a CPU chassis.

irregular field
A data array in which the coordinate space may not have the same number of dimensions as computational space. Each data element in computational space is mapped is explicitly mapped to a point in coordinate space.

J

job scheduler
The operating system program that schedules and manages the execution of all processes.

JTAG
Joint Test Action Group. This is a device-scanning standard that is used in the Convex SPP system architecture.

jump
Departure from normal one-step incrementing of the program counter.

K

kernel
The core of the operating system where basic system facilities, such as file access and memory management functions, are performed.

L

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.
large application
An application requiring a virtual address space larger than 4 GB.

latency
The time delay between the issuing of an instruction and the completion of the operation. A common benchmark used for comparing SPP systems is the latency of coherent memory access instructions involving non-CPU private memory; Convex SPP system latency values are among the best (lowest) in the industry. This particular latency measurement is believed to be a good indication of the *scalability* of an SPP system; low latency equates to

low system overhead as system size increases.

least significant bit (LSB)
The significant bit contributing to the smallest quantity to the value of a binary numeral. Bit numbering in Convex computers is left to right, N-1 through 0. The LSB is 0.

Libpvm
The C programming library (libpvm3.a), FORTRAN programming library (libfpvm3.a), or group library (libgpvm3.a).

linker
A software tool that combines separate software modules into a single module.

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.

local optimization
Restructuring of program statements within the scope of a basic block. Local optimization is done by all Convex compilers at optimization level `-O0` and above.

logical address
Logical address space is that address as seen by the application program.

logical cache
A cache that is accessed with logical addresses for fast retrieval of data. It resides in the CPU.

logical memory
The memory space as seen by the program. Also called virtual memory. See also *page*.

longword (l)
Eight bytes (64 bits), the largest data type directly supported by hardware in the Convex C120. See also *halfword; word*.

loop-carried dependency (LCD)
A dependency between two operations executed on different iterations of a given loop and on the same iteration of all enclosing loops. A loop carries a dependency from an indexed assignment to an indexed use if, for some iteration of the loop, the assignment stores a value that is referred to on a later iteration of the loop. For example, an LCD from `A(I+1)` to `A(I)` exists in the following

loop:

```
DO I = 1, 100
  A(I + 1) = A(I) + B(I)
ENDDO
```

An LCD from `B(I+1)` to `B(I)` exists in the following loop:

```
DO I = 1, 100
  A(I) = B(I) + C(I)
  B(I + 1) = D(I) * 3.14
ENDDO
```

loop constant

A constant or expression whose value does not change within the loop.

loop distribution

The restructuring of a loop nest to create additional innermost loops and to enhance opportunities for loop interchange. Loop distribution creates two or more loops, called distributed parts, isolating code that must run serially from parallelizable or vectorizable code.

loop-independent dependency (LID)

A dependency between two operations executed on the same iteration of all enclosing loops such that one operation must precede the other to produce correct results. For example, an LID from the use of `B(I)` to the assignment to `B(I)` exists in the following loop:

```
DO I = 1, 100
  A(I) = B(I) + C(I)
  B(I) = 0.0
ENDDO
```

An LID from `B(100)` to `B(I)` exists in the following loop, though only on the hundredth iteration:

```
DO I = 1, 100
  A(I) = B(100) + C(I)
  B(I) = 0.0
ENDDO
```

loop induction variable
A variable that changes linearly within the loop, that is, whose value is incremented by a constant amount. For example, in the following loop, `J` and `K` are induction variables, but `L` is not.

```
DO I = 1, N
  J = J + 2
  K = K + N
  L = L + I
ENDDO
```

loop interchange
The reordering of nested loops to increase the granularity of the parallelizable outer loop, to increase the iteration count of the vectorizable inner loop, or to achieve the most efficient vector stride in the inner loop.

loop invariant
See *loop constant.*

loop invariant computation
An operation that yields the same result on every iteration of a loop.

LSB
See *least significant bit (LSB).*

LSI
See *language specific information (LSI).*

M

machine exception
A fatal error in the system that cannot be handled by the operating system. See also *exception.*

main memory
See *physical memory.*

maskable interrupt
An interrupt to which the operating system may choose not to respond.

massively parallel processing (MPP)
A computer architecture in which tens to thousands of processors are combined to work simultaneously on solving complex problems.

