CONVEX Theory of Operation
(C200 Series)

Document No. 081-005030-000

Second Edition
September 1990

CONVEX Computer Corporation
Richardson, Texas USA

CONVEX Theory of Operation
(C200 Series)
Order No. DHW-095
Second Edition

Printed in the United States of America

Revision Sheet

CONVEX Theory of Operation
(C200 Series)

Edition	Document	Date	Description
Second Edition	081-005030-000	September 1990	Changed document number, added C200 Series information, added C232i information and various other changes.
First Edition	081-000350-200	September 1988	First release of the CONVEX Theory of Operation.

FCC NOTICE

Warning: This equipment generates, uses, and can radiate radio frequency energy and if not installed and used in strict accordance with the instructions manual, may cause interference to radio communications. It has been tested and found to comply with the limits for a Class A computing device pursuant to Subpart J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such interference when operated in a commercial environment. Operation of this equipment in a residential area is likely to cause interference in which case the user at his own expense will be required to take whatever measures may be required to correct the interference.

Do not connect external equipment to the utility outlets in CONVEX equipment cabinets. Unauthorized connection voids all agencies' emissions certifications.

Table of Contents

3 Scalar Subsystem

7 Utility Subsystem

Appendixes

A Essential System Files

B C200 Vector Instruction Operation

C Reporting Problems

List of Tables

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

Preface

Purpose and Intended Audience

The *CONVEX Theory of Operation* is the second of six volumes in the *CONVEX Maintenance Documentation (C200 Series)*. The other volumes include the following:

- *CONVEX Maintenance Documentation Overview (C201, C202, C210, C220)*
- *CONVEX Installation Guide (C200 Series)*
- *CONVEX General Maintenance Guide (C200 Series)*
- *CONVEX Troubleshooting Guide (C201, C202, C210, C220)*
- *CONVEX Removal/Replacement and IPB Guide (C201, C202, C210, C220)*

The primary purpose of this document is to assist a Field Engineer (FE) in getting a CONVEX C200 Series machine operational as quickly and easily as possible. Additionally, this volume may be used as a teaching text by the CONVEX Training department.

Scope

The CONVEX Maintenance Documentation applies to the CONVEX C200 Series supercomputers.

Outline

The content of each chapter is outlined below:

Chapter 1. Introduction—This chapter contains a general description of the C200 series systems and the primary functional areas: a first-level theory of operations.

Chapter 2. C200 Series Architecture Overview—This chapter provides an architectural overview of the C200 series single processor and multiprocessor systems. Specifically, this architecture overview discusses all the additions to the C100 Series architecture that comprise a C200 Series architecture.

The following chapters provide a second-level theory of operation generally sufficient to understand what a functional block does, how it works, and how it integrates into the C200 Series system. This is not a rigorous treatment of the subject; it stops some distance short of third-level theory of operations (bit-component-schematic level).

Chapter 3. Scalar Subsystem—This chapter presents the C200 Series Address and Scalar Unit (ASU) subsystem: the scalar processor, the instruction processor, and the memory interface. The ASU consists of four circuit boards: the ASP, IPP, SFU, and DCU.

Chapter 4. Vector Subsystem—This chapter presents the vector subsystem: the vector processor. The vector subsystem consists of two boards: the VPC and VPD.

Chapter 5. Memory Subsystem—This chapter presents the memory subsystem which consists of a minimum of one (and up to four) even and odd MCM board pairs.

Chapter 6. I/O Subsystem—This chapter presents the Input/Output (I/O) subsystem. The I/O subsystem consists of the PIA, SP2, SP4, and various CCUs (MIOP, VIOP, and HSP).

Chapter 7. Utility Subsystem—This chapter presents the utility subsystem. The utility subsystem consists of the SP2/SP4, CPX and SCM boards (for one and two CPU systems) or the SP4, CUO, CUE, and ESM boards (for three and four CPU systems).

Appendix A. Essential System Files—This appendix contains information on essential ConvexOS system files necessary for normal operation of the ConvexOS operating system.

Appendix B. C200 Vector Instruction Operation—This appendix contains information about the C200 vector instruction operation.

Appendix C. Problem Reporting—This appendix contains information about using the *contact* facility to report problems.

Notational Conventions

The notational conventions used in this text are listed below:

- Bit numbering is left to right, `N-1` through `0`. The most significant numerical bit is `N-1`, the least significant `0`. The bit numbering represents the binary weight of a position.
- Bit fields are specified using the following convention: *name*<*x..y*> where the bit field is *name* from bits *x* through *y*.
- Individual bit positions within a register are denoted by specific positions separated by commas. For example, REG<15,4,0> denotes bits 15, 4, and 0 of REG.
- Byte numbering is from left to right
- A *bit* is a single binary value or entity
- A *byte* is 8 bits
- A *halfword* is 16 bits
- A *word* is 32 bits
- A *longword* is 64 bits
- *Single precision* is a 32-bit floating point word
- *Double precision* is a 64-bit floating point longword
- An *instruction* is a multihalfword operand
- A bit is *set* when it contains a binary value of `1`.
- A bit is *clear* when it contains a binary value of `0`.
- All memory and I/O addresses are written in hexadecimal notation unless explicitly stated otherwise.

- All register contents are written in hexadecimal notation unless explicitly stated otherwise.
- A *register* is a programmer-visible hardware storage element internal to the processor
- *Physical memory* is the physical memory installed in the processor
- *Virtual memory* is the perceived amount of physical memory as seen by the application programmer
- The symbol *K* is an abbreviation for *kilo* or 1,024
- The symbol *M* is an abbreviation for *mega* or 1,048,576
- The symbol *G* is an abbreviation for *giga* or 1,073,741,824
- A *stack* is a linked-list group of words useful for dynamic allocation and deallocation of memory
- A *return block* is a collection of registers that is pushed or popped from a context stack in response to an instruction or other event
- *Reserved* or *undefined* convey what to expect, if anything, from unused fields in registers, reserved memory, or reserved I/O space. Algorithm implementation based on the use of undefined or reserved fields is not recommended.

Warnings, Cautions, and Notes

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

WARNING

Warnings highlight procedures or information necessary to avoid injury to personnel. A warning immediately precedes the critical information and includes a description of the hazzard.

CAUTION

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

NOTE

A note highlights useful information that is supplemental in nature. A note may immediately precede or follow the information that is being highlighted.

Associated Documents

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

- *CONVEX C1/C120/C210A Architecture Reference*, Product No. DHW-005
- *CONVEX Processor Operations Guide (C100 Series, C200 Series)*, Product No. DHW-015
- *CONVEX UNIX Primer*, Product No. DSW-133
- *CONVEX Computer Site Preparation Guide (C210, C220, C230, C240)*, Product No. DHW-009
- *CONVEX SPU UNIX Utilities Manual*, Product No. DHW-021
- *CONVEX UNIX Tutorial Papers*, Product No. DSW-002
- *CONVEX Network File System System Manager's Guide*, Product No. DSW-113
- *CONVEX Diagnostic Utilities Manual (C130, C210, C220)*, Product No. DHW-082
- *CONVEX System Manager's Guide*, Product No. DSW-004

Ordering Documentation

To order the most current version of this or any other CONVEX document, use the CONVEX product number. If the product number is not known, order by the exact title. In some situations, the most current version may not be desired. To receive a specific version of a manual, order the manual by its CONVEX document, or part, number, which can be obtained by contacting the local CONVEX office or by calling the Technical Assistance Center.

The product number for this manual is DHW-095.
The document number for this manual is 081-005030-000.

CONVEX documents can be ordered by mail by sending a request to:

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

Technical Assistance

Hardware and software support can be obtained through the CONVEX Technical Assistance Center (TAC). In the continental United States, the TAC can be reached by calling 1(800)-952-0379. From locations in Alaska, Hawaii and Alaska, call 1-(214)-952-4379. From all other locations, contact the nearest CONVEX office.

Acknowledgments

I would like to thank the following people for their contributions to this manual:

- Technical contributors: Jeff Gruger, Carl Jackson, Steve Fieler, Gary Gostin,
- Document review team: Larry Bonura, Al Haddix
- Contributing writers: Bill Benson, Randy Stiles
- Hardware documentation staff: Larry Bonura, Josie Davis, Peggy Gilloon

Without the efforts of all the aforementioned, this document would not have been possible.

Roger Morris, Technical Writer
CONVEX Hardware Documentation

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1
Introduction

1.1 Overview

The CONVEX C200 Series computer systems use CPUs that contain integrated scalar and vector processors. A system will have at least one, and as many as four CPUs that share a common memory and I/O. The multiple-CPU systems use tightly-coupled, self-allocating processors. The C200 Series computer systems include the following processors: C201, C202, C210, C220, C230, C232i and C240 processors. This document discusses the CONVEX C200 Series processors.

The C200 Series multiprocessor system provides the user with the capability to exploit program parallelism by supporting the allocation of multiple threads of execution within a single process. These same functions are used by the operating system to provide simultaneous execution of multiple processes for system load balancing and timesharing. The multiprocessor functions of the CONVEX C200 Series processors were designed to:

- Provide a set of instructions for thread creation which require no knowledge of the physical CPU configuration.
- Optimize the execution cycles of each CPU by defining the instruction set so that no CPU ever waits for another CPU to become available.
- Provide fast thread creation and termination functions to users so they may take advantage of small regions of parallelism within programs without operating system involvement.
- Support for any number of CPUs within a C200 Series system for configurability, software compatibility, performance, and future expansion.

The fundamental principal of operation for multiprocessing is that each CPU within the a C200 Series system is solely responsible for scheduling itself. No "master process" exists which finds idle CPUs and schedules processes or threads on them. Each process "posts" the need for another thread to join in its computation, or in the case of the operating system, switch from one process context to another.

Subsections 1.1.1 through 1.1.5 contain brief summaries of the C200 Series architecture. These summaries include information about important design features of the the CONVEX C200 Series system architecture. The subsections are arranged in the following order:

- Data Representations (1.1.1)
- Register Sets (1.1.2)
- Memory Management (1.1.3)
- Multiprocessor Management (1.1.4)
- Exceptions and Interrupts (1.1.5)

1.1.1 Data Representations

CONVEX processors recognize three numeric data types: floating point, unsigned fixed point integer, and signed fixed point integer. The system architecture interprets signed fixed point numbers as 2's complement representation. The CONVEX processors support four fixed point integer precisions:

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

The CONVEX C200 Series processors support both native and IEEE standard floating point number representations in two formats:

- **Single Precision Word (32 bits)**
- **Double Precision Longword (64 bits)**

Both formats are interpreted as binary normalized fractions with an implicit "1" bit in the most significant bit position of the fraction, and an exponent that is a biased power of 2 scale factor.

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

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

1.1.2 Register Sets

The processor logic contains general register sets and status registers. The registers are partitioned according to the operand to be manipulated. The operands can be addresses (and scalar indices), scalars, or vectors. There are three general register sets. They are:

- Address registers (eight 32-bit registers)
- Scalar Registers (eight 64-bit registers)
- Vector registers (eight registers, each with 128 64-bit elements)

1.1.3 Memory Management

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

The virtual address space of the CONVEX system architecture may contain a valid virtual address, but the referenced data may or may not be in main or physical memory. Memory is managed on a fixed page size basis. To manage memory, the CONVEX system architecture defines and supports several features:

- **Segment**—A logically contiguous 512-Mbyte block of memory
- **Segment Descriptor Register (SDR)**—A 32-bit register that contains the pointer to the first level page table
- **Page**—A 4-Kbyte contiguous block of memory. The bytes within a page are both logically and physically contiguous.
- **Page Frame**—A page that is stored in main memory
- **Page Tables**—A page that contains entries called Page Table Entries (PTE). A page table begins on an integral page boundary and is contained in one page frame or less. First level page tables contain PTEs that have pointers to second-level page tables; second-level page tables contain pointers to physical page frames.
- **Page Table Entry (PTE)**—A 32-bit entry that conveys information to determine, for instance, whether or not a page is resident in main memory
- **Referenced and Modified Bits**—Bits that determine whether a valid memory read or write has occurred
- **Address Translation Unit (ATU)**—A programmer-invisible address cache that maintains the most recently used logical-to-physical address translations

Since the operating system is embedded within the user logical address space, it must be protected from the user. The memory protection system also protects user programs from each other, while supporting timesharing and operating system structures. This system is based on hierarchical structures called *rings* and provides the following:

- Supports embedding the operating system in the user logical address space
- Contains certain access violations to the user's process
- Permits implementing the ConvexOS operating system
- Enhances operating system call processing by reducing the time for context switching

1.1.4 Multiprocessor Management

The multiprocessor management hardware of a CONVEX C200 Series processor provides the operating system and user with a flexible instruction set for dynamic CPU allocation, de-allocation, and communication. Each CPU in a CONVEX C200 Series multiprocessor operates independently, as a standard CONVEX 64-bit supercomputer. The multiprocessor management hardware combines these CPUs into a set of tightly-coupled processors, with shared memory, to provide a parallel execution environment.

In order to manage multiple CPUs, the CONVEX multiprocessor architecture defines and supports several attributes:

- **CPU**—One physical central processing unit.
- **Complex**—The entire set of physical CPUs in a configuration.
- **Sub-Complex**—Any subset of a Complex.
- **Process**—A collection of instruction streams within a single logical address space.
- **Thread**—Any single instruction stream executing within a process.

Multiprocessing is defined as the creation and scheduling of individual processes on any Sub-Complex. In a CONVEX multiprocessing environment, the total number of threads supported in a process can never exceed the number of CPUs within the Complex.

1.1.5 Exceptions and Interrupts

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

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

All I/O data references by the processor (CPU) are memory mapped. This means there are no explicit I/O instructions. The I/O registers and memory status bits are referenced through the appropriate virtual-to-physical address mapping. The I/O register space spans 4 Gbytes, all of which may reference I/O registers. Generally, I/O operand references must be on an integral boundary so the least significant address bits (that are equal to the precision of the referenced operand) are all zeros.

1.2 System Hardware Descriptions

This section contains descriptions of the physical components of the CONVEX C200 Series supercomputer cabinets. Refer to the *CONVEX Computers Site Preparation Guide* for additional information on CONVEX equipment.

1.2.1 CONVEX Cabinets

CONVEX C201, C202, C210, and C220 supercomputer cabinet configurations consist of one *processor* cabinet, and one or more *expansion* cabinets.

CONVEX C230, and C240 supercomputer cabinet configurations consist of two *processor* cabinets, and one or more *expansion* cabinets.

A CONVEX C232i supercomputer cabinet configuration consists of one *processor* cabinet, one *combination, processor and I/O* cabinet, and one or more *expansion* cabinets.

1.2.1.1 Processor Cabinets — C201, C202, C210, C220

A CONVEX C201, C202, C210, C220 processor cabinet can contain the following hardware components:

- **Required**
  - Processor board set for **one** processor—ASP, SFU, DCU, IPP, VPC, VPD, SCM, CPX.
  - Two Memory boards—ME0 (even), MO0 (odd).
  - Service Processor Unit (SPU) board—SP2 or SP4 (SP2 = up to 2 C200 processors, SP4 = up to 4 C200 processors). The SP2 board is no longer built. It has been superseded by the downward compatible SP4 board.
  - Channel Control Unit(s) CCUs—MIOP, VIOP.
  - Service Processor Unit (SPU) cartridge tape drive and fixed hard disk (standard configuration). If a Removable Disk System (RDS) has been installed, instead of a fixed disk, the RDS is located in the expansion cabinet.
  - Power supplies and power supply control panel (AC power-controller panel).
  - Operator control panel (front control panel).
- **Optional**
  - Board set for an additional processor—ASP, SFU, DCU, IPP, VPC, VPD.
  - Additional memory boards. Note that memory boards are added to the system in pairs (one even board, and one odd board).

Figure 1–1 shows the basic hardware components of the CONVEX C201, C202, C210 and C220 models:

Figure 1–1, CONVEX C201, C202, C210, C220 Hardware Components

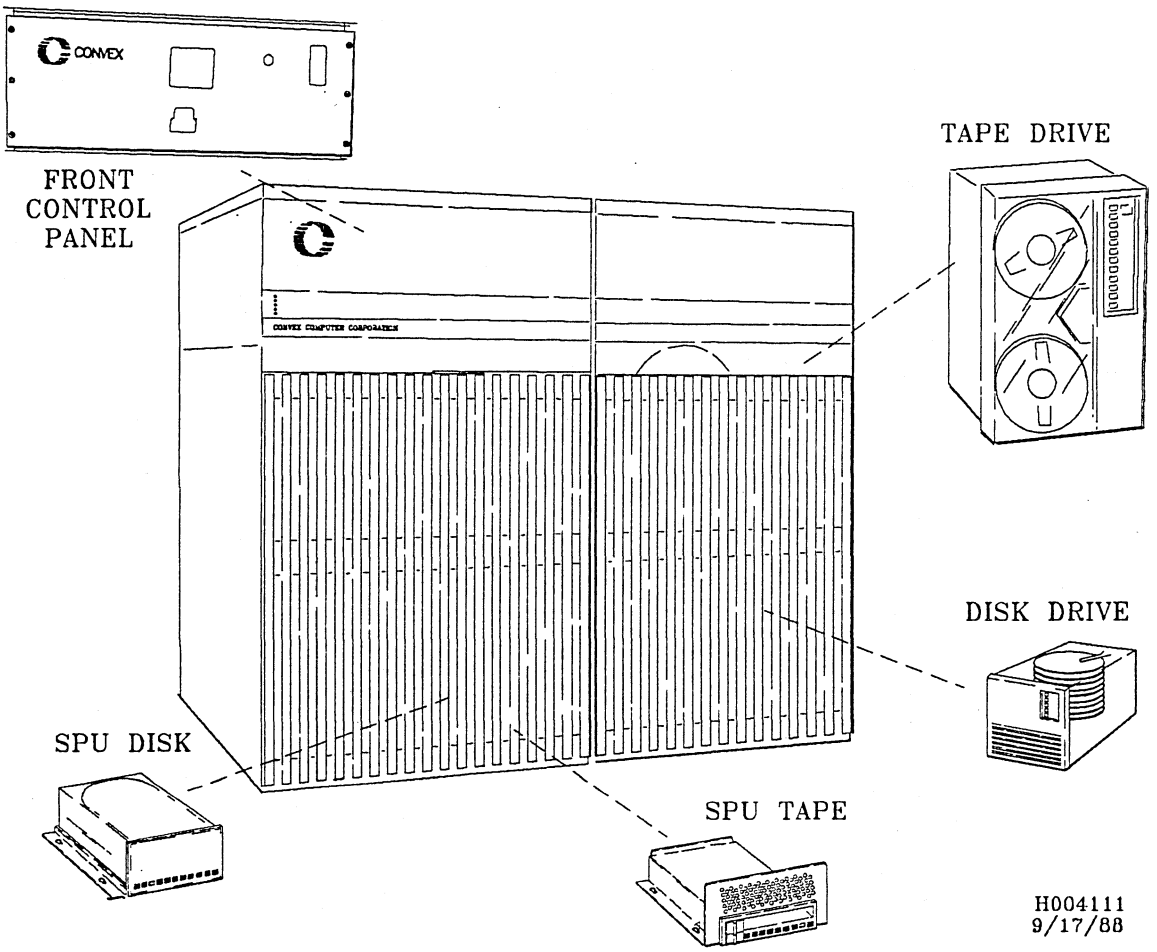
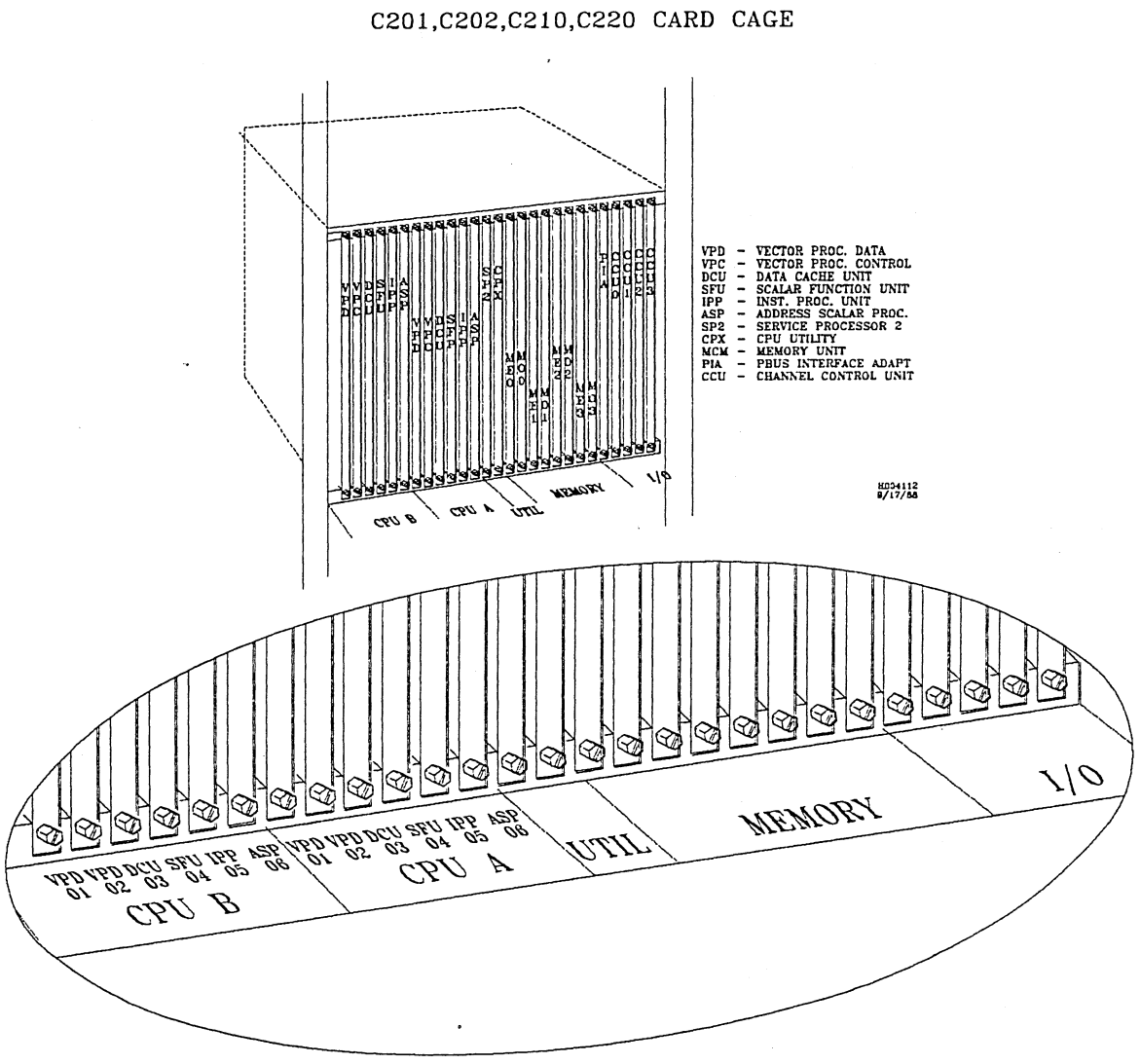


Figure 1–2 shows the card cage of a C201, C202, C210, or C220 processor cabinet with each board location called out:

Figure 1–2, Processor Cabinet, Card Cage—C201, C202, C210, C220



1.2.1.2 Processor Cabinets — C230, C240

A CONVEX C230, C240 processor cabinet can contain the following hardware components:

- **Required**
  - Processor board set for **one** processor—ASP, EFU, EDC, IPP, VPC, VPD, ESM, CUE, CUO.
  - Two Memory boards—ME0 (even), MO0 (odd).
  - Service Processor Unit (SPU) board—SP4 (SP4 = up to 4 C200 processors).
  - Channel Control Unit(s) CCUs—MIOP, VIOP.
  - Service Processor Unit (SPU) cartridge tape drive and fixed hard disk (standard configuration). If a Removable Disk System (RDS) has been installed, instead of a fixed disk, the RDS is located in the expansion cabinet.
  - Power supplies and power supply control panel (AC power-controller panel).
  - Operator control panel (front control panel)
- **Optional**
  - Board set(s) for additional processor(s)—ASP, EFU, EDC, IPP, VPC, VPD
  - Additional memory boards. Note that memory boards are added to the system in pairs (one even board, and one odd board).