MAUI
The Convex SPP system MPP Agent for Utilities and I/O gate array.

Mbyte
See *megabyte.*

megabyte (Mbyte)
1 million bytes.

memory management
The hardware and software features that control memory page mapping and memory protection.

message
Data copied from one thread to another (or the same) thread. The copy is initiated by the sending thread, who specifies the receiving thread. The sending and receiving threads need not share a common address space.

Message passing
A type of programming in which program modules (often running on different processors or different hosts) communicate with each other by means of system library calls that package, transmit, and receive data.

A hardware architecture in which memory accesses involving physically distributed banks of memory are implemented by messages passed between the processors that own the respective banks of memory.

microcode
A program or set of assembly language instructions that reside within a storage device. Microcode is also used as a synonym for firmware.

microkernel
An operating system architecture in which only a small portion of the operating system kernel (the microkernel) is resident in each processor; the remainder of the operating system is provided by the operating system server. In Convex SPP systems, there is a single image of the microkernel resident in each *node* of the system.

Microstrip connector
The AMP connector used for the daughterboard to CSB connection.

MOD
Meaningful Optical Display. A term used to describe the Convex SPP system cabinet flashing lights.

most significant bit (MSB)
The significant bit contributing to the largest quantity to the value of a binary numeral. Bit numbering in Convex computers is left to

right, N-1 through 0. The MSB is N-1

move in
The operation of bringing information from memory into a cache.

MPP
See *massively parallel processor.*

MSB
See *most significant bit.*

MTV
Memory Tag Vortex. The Convex SPP system memory board.

MU
The Convex SPP system utility board.

Multiplexer (also known as a mux)
A device that takes multiple data streams and combines them into a single stream for transmission over a circuit. Another mux demultiplexes the data stream on the receiving end.

multiprocessing
The creation and scheduling of processes on any subset of CPUs in a system configuration.

multiuser mode
In Convex UNIX, the mode of operation where the supercomputer is being run in a general timesharing environment with multiple users. This is the normal operating mode for Convex UNIX. See also *single-user mode.*

mutual exclusion
A protocol that prevents access to a given resource by more than one thread at a time.

N

named pipe
A pipe that exists permanently in the file system with directory entries and path names. Because you can access these pipes by name, you can use them for a variety of applications that you cannot accomplish with ordinary pipes. Typically, named pipes are used to allow a number of processes to communicate with a daemon.

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

National Institute of Standards and Technology (NIST)
Formerly known as the National Bureau of Standards (NBS),NIST is responsible for defining the set of standard protocols required for systems used by U.S. government agencies.NIST activities produced the Government OSI Profile (GOSIP).

near-shared memory
Memory that is globally accessible by all nodes of a Convex SPP system, but has affinity for its home node. Latency is lowest from the home node, and higher from other nodes. Compare with *CPU-private memory, node-private memory,* and *far-shared memory.*

negate
An instruction that performs a 2's complement on a number.

negative true logic
In Convex hardware descriptions, negative true logic levels are always identified with an asterisk (*) immediately after the signal name. The signal is said to be true, or asserted, when the signal is at the lower level of the two voltage levels.

network
A system of interconnected computers that enables machines and their users to exchange information and share resources. Convex SPP systems provide support for *FDDI* networks.

network architecture
The logical organization of a networking system.

NIST
See *National Institute of Standards and Technology.*

noncoherent memory reference
A memory reference which 1) does not cause a cache move in, or 2) causes a cache move in, but fails to obey cache coherency rules.

O

object management computing (OMC)
A way to represent information pictorially so a user does not need to understand the mechanics of the computer/networking operating system. OMC assigns icons to each network component, including programs, documents, and drawings, letting users manipulate images to access resources.

Object Management Group (OMG)
An industry consortium created to develop an easy-to-use graphic interface for distributed networks. The group adopted New Wave Object Management Facility from Hewlett-Packard as its core technology.

offset
An integer value that is added to a base address to calculate a memory address. Offsets in Convex SPP systems are 32-bit values, and must keep address values within a single 4-GB memory *space*.

opcode
A predefined sequence of bits in an instruction that determines the operation to be performed.

Open Software Foundation (OSF)
A consortium of industry leaders working to standardize the UNIX operating system.

open systems
Systems that conform to any nonproprietary, publicly-available standards. In recent years, however, the term has come to mean only those systems that use the international standards for network architecture, as specified by the OSI model.

Open Systems Interconnection (OSI)
The ISO definition of a system that provides reliable, data-transparent, host-independent services.

operand
A register or memory location referenced by an instruction. It is the code or sequence of bits in an instruction that determines the data to be operated on.

operating system
The program that manages the resources of a computer system. The Convex SPP system uses the MPPOS operating system. MPPOS is compatible with Hewlett-Packard HP-UX operating system.

optimization
The refining of application software programs to minimize processing time. Optimization takes maximum advantage of a computer hardware features and minimizes input/output traffic and idle processor time.

optimization level
The degree to which source code is optimized by the compiler. The Convex FORTRAN and C compilers have five levels of optimization, level `-no`, `-O0`, `-O1`, `-O2`, and `-O3`. CXdb can debug code that has been optimized to any of these levels.

OSF
See *Open Software Foundation.*

oversubscript
An array reference that falls outside declared bounds.

P

PA-RISC
The Hewlett-Packard Precision Architecture reduced instruction set processor chip. This is the processor chip used in Convex SPP systems.

packet
A group of related items. A packet may refer to the arguments of a subroutine or to a group of bytes that is transmitted over a network.

page
A page is the unit of logical memory controlled by the memory management algorithms. A page is 4 K (4,096) contiguous bytes. See also *logical memory*.

pagefault
A pagefault 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 will create 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 that it was in when it determined that the data it needed was nonresident. The CPU can then continue with normal operation of the process.

page frame
A page frame is the unit of physical (main) memory in which pages are placed. Referenced and modified bits associated with each page frame aid in memory management.

parallel optimization
The transformation of source into parallel code (parallelization) and restructuring of code to enhance parallel performance.

parallel vector loop
A nested loop structure such that the innermost loop is vectorized and the outer strip-mine loop can run in parallel if a CPU is available.

parallelization
The act of creating code that enables sections of code to run simultaneously on multiple CPUs. At optimization level `-O3`, the Convex FORTRAN compiler automatically parallelizes your program

and recognizes compiler directives with which you can specify parallelization.

parameter
In C, either a variable declared in the parameter list of a procedure (function) that receives a value when the procedure is called (*formal parameter*) or the variable or constant that is passed by a call to a procedure (*actual parameter*). FORTRAN conventions refer to parameters as *arguments*.

parent node
A node in any hierarchical system that has one or more child (immediately lower) nodes. A child node is said to *descend* from its parent node.

parity
Even parity is an even number of one bits, and is used throughout the Convex I/O system.

Parity <0..7>
This is a single parity byte, and each bit is the even parity bit for one of the data bytes in the 64-bit longword. Parity<0> is the parity bit for Data <63..56>; Data <63..56> is byte 0 on a 64-bit transfer. Parity<7> is the parity bit for Data <0..7>. The SizeSel field defines the parity bit that must be driven. If the channel on the user port is configured for 16 bits, then only Parity bits <0..1> are required (and driven) for Data <63..56> and Data <55..48>.

parity RAM
This register contains an odd parity bit for each byte of data in buffer memory 0 and 1.

path
An environment variable that you set in you shell configuration file that allows you to access commands in various directories without having to specify a complete path name.

PB0
Power board 0. The lower of the 2 front power boards in Convex SPP systems.

PB1
Power board 1. The upper of the 2 front power boards in Convex SPP systems.

PB2
Power board 2. The rear power board in Convex SPP systems.

PCC
Power concentrator card. The power board that is located in the

power bay in Convex SPP systems.

PCI
Peripheral Controller Interface allows the connection of up to four I/O controllers on a single bus.