Figure 1–3 shows the basic hardware components of the CONVEX C230, and C240 models:

Figure 1–3, CONVEX C230, C240 Hardware Components

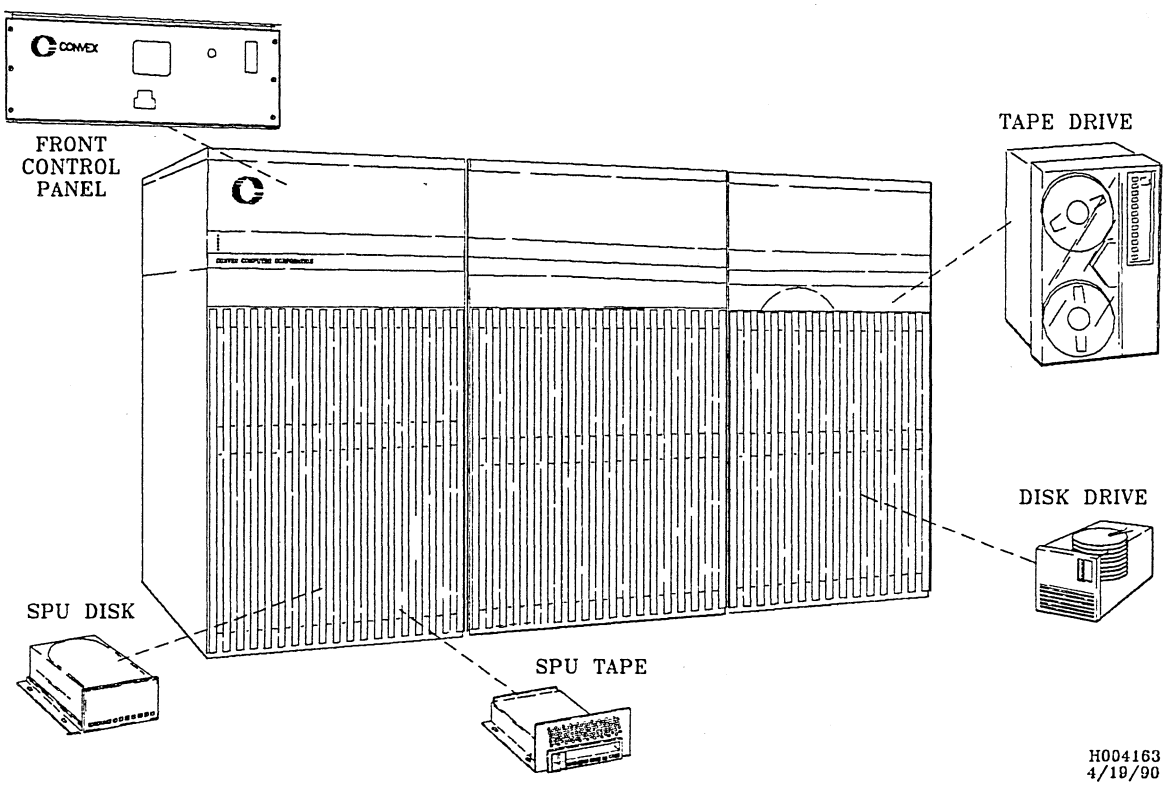
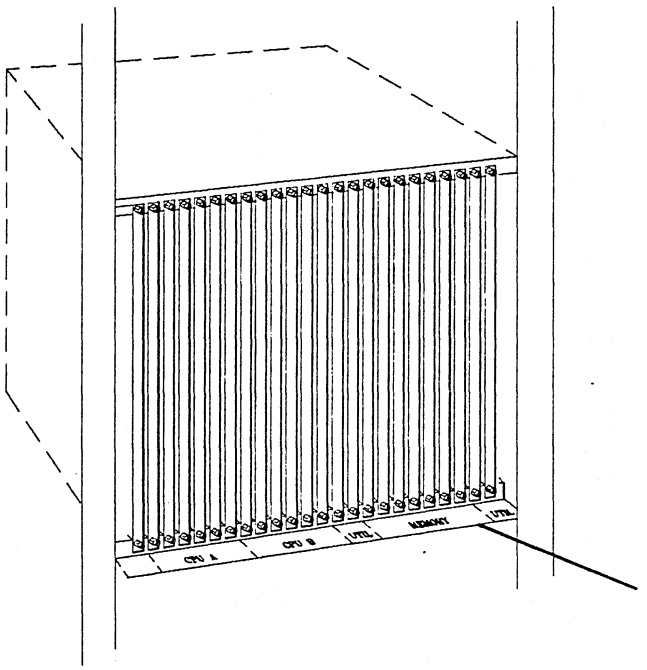


Figure 1–4 (two pages) shows the two card cages for the C230, or C240 processor cabinets, with each board location called out:

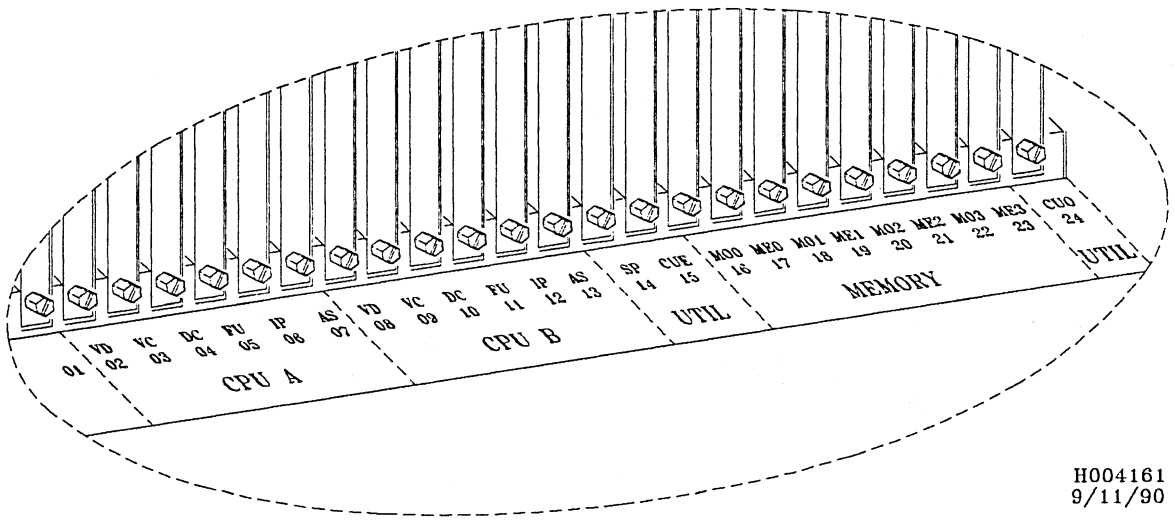
Figure 1–4, Processor Cabinet, Card Cages—C230, C240

C230/C240 FIRST CARD CAGE
(FRONT VIEW)



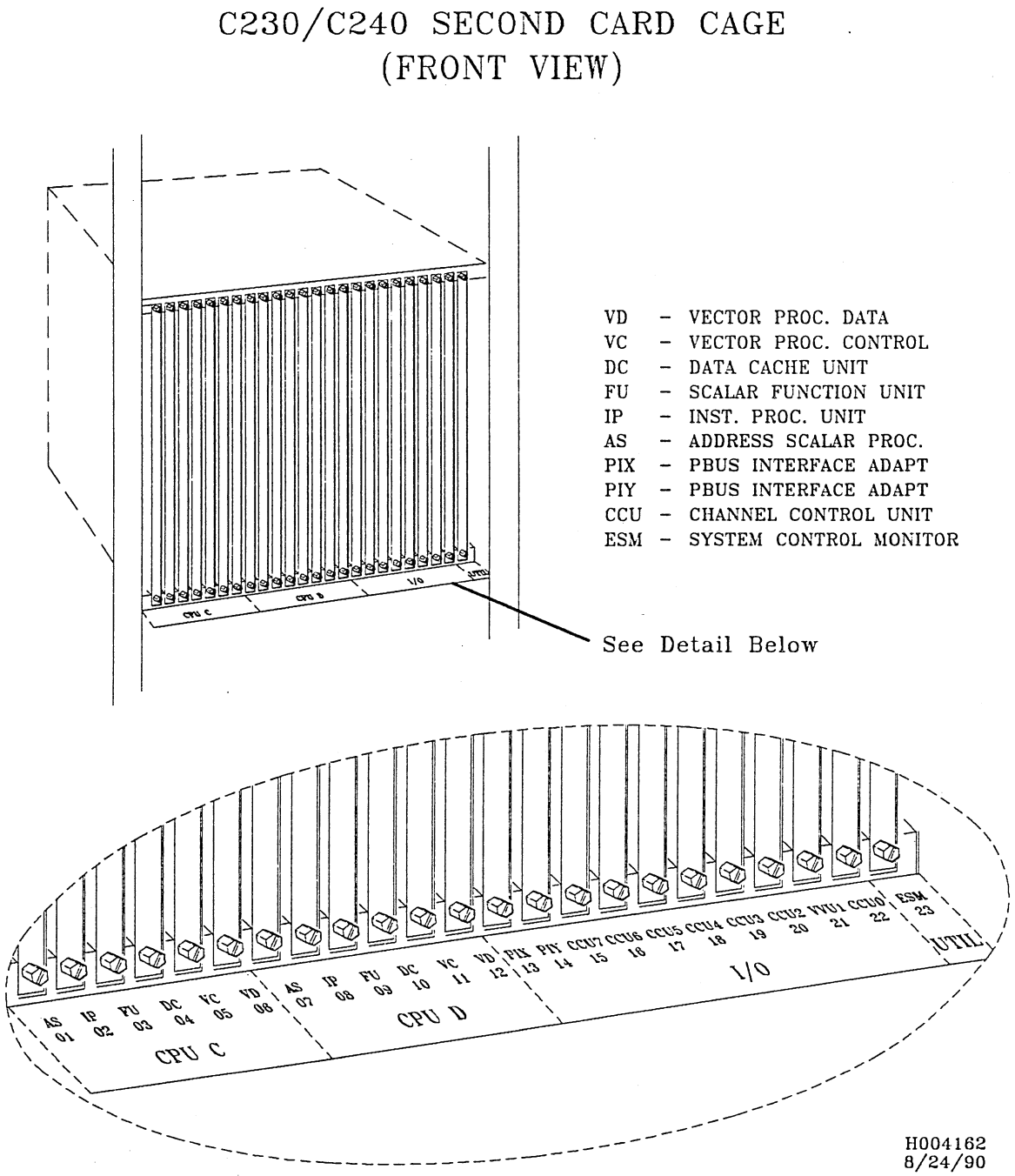
VD – VECTOR PROC. DATA
VC – VECTOR PROC. CONTROL
DC – DATA CACHE UNIT
FU – SCALAR FUNCTION UNIT
IP – INST. PROC. UNIT
AS – ADDRESS SCALAR PROC.
SP – SERVICE PROCESSOR 4
CUE – CPU UTILITY (EVEN)
CUO – CPU UTILITY (ODD)
MOx – ODD HALF, MEMORY BOARD PAIR
MEx – EVEN HALF, MEMORY BOARD PAIR

See Detail Below



H004161
9/11/90

Figure 1–4, Processor Cabinet, Card Cages—C230, C240 (continued)



1.2.1.3 Processor Cabinets — C232i

A CONVEX C232i processor cabinet can contain the following hardware components:

- **Required**
  - Processor board set for **one** processor—AS, IP, FU, DC, VC, VD, SP, CUE.
  - Two Memory boards—ME0 (even), MO0 (odd).
  - Service Processor Unit (SPU) board—SP4 (SP4 = up to 4 C200 processors).
  - Channel Control Unit(s) CCUs—MIOP, VIOP.
  - Service Processor Unit (SPU) cartridge tape drive and fixed hard disk (standard configuration). If a Removable Disk System (RDS) has been installed, instead of a fixed disk, the RDS is located in the expansion cabinet.
  - Power supplies and power supply control panel (AC power-controller panel).
  - Operator control Panel (front control panel).
- **Optional**
  - Processor board set(s) for additional processor(s)—AS, IP, FU, DC, VC, VD, SP.
  - Additional memory boards. Note that memory boards are added to the system in pairs (one even board, and one odd board).

Figure 1–5 shows the basic hardware components of the CONVEX C232i model:

Figure 1–5, CONVEX C232i Hardware Components

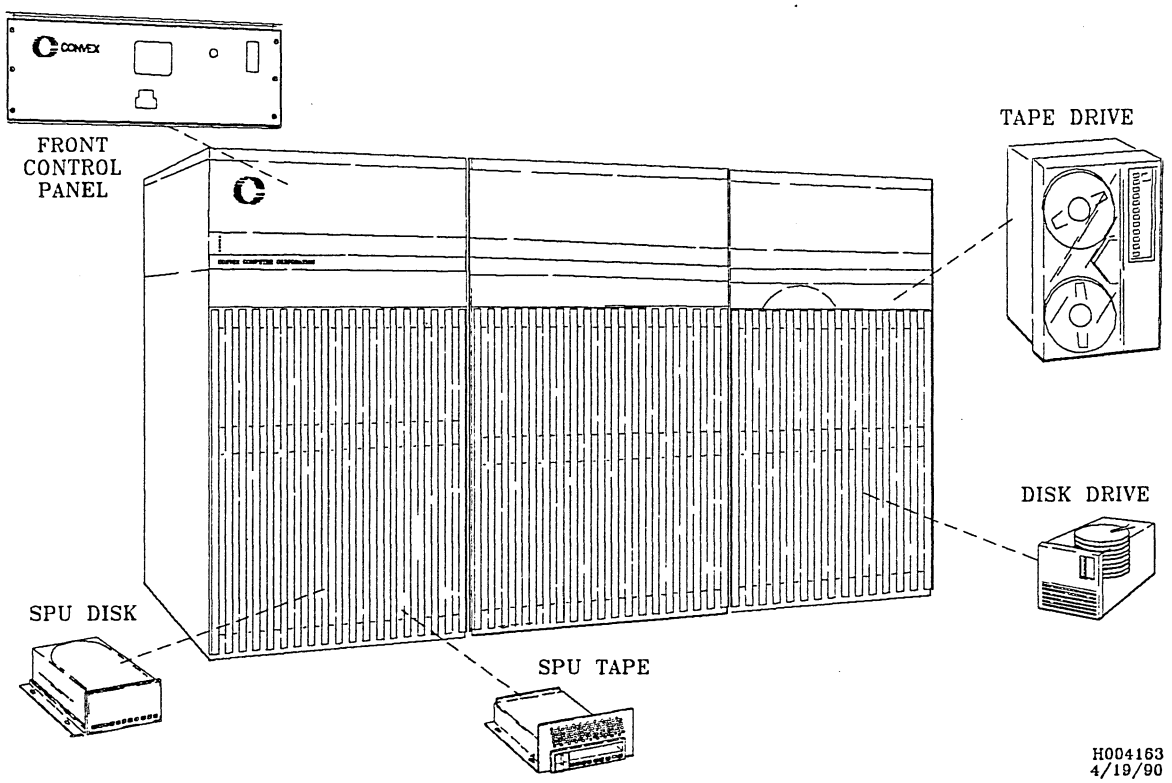


Figure 1–6 (two pages) shows the two card cages of the C232i processor cabinets, with each board location called out:

Figure 1–6, Processor Cabinet, Card Cages—C232i

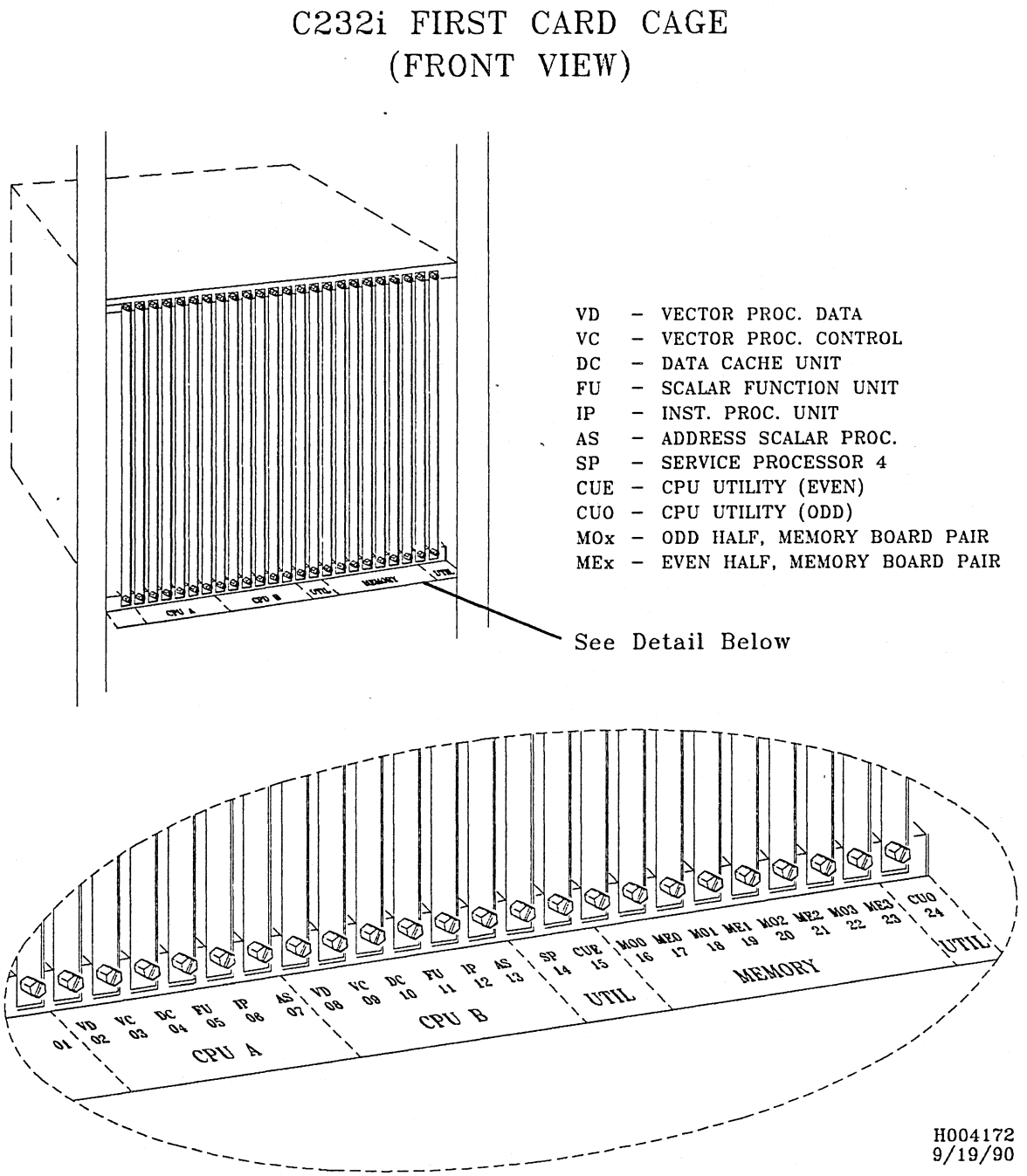
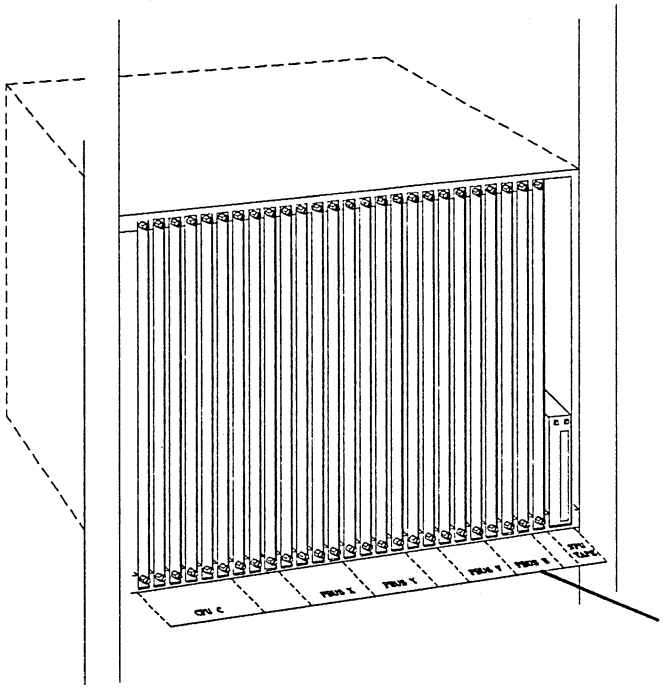


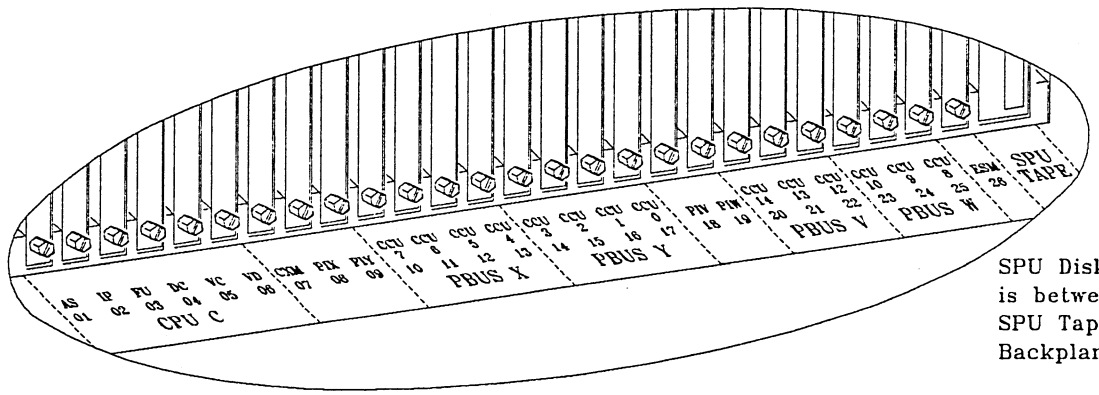
Figure 1–6, Processor Cabinet, Card Cages—C232i (continued)

C232i SECOND CARD CAGE
(FRONT VIEW)



VD – VECTOR PROC. DATA
VC – VECTOR PROC. CONTROL
DC – DATA CACHE UNIT
FU – SCALAR FUNCTION UNIT
IP – INST. PROC. UNIT
AS – ADDRESS SCALAR PROC.
PIV – PBUS INTERFACE ADAPT.
PIW – PBUS INTERFACE ADAPT.
PIX – PBUS INTERFACE ADAPT.
PIY – PBUS INTERFACE ADAPT.
CCU – CHANNEL CONTROL UNIT
ESM – SYSTEM CONTROL MONITOR

See Detail Below



SPU Disk is between SPU Tape and Backplane

H004164
9/19/90

1.2.1.4 Expansion Cabinets

CONVEX expansion cabinets provide the physical means to increase peripheral and I/O capacity. Additional expansion cabinets may be added to CONVEX systems as required. The first (or primary) expansion cabinet usually contains the following hardware components:

- **Required**
  - Multibus, VMEbus, or "combination" Multibus/VMEbus chassis
  - Peripheral controllers
  - One disk drive
  - RS-232-C interface panel
- **Optional**
  - Tape drives
  - Additional disk drives
  - Modem

1.2.2 System Power Requirements

The supercomputers within the CONVEX C200 Series have different AC power requirements and consumption levels. It is critical that proper power supplies be used to avoid damage to the equipment. Additional information on power requirements is provided in the *CONVEX Computers Site Preparation Guide.*

1.3 System Architecture

A C200 Series system configuration can contain from 1 to 4 tightly-coupled CPUs which all share the same physical memory and I/O subsystems. Each processor in a C200 Series system is self-allocating, so there is no "master" CPU.

1.3.1 C200 Series Central Processor Units

Each CPU is divided into an Address/Scalar Unit (ASU) and a Vector Processor (VP). The Address/Scalar Unit performs all functions, except vector operations, and the Vector Processor executes all vector operations. The Address/Scalar and Vector Processor can run simultaneously, and each may run multiple processes concurrently.

Each CPU (Address/Scalar Unit and Vector Processor) contains six boards that execute all address/scalar and vector functions for that CPU. The boards are named for the primary function which they perform, but may contain parts associated with other functions.

1.3.1.1 C210, C220 CPU

The CPU in a C210 or C220 system contains the following board sets:

- **ASU (4 Boards)**
  - ASP—Address/Scalar Processor
  - IPP—Instruction Pre-Processor
  - SFU—Scalar Function Unit
  - DCU—Data Cache Unit
- **VP (2 Boards)**
  - VPC—Vector Processor Control
  - VPD—Vector Processor Data

1.3.1.2 C230, C240 CPU

The CPU in a C230 or C240 system contains the following board sets:

- **ASU (4 Boards)**
  - ASP—Address/Scalar Processor
  - IPP—Instruction Pre-Processor
  - EFU—Scalar Function Unit
  - EDC—Data Cache Unit
- **VP (2 Boards)**
  - VPC—Vector Processor Control
  - VPD—Vector Processor Data

1.3.1.3 C232i CPU

The CPU in a C232i system contains the following board sets:

- **ASU (4 Boards)**
  - AS—Address/Scalar Processor
  - IP—Instruction Pre-Processor
  - FU—Scalar Function Unit
  - DC—Data Cache Unit
- **VP (2 Boards)**
  - VC—Vector Processor Control
  - VD—Vector Processor Data

1.3.2 CPU To Memory Access

A CPU in a C200 Series system accesses system memory through an individual port, for example CPU A can only access system memory through port A. Since system memory is divided into even and odd memory, a port transfers separate signals to even and odd memory. These even and odd routes are unidirectional buses for read data, write data, memory addresses, and control signals.

Figure 1–7 shows a C210, C220 system, functional block diagram of a CPU and the paths through its port to even and odd memory:

Figure 1–7, C210, C220 CPU Functional Block Diagram

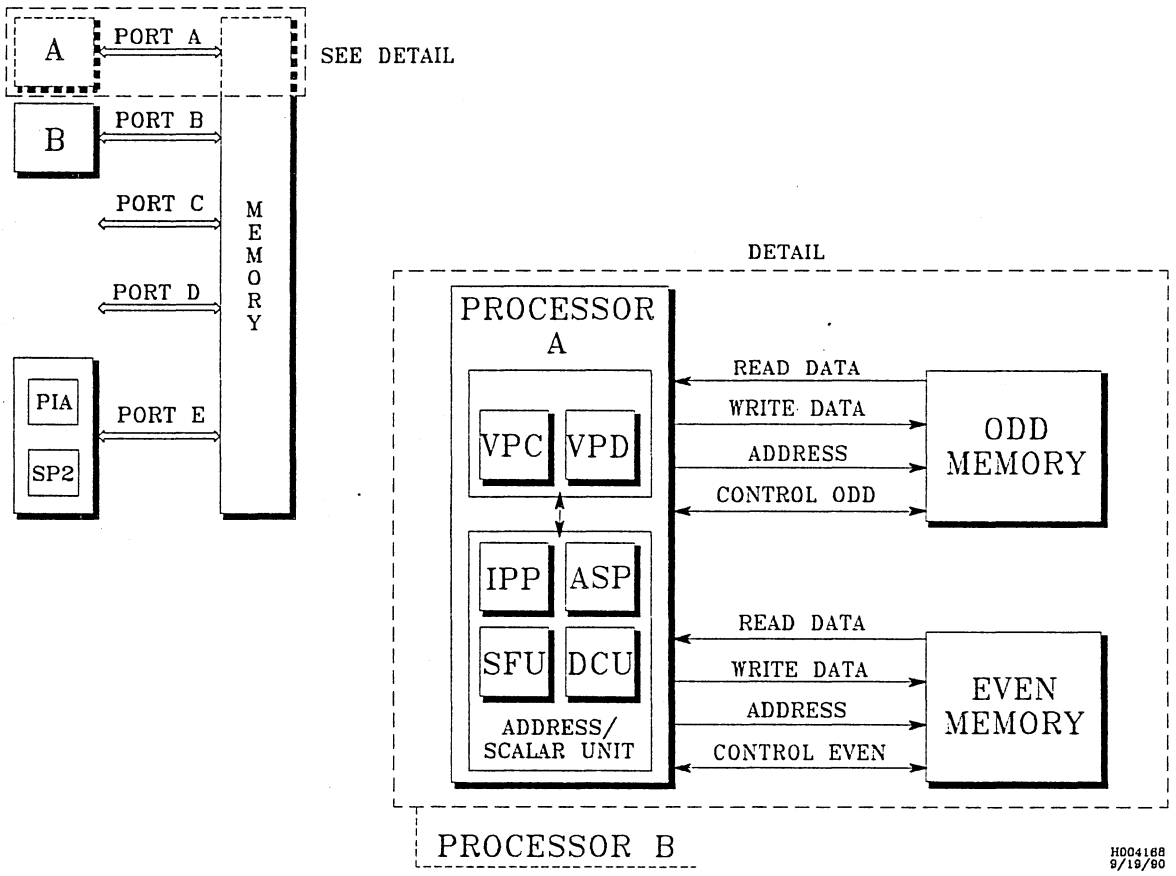


Figure 1–8 shows a C230, C240 system, functional block diagram of a CPU and the paths through its port to even and odd memory:

Figure 1–8, C230, C240 CPU Functional Block Diagram

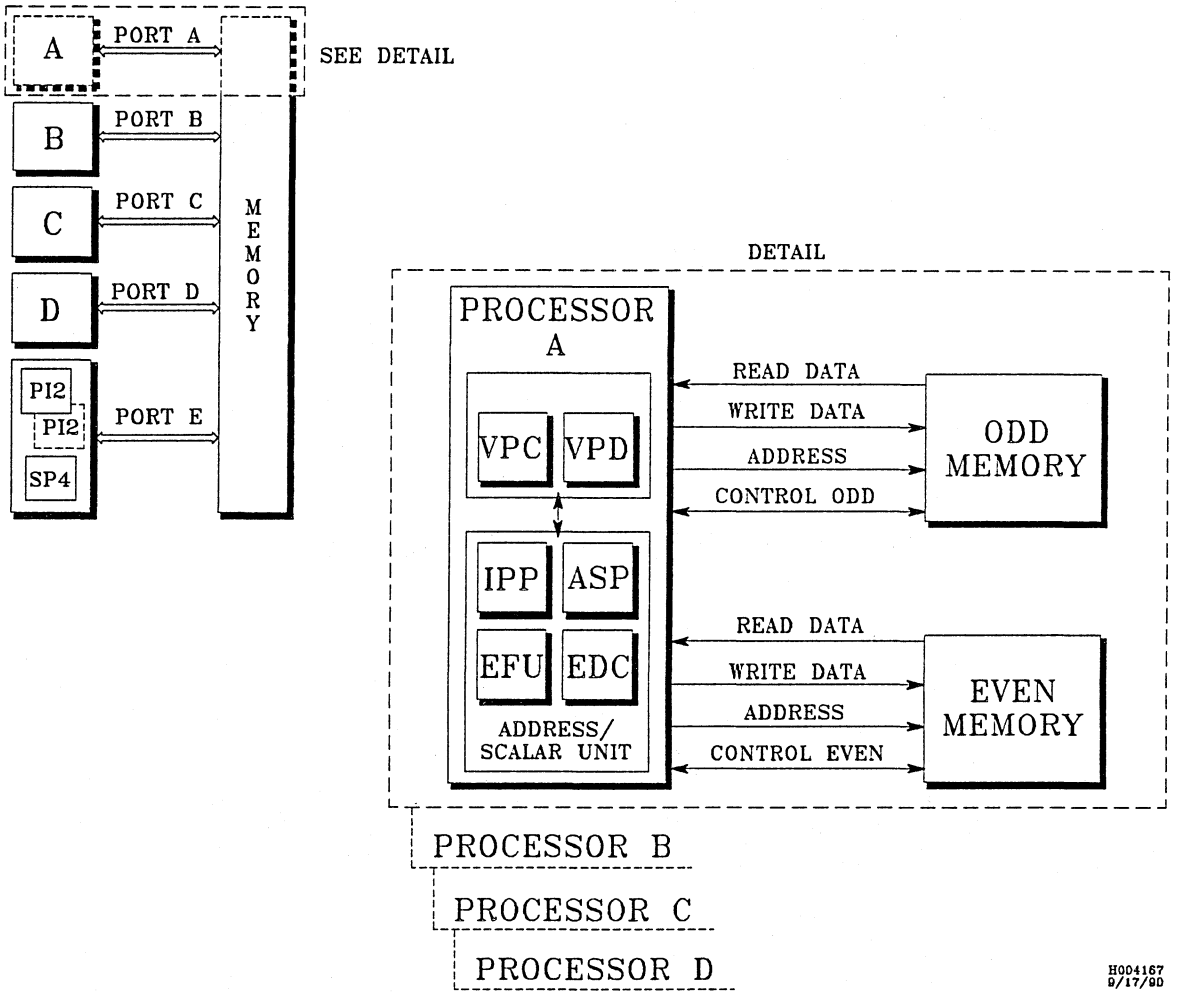
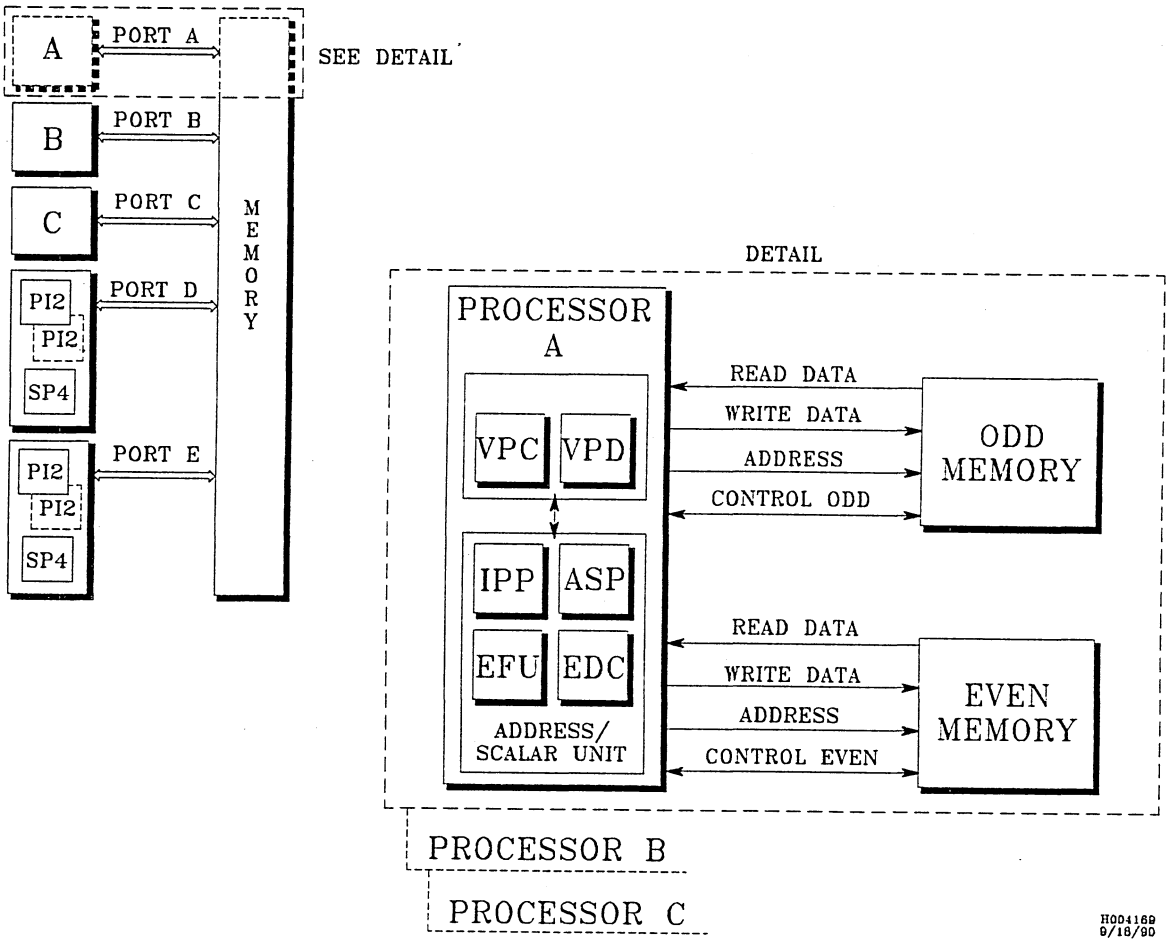


Figure 1–9 shows a C232i system, functional block diagram of a CPU and the paths through its port to even and odd memory:

Figure 1–9, C232i Functional Block Diagram



1.3.3 C200 Series Major Functional Areas

A C200 Series system is divided into five major functional areas. They are:

- Address/Scalar Unit Subsystem
- Vector Processor Subsystem
- Memory Subsystem
- Input/Output Subsystem
- Utility Subsystem

The C200 Series subsystem hardware architecture is discussed and illustrated in the text and block diagrams contained in the following sections.

Figure 1–10 (two pages) shows a C210, C220 system functional block diagram:

Figure 1–10, C210, C220 System Functional Block Diagram

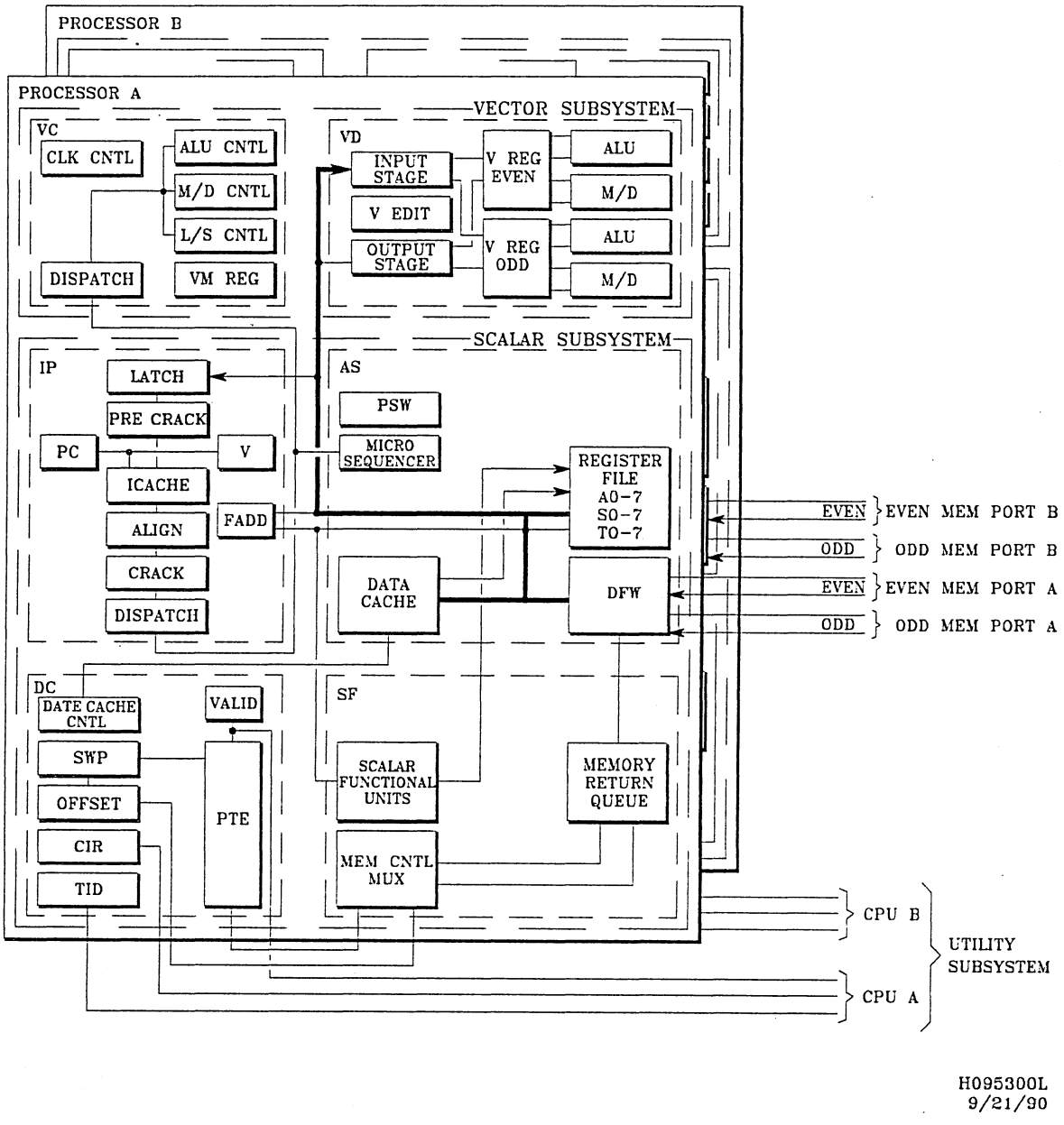
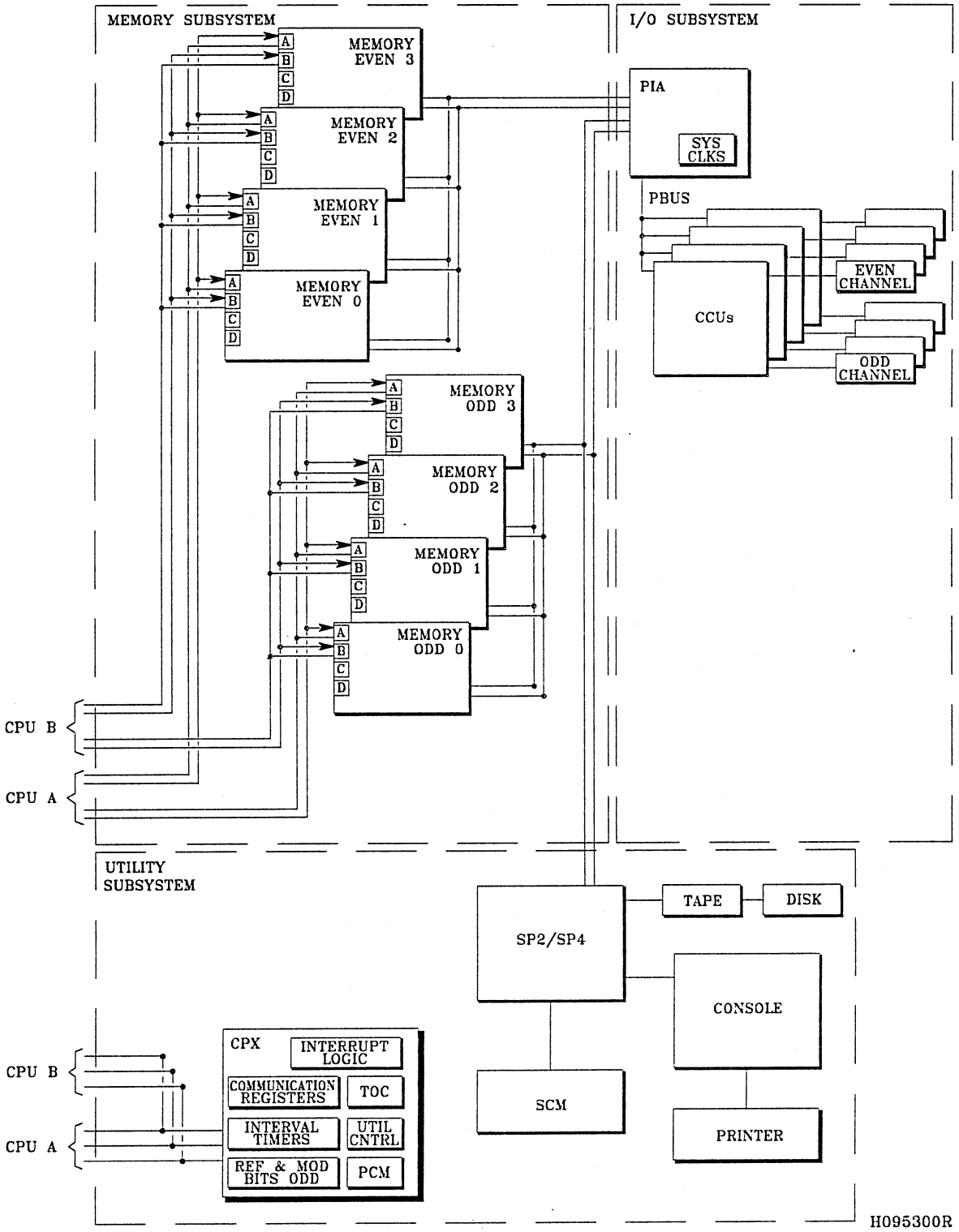


Figure 1–10, C210, C220 System Functional Block Diagram (continued)



NOTE: System Clocks are on PIA on C210/C220, not on CPX.

Figure 1–11 (two pages) shows a C230, C240 system functional block diagram:

Figure 1–11, C230, C240 System Functional Block Diagram

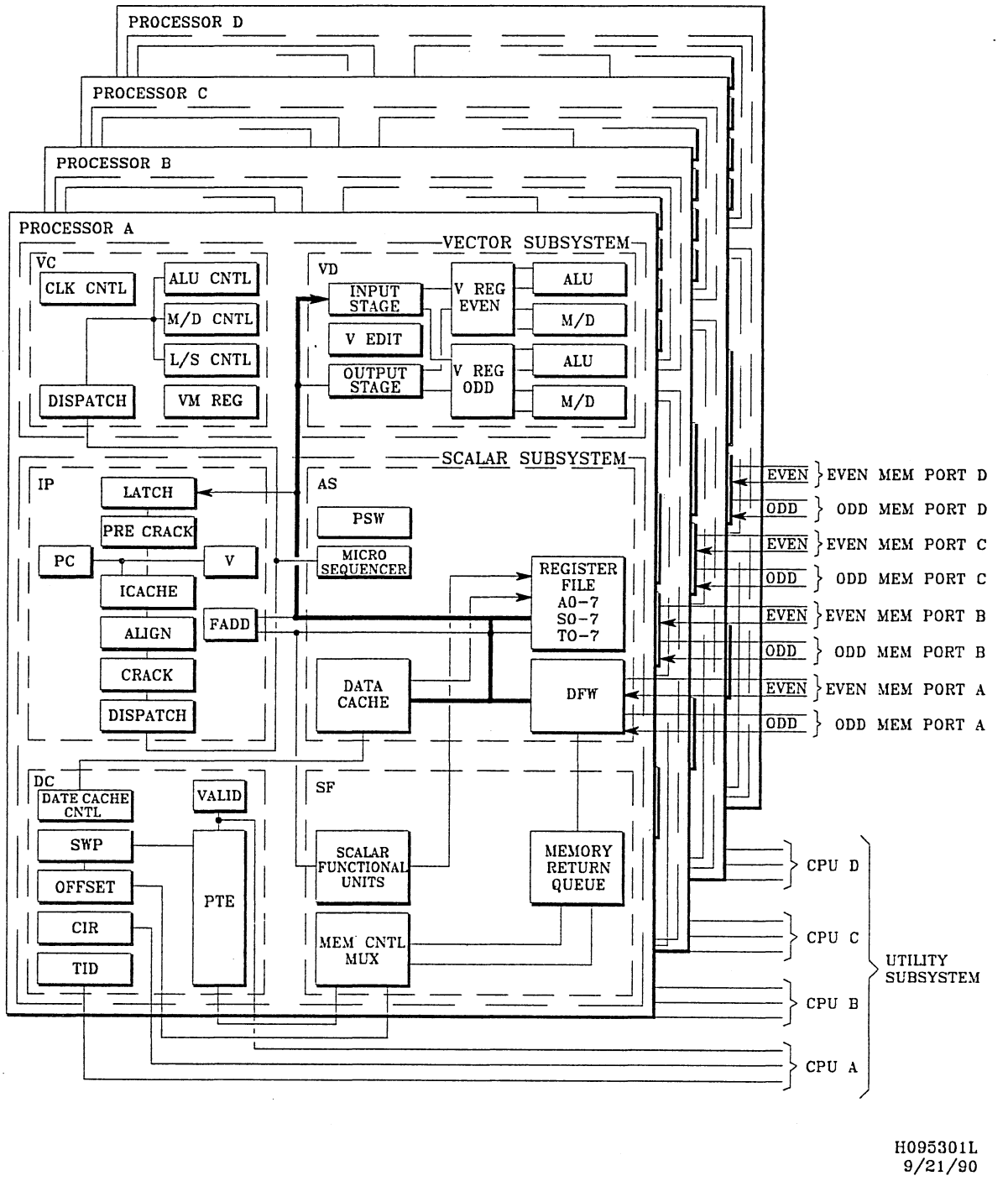


Figure 1–11, C230, C240 System Functional Block Diagram (continued)

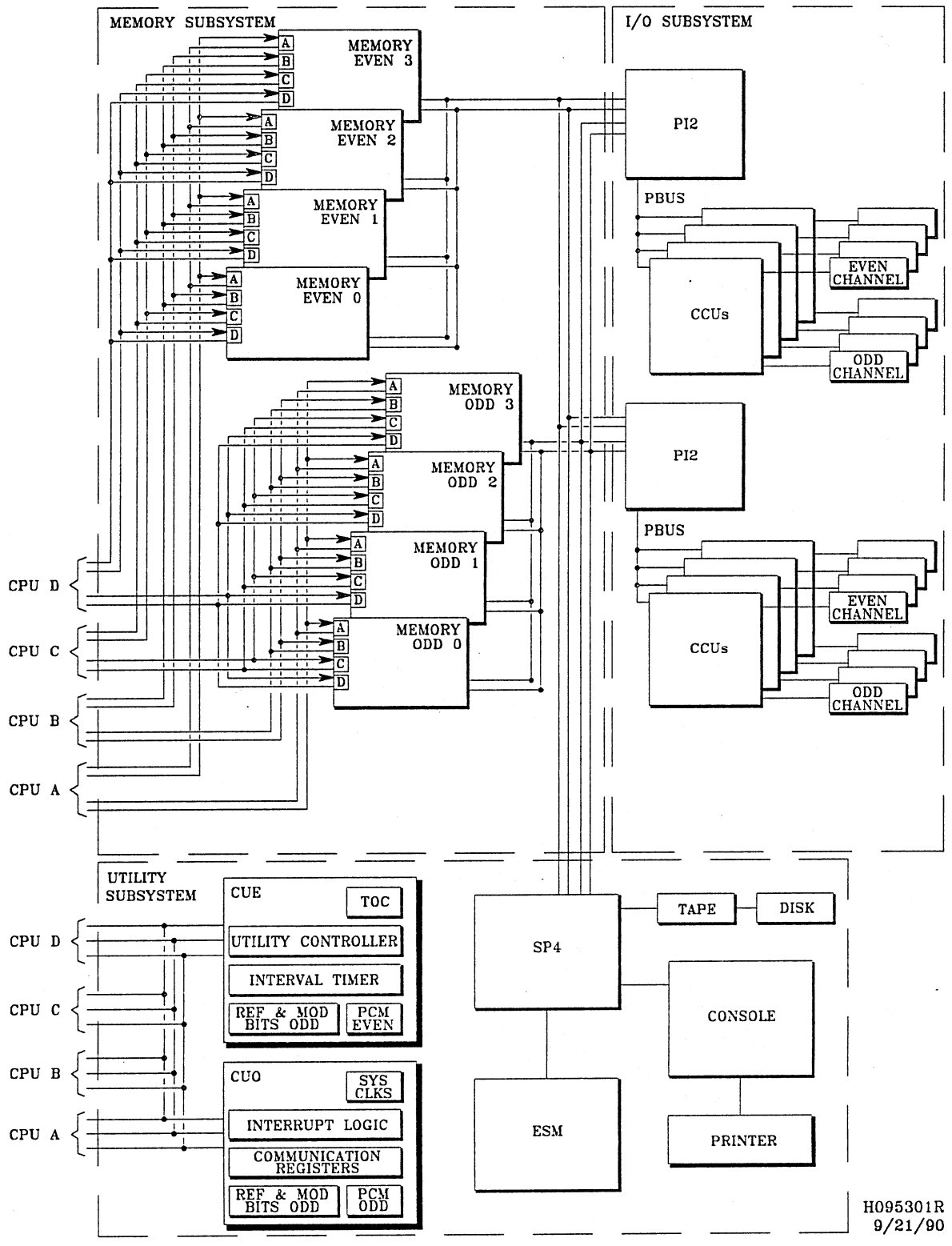


Figure 1–12 (two pages) shows a C232i system functional block diagram:

Figure 1–12, C232i System Functional Block Diagram

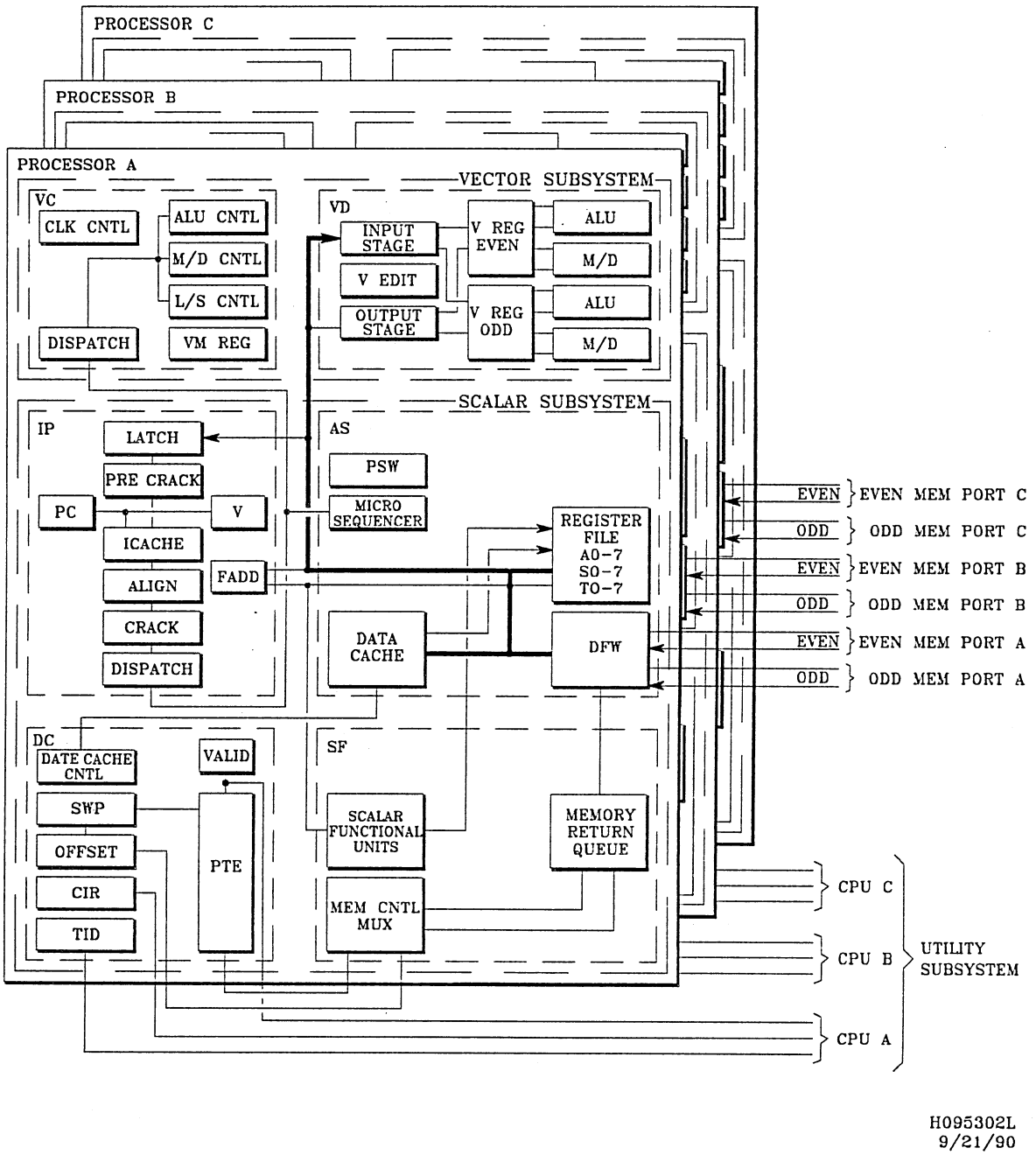
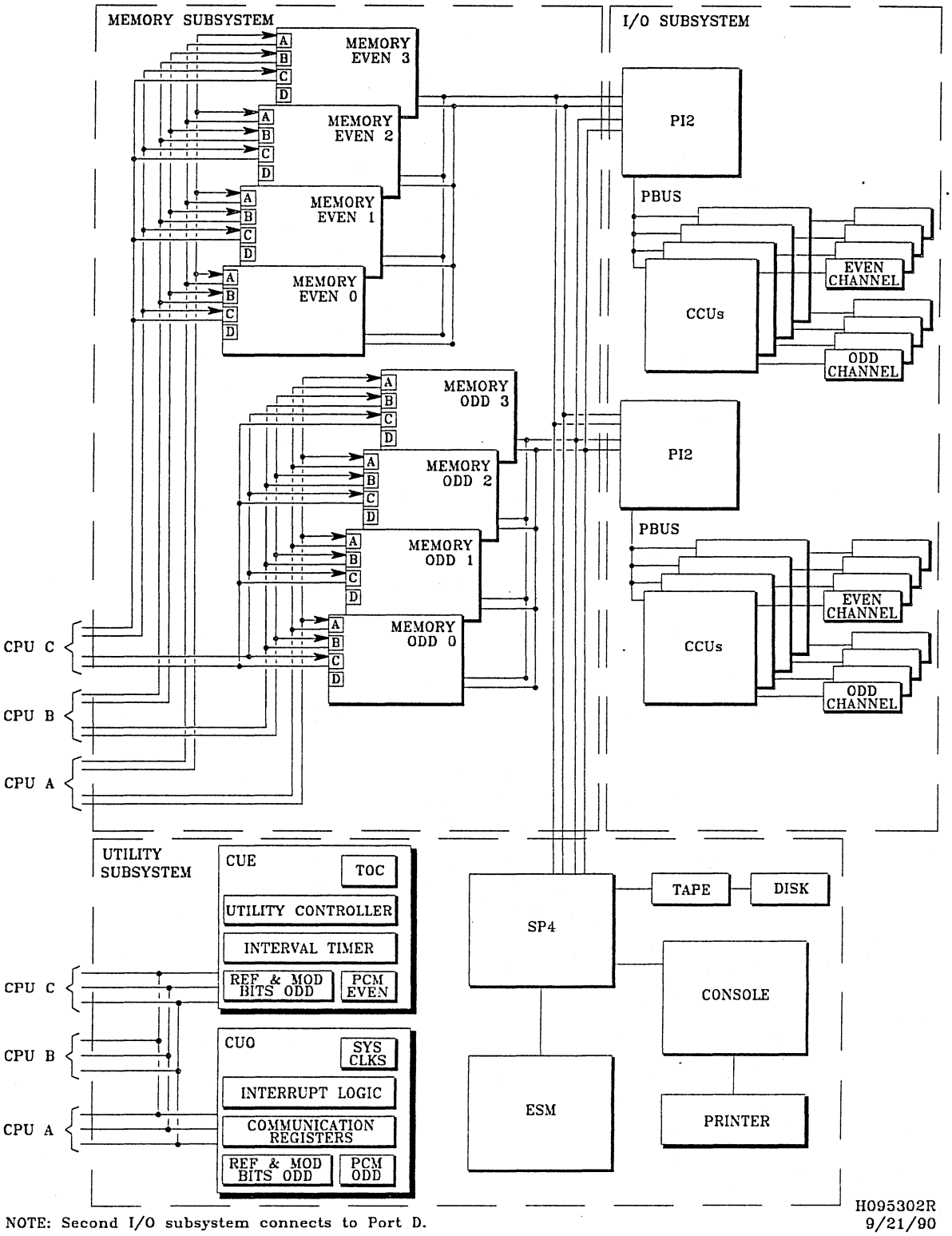


Figure 1–12, C232i System Functional Block Diagram (continued)



NOTE: Second I/O subsystem connects to Port D.

1.3.3.1 Address/Scalar Unit (ASU) Subsystem

The Address/Scalar Unit Subsystem controls machine instruction execution and all related activities. These activities include:

- Executing all scalar instructions and maintaining process control and status.
- Controlling the memory interface (memory read and write operations).
- Controlling all addressing functions for scalar operations and most addressing functions for vector operations.

The Address/Scalar Unit Subsystem interacts with the Vector Subsystem only when a vector operation is initiated by a vector instruction. The Address/Scalar Unit Subsystem performs most memory addressing, data read operations, and data write operations, with respect to vector data. The C200 Series ASU is *functionally* separated into three areas:

- SP—Scalar Processor
- IP—Instruction Processor
- MI—Memory Interface

Table 1–1 lists the operations performed by each functional area in the ASU:

Table 1–1, Address/Scalar Unit Functions

PROCESSOR	FUNCTION
SP	Handle addressing functions for SP Handle some addressing functions for VP Execute scalar operations
IP	Maintain the Program Counter (PC) Dispatch instruction pointed to by PC, to ASP or VPC Execute no-operation, jump, and branch instructions
MI	Handle read data memory requests for IP, SP, VP Handle write data memory requests for SP, VP

Scalar Processor

The Scalar Processor (SP) executes the addressing functions for the SP, some addressing functions for the VP, and all scalar functions. The following four circuits form the heart of the scalar processor data path:

- The register file
- An integer function unit
- Nine Data Flow Gate Arrays (DFW)
- Data cache RAMs

The functions performed by the SP are spread across three of the six system CPU boards. These boards include the following:

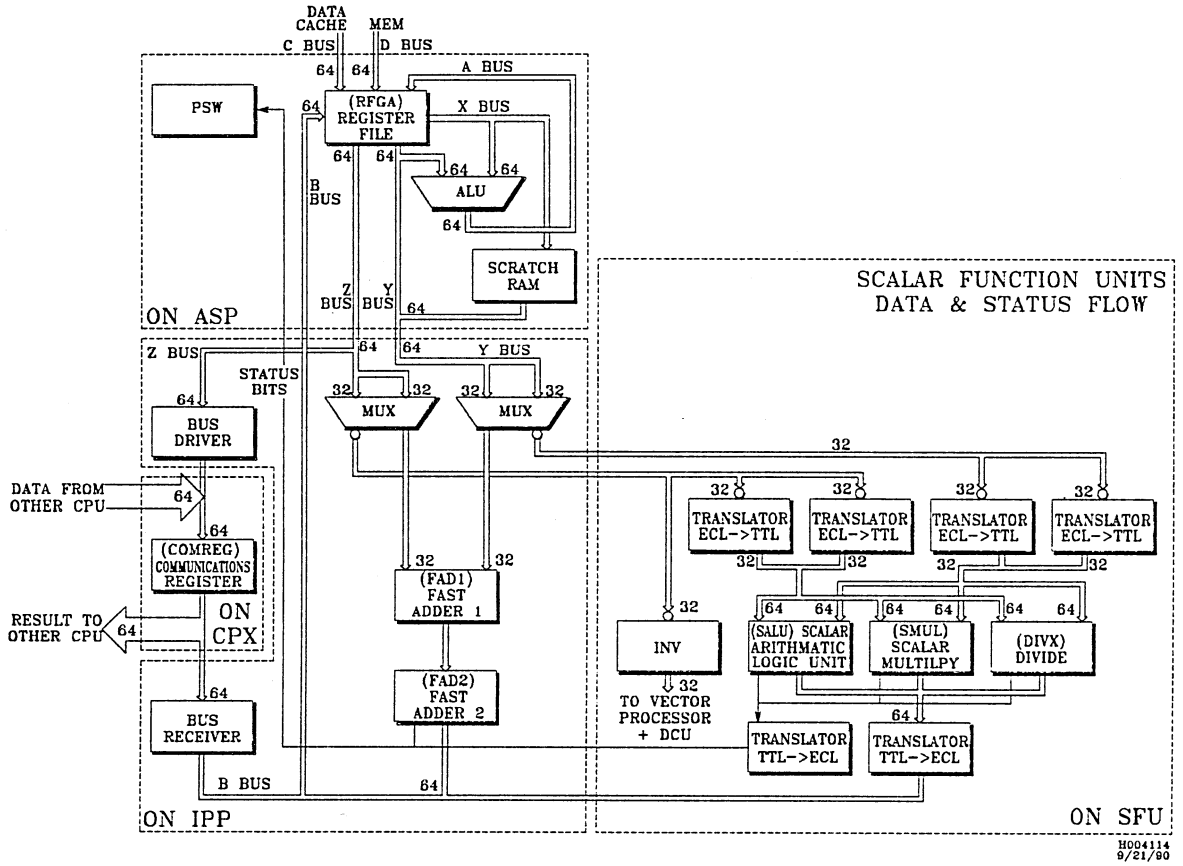
- Address/Scalar Processor (ASP)
- Instruction Pre-Processor (IPP)
- Scalar Function Unit (SFU)

The SP performs the following activities:

- Processes scalar arithmetic operations
- Performs logical functions
- Generates some memory addresses
- Shifts data within the SP

Figure 1–13 shows a functional block diagram of a C200 Series Scalar Processor (SP):

Figure 1–13, CPU Scalar Processor (SP) Functional Block Diagram



Instruction Processor

The Instruction Processor (IP) retrieves instructions from memory. It uses the lookahead logic to request instructions from memory through the Pre-Crack logic, then places them into the Instruction Cache (Icache). Instructions are then read from the Icache.

> **NOTE**
>
> The C200 Series systems **cannot** bypass the Icache. The C100 Series systems **can** bypass the Icache.

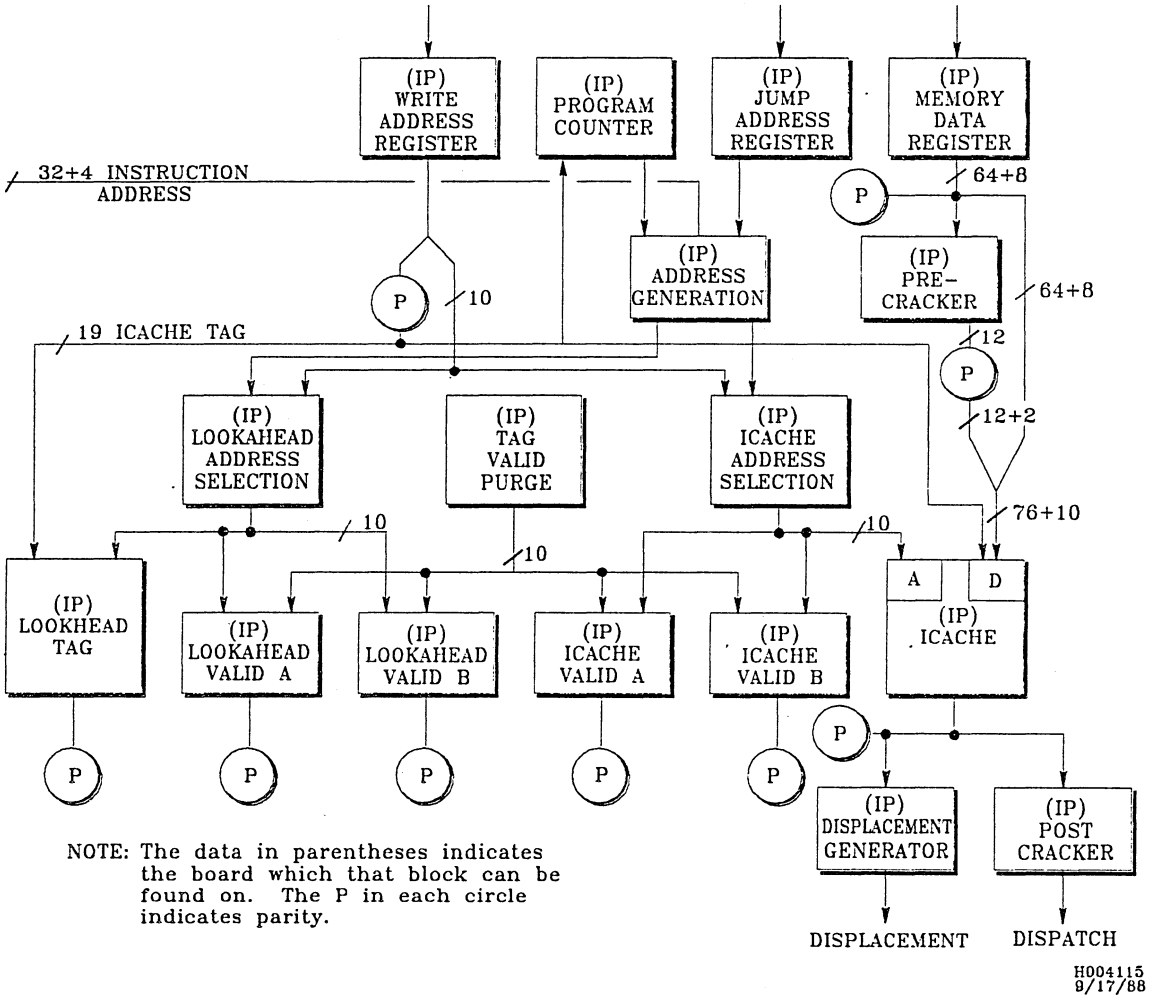
The instruction dispatch logic determines which instruction should execute next, then dispatches it to the instruction sequencer on the ASP board. The IP determines whether the instruction is used by the SP, VP, or both, then sends it to the appropriate processor(s).

Some instructions require scalar processing, others require vector processing, and a few require both. The information transferred contains the instruction and its microcode starting address.

The IP only executes conditional, unconditional branch, or no-op instructions. And it uses circuits on the ASP and IPP boards while executing those instructions.

Figure 1–14 shows a functional block diagram of the Instruction Processor (IP) of a C200 Series system:

Figure 1–14, CPU Instruction Processor (IP) Functional Block Diagram



Memory Interface

The Memory Interface (MI) issues all requests to memory, including read data and write data requests for both the Scalar Processor (SP) and the Vector Processor (VP), and read data requests for the Instruction Processor (IP).

The ASP board makes requests to memory for any type of data, including requests dealing with vector load or store data, and IP longword aligned longwords of data.

The MI consists of the following sections:

- **Memory Control**—Memory Control selects a memory request, identifies its type, and aligns and passes the address and control information to memory. The memory control operations include read data, write data, test and modify bytes, and no-op functions. It uses a cycle type to tell the memory boards which operation is requested. The cycle type includes memory opcode, operand size, and return code. Zone bits are used during a store operation, to list which bytes in memory to write. The zone bits may include one or two 32-bit memory addresses.
- **Logical to Physical Address Translation**—This system generates the physical memory address that is mapped to the virtual address used by the requesting processor. Once a request is the active, or current, request on the system, the virtual address is found in a memory Page Table Entry (PTE). The PTE is used to generate the physical address that is mapped to the virtual address.

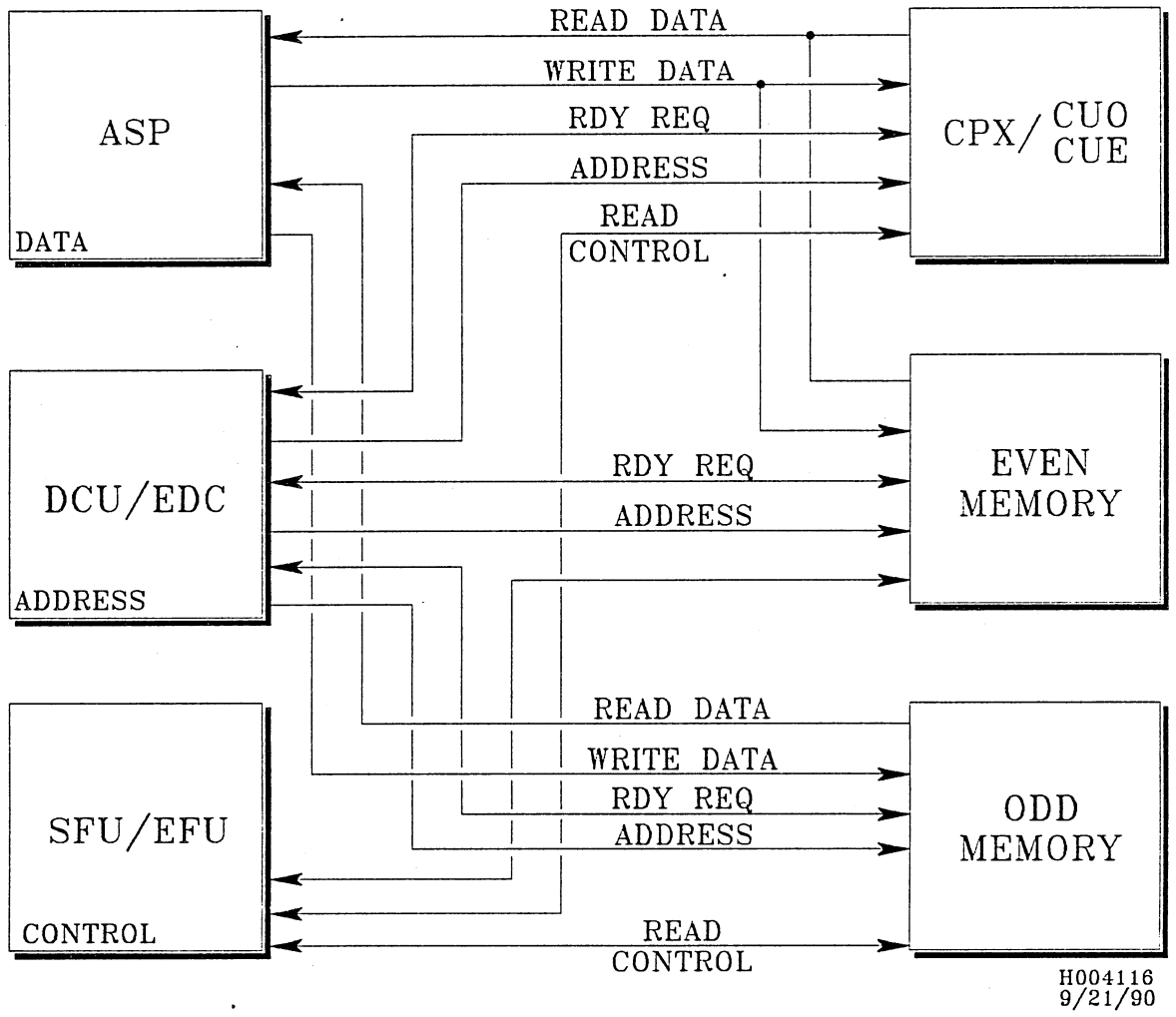
> **NOTE**
>
> The C200 Series systems **cannot** bypass the PTEs. The C100 Series systems **can** bypass PTEs.

- **Vector Address Generator**—The Vector Address Generator (VAG) is a register, on the SFU and DCU boards, which handles requests from the VP for memory.
- **Memory Return Queue**—The return control, from memory control on the DCU, uses the Memory Return Queue to determine which board should receive the information from memory. It then asserts a handshake to that board.

Most of the MI is located on the DCU board, including part of the data cache data path and all of the data cache control. The outgoing memory interface control logic is also on the DCU board. The DCU and SFU boards both generate addresses. The SFU board contains the remainder of the data cache data path, the return control queue and the interface logic, which processes the handshakes to the memory system for the returning read data. The ASP board sends to, and receives from memory, the write and read data requests.

Figure 1–15 shows a functional block diagram of the Memory Interface (MI) of a C200 Series system:

Figure 1–15, CPU Memory Interface (MI) Functional Block Diagram



1.3.3.2 Vector Processor Subsystem

The Vector Processor Subsystem executes all vector operations. The Vector Processor (VP) interfaces with three other processors. The following lists the processors and why they interface with the VP:

- **The Scalar Processor**—The Scalar Processor (SP) interfaces transfer data to and from the Vector Processor. This includes transfers from the memory system or the SP to the VP, from the VP to the memory system or the SP, or to the vector length register.
- **The Instruction Processor**—The Instruction Processor (IP) dispatches instructions to the VP. A set of handshake signals control the transfer of instructions to the VP vector dispatch logic.
- **The Service Processor**—The Service Processor Unit (SPU) handles clock control and error detection.

> NOTE
>
> The term "**Service Processor Unit (SPU)**" is used to generically represent SP2 or SP4, depending which C200 system is being discussed.

The Vector Processor Subsystem consists of the Vector Processor Control board (VPC) and the Vector Processor Data board (VPD). These two boards contain the logic used to process the vector instructions. The VPC contains control lines, the vector merge register for data flow, and some clock generation logic. The VPD board contains additional clock generation logic and most of the data path logic.

Divide operations are performed at a slower speed than other operations. In order to reduce speed for divide operations without impairing performance for other operations, the VP uses odd and even data paths and a clock (80 ns) that runs at half the speed of the system clock (40 ns).

The SP begins all VP memory requests with a scalar memory request. The Vector Address Generator (VAG) then makes any additional VP memory requests.

The SP loads all vector processes onto two boards, the VPC and the VPD. A 64 bit bus from the SP parallel loads the Vector Merge Register (VM) and the VPC state. A second 64 bit bus to the VPD board stores the value of the VM register, the VPC state, and all vector elements.

The VPD board contains the data path logic and some clock generation logic for the processor. The data flows through the following circuits:

- Vector Register Files
- Memory and Scalar Processor Interface
- Function Units
- Vector Merge Register

Though functionally part of the data path logic, the Vector Merge Register is physically located on the VPC control board.

The VPC board contains the control lines and some clock generation logic for the processor. It maintains control of the following:

- The Instruction Dispatch Logic
- Load and Store Function Pipelines, including micro control and write control
- The Vector Length Register
- Output Staging
- VM Staging
- ALU Function Pipelines, including unit control, micro control and write control
- Multiplier Function Pipelines, including unit control, micro control and write control
- The Divide Function Unit Pipeline

Memory and Scalar Processor Interface

The VP passes data to and from both the memory and the SP over two unidirectional buses. The SP is the source for the bus that transfers data to the VP from the Data Flow Gate Arrays on the ASP board. The VP is the source of the data return bus to those same arrays.