PDIR
Physical page directory. PA-RISC CPUs that implement hardware TLB miss handlers, may fetch TLB entries from a PDIR entry in the event of a TLB miss. The PDIR serves as a cache of virtual-to-physical page translations, and is maintained by the operating system.

peripheral chassis
The chassis that is situated in the cabinet between the two nodes. Also referred to as the *I/O chassis*.

physical address
A unique identifier that selects a particular device from the set of all devices connected to a particular bus.

physical address space
The set of possible addresses for a particular bus.

PID
A UNIX process ID. See *process identification*.

pipe
A pair of file descriptors that provide the mechanism for a one-way flow of data.

pipeline
An overlapping operating cycle function that is used to increase the speed of computers. Pipelining provides a means by which multiple operations occur concurrently by beginning one instruction sequence before another has completed.

pipelining
Grouping register loads together for concurrent execution.

PMC
PCI Mezzanine Card

pointer tracking
A powerful aliasing algorithm used by the Application Compiler, which tracks the memory locations to which each pointer can be set.

porting
Converting software from one type of machine to another.

Positive True Logic
All signals in Convex equipment that do not have an asterisk immediately after the signal are assumed to be positive true logic. The signal is said to be true, or asserted, when the signal level is a the higher level of the two voltage levels.

POSIX standards
A family of open systems standards developed by the IEEE Technical Committee on Operating Systems.

Power bay
The area at the bottom of the cabinet that has input power components & the 48VDC power supplies. It is not called a chassis because it is not a self-contained assembly; for example, the power supplies are mounted directly to the bottom of the cabinet.

PRAM
Parallel random-access machine. An abstraction of a parallel computer that has the properties of (a) *p* serial processors, (b) a single, shared global memory for all of the *p* processors, (c) all *p* processors can read from and write to the global memory in parallel, and (d) the processors can perform arithmetic and logical operations in parallel with each other. The major difference between a PRAM model and a real machine is that the latter has different memory access times depending on the access patterns made by the processors' execution.

privilege level
The bottom two bits of the Instruction Address Offset Queue (IAOQ) specify the privilege level of the current instruction. A privilege level of zero is most privileged. The TLB contains the minimum and maximum privilege levels required to gain access to each page in the system.

primitive
The basic geometric building blocks that make up complex objects.

priority
An ordering of events. May be applied to protection levels as well as to I/O interrupt levels.

privileged instruction
An instruction used by the operating system or privileged systems programs. It must execute at privilege level, or an exception occurs.

process
The running version of your program. The current state of the process is stored in the process image.

A process is the fundamental unit of a program that is managed by the job scheduler.

A collection of one or more execution streams within a single logical address space; an executable program. A process is made up of one or more threads.

process exception
Belongs to the currently running process and may be handled with an exception handler in that process. The exception handler is in the current ring of execution.

process identification (PID)
A unique number assigned to a process when initiated. It may be found with the operating system command referenced by various commands, such as *kill*.

process memory
The portion of system memory that is used by an executing process.

process settings
The collection of settings that control various aspects of your process, such as its floating point mode. You can modify the process settings to match the needs of your application.

processor set
In Convex SPP systems, a collection of zero or more processors from a single node. Each microkernel task is assigned to a particular processor set, and the threads of the task may only run on processors belonging to that processor set.

processor status word (PSW)
A word that contains control flags used to control and indicate the states of various computations and sequences within the processor.

process stack
A dynamic, LIFO storage list of frames that the operating system uses to track the flow of execution in your process. Each frame stores the registers and local variables from the previous execution context, the addresses used to manage the current stack frame, and a pointer to the previous stack frame.

process working directory
The directory from which the process is run. It is also the base directory for the relative path names of all files accessed by your process.

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

program working environment
The environment that controls various aspects of your program. You can tailor it to meet the needs of your program.

program segment
A block of code identified by a program-segment directive.

program unit
A subroutine, function, or main section of a program.

prompt
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.

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.*

push
The act of storing the present machine state on a stack.

Q

quadrant
One quarter of a space. In Convex SPP systems, each quadrant of memory is 1 GB. in size.

queue
A data structure in which entries are made at one end and deletions at the other. Often referred to as first-in, first-out (FIFO).

quotient
The result of a division operation.