> **NOTE**
>
> Any data request by the Memory Interface is also gated by the Data Flow Gate Arrays to the Memory Data Register. It is placed in the Memory Data Register, on the Instruction Pre-Processor (IPP) board, whether or not it is intended for the Instruction Processor (IP) and is parity checked. Therefore, an IP parity error may be displayed for any incorrect data from memory.

Instruction Dispatch Control

The Instruction Pre-Processor (IPP) board loads the Instruction Dispatch Registers when a vector instruction is parsed. The VP dispatch logic uses the opcode from the IPP to index an entry table to obtain specific information about the opcode. The fields of the table include the following:

- Function Pipe Reservation Logic—The function pipe reservation logic checks if the instruction can chain to the current executing instruction. It also tells when the function pipe is available.
- Chaining Hazards—This field determines if the instruction uses results from the current executing instruction.
- Port Reservation Logic—The port reservation logic checks if vector register ports are available for the operands that the instruction requires. It also indicates which port the vector register allocated to the instruction.

For instructions that must obtain a scalar value to begin, the dispatch control requests the data from the SP.

The dispatch information loads into the appropriate function pipe dispatch registers. The function pipe controller then begins at the entry point address in the dispatch table and controls the operation.

Function Units

The VP uses three pipelined function units to perform all vector operations and operate on vector data. Each function pipe has its own micro controller to control the sequence of actions to perform an instruction. These function units are:

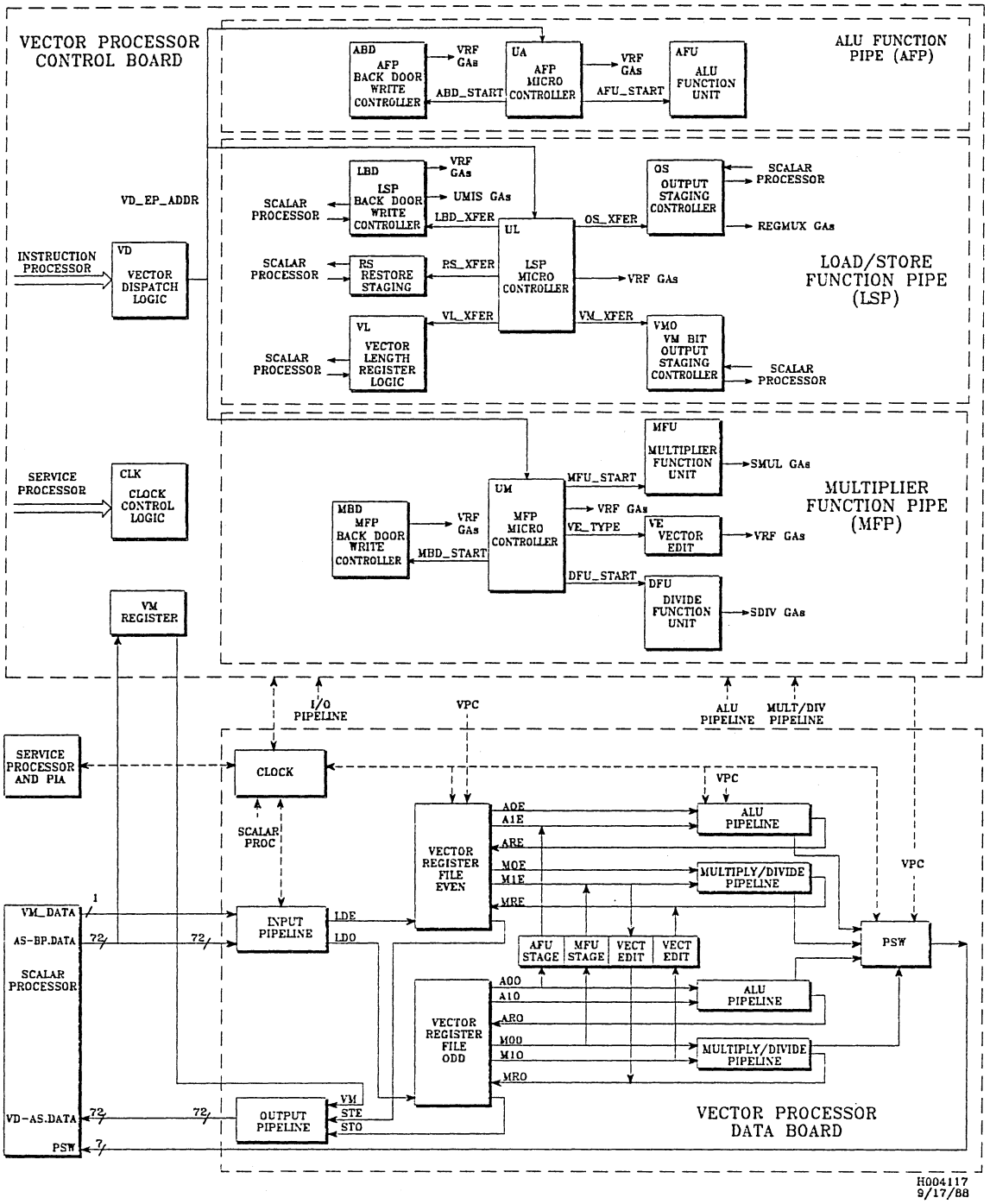
- ALU Function Unit (AFU)—Add, subtract, compare, convert, logical, shift, and population count operations
- Multiply Function Unit (MFU)—Multiply operations
- Divide Function Unit (DFU)—Divide and square root operations

State Save and Restore Data

The SP informs the VP that a fault has occurred. The VP ignores this fault condition until the SP memory queue is empty. When both conditions exist, the processor recognizes the fault on the next clock cycle.

Figure 1–16 shows a functional block diagram of the Vector Processor Subsystem of a C200 Series system:

Figure 1–16, CPU Vector Processor (VP) Subsystem Functional Block Diagram



1.3.3.3 Memory Subsystem

The memory subsystem is a five port design (ports A, B, C, D, and E) that allows simultaneous access to physical memory (up to 2 Gbytes) by up to four CPUs and the I/O Subsystem or the Service Processor Unit. Ports A through D are reserved for CPUs (CPU A to Port A, CPU B to Port B, etc.). Port E is reserved for the I/O subsystem and the Service Processor Unit.

> **NOTE**
>
> The C232i system has an extra I/O Subsystem, in place of CPU D. It accesses the Memory Subsystem through two additional PIA2 units (PIV and PIW) that are connected to Port D. Since it is connected to memory through what is normally a CPU port, the second I/O Subsystem has the same priority as CPU D would have.

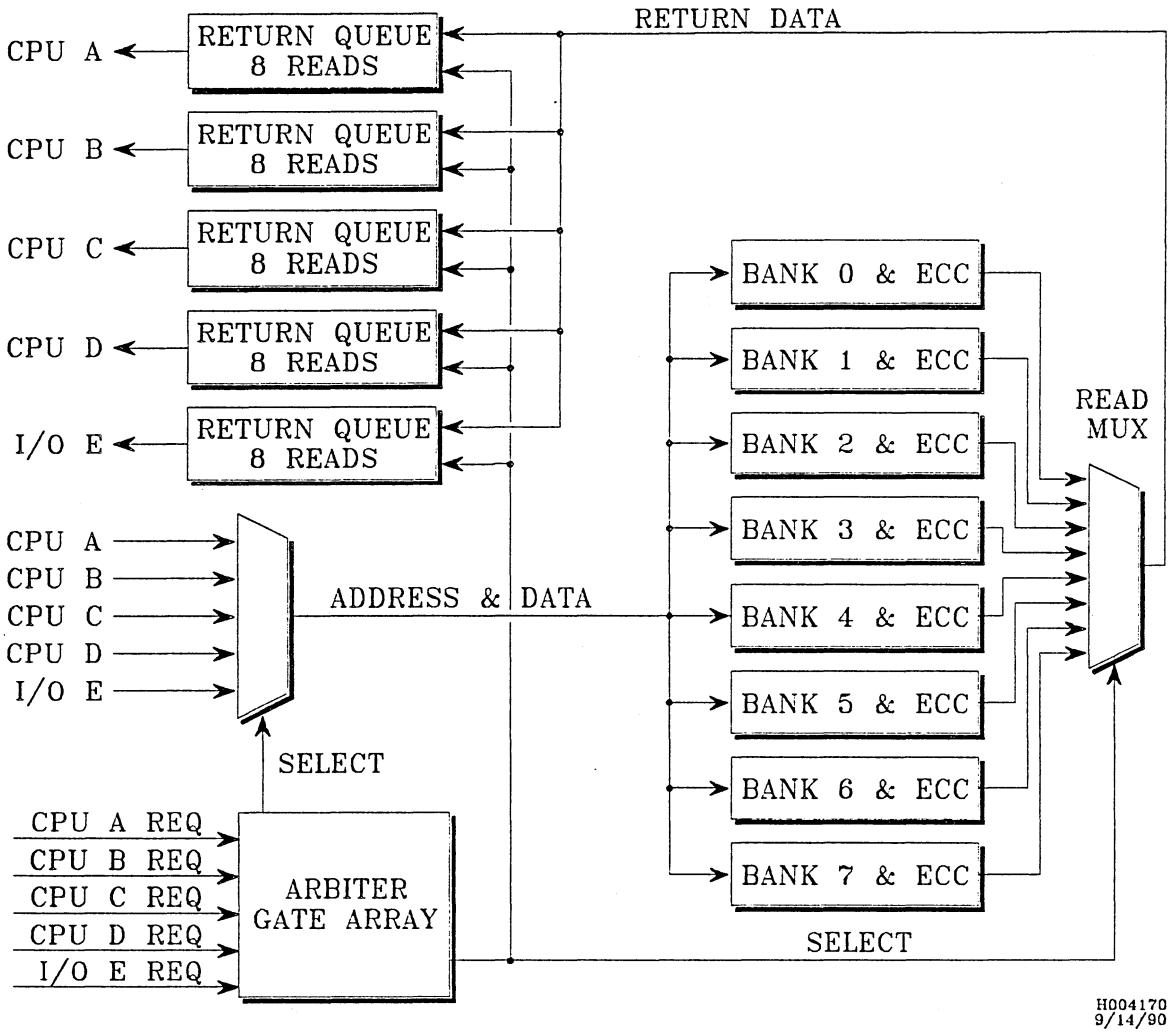
Memory subsystems can contain from one to four pairs of Memory Control Modules (MCMs). The MCM boards are paired to provide even and odd memory. Each MCM includes a five port crossbar, arbitration logic, and four Memory Array Modules (MAMs). Each MAM contains two banks of memory for a total of eight banks per MCM.

Each MCM has its own arbitration controller to determine which memory request to honor. I/O subsystem and SPU requests (Port E) have the highest priority, followed by the CPU ports (A, B, C, and D). The arbitration between CPUs is based on previous accesses. The more recently a CPU is granted access to memory, the lower its priority will be. The use of a separate arbitration controller on each MCM, allows parallel memory operations, which improves memory bandwidth.

The memory subsystem uses 32-bit words (4-bytes). Word-aligned operands use a simple write cycle instead of a longer read-modify-write cycle. Word operands can be retrieved from either independent half of memory, which reduces memory contention.

Figure 1–17 shows a functional block diagram of the Memory Subsystem of a C200 Series system:

Figure 1–17, Memory Subsystem Functional Block Diagram



1.3.3.4 I/O Subsystem

The C200 Series I/O Subsystem connects to the rest of the system through Port E of the Memory Subsystem via a PBUS Interface Adapter (PIA). One PIA is used in single CPU or two CPU systems. For larger systems (three or four CPUs), two PI2s (enhanced version of the PIA) are configured. The C232i system has a second I/O Subsystem, and requires two additional (four total) PI2s.

The C200 Series I/O Subsystem uses the same I/O processors as the CONVEX C100 Series I/O subsystems. The I/O processors connect to the PIA/PI2 through the PBUS. Each I/O processor is a potential PBUS master.

Port E of the Memory Subsystem is dedicated to the I/O Subsystem. The PIA/PI2 and the Service Processor Unit (SP2/SP4) both use this port. The service processor for the C200 Series systems does not reside on the PBUS. It connects directly to memory through port E.

PBUS Interface Adapter (PIA/PI2)

The system clock generator is contained in the PIA or the CUO, depending on which system you have. The oscillators, clock switching (for margin testing) and drivers provide master clock and phase signals for all of the boards in the system.

System interrupt arbitration is contained in the PIA/CUO. Devices are polled in a round robin fashion until an interrupt request is found. Upon completion of the interrupt sequence, the next device is polled. The PIA/PI2 also provides TTL to/from ECL translation for signals going between the PBUS and EBUS.

The PIA/PI2 and the Service Processor Unit (SP2/SP4) share the address and data lines for port E of the memory bus (the EBUS). The PIA/PI2 provides EBUS arbitration and return data ordering for these two potential EBUS masters. EBUS arbitration determines when the selected EBUS master is allowed to drive the memory system address, control, and data signals. Return data ordering ensures that memory read data are returned to the requesting EBUS master in the same order that their requests were issued.

When multiple PBUS device requests are present, requesting devices are serviced in a round robin fashion. Once a transfer is initiated, the PIA/PI2 allows it to complete before moving on to the next PBUS device. Between each PBUS request, the SP2/SP4 is guaranteed at least a one cycle window to master the EBUS. If a request occurs during this window, a single transfer will be allowed to complete before the next PBUS device is allowed control of the bus.

PBUS Description

The I/O Subsystem consists of from one to four Channel Control Units (CCU) connected to the PIA/PI2 on a single PBUS interface. The PIA/PI2 is designed to emulate a memory interface to the CCUs that is upward compatible with the memory interface to CCUs in the CONVEX C100 Series systems.

The PBUS interface is a synchronous, 10-MHz multi-master, block transfer bus protocol, used as an I/O path to and from main memory. Each CCU is capable of being the bus master during any given bus cycle. All transfers on the PBUS are to or from the PIA/PI2. CCU to CCU transfers are not supported. Bus arbitration is performed in the PIA/PI2.

The C232i configuration of the CONVEX C200 Series Systems has four PI2 boards. The boards and their relationship to C232i I/O architecture are shown in Table 1–2.

Table 1–2, C232i I/O Architecture

PI2 UNIT	MEMORY PORT	CPU UTILITY	CH CONTROL UNIT
PIV	D (odd memory)	CUO	CCU 8, 9, 10
PIW	D (even memory)	CUO	CCU 12, 13, 14
PIX	E (odd memory)	CUE	CCU 4, 5, 6, 7
PIY	E (even memory)	CUE	CCU 0, 1, 2, 3

EBUS Description

The Memory Subsystem is organized as a 5 port memory array. Each port is capable of transferring 64 bits (longword) during each 40 ns bus cycle. Each longword transfer references an even bank for 32 bits and an odd bank for the other 32 bits.

In addition to the memory space, there exists an I/O space for miscellaneous timer registers and CPU referenced and modified bits. I/O space consists of even banks only, but otherwise behaves identically with memory space. I/O space resides on the CPU utilities board (CPX).

The PIA/PI2 connects to the Memory Subsystem through Port E. This interface provides full memory capabilities for all I/O processors as well as the SP2/SP4. The data path portion of the memory interface is shared between the SP2/SP4 and the PIA/PI2. All memory control signals are sourced by the PIA/PI2.

Channel Control Unit (CCU)

The C200 I/O Subsystem is based around a 64-bit 80-Mbyte-per-second bus known as the PBUS. The PBUS connects main memory, the Service Processor Unit (SP2/SP4), and the CCUs on the backplane. It uses a block transfer protocol to provide high data throughput with a minimum of control overhead. Each C200 Series system PBUS can support up to four CCUs.

The SP2/SP4 has a diagnostic interface to the CCUs and the CPU called scan rings. Many of the CPU and CCU registers are built out of shift registers that the SP2/SP4 initializes at boot time and manipulates for diagnostic purposes.

Most of the device driver code on the CCU is executed by the 68000. A small amount of the high level-driver code executes on the CPU. This portion of the driver translates an I/O request from the ConvexOS operating system into a queue entry in main memory for the CCU.

The CCU reads the queue entry from main memory, executes the requested data transfer, and places transfer status back in the queue for the CPU. A test-and-set mechanism locks queues during updates. If an entry is placed in an empty queue, the processor making the queue entry sends an interrupt to the other processor to initiate queue service.

Following is a list of software/hardware relationships:

- **CPU**—The Central Processing Unit (CPU) controls job processing on the C200 Series system. This includes user processes, the shell, the ConvexOS kernel, the ConvexOS device driver, and a portion of the Message Based System (MBS).
- **Main Memory**—The main memory domain contains the MBS data structures used for communication between the Channel Control Units (CCUs) and the CPU.
- **CCU**—The rest of the MBS, the CCU device driver, and the Event Governed Operating System (EGOS) run on the CCU hardware.
- **Device**—Devices and device controllers interface to CONVEX hardware and software at this level.

I/O Processors

CONVEX I/O Processors (IOPs) provide the connection between the PBUS and Multibus (MIOP)-based or VMEbus (VIOP)-based peripheral controllers.

A Multibus I/O Processor Subsystem consists of a single CONVEX daughter card (an MIOP), connecting cables, and a Multibus Control Unit (MBCU) card within each Multibus cage. An MIOP supports up to two Multibus card cages with up to eight controller slots each, and an aggregate data bandwidth of 8 Mbytes per second.

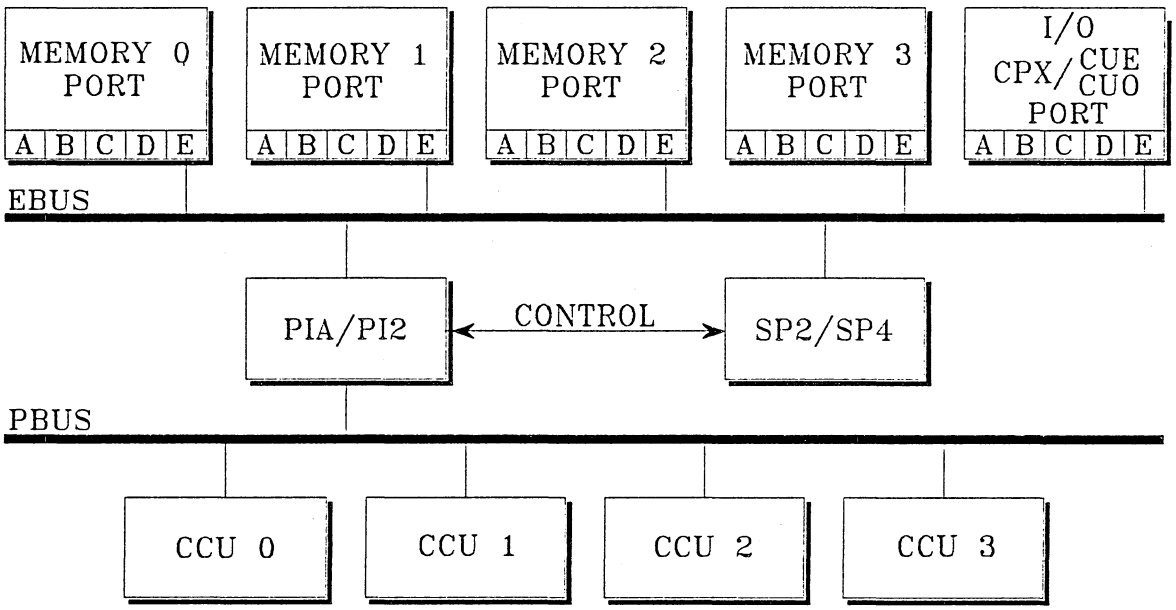
A MIOP can be inserted into any I/O backplane slot (CCU0 through CCU3). Each MIOP has two Multibus Adapter buses (0 and 1) to which the Multibus controllers are attached.

Each controller then connects to a local controller bus, which in turn, connects to the peripheral devices. A CONVEX MIOP supports devices such as disks, tape drives, printers, and terminals using Multibus (IEEE P796) controllers.

A VMEbus I/O Processor Subsystem consists of a single CONVEX daughter card (an VIOP), connecting cables, and a VMEbus Control Unit (VBCU) card within each VMEbus cage. A VIOP can be inserted into any I/O backplane slot (CCU0 through CCU3). Each VIOP has two VMEbus Adapter buses (0 and 1) to which the VMEbus controllers are attached.

Figure 1–18 shows a functional block diagram of the I/O Subsystem of a C200 Series system:

Figure 1–18, I/O Subsystem Functional Block Diagram



NOTE: C232i Second I/O Subsystem connects to Memory Port D.

H004119
9/20/90

Figure 1–19 shows a functional block diagram of a C200 Series CCU Subsystem:

Figure 1–19, CCU Subsystem Functional Diagram

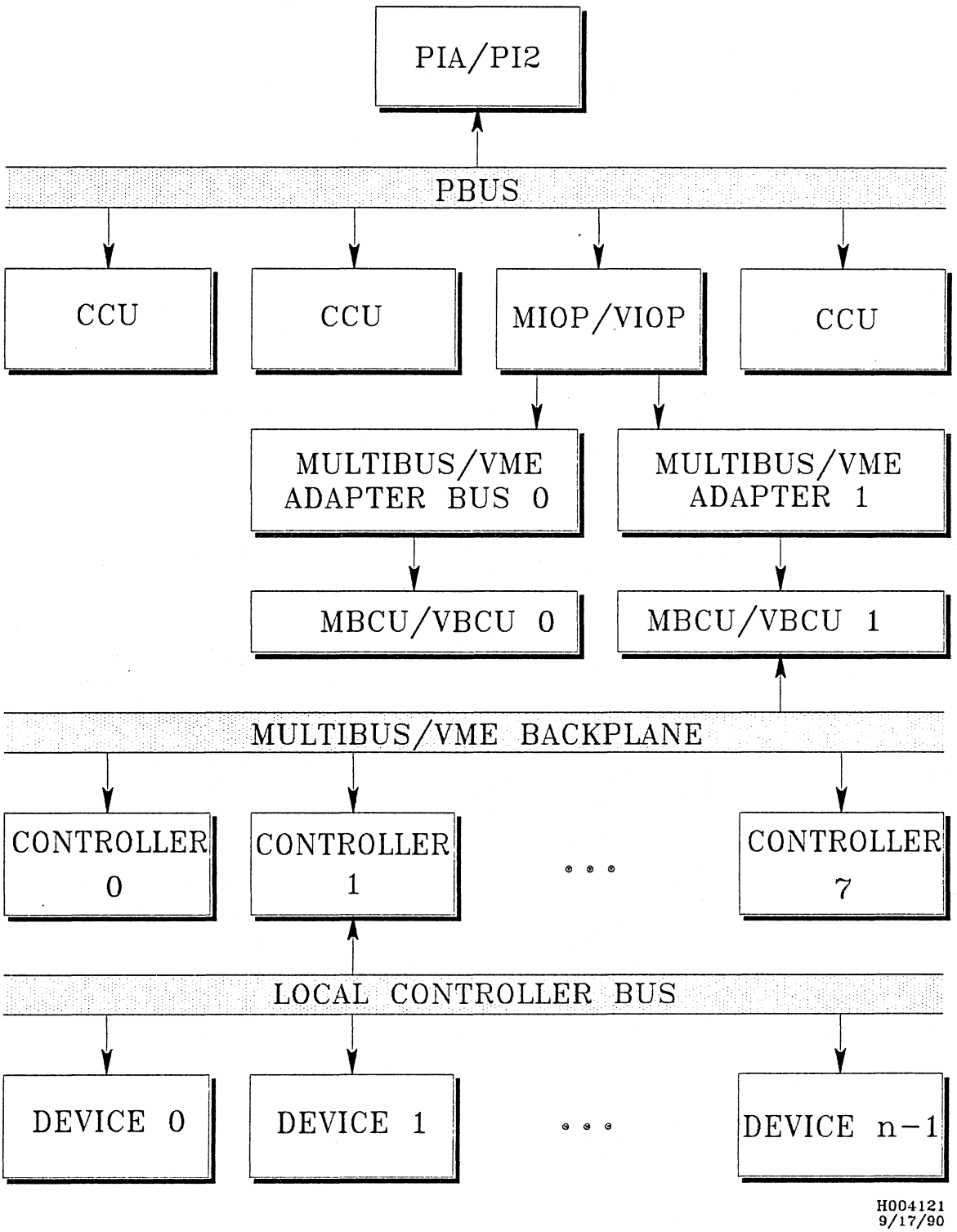
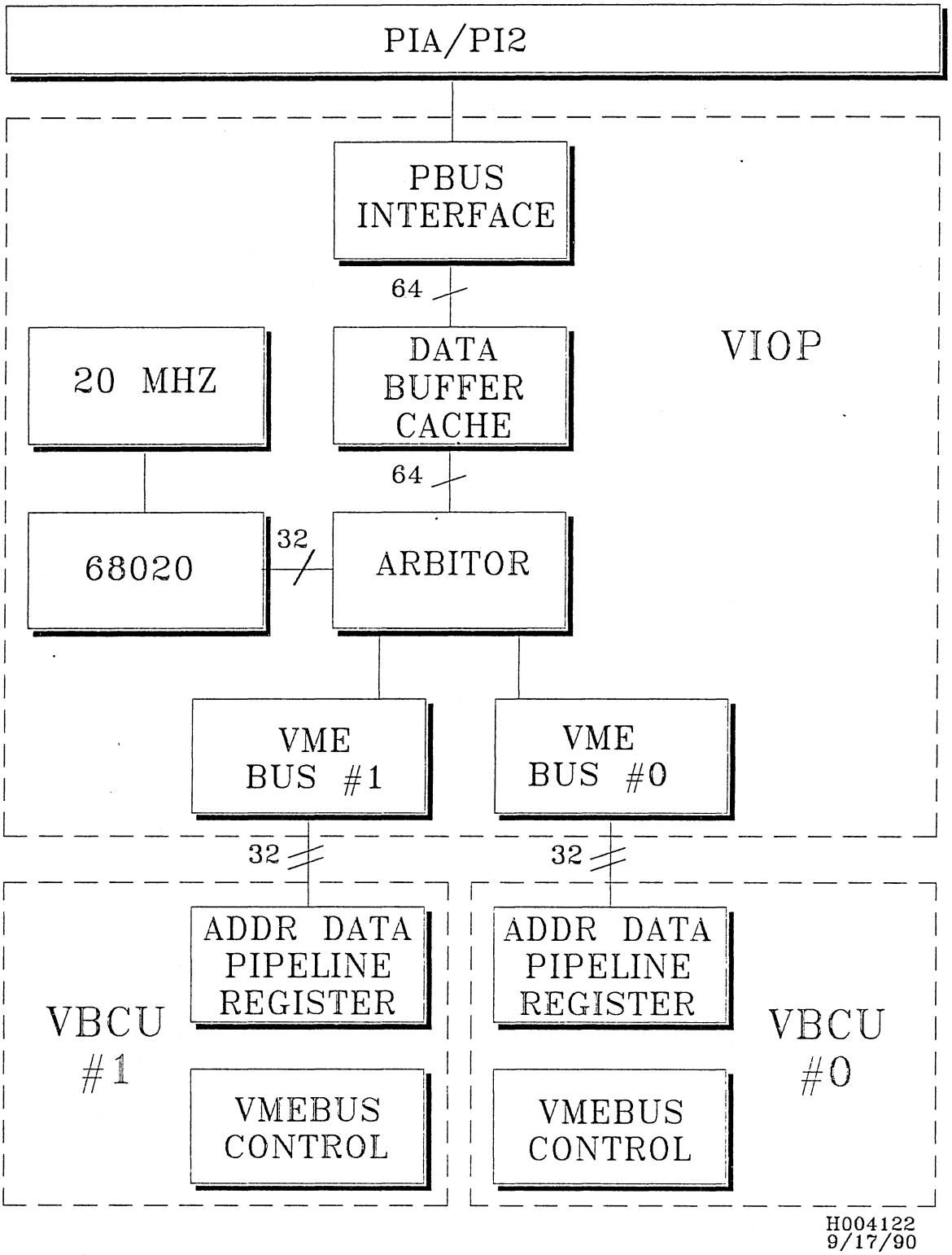


Figure 1–20 shows a functional block diagram of a C200 Series VMEbus I/O Processor:

Figure 1–20, I/O Processor Subsystem Functional Diagram



1.3.3.5 Utility Subsystem

The C200 Series Utility Subsystem performs most miscellaneous and monitoring functions. This subsystem is comprised of the following boards:

- CPX—CPU Utility Unit (C201, C202, C210, and C220 models)
- CUE and CUO—CPU Utility Unit (C230, C240, and C232i models)
- SCM—System Control Monitor (C201, C202, C210, and C220 models)
- ESM—System Control Monitor (C230, C240, and C232i models)
- SP2/SP4—Service Processor Unit

CPU Utility Unit (CPX or CUE and CUO)

The CPU Utility Unit contains most of the utility functions used by the other subsystems. These utility functions are separated into two groups:

- Functions that occupy I/O space and are accessible by the CPU(s) and the PIA/PI2.
- Functions that are accessible only by the CPU(s).

Several of the I/O space functions are similar to those on the C100 Series systems. They are:

- The Referenced and Modified (R&M) bits are used by the operating system to optimize the use of accelerated blocks of memory and monitor activity on up to four memory buses.
- The Physical Configuration Map (PCM) indicates physical blocks of memory present in the system and monitors all memory accesses to verify address validity on up to four memory buses.
- The Interval Timer is a 16-bit programmable timer that generates an interrupt to a programmable vector location.
- The Time of Century Counter is a 64-bit, 1-MHz counter used for such functions as time stamps and process timing.

All I/O space functions are accessed via the even memory bus, and all data is passed on the upper 16 bits of the 32-bit even memory data bus along with parity for the entire data bus.

The functions on the CPU Utility Unit that are accessible only by the CPUs are the Communication Registers and the CPU Interrupt Arbiter.

- The communication registers are 1K by 72 bits, including 8 parity bits and an associated lock bit.
- The interrupt arbiter monitors the system interrupt bus and processor status to determine when a CPU interrupt is generated and which processor should process the interrupt.

System Control Monitor (SCM or ESM)

CONVEX C200 Series models have a built-in safety mechanism called the System Control Module (SCM for C201, C202, C210, and C220 models, or ESM for C230, C240, and C232i models). The SCM circuit board is mounted behind the air plenum in the C201, C202, C210, C220 processor cabinets. The ESM board plugs into card slot 23 of the second card cage for C230 and C240 processor cabinets, and card slot 26 of the second card cage for C232i processor cabinets. The system control monitor logic is enabled when the main circuit breaker is in the **ON** position.

The SCM/ESM monitors various elements of system hardware and operating environment and will stop system operation if improper conditions occur. When an error condition is detected by the SCM/ESM during the power-up check, power-up is halted and an error code(s) is displayed on the hexadecimal system status display.

If an error is detected while the system is running, the SCM/ESM may shut down the system and display an error code. SCM/ESM errors are classified into two types:

- Warning (display error code)
- Fatal (drop power)

The following conditions cause the SCM/ESM to inhibit power up or down of CONVEX systems:

- Improperly installed Printed Circuit Boards (PCBs)—If the SCM determines that a PCB has been installed incorrectly, the SCM/ESM prevents the machine from powering up, regardless of the keyswitch position.
- Power supply margins (normal in some diagnostic modes)—If the SCM/ESM detects voltage levels outside specific tolerances, the SCM/ESM prevents machine power up.
- An excessive ambient temperature—If the SCM/ESM detects temperatures above specified levels, it will drop machine power and display an error code on the **SYSTEM STATUS** display. Temperature sensors are located throughout the processor cabinet.
- Cooling fan failure—If the SCM/ESM detects a **single** cooling fan malfunction, an error code appears on the **SYSTEM STATUS** display. If a multiple cooling fan malfunction is detected, the SCM/ESM will drop machine power.
- Power supply malfunction—The SCM/ESM continually monitors power supplies, and will drop machine power if a fatal error condition is detected.

Service Processor Unit 2/4 (SP2/SP4)

The Service Processor Unit (SPU—the SP2 variant is used in one and two CPU C200 Series systems and the SP4 variant is used in three and four CPU systems). The SP2/SP4 has several primary functions. They are described in the following list:

1. Operates before the processor(s) is running.
2. Initializes and boots the processor(s).
3. Logs errors that might be generated by the system during its operation.
4. Continues to run after the processor(s) is halted due to a function error.
5. Conducts diagnostics on the C200 Series system to locate faults.

SP2/SP4 Hardware

The SP2/SP4 is an independent, single-board, minicomputer system based on the Motorola 68000 microprocessor. It plugs into the SP2/SP4 slot in the C200 Series system card cage. The SP2/SP4 CPU board contains on-board local memory and peripheral interfaces.

The SP2/SP4 system console, remote console, and boot devices are completely separate from the C200 Series processor system resources. They are connected via special connectors on the rear of the C200 Series system backplane.

SP2/SP4 Indicators

There are five LED indicators on the front edge (foreplane) of the SP2/SP4 board. They are visible from the front of the C200 Series processor card cage. A green LED is on top, followed by four red LEDs. During normal operation these LEDs are defined as follows:

1. The green LED indicates the status of the SP2/SP4 halt line. This indicator is illuminated as long as the SP2/SP4 68000 processor is running.
2. The first (top) red LED indicates the SP2/SP4 clock line heartbeat—a 1-Hz flash.
3. The second red LED is illuminated when the SP2/SP4 processor is busy.
4. The third red LED is user (software) definable.
5. The fourth (bottom) red LED is user (software) definable.

> **NOTE**
>
> The four red LEDs are software definable. In accordance with the current convention, they defined as shown in items 2 through 5 of the above list.

The EPROM-based power-up self-test contains fifteen subtests (1 through E). During power-up, the number of the subtest in progress is displayed on the console. The subtest number is also displayed on the four red LEDs (even if the console is inoperative). If a subtest should fail, it will loop with its number displayed on the console and the four red LEDs.

> **NOTE**
>
> The four red LEDs are used as a 4-bit display register during the power-up self-test. The bottom LED is the Least Significant Bit (LSB) and the top LED is the Most Significant Bit (MSB).

SP2/SP4 EBUS Interface

The EBUS interface allows the SP2/SP4 to communicate with the rest of the machine through main memory. All shared memory resources of the C200 Series processor are accessible to the SP2/SP4 through the EBUS and its interface on the SP2/SP4.

Control of the EBUS is the responsibility of the EBUS arbiter, located on the PIA/PI2 (which is also attached to the EBUS). This arbiter controls the SP2/SP4 EBUS interface and determines which CCU or SP2/SP4 will get control of the EBUS. The SP2/SP4 has priority over the CCUs in bus accesses.

System Clock Control

The SP2/SP4 controls all clock generation. Although all clock signals (in the form of phase bits) go to the boards where they will are needed, the SP2/SP4 has gating control over them as they go onto the boards. It can turn on or off the run bits to the clock gates at (essentially) the board connector to let the clocks onto the board or not.

Each board in a C200 Series system (except the SP2/SP4 and System Control Module (SCM)) has a separate RUN line to the SP2/SP4 for clock control (gating). This allows the clocks to be turned on or off individually for each board. The RUN line to a particular board controls all clocks that are used on that board.

Main Memory Refresh Control

The processor memory in a C200 Series system does not refresh automatically. The SP2/SP4 has a hardware counter that determines when the system memory should be refreshed and sends the appropriate signals. If the memory system is busy when the refresh request is received, it can determine whether it must force the refresh immediately or if it can wait until the memory cycle is complete.

Diagnostic Interface

The diagnostic interface is connected to scan rings, clock control, and error lines which allows for access to programs that run on the SP2/SP4 processor (diagnostics). The SP2/SP4 includes a micro machine to operate the scan interface.

Special single-bit serial buses (Scan Data lines (SCNDAT)) connect the SP2 (seven lines)/SP4 (nine lines) diagnostic interface directly to the scan rings in these functions:

- Processor A (SP2/SP4)
- Processor B (SP2/SP4)
- Processor C (SP4 only)
- Processor D (SP4 only)
- I/O (SP2/SP4)
- CCUs (SP2/SP4)
- Memory even (SP2/SP4)
- Memory odd (SP2/SP4)
- Miscellaneous log (CPX, PIA/PI2, CUE,CUO) (SP2/SP4)

These lines allow data (initialization and diagnostic) to be loaded into the C200 Series system boards, and data (machine state and diagnostic result) to be read from the boards. Other lines set boards to the diagnostic mode (DMODE), specify the diagnostic operation to be performed (SCTL), issue clocks to scan rings (RUN), and set the board to the diagnostic read or write mode (ODENA).

SP2/SP4 Peripherals

The SP2/SP4 supports a local console and a remote console through asynchronous ports. It also supports boot devices through a standard Small Computer System Interface (SCSI) port. The SCSI port is normally interrupt driven, but can be software selected for DMA operation.

System Console

The system console communicates with the SP2/SP4. It also communicates with the C200 Series processor (through the SP2/SP4). The the local console or the remote console can be designated as the **system console**, according to the position of the front panel keyswitch.

Under normal conditions, the system console is connected to ConvexOS running on the C200 Series processor. However, the position of the keyswitch can be changed to switch the console back and forth between ConvexOS and CONVEX SPU OS.

> **CAUTION**
>
> Changing the system console from ConvexOS to CONVEX SPU OS and then running certain SPU job processes (such as C200 Series processor diagnostics) while the C200 Series processor is executing a program may cause the system to go down.

The **local system console** is connected to the SP2/SP4 via an RS-232 port which normally operates at 9600 bits per second (bps). This port is enabled only when the front panel keyswitch is set to the **1 LOCAL MAINTENANCE** position.

The **remote system console** port is enabled when the front panel keyswitch is set to the **1 REMOTE MAINTENANCE** position. The remote console is the same as the local console except there is a modem and a remote connection between the terminal and the SP2/SP4.

This RS-232 port normally operates at 1,200 bps and is connected to a modem in the processor cabinet. This allows Technical Support to communicate with the machine via the modem for diagnostics and evaluation of a problem.

SPU Disk

The SPU disk is a 5.25-inch, 170-Mbyte hard disk drive with an embedded disk controller. The SPU disk is the default SPU boot device. The SPU disk is daisy-chained with the SPU tape unit, then connected to the SP2/SP4 through the SCSI interface.

The following items are stored on the SPU disk:

- CONVEX SPU OS
- Part of ConvexOS
- EGOS (the IOP operating system)
- System boot and error utilities
- All diagnostics
- Error logs
- Files containing processor configuration and performance information

SPU Tape Drive

The SPU tape drive is used for SPU software distribution (CONVEX SPU OS, control stores, diagnostics, and upgrades). This software is supplied on a tape cartridge which is read by the SPU tape drive whenever new or additional software is to be loaded into the SPU disk.

It is also possible to boot the SPU from the SPU tape drive (assuming that a bootable tape is available). Booting from the SPU tape is selected by the operator at the system console.

SP2/SP4 Software

SP2/SP4 software is stored on the SPU disk. This software includes operating systems, diagnostics, and utilities that are used by the SPU and the C200 Series system.

Operating Systems

The SPU disk contains all or portions of the three operating systems required to run the C200 Series processor (CONVEX SPU OS, ConvexOS, and EGOS).

Currently, the SP2/SP4 runs CONVEX SPU OS, the C200 Series processor runs ConvexOS, and each individual I/O processor runs EGOS (Event Governed Operating System).

Once the C200 Series processor has been booted, there are three different operating systems running somewhere in the system. They are:

- **CONVEX SPU OS**—Operates the SP2/SP4. CONVEX SPU OS controls any processes that are requested to run in the SP2/SP4. These processes include tasks such as finding a particular file on the SP2/SP4 disk or tape drive, scanning C200 Series processor boards, and handling messages between the CCUs. CONVEX SPU OS also schedules all of the on-going jobs running on the SP2/SP4 which are swapped in and out as needed (memory scrubber daemon, and so on).
- **ConvexOS**—(the mini root—a small portion of the root partition) is contained on the SP2/SPU4 disk so that it can be loaded into system memory at boot time. This code is sufficient to configure the C200 Series processor so that it can find its own root partition on the system disk and complete the boot process.
- **EGOS**—Is the operating system for the I/O processors. It is kept on the SP2/SP4 disk and loaded into main memory at system boot time. At the time of I/O processor initialization, a copy of EGOS is loaded from main memory into IOP local memory, and then executed.

Diagnostics

Most diagnostics consist of two parts—one part runs in the SP2/SP4 and the other part runs in the particular functional unit that is being tested (IOP, CPU, etc.).

Generally, the SP2/SP4 runs its portion of the diagnostic and the functional unit under test runs its portion of the test concurrently. As the test proceeds, the SP2/SP4 and the functional unit under test communicate progress to each other.

Both the SP2/SP4 portion, and the C200 Series unit under test portion of the diagnostics are kept on the SP2/SP4 disk.

Utilities

The SP2/SP4 disk contains all utilities required to initialize and boot the SP2/SP4 and C200 Series system, maintain system performance, and create and retain records of system performance (and problems).

Figure 1–21 shows a functional block diagram of the CPU Utility Unit (CPX) of a C200 Series system:

Figure 1–21, CPU Utility Unit (CPX) Functional Block Diagram

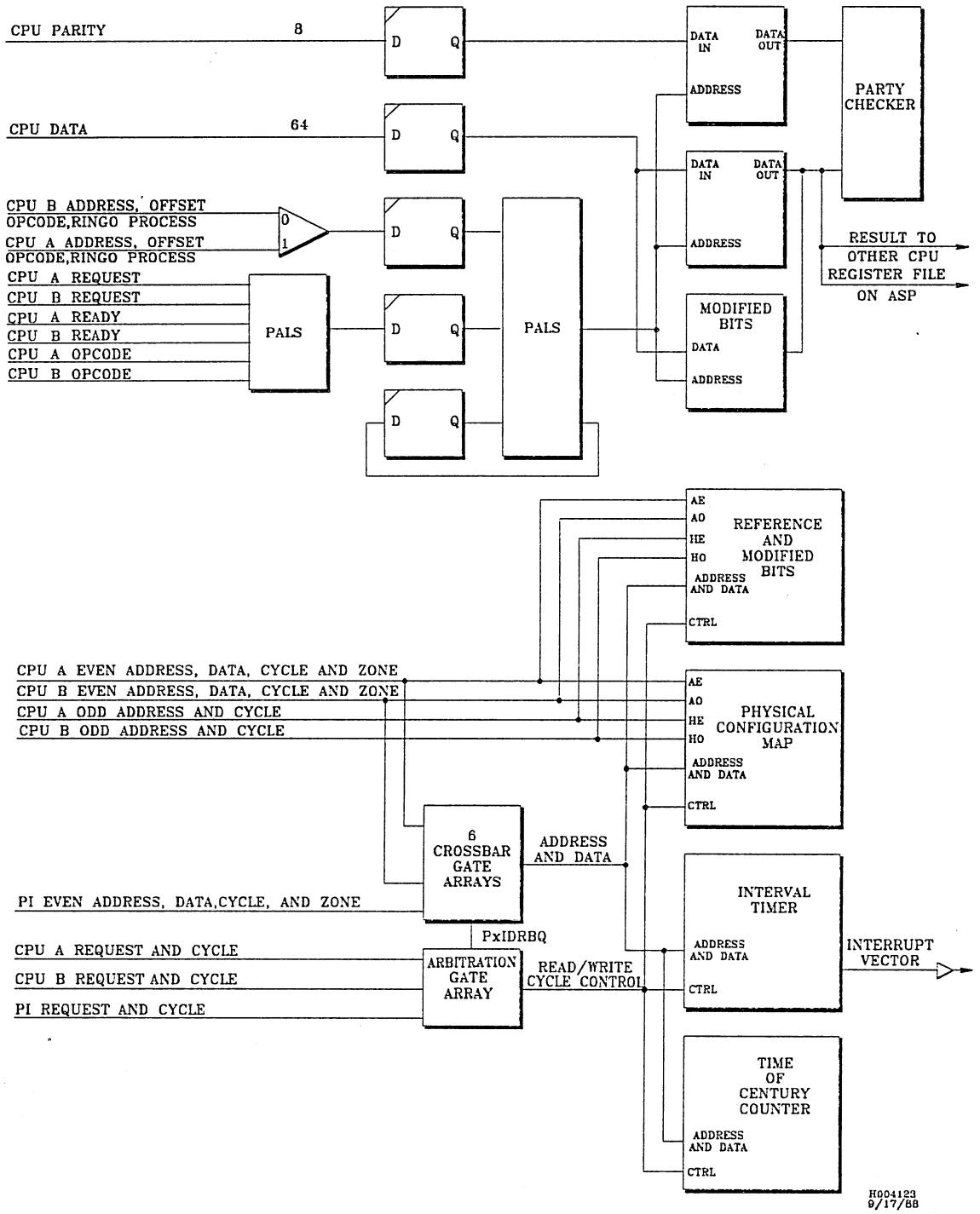


Figure 1–22 shows a functional block diagram of the System Control Monitor (SCM) of a C200 Series system:

Figure 1–22, System Control Monitor (SCM) Functional Block Diagram

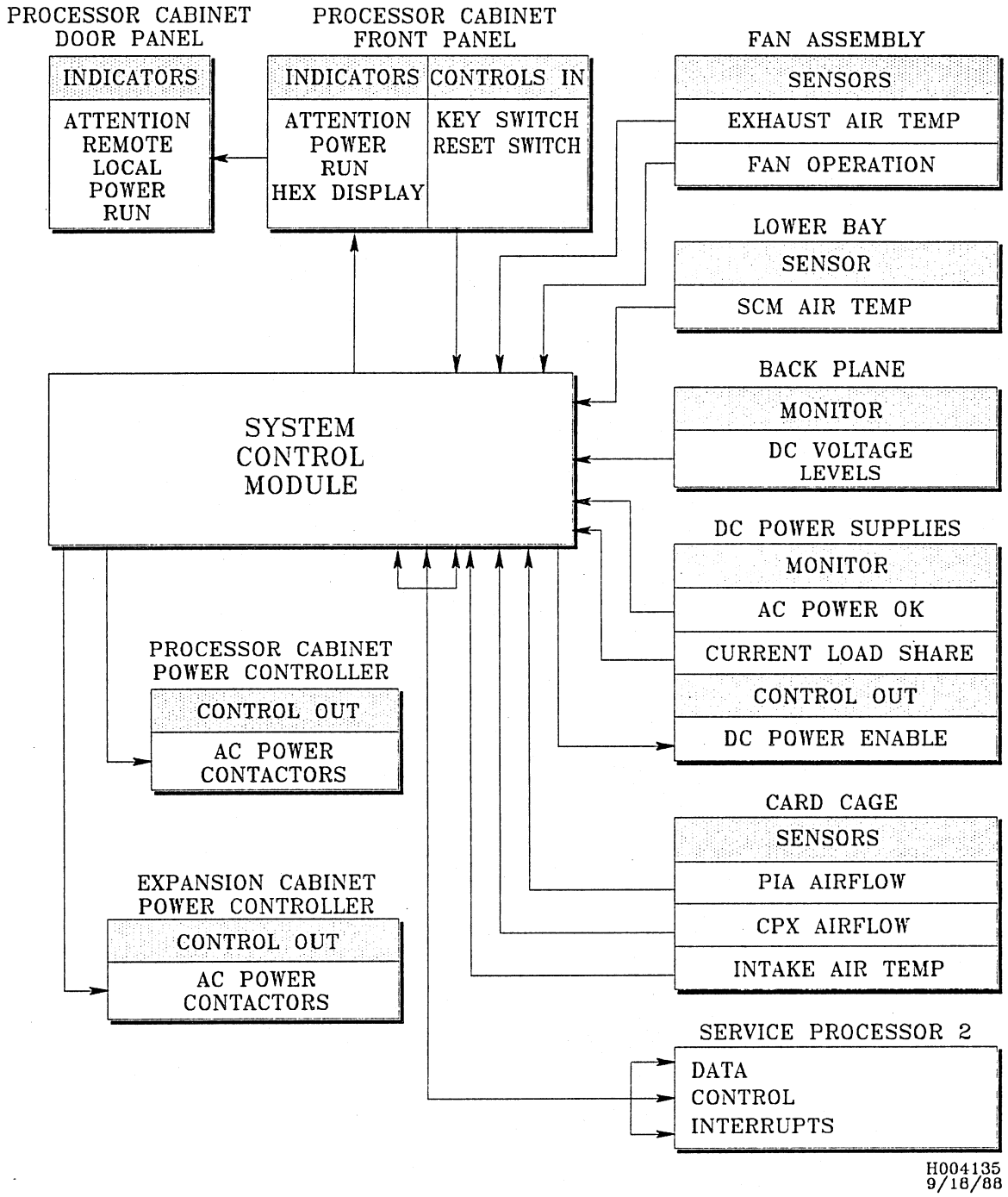


Figure 1–23 shows a functional block diagram of the Service Processor Unit 2 (SP2) of a C200 Series system:

Figure 1–23, Service Processor Unit 2 (SP2) Functional Block Diagram

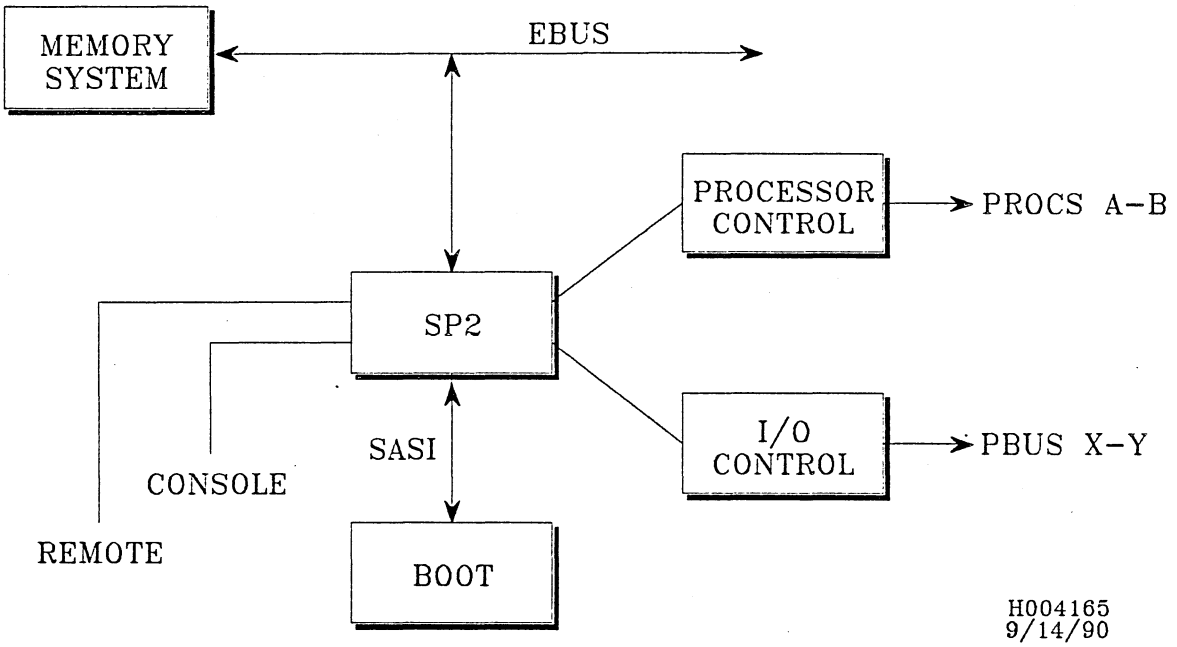
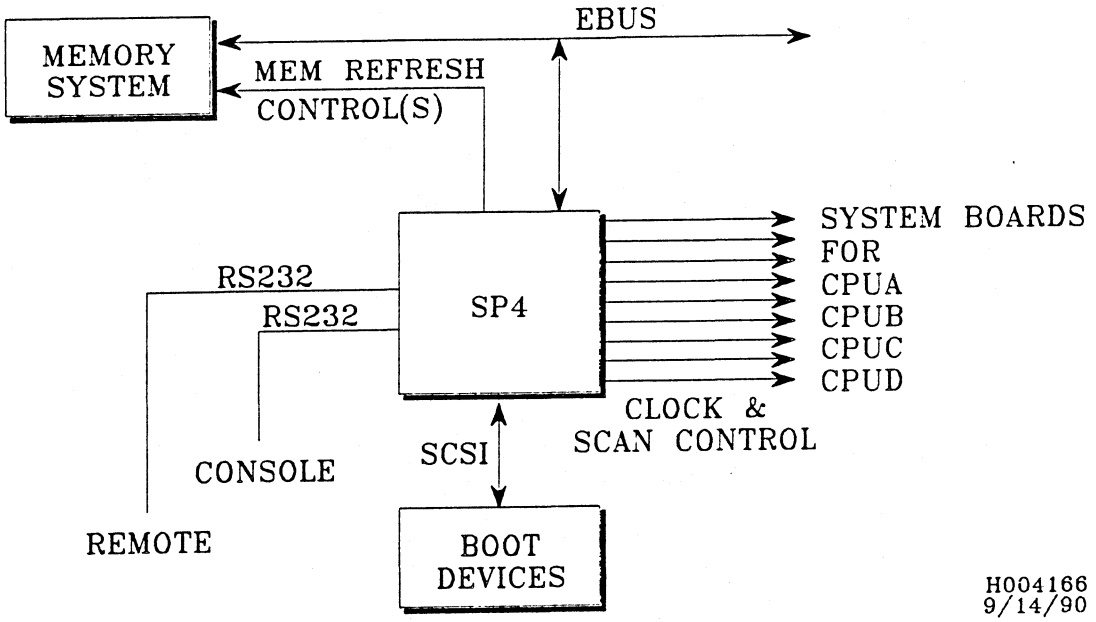


Figure 1–24 shows a functional block diagram of the Service Processor Unit 4 (SP4) of a C200 Series system:

Figure 1–24, Service Processor Unit 4 (SP4) Functional Block Diagram)



1.4 Processor Operation Environment

There are *five* operation states (modes) in CONVEX supercomputers:

- Power up/power down
- Soft front panel (firmware)
- Service Processor Unit Operating System (CONVEX SPU OS)
- ConvexOS single-user
- ConvexOS multi-user

These five CONVEX operation states are described below. Refer to the *CONVEX System Manager's Guide* for additional information on these operation states. This section also contains descriptions of the System Monitor Board (SMB), System Control Module (SCM), system prompts, the *fsck* filesystem check program, and SPU self-test diagnostics.