R

raw socket
An IPC facility that provides users with access to the underlying network protocols.

read
A memory operation in which the contents of a memory location are copied and passed to another part of the system.

realtime programing
Programming for computation to be performed during an external process, so that the computation may be used to respond to the process in a timely fashion

recurrence
A cycle of dependencies among the operations within a loop. (See also *data dependency*.)

recursion
Continued repetition of the same operation or group of operations. It is a type of operation flow where subsequent calculations depend on previous results, which may prohibit some vectorization/parallel operations.

reduced instruction set computer (RISC)
This is 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.

re-entrancy
The ability of a program unit to have multiple versions in existence that may execute in parallel. Each version maintains a thread-private copy of its local data and a thread-private stack to store compiler-generated temporary variables.

register
A hardware entity that contains an address, operand, or instruction status information.

relocatable expressions
Expressions that reference addresses that cannot be resolved at assembly time.remote computer system

Remote Procedure Call (RPC)
A facility provided by the Convex NFS product that allows a client process to have another process execute a procedure call, as if the caller had executed the procedure call in its own address space.

reset
The process of establishing a known state in a machine register.

RISC
See *reduced instruction set computer.*

root directory
The base directory in UNIX from which all other directories stem, directly or indirectly.

root file system
In UNIX, the file system associated with the root directory.

root node
A node in a hierarchy that has no *parent node.*

router
A hardware device that directs packets across networks. A router attempts to choose the most effective route to a packet destination if multiple routes exist.

A hardware device used in Convex SPP systems to pass data across *nodes.*

row-major order
Memory representation of an array such that the rows of an array are stored contiguously. For example, given a two dimensional array `A(3,4),A(1,4)` immediately precedes `A(2,1)` in memory.

RPC
See *Remote Procedure Call.*

rsh
The UNIX remote shell command. This command enables you to execute a command on one computer from the command line of another computer.

runtime
A software module that is referenced as a procedure. A runtime represents a required function that is not directly supported by the hardware, but is required by the software.

S

Scalable parallel processing (SPP)
A computer architecture that permits an application to run with a relatively small number of processors or in a processor array containing hundreds to thousands of processors. A key design goal for SPP systems is to enable performance to increase linearly with respect to its number of processors.

SBus
The Sun SPARC Bus. This is the bus standard used in SPARC-compliant computers such as Sun 4. SPARC stands for Scalable Processor Architecture. Convex SPP systems will use SBus for I/O connections to disk drives and external networks. SBus is both an electrical standard and a mechanical standard

which includes defining the board size, connector type, and other physical details.

SCI
Scalable Coherent Interface. This is defined by IEEE standard 1596-1992. The interface is physically defined as a pair of 18-bit, differential ECL, unidirectional links which are clocked at 250 Mhz. Each link provides 16 bits of data with two control signals. Data is sampled on both the rising and falling edges of the clock. This interface is used in Convex SPP systems.

SCSI
Small computer system interface (SCSI).

server
A process that fulfills a request issued by a client process, and transmits a response back to the client. Also called a *service provider.*

server set
In a Convex SPP system, the set of processors on a node that will run server tasks for the node. Server tasks and operating system microkernel tasks run only on processors belonging to the server set. Every node has a server set, and it must contain at least one processor at all times.

server stub
Used in implementing remote procedure call (RPC) facilities, the server stub waits for an RPC request from a client, executes a local procedure call on behalf of the client, and returns results to the client. Server stubs are used to abstract the details of passing messages over the network.

s-field
A field in some PA-RISC instruction formats that specify the memory *space* the instruction operates on.

shared virtual memory
See *globally shared memory (GSM).*

sharing list
A structure used by SCI for maintaining cache coherence state information. The sharing list is structured as a linked list.

shift
A class of instructions used to shift the contents of a register right or left.

signal
The standard UNIX entity used to control a program. Refer to the

signal(3c) man page for a list of signal names.

SIMD (single instruction stream multiple data stream)
A computer architecture that performs one operation on multiple sets of data. One processor is used for the control logic and the remaining processors are used as slaves. Compare with SIMD.

single (s)
A single-precision floating-point number stored in 32 bits. See also *double.*

single-user mode
In Convex computer systems, the mode of operation in which the computer system 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.*

sinking
An optimization process that moves a store from within a loop to the basic block following the loop.

snooping
The act of externally flushing or invalidating a cache line from a processor cache. The flushing or invalidating transaction is issued by the agent to the processor and is referred to as a "snoopy transaction". This action occurs when one processor in the system loads or stores to a *dirty* line in the cache of another processor, or when a processor stores to a line that is shared by one or more processors in the system.

SMP
A symmetric multiprocessor.

soft error
Correctable single-bit memory error. May further be defined as transient (non-reproducible) or stuck (reproducible). Recorded in the softlog.

software device driver
A program that controls the operation of attached I/O peripheral devices.

source
A register or memory location used as an input to an instruction.

source code
The uncompiled version of a program, written in a high-level language such as C or FORTRAN.

source file
A file that contains program source code.

source unit
A logical subdivision of source code. The five types (or granularity) of source units are routine, loop, block, statement, and expression.

space
A contiguous range of virtual addresses within the system wide virtual address space. Spaces are 4 GB. in size in MP-1.

space id
A 16, 24, or 32 bit value which occupies the upper portion of a virtual address and specifies the virtual space portion of the virtual address.

space registers
Eight registers (SR[0]..SR[7]) defined in the PA-RISC architecture that are used by CPU instructions to reference different spaces.

span
The distance between a jump or branch instruction and its target.

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.

spawn
To create new tasks.

SPOK
System Power OK. The name of the cable in Convex SPP systems that connects the PPC to each of the nodes for the purpose of passing them the status of the power supplies in the power bay.

SPMD
Single program multiple data. A single program executing simultaneously on several processors. This is usually taken to mean that there is redundant execution of sequential scalar code on all processors.

SPP
See *scalable parallel processing (SPP).*

SST
System Scan Test. A diagnostic procedure that uses the scan (diagnostic) bus.

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). See also *process stack.*

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 me

stream socket
IPC facility that provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries.

strip length, parallel
The amount by which the induction variable of the inner loop is advanced on each iteration of the outer loop.

strip mining
The transformation of a single loop into two nested loops. Convex compilers perform parallel strip-mine optimizations. In a parallel strip-mine optimization, the outer loop (the parallel strip-mine loop) advances the initial value of the inner loop's induction variable by the parallel strip length. When more than one processor is detected (or specified with the `-ep` option), the parallel strip length is based on the trip count of the loop and the amount of code in the loop body.

subcomplex
In Convex SPP systems, a logical entity that provides control over the allocation of processors and physical memory to different applications and users.

subcomplex server
In Convex SPP systems, a subsystem within the process manager that provides the system call interface for creating and manipulating sub-complexes and server sets on the system.

subroutine
A frequently used software module that is called from various places in a program.

superscalar
A class of *RISC* processors that allow multiple instructions to be issued on each clock period.

superuser
The UNIX term for a special privilege level that allows the system manager to perform system maintenance functions that ordinary users cannot perform.

symbolic debugger
A debugger that can map source code to executable machine instructions. CXdb is a symbolic debugger with the added functionality and capability to debug optimized code symbolically.

symbolic link
A connection that associates a node with a tape device.

A feature present in most UNIX systems that allows a file to exist as a link to a file in another location.

synchronization
A way to keep two threads from accessing the same critical region simultaneously. You can synchronize programs using compiler directives or assembly-language instructions. You do so, however, at the cost of additional overhead; synchronization may cause at least one CPU to wait for another.

synchronous mode
Execution mode in which a user calls a library function and control does not return until after the corresponding asynchronous event occurs.

system manager
The person responsible for the management and operation of a computer system.

system console
The terminal or workstation that serves as a communication device between the system manager and the computer system. On Convex SPP systems, the *test station* serves as the system console.

system sub-complex
In a Convex SPP system, a sub-complex that is automatically created at boot time by the operating system to run system processes, including `init` and processes spawned by `init`. The Complex Manager will not allow users to destroy this sub-complex, nor to remove the last processor from this sub-complex.