1.4.1 Power-Up/Power-Down

The *power-up* operation state occurs when processor cabinet main circuit breaker is in the **ON** position, and the front panel control keyswitch is in one of the following three positions:

- **LOCAL MAINTENANCE**
- **SECURE EXECUTION**
- **REMOTE MAINTENANCE**

CONVEX supercomputer processing is **enabled** when the system is in the power-up state.

The *power-down* operation state occurs when the processor cabinet main circuit breaker is in the **OFF** position, or the front control panel keyswitch is in the **OFF** position.

CONVEX supercomputer processing is **disabled** in the power-down state.

1.4.2 Soft Front Panel (Firmware)

The soft front panel is an interactive program stored in Erasable Programmable Read-Only Memory (EPROM), on the SP2/SP4 circuit board. The soft front panel is accessed primarily to modify the settings of the firmware switches (options) used to configure the system. The SP2/SP4 stores the settings of the firmware options in nonvolatile EPROM memory, so they're saved even when the system is powered off.

The soft front panel program executes whenever the power-up procedure is completed and the keyswitch is turned from **OFF** to one of the following three positions:

- **LOCAL MAINTENANCE**
- **SECURE EXECUTION**
- **REMOTE MAINTENANCE**

The soft front panel is the first level of software executed as the system boots from a power-down state.

> **NOTE**
>
> The soft front panel program is executed when the keyswitch is turned from the **OFF** position to **SECURE EXECUTION** position. However, the system boots directly to ConvexOS multi-user mode and does *not* stop at the soft front panel prompt.

Entering commands into the soft front panel program on the system console allows the system manager to examine and modify system operation. For example, the soft front panel can be programmed to automatically boot to a specific level of software, or to boot to the SPU or CONVEX operating systems (CONVEX SPU OS or ConvexOS).

1.4.3 SPU Operating System (CONVEX SPU OS)

The Service Processor Unit Operating System (CONVEX SPU OS) is based on AT&T's Version 7 UNIX operating system, an early version of the UNIX operating system designed for smaller applications. Unlike 4.2 BSD, Version 7 is not a virtual-memory system.

The CONVEX SPU OS controls CONVEX diagnostic software, coordinates error logging, and boots the CPU. Refer to the *CONVEX SPU UNIX Utilities Manual*, the *CONVEX Diagnostic Utilities Manual (C1, C120)*, and the *CONVEX Diagnostic Utilities Manual (C200 Series)* for a complete description of CONVEX SPU OS functions and related utilities.

1.4.4 CONVEX Operating System (ConvexOS)

ConvexOS is a demand-paged, virtual-memory operating system derived from Berkeley UNIX 4.2 BSD. The system consists of the following:

- Command interpreters (shells)
- Utilities
- A filesystem
- A kernel that manages all system resources, including some interrupts, memory management, process control, and I/O.

The ConvexOS acts primarily as a coordinator and scheduler of system resources. The ConvexOS supports three command interpreters (also called shells). They are:

- The Bourne shell
- The C shell
- The COVUE shell

These shells enable users to issue commands to the system and to combine commands together. Refer to the *UNIX Primer Plus* for additional information about the COVUE shell.

Two modes of operation exist within the ConvexOS computing environment:

- **Single-user mode**—The system manager runs maintenance software, performs filesystem checks, and mounts and unmounts filesystems.
- **Multi-user mode**—The system manager performs periodic system checks, such as monitoring user file space, number of users, and amount of user processing time.

1.4.4.1 Basic Booting Procedures

When booting CONVEX supercomputers with the keyswitch in the **LOCAL MAINTENANCE** or **REMOTE MAINTENANCE** position, the soft front panel prompt, `(fp)>` is displayed.

From the `(fp)>` prompt, the system can be booted to another environment by entering the desired mode of operation command (**sm=n**[*ormal*], **sm=d**[*iagnostic*], or **sm=a**[*lternate*]) followed by the **b**[*oot*] command:

```
(fp)> sm=n

(fp)> b
```

CONVEX supercomputers can also be booted directly to ConvexOS multi-user mode by turning the keyswitch to the **SECURE EXECUTION** position.

1.4.4.2 Power-Up Procedures

Reasons for Using This Procedure

To power up the machine.

Procedure Description

This procedure describes the steps required to power up CONVEX supercomputers from a power-down state. The procedures described in this section assume that all system components, e.g., hard disks and tape drives, have been powered down normally.

> **CAUTION**
>
> Power up the disk drives *last*. Powering up the disk drives last ensures that disk files are not corrupted by power fluctuations during the power-up sequence.

The power-up procedure shown in Table 1-3 consists of nine steps. Follow each step *in sequence.* Any attempt at short-cuts may result in serious damage to the system. The easiest way to make sure the various breakers and switches are in the right position is to turn them all *off* as a preliminary step. Then, follow the steps of the power-up procedure. User terminals can be turned on (or off) at any stage in the procedure.

Table 1–3, Power-Up Procedures

STEP	PROCEDURE
1	This procedure begins from a power down state. Remove the SPU tape cartridge from the SPU tape drive.
2	Turn the front control panel keyswitch to the **OFF** position.
3	Turn the AC power-controller mode switch to the **REMOTE** position.
4	Turn the main circuit breaker on the processor cabinet's AC power-controller panel to the **ON** position.
5	If there are additional AC power-controller panels in expansion cabinets, turn the mode switches to the **REMOTE** position and turn the circuit breaker switches to the **ON** position.
6	Apply power to the tape drive(s).
7	Apply power to the disk drive(s).
8	Turn the front control panel keyswitch to the **LOCAL MAINTENANCE, SECURE EXECUTION,** or **REMOTE MAINTENANCE** position.
9	Turn the system console power switch to the **ON** position. The soft front panel prompt (`(fp)>`) should display on the system console.

NOTE

If the keyswitch is turned to the **SECURE EXECUTION** position in step 8, the system boots directly to ConvexOS multi-user mode and displays the login prompt.

NOTE

If the system is down and the *power on-reboot* option is disabled, the system will not boot to ConvexOS when the keyswitch is turned to the **SECURE EXECUTION** position. Enable the *power on-reboot* option by entering the following command at the soft front panel prompt:

```
(fp)> set power on-reboot=enable
```

NOTE

If the *automatic-reboot* option is disabled, the system does not automatically reboot when the **SYSTEM RESET** switch pressed, nor does it automatically reboot after a system crash. Enable the *automatic-reboot* option by entering the following command at the soft front panel prompt:

```
(fp)> set automatic-reboot=enable
```

The power-up procedure should successfully complete if the keyswitch is turned to any position except **OFF** (step 8). If the machine fails to power up, check the following:

- The **POWER** LED (on the front control panel) and the appropriate AC power status LED indicators should be lit (one for each power supply configured).
- The power lights on all peripherals should be lit, and the disk drives should spin up.
- The mode switch on the AC power-controller panel should be in the **REMOTE** position.
- The system status code **FF** should appear on the **SYSTEM STATUS** display on C200 Series models. Any other status code indicates that the system has a problem.

If the system still fails to power up, turn the keyswitch to the **OFF** position and repeat the power-up procedure from step 1. Contact the CONVEX Technical Assistance Center (TAC) if the system does not power-up properly.

Additional Considerations

None

1.4.4.3 Booting From Power-Up To ConvexOS Multi-User Mode

Reasons for Using This Procedure

The ConvexOS, in multi-user mode, is used for general timesharing. Invoking the multi-user system mounts all filesystems, starts the daemons, and runs *init* to enable logins on all terminals.

The system displays the ConvexOS login prompts (`login:` and `Password:`) if the keyswitch is turned directly to **SECURE EXECUTION**. After logging in, a default shell prompt for the current environment is displayed on the system console. For example, the default C Shell (csh) prompt is `%`, and the default Bourne Shell (sh) prompt is `$`.

The system manager sets a default shell environment in the */etc/passwd* file for each user. To execute a shell environment of choice, the shell's executable binary must be available and the system manager must list the desired shell in that user's entry in the */etc/passwd* file.

Procedure Description

The procedure in Table 1-4 lists the steps required to boot directly to ConvexS multi-user mode at power-up.

From a power-down state, the system goes directly from the soft front panel to ConvexOS multi-user mode when *all* the following conditions apply:

- The keyswitch is turned to the **SECURE EXECUTION** position.
- The mode of operation is preprogrammed to *normal-os*.
- The *power-up-reboot* is preprogrammed to *enable*.

Additional Considerations

None

Table 1–4, Booting From Power-Up To ConvexOS Multi-User

STEP	PROCEDURE
1	Follow steps 1 through 7 of the power-up procedures in Table 1-3.
2	Turn the keyswitch to the **SECURE EXECUTION** position. The system performs CONVEX SPU OS and ConvexOS file checks (fsck), and memory initialization. A variety of information reflecting these processes is displayed on the system console.
3	The system automatically boots to the ConvexOS multi-user mode and displays the ConvexOS multi-user login and password prompts. Enter the login ID and password: `login:` **(login ID) <CR>** `Password:` **(valid password) <CR>**

1.4.4.4 Booting in Diagnostic Mode

To access the diagnostics mode, boot to the soft front panel, then to the CONVEX SPU OS prompt, `(spu)>`. Execute the *dshell* command to access the diagnostic shell. Its shell prompt is a colon (`:`). `This power-up procedure requires that the keyswitch be turned to` **LOCAL MAINTENANCE** or **REMOTE MAINTENANCE**. Refer to the *CONVEX Diagnostic Utilities Manual (C200 Series)* for information on diagnostic mode operations.

> **NOTE**
>
> When the diagnostic mode of operation is selected (**sm=d**), the *boot* command invokes a limited-diagnostics mode. This mode allows the system manager to execute utilities that manipulate SPU peripherals without having to boot CONVEX SPU OS. Entering a **<CR>** at the limited-diagnostics colon prompt (`:`) brings the system up to CONVEX SPU OS.

1.4.4.5 System Generation Overview

System generation creates a new version of the ConvexOS operating system based on global system-configuration parameters and hardware specifications in the system-configuration file. From this system-configuration file and files in the */sys/sysgen* directory, the *sysgen* utility creates the files and directories necessary to build a system.

> **NOTE**
>
> Throughout this chapter, the name used for the system-configuration file is *GENERIC*.

sysgen creates five directories to hold object-code files.

- */sys/GENERIC*
- */sys/GENERIC_hsp*
- */sys/GENERIC_iop*
- */sys/GENERIC_viop*
- */sys/GENERIC/os*

In each of the first four directories, *sysgen* creates

- Makefiles listing program and file dependencies for the *vmunix* system image and the system images for the CCUs
- Header files (with a *.h* suffix) listing the devices that are compiled into the system
- A header file (with a *_conf.h* suffix) listing the entry points for the driver

The */sys/GENERIC/os* directory holds the system images generated by *make*. The *sysgen* utility also creates the file */sys/GENERIC/swapvmunix.c* that describes the locations of the root partition, the argument-processing partition, and the swap partition.

After executing *sysgen*, the *make* utility compiles and links the user-configured code with the code provided by CONVEX, creating bootable system images for ConvexOS (*vmunix*) and for the CCUs (*hsp*, *iop*, and *viop*) and places these system images in standard locations.

- The *make depend* command uses the makefiles created by *sysgen* to create a list of dependencies that determine the code and data files that must be compiled. This rule list is a string of *include* files used by each source file in the system.

- When the dependency lists are created, the *make* utility is used to compile and link all system files that contain the object code required to generate the bootable system-image files.
  - */sys/GENERIC/vmunix*
  - */sys/GENERIC_hsp/hsp*
  - */sys/GENERIC_iop/iop*
  - */sys/GENERIC_viop/viop*

  If the system has been issued a source license, the *make* utility uses library files that are created during system generation and placed in these directories. If the system has been issued a binary license, *make* uses the following precompiled library files distributed with ConvexOS:
  - */sys/CCU_OBJ/libhsp.b*
  - */sys/CCU_OBJ/libiop.b*
  - */sys/CCU_OBJ/libviop.b*
  - */sys/CPU_OBJ/libunix.a*
- The *make install* command combines the bootable system-image files and places them in the directory */sys/GENERIC/os* that was created by *sysgen*.

When the bootable system-image files have been created, they must be moved to the SPU disk, from which the new operating system is booted. After rebooting the system, the new *vmunix* should be copied from the */sys/GENERIC/os* directory to the / (root) directory.

1.4.4.6 Essential System Files

A system file is considered essential, if ConvexOS requires it for proper system operation. Essential system files require periodic maintenance and should be checked if ConvexOS is behaving abnormally. For descriptions of the essential system files, refer to Appendix A.

Chapter 2
C200 Series Architecture Overview

2.1 Overview

The purpose of this chapter is to provide an introductory overview to the C200 Series architecture in order to assist an engineer's or technician's understanding of the C200 Series architecture. The primary audience for this chapter is Manufacturing and Field Engineering. The goal is to provide enough tutorial information so that the *CONVEX Architecture Reference* and other hardware functional specifications can be used effectively as a reference.

Although this chapter is somewhat duplicative of the *CONVEX Architecture Reference*, the information is presented in a tutorial fashion, including implementation-dependent information to enhance understanding. In some places, information from the *CONVEX Architecture Reference* is included verbatim since the description provided there was sufficiently tutorial in nature. The information is included in this document to provide a suitably complete introduction.

This document does **not** attempt to explain the entire architecture in detail. Refer to the *CONVEX Architecture Reference* for more detail.

The audience is assumed to be somewhat familiar with the C100 Series architecture. Often, descriptions will contrast the C200 Series architecture with the C100 Series architecture to provide a familiar analogy. In some cases, an implementation-dependent description is provided to enhance understanding. This will be the C201/C202/C210/C220 implementation. The C230/C240 implementation may be slightly different.

First, a definition of the C100 and C200 terms is in order. The C100 Series architecture is the marketing name given to the original CONVEX architecture as implemented on the C100 Series systems. The C200 Series architecture's first implementation is on the C210/C220/C201/C202 machines under ConvexOS version 7.0. This is the same architecture implemented on the C230/C240 machines.

2.1.1 What Makes a C2 a C200 Series Machine?

The C2 Series hardware has six writable control stores that must be loaded for it to function properly. The control stores are the following:

- *us.wcs* or *us.200.wcs* — Scalar processor sequencer control
- *sr.wcs* — Scalar processor scratch RAM
- *ua.wcs* — Vector processor add microcontroller
- *um.wcs* — Vector processor multiply microcontroller
- *ul.wcs* — Vector processor load/store microcontroller
- *vd.wcs* or *vd.200.wcs* — Vector processor dispatch table

Two of these control stores may be loaded with one of two different files. If *us.wcs* and *vd.wcs* are loaded, the machine runs the C100 Series architecture. This would be the case for a C210A. If *us.200.wcs* and *vd.200.wcs* are loaded, the machine runs the C200 Series architecture. This would be the case for all other C200 Series hardware class machines. The difference in these microcode files handle all differences between the instruction sets.

For example, the new *sqrt.s Vj,Vk* C200 Series instruction must be detected as an unimplemented opcode on a C2 Series machine running the C100 Series architecture. To handle this, the *vd.wcs* file dispatches the vector processor to a no-operation, and *us.wcs* performs the unimplemented instruction trap. If the machine is running the C200 Series architecture, *vd.200.wcs* dispatches the multiply pipe to perform the vector square root, and *us.200.wcs* performs a no-operation on the scalar processor.

There are also various bits in the scan rings that must be initialized in a particular manner to define a C100 or C200 Series architecture. Refer to the *CONVEX SPU UNIX Utilities Manual*, *CONVEX Diagnostic Utilities Manual (C1, C120)*, and the *CONVEX Diagnostic Utilities Manual (C130, C210, C220)* for more information.

2.2 Extended Opcodes

In order to add many new instructions to the C100 Series architecture, the C200 Series architecture includes an extended opcode format. In this format, any one- to three-halfword CONVEX opcode may be preceded by a 16-bit value known as a prefix opcode. The C200 Series architecture defines two prefixes with values 7EF0 (extended-0) or 7EF8 (extended-1). When either of these prefixes is encountered in the instruction stream, the subsequent halfword is used as the root opcode of the instruction, thereby generating a new set of opcodes for the C200 Series architecture.

For example, 6800 is the opcode for *eq.b Vj,Vk*, while 7EF0 6800 is the extended opcode for *eq.b.f Vj,Vk*, and 7EF8 6800 is the extended opcode for *eq.b.t Vj,Vk*. This example also shows the major use for the extended opcode prefix—operation under mask (the use of operation under mask will be described in the next section). In general, the root opcode will be the nonmasked version, the root prefixed with 7EF0 will be under mask false, and the root prefixed with 7EF8 will be under mask true.

The mask polarity is the **only** distinction that can be made with the two prefixes; if an extended opcode is not used for an under mask instruction then there are two opcodes which will cause its execution. For example, both 7EF0 3700 and 7EF8 3700 are equivalent opcodes for the *snd.l Sk,Ceffa* instruction. In addition, the last three bits of the prefix opcode are not used. This means that prefixes 7EF0 through 7EF7 are equivalent. Similarly, 7EF8 through 7EFF are equivalent.

2.3 New Vector Operations

Many of the new C200 Series instructions are vector operations. Some new functions have been added, such as vector conversions and square root, but these are quite straightforward in nature and will not be explained here. The major architectural addition to vector processing in the C200 Series is the *operate under mask* capability.

In this mode, the bit of the Vector Merge (VM) register corresponding to each vector element is examined to either enable or disable that vector element from the operation. The least significant bit of VM corresponds to vector element 0, on up to the most significant bit which corresponds to vector element 127. Operate under mask is indicated in the assembly language by adding a suffix to the instruction; either *.t* for under mask true or *.f* for under mask false.

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

An example of the use of this feature follows. Consider the following code sequence:

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

Ignoring the length and stride setup, this can be vectorized with the following instruction sequence:

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

2.4 Intrinsics

The C200 Series architecture added a group of instructions called *intrinsics*. They are microcode and hardware implementations of frequently used runtime library routines. They are available in *.s* (single precision) and *.d* (double precision) versions. For example, *sin.(s|d)* means that both *sin.s* and *sin.d* exist. The intrinsic instructions include the following:

- *sin.(s|d) Sk* — Calculate trigonometric sine of radian quantity in Sk, return result in Sk
- *cos.(s|d) Sk* — Calculate trigonometric cosine of radian quantity in Sk, return result in Sk
- *exp.(s|d) Sk* — Raise *e* to the Sk power, return result in Sk
- *ln.(s|d) Sk* — Calculate natural logarithm of Sk, return result in Sk
- *atan.(s|d) Sk* — Calculate trigonometric arctangent of Sk, return result in Sk
- *sqrt.(s|d) Sk* — Calculate square root of Sk, return result in Sk
- *sqrt.(s|d) Vj,Vk* — Calculate square root of each element of Vj, return result in corresponding element of Vk

All of these instructions operate in either CONVEX native or IEEE floating point format, based on the state of the IEEE bit in the PSW. Some new arithmetic exception conditions are raised by these instructions. Refer to the "New PSW Bits and Traps" section in this chapter for more information about these new exceptions.

With the exception of the square root instructions which are implemented in hardware, all of these instructions use the same algorithm as the runtime libraries and therefore should produce the same results. The square root instructions produce slightly different (more accurate) results.

2.5 Memory and Cache Management

This section presents the extensions, added by the C200 Series architecture, that deal with virtual memory management. All of these architectural additions were required due to the ability of a process to run simultaneously on two or more CPUs. The C200 Series architecture defines the following attributes:

- **CPU** — One physical processing unit, consisting of a scalar and vector subsystem
- **Complex** — The entire set of one or more CPUs in a configuration
- **Sub-Complex** — Any subset of a Complex
- **Process** — A collection of instruction streams within a single virtual address space, i.e., sharing the same Segment Descriptor Registers (SDRs)
- **Thread** — Any single instruction stream executing within a process

A process is made up of a collection of one or more threads. For example, a C220 could have a process executing on the entire Complex, with one thread executing on each CPU. There are two registers on each CPU which help define the memory management scheme for threads:

- **Communication Index Register (CIR)** — This 3-bit register defines which subset of the communication registers is being used by the process executing on a CPU (refer to "Communication Registers" section for more information). The CIR can represent 8 different index values. The CIR index is the primary definition point of an operating system process (each executing process has a different CIR index).

  The CIR defines which Segment Descriptor Registers (SDRs), that reside in the communication register, are in use by a process. The C2 Series Complexes supporting the C200 Series architecture implement a CIR as a 3-bit register. There can be more than 8 processes, but only 8 processes can be loaded in the communication registers at one time.
- **Thread Identifier (TID)** — This 5-bit register is used to subdivide a process into disjoint threads. Up to 32 threads may exist in the same process, i.e., have the same CIR. The TID makes the thread unique where necessary. The TID is used for implementing unshared memory. The manner in which a processor "becomes" a particular TID is discussed in the "Multiprocessing" section.

When the value in either the CIR or TID is changed, the virtual-to-physical address translation on the CPU has been altered. In the C2 Series Complexes, the address translation (ATU) cache is tagged with CIR and TID values to remove the need to purge the ATU when the CIR or TID value is changed. The instruction cache is not tagged in this manner, so it must be purged when the CIR index value is changed. It only needs to be purged on TID changes if unshared text (instruction) pages are used (not supported in ConvexOS).

2.5.1 Resource Structure

Communication registers can be viewed as a form of fully semaphored memory, available in considerably smaller quantities than virtual memory. One of the primary functions of communication registers is giving software a means to relocate frequently accessed data from virtual memory into a location with internal locks.

Memory duals of the communication instructions were added to perform primitive functions to manipulate virtual memory similar to the functions that manipulate communication registers. Software can use the memory duals of the communication instructions to create data structures in memory, and then relocate the critical data structures to communication registers.