T

task
A collection of system resources. A task itself does not execute, but at least one *thread* executes within each task.

task context
Relevant data for each task running on a host.

task ID (tid)
An integer that uniquely identifies a task across a virtual machine.

Tbyte
See *terabyte*

term
A constant or symbolic name that is part of an *expression.*

terabyte
one trillion bytes

TLB
Translation Lookaside Buffer. A hardware structure in each CPU in a Convex SPP system that contains the information necessary to translate a virtual memory reference to a physical page and to validate memory accesses.

TOC
Time of Century. This value must be consistent across an entire computer system. Convex SPP systems achieve this consistency by using cables that connect all of the nodes together in a bus fashion.

Test Station
The term that is used for the workstation that is used to diagnose problems and install system software on a Convex SPP system. The test station takes the place of the Service Processor Unit (SPU) on other Convex systems, and the system console on other computer systems.

thread
An independent execution stream that is fetched and executed by a CPU. One or more threads, each of which can execute on a different CPU, make up each process. Memory, files, signals, and other process attributes are generally shared among threads in a given process, enabling the threads to cooperate in solving the common problem. Threads are created and terminated by instructions that can be automatically generated by Convex compilers, inserted by adding compiler directives to source code, or coded explicitly in assembly-language programs.

thread data
Initialized data that is not shared among all threads comprising a process, but is unique to that one thread.

thread-private or thread-specific
Data that is accessible by a single thread only (not shared among the threads constituting a process). Thread-specific data allows the same virtual address to refer to different physical memory locations.

transaction
A data or control element, signal, event, or change of state that causes, triggers, or initiates an action or a sequence of actions.

trap
A type of *interruption* caused when either the function requested by the current instruction cannot or should not be carried out, or system intervention is desired by the user before or after the current instruction is executed. Typically, this condition is a result of unexpected arithmetic results. See also *exception.*

tree-height reduction
Expressions are represented internally as trees whose height corresponds to the depth of the expression. These trees are optimized by tree-height reduction or balancing. For example, the height of `A+B+C+D+E+F+G+H` could be seven: `(((((((A+B)+C)+D)+E)+F)+G)+H)`. However, the compiler orders this expression so that more than one addition can occur at the same time: `(((A+B)+(C+D))+((E+F)+(G+H)))`. The height of this tree is three. Shorter heights mean faster execution. Tree height reduction occurs only for floating-point expressions.

trip count
The number of times a loop executes.

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.

unit address
The component of a *device node name* that indicates the *device node* position within the address space defined for its *parent node.*

unsigned
A value that is always positive.

use-def chain
A data structure created and used by optimizing compilers to track every site within a program where a particular variable or array subsection could possibly be defined, starting from the point where the item is used. See also *use-def chain.*

user data
Data that originates from a service user.

user interface
The portion of a computer program that processes input entered by a human and provides output for human users.

utility
A software tool designed to perform a frequently used support function.

V

vector
An **array** with one dimension.

virtual aliases
Two different virtual addresses which map to the same physical page.

virtual machine
A collection of computers appearing as one heterogenous computer.

virtual memory
See *logical memory.*

virtual page number (VPN).
The page number portion of a virtual address.

W

wall-clock time
The time an application requires to complete its processing. If an application starts running at 1:00 p.m. and finishes at 5:00 a.m. the following morning, its wall-clock time is sixteen hours. See *CPU time.*

weak ordered memory reference
A memory reference which is not guaranteed to be completed 1) after all previous memory references are completed, or 2) before any subsequent memory references are completed.

word
A contiguous group of bytes starting on an addressable boundary. In Convex computer systems, a word is four bytes (32 bits) in

length. See also *halfword, longword.*

working set
That portion of a user program currently in physical memory. Typically, the working set is much smaller than the user program

workstation
A stand-alone computer that has its own processor, memory, and possibly a disk drive.

write
A memory operation in which a memory location is updated with new data.

X

X Window System
A graphic windowing system designed for networked computer systems with bitmap displays.

Y

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.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

Q

R

S

T

U

V

W

X

Z

ORDER NUMBER
DHW-014

DOCUMENT NUMBER
081-023430-003