2.5.1.1 Shared Resource Structure

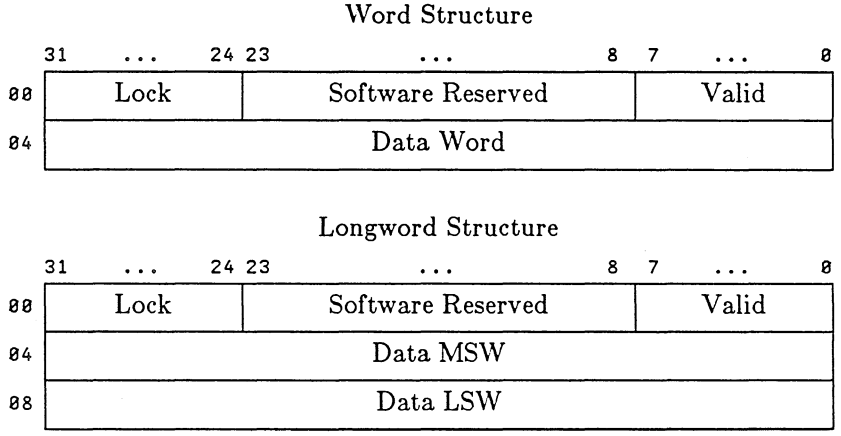
The memory duals operate on a data structure called a *shared resource structure*. This structure defines a memory format that includes a data word or longword and synchronization bytes to synchronize access to the structure.

The first byte of the structure is a lock byte, which must be successfully test-and-set as the first level access of semaphoring. Test-and-set is an indivisible operation provided by the memory system. An indivisible operation is sometimes referred to as an *atomic* operation, i.e., once the operation begins no other operation, such as interrupts, may intervene until the operation is complete.

The next synchronization byte is the valid byte, which is set if valid data follows the synchronization (lock) byte. The two synchronization bytes model the semaphoring inherent in the communication registers. The lock byte models the inherent indivisible access to the communication registers provided by their primitive functions, i.e., the memory system does not provide operations like *send* and *receive*. The valid byte models the communication lock bit; it indicates whether valid data is in the register, i.e., the structure is "valid."

The format of word and longword shared resource structures is shown in the following figure:

Figure 2–1, Word and Longword Shared Resource Structures



As an example of how this structure works, consider the *sndr.w Ak,effa* instruction. This instruction is the memory dual of the *snd.w Ak,Ceffa* instruction discussed in the "Communication Register" section.

First, a test-and-set is performed on the lock byte. If this succeeds (the lock byte was initially 0x00), the valid byte is read. Also, the data word is read into address register Ak. If the valid byte is 0xFF, then valid data exists and a success status of "1" is returned in PSW<C>. Otherwise, a failure status of "0" is returned.

If the test-and-set of the lock byte succeeds, the data is always read into register Ak regardless of the state of the valid byte because the *snd.w* communication register instruction always reads the contents of the communication register into register Ak. This is because a single level of synchronization is required for the communication registers as mentioned earlier. The full set of instructions that operate on this structure is documented in the *CONVEX Architecture Reference.*

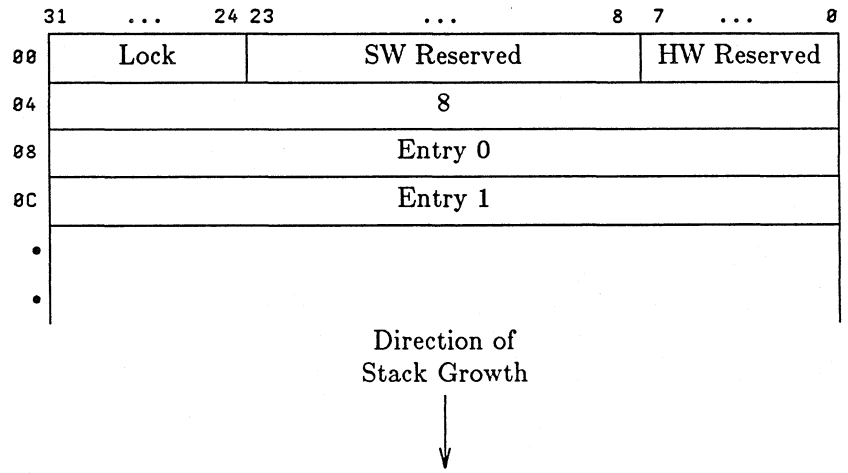
2.5.1.2 Stack Resource Structure

An extension of the shared resource structure, called a ***stack resource structure***, is provided to allow stack operations, such as push and pop on a stack of word resource structures. The stack resource structure is defined to have only word entries. Instead of the valid byte, the word resource structure contains a depth word (index) that shows the number of elements in the structure.

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

The following figure shows an example of a word resource structure containing two pushed entries:

Figure 2–2, Word Resource Structure With Two Pushed Entries



2.5.1.3 System Resource Structure

However, since more than one thread in a process could be crossing rings or faulting at the same time, the C200 Series architecture defines a *system resource structure* to manage allocation of available stacks in Ring 0. The system resource structure is essentially a stack of pointers to available stacks which are allocated to threads. Accesses to the system resource structure are synchronized by placing part of this structure in a communication register with the other part contained in memory. The communication register lock bit is used as the semaphore in order to control contention between multiple threads.

The system resource structure for Ring 0 is managed differently than a process stack resource structure in Ring 4. Whenever a thread crosses rings, or faults in any ring, the virtual address of the communication register contained in byte address 0000 0048 in page 0 of the ring being entered is read (Ring 0 for faults, interrupts, and system exceptions). This communication register contains the base address and stack index to a list of available stack pointers that are located in memory. These stack pointers point to system stacks used for cross ring calls and returns, and for saving and restoring context blocks.

After the address of the communication register is read, the communication register must then be successfully received. If the communication register cannot be received immediately, the receive operation retries until the register is successfully received.

This 64-bit register is divided into two 32-bit words. The most significant word is the base address which contains the virtual memory address of a list of stack pointers for available stacks. The least significant word is the stack index which is the byte offset from the virtual address of the next available stack pointer plus 4.

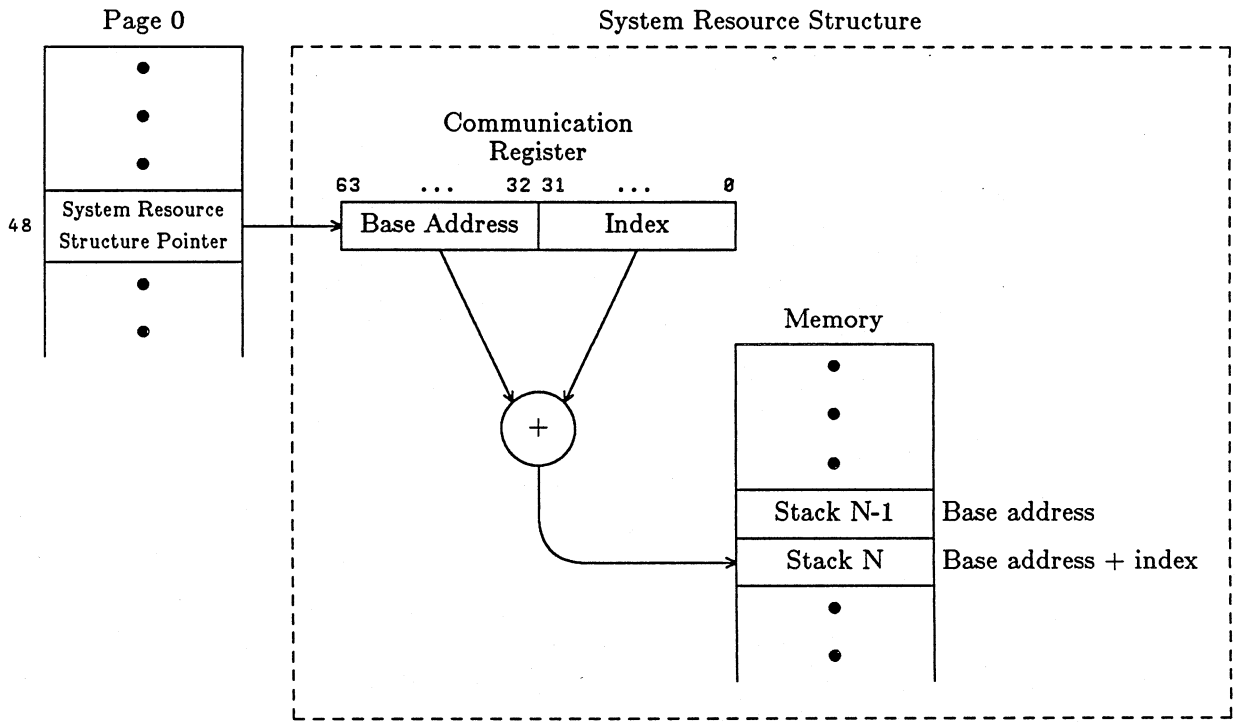
When a stack is allocated form the list, the stack pointer is fetched by decrementing the stack index by 4 and the contents (base address + decremented index) are read. The decremented value of the index is sent back to the communication register, making the system resource structure available for access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

When the thread eventually exits Ring 0 or the page fault handler via the *rtn* or *rtnc* instructions, the stack is returned to the structure in the following sequence:

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

The path followed to allocate and deallocate a stack from the stack resource structure is shown in the following figure:

Figure 2–3, System Resource Structure Accessing



2.5.2 Shared and Unshared Memory

A process can view virtual memory as either shared or unshared. Shared memory means that more than one thread may use the same virtual address to read or write the same physical location in memory. Proper synchronization must be maintained by software in this case.

Unshared memory means that each thread uses the same virtual address to access different physical locations in memory, e.g, the stack. A process that may or may not execute in parallel (depending on whether idle CPUs are available) needs to be able to use the stack without additional software overhead. This ensures that one processor is not popping something that another CPU pushed, etc. Some of the upcoming subsections describe architectural features included to facilitate thread shared memory.

2.5.3 Thread-Level Page Table Entry (PTET)

The C200 Series architecture supports hardware-implemented unshared memory by adding another level of Page Table Entry (PTE) in the address translation process. The second-level PTE (PTE2) contains the base address of a third-level table of PTEs called the thread-level PTE (PTET). If the Level 3 (LT) bit in the PTE2 is set, the table of PTET entries pointed to by the PTE2 is traversed using the processor's TID to find the physical page frame base address.

The indexing into this table is based on TID. The extra translation is performed by the processor's PTE miss resolution routines contained in microcode. When the PTE2 is returned from memory, the state of the LT bit is examined. If the LT bit is not set, the referenced page is shared memory, so the PTE2 is encached in the ATU cache as the final-level PTE, since that particular PTE2 includes the physical page frame base address.

If LT is set, the page is unshared and the microcode uses the page frame bits of the PTE2 to request the PTET for the processor's current TID. The PTET is then encached in the ATU cache as the final-level PTE. In this manner, two processors running as different TIDs in the same CIR (process) can use the same virtual address and still have unique physical memory. There are figures in the *CONVEX Architecture Reference Manual* which detail the virtual-to-physical translation process for both shared and unshared pages.

2.5.4 Data Cache Management and Remote Invalidates

One of the major problems in the design of a multiprocessor system with shared memory is that of maintaining processor cache consistency. The I/O subsystem has a separate port to main memory since it performs many of the same functions as a CPU. Since the memory system is multiported, a CPU Complex with shared memory must maintain the consistency of each CPU's data cache. The consistency of a CPU's data cache is maintained by ensuring that when one CPU has loaded data and caused a local encachement, that CPU is informed when another processor stores data to that address location so both CPUs maintain a consistent view of physical memory.

The C2 Series CPU hardware supporting the C200 Series architecture uses a technique called remote invalidation. This technique has each CPU watching all memory ports for data stores and invalidates its local cache when a data store is made. There are some cases when invalidates are missed.

For example, if one CPU is spinning in a loop loading and comparing a word of memory, the data could become "stuck" in that CPU's data cache and not be invalidated when another CPU stores data to that same address location. This would happen if the remote invalidate from the storing CPU was received by the loading CPU before the load data returned from memory and wrote the cache.

Software must follow the following rules for communication through shared memory.

- A lock byte with full semaphoring must be maintained around the shared region.
  - A *tas effa* instruction must be successfully performed (i.e., the carry bit (PSW<C>) returns as 1) before writing or reading the shared region
  - A *tac effa* instruction must be performed after reading or writing is complete. Since the *tas* instruction ensures the lock byte is set when it succeeds, the return status from the *tac* instruction need not be checked before continuing.

Both the *tas* and *tac* instructions perform an *msync* instruction internally, i.e., the instructions wait for all stores on the processor to reach the memory system boundary before performing the test and modify (set for *tas*, clear for *tac*).

The C200 Series architecture also provides an explicit *msync* instruction in case it is needed for other shared memory applications. One such application is memory structures locked with communication registers, which is discussed in detail at the end of the "Communication Register" section of this document.

2.5.5 Unencacheable Pages

In some situations the operating system views the cache consistency rules to be too stringent; i.e., the overhead for full synchronization is too expensive or in some cases impossible to enforce. Therefore, the C200 Series architecture added an additional bit, the Cache Load Bypass (CLB), to the bottom level PTE (PTE2 or PTET).

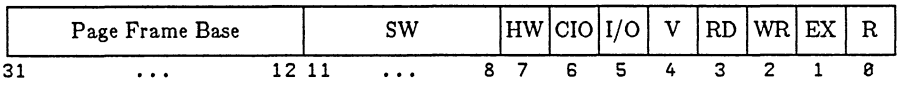
The CLB bit forces a referenced page to appear unencacheable. On a load for an unencacheable page, the data cache is bypassed forcing the load to reference main memory. The CLB bit does not keep the data from being encached on stores.

2.5.6 Page Table Entry (PTE)

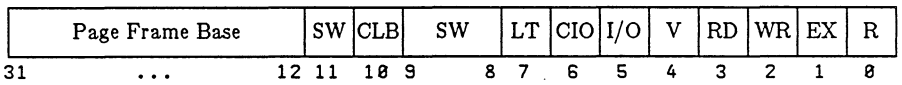
The PTE format for the C200 Series architecture is shown in the following figure:

Figure 2–4, C200 Series PTE Format

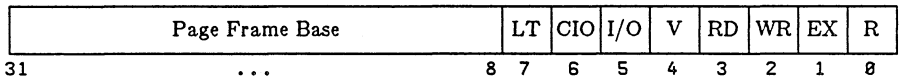
First-Level Page Table Entry



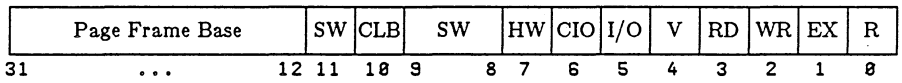
Second-level PTE, Shared Page (LT = 0)



Second-level PTE, Unshared Page (LT = 1)



Thread-level Page Table Entry



2.6 Communication Registers

The C200 Series architecture defines a new hardware structure, a *communication register,* to facilitate multithreaded execution of a process across multiple CPUs in a Complex. A communication register is a 32-bit or 64-bit addressable register with an associated hardware-maintained lock bit. The lock bit is used by software and hardware as a semaphore on the contents of the register.

A single set of communication registers is provided for a Complex. For C200 Series machines, the communication registers are implemented in a 1,024 location, 64-bit RAM, with a 1,024 location, 1-bit lock RAM using the same physical RAM address.

2.6.1 Communication Register Addressing

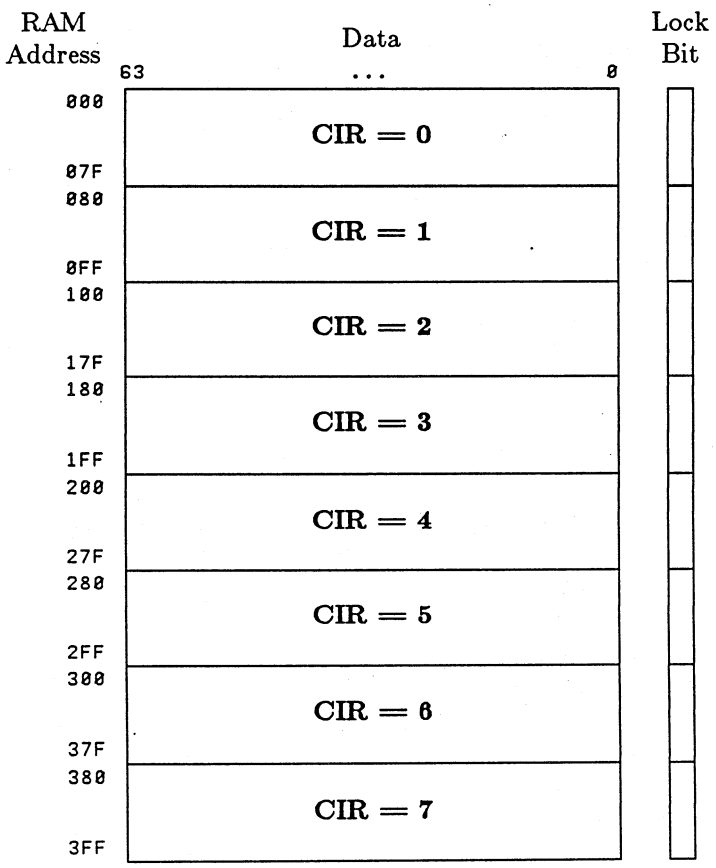
Communication registers are addressed with a 16-bit virtual address supplied in the instruction. This address is referred to as the Communication Register Effective Address (Ceffa). This virtual address is translated into a physical address using a process analogous to the virtual-to-physical translation of virtual memory addresses.

For communication registers, the Communication Index Register (CIR), takes the place of the SDRs and PTEs in the memory scheme. The CIR is the root identifier of a operating system process; i.e., when the CIR is changed the entire process context is different. The *CONVEX Architecture Reference* has a general architectural description of this translation with appropriate figures. In this document, the implementation of the C200 Series architecture is described with figures to illustrate the architecture and explain the particular implementation.

In the C200 Series Complex, the 1,024-location communication register address range is subdivided into eight 128-location partitions, each of which is accessible from a process by loading an index value of 1 in the CIR of any CPU. The CIR can be viewed as a base address of a unique region of communication registers. Except for a special physical addressing scheme that is independent of the CIR, the CIR completely restricts the processor to one partition of the communication registers (for the C200 Series Complex, one of the eight partitions of 128 registers).

Figure 2–5 shows the division of the C200 communication registers with the associated CIR, and the physical communication address range to each CIR:

Figure 2–5, C200 Series Communication Register CIR Division



Each partition of the communication registers is assigned to a particular process. The communication registers are subdivided within that process to accommodate a ring protection mechanism for communication register addressing. Each process allocates some communication registers for Ring 4 (user) programs, some for Ring 0 software (operating system) programs, and some for Ring 0 hardware (microcode use).

Each implementation of the C200 Series architecture can have a different number of communication registers in each division, but the communication registers all have the same basic division.

The 16-bit virtual address space of 0000 through FFFF must be translated based on the ring of execution to the physical addresses available in the implementation. This ring mechanism is slightly different than the memory address ring wrapping scheme employed in the CONVEX virtual memory architectures.

Virtual memory addresses have 5 distinct rings; Rings 0, 1, 2, 3, and 4. Virtual memory address Rings 0, 1, 2, and 3 are considered the same. Communication register virtual addressing has only 2 distinct rings; either Ring 0 or Ring 4.

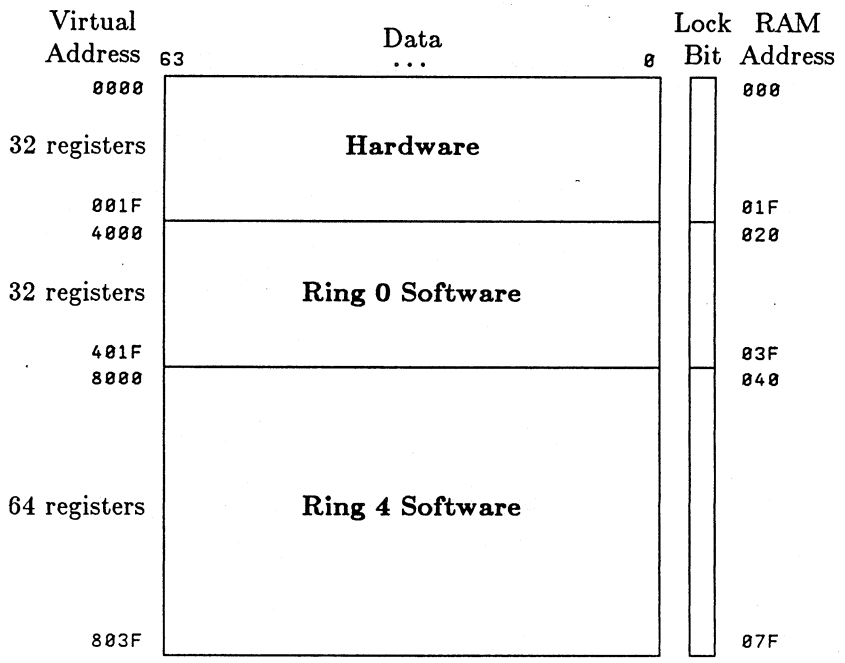
The C200 Series architecture divides the communication register address space into ranges 0000 through 7FFF for hardware and Ring 0 software, and 8000 through FFFF for Ring 4 software. An *invalid communication address* system exception is executed if a program violates this ring protection.

For example, if a user (Ring 4) process (thread) references the communication addresses 0000 or 4000, the thread will cause a system exception. In addition, any virtual communication address that does not map to a physically present communication register will also cause an invalid communication address system exception. More details on this type of exception are provided in the "Page 0, Traps, and Exceptions" section.

The C200 Series hardware has 128 registers accessible in each CIR index. These are divided into 32 for Ring 0 hardware, 32 for Ring 0 software, and 64 for Ring 4 software. The true RAM address is obtained by adding the offset, Ceffa, to the base physical address of the communication register partition referenced by that CIR index.

Figure 2–6 shows the virtual-to-physical address mapping for CIR = 0 (except for a special physical addressing mode described subsequently):

Figure 2–6, C200 Series Virtual-to-Physical Address Mapping For CIR = 0



This organization indicates that for the C200 Series hardware, the virtual address ranges 0020 – 3FFF, 4020 – 7FFF, and 8040 – FFFF are unimplemented and cause invalid communication address traps. This is true with the exception of the range 3C00 – 3FFF, which is used to implement the special physical addressing scheme subsequently described.

Although each CPU has direct access to a portion of the communication registers based on the CPU's current CIR index, hardware and Ring 0 software can also access communication registers in partitions other than their own. For this purpose, a special physical address mapping called *communication physical addressing*, is provided. This type of addressing should not be confused with the virtual-to-physical memory address translation process.

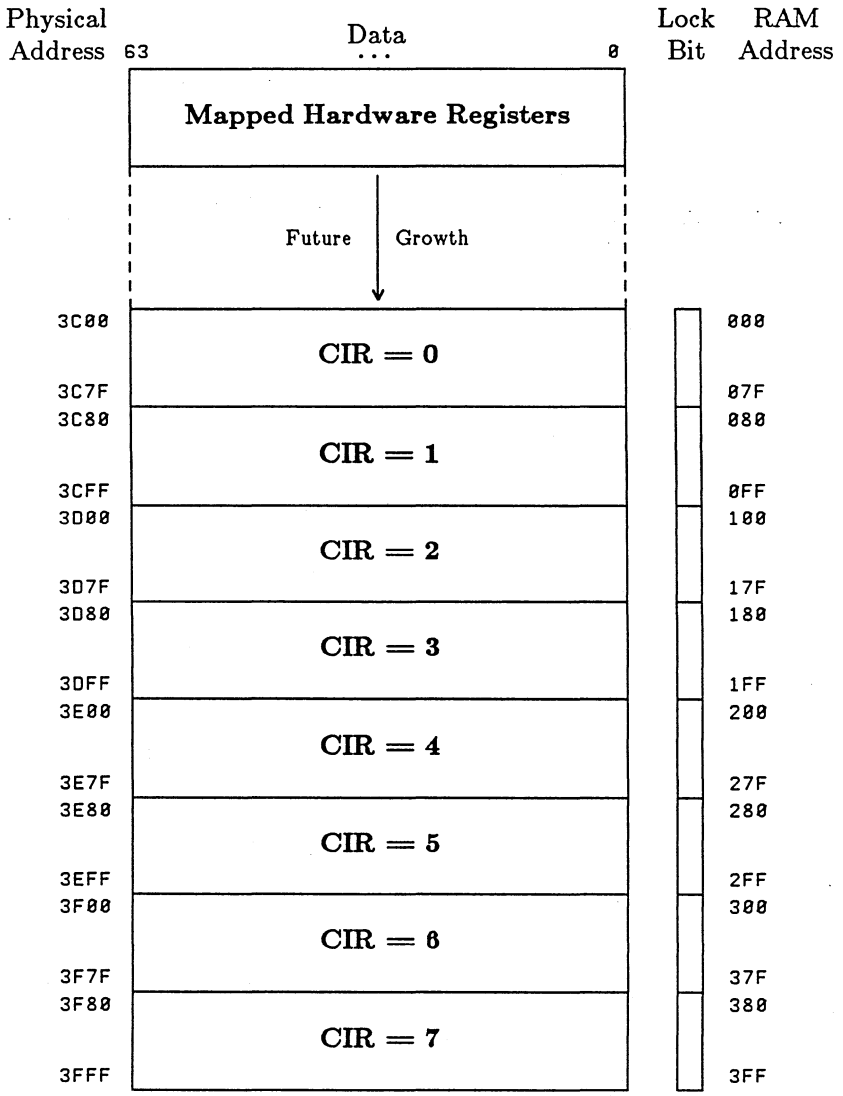
There are two virtual addresses that refer to each communication register — one is CIR based (virtual) and one is CIR independent (physical). Each implementation of the C200 Series architecture sets aside a range of virtual communication register addresses with enough address space to provide a second one-to-one address mapping. The second address mapping always ends at virtual address 3FFF, which places it in the Ring 0 (privileged) communication address space.

These two addressing schemes (CIR based and CIR independent) together comprise the entire communication addressing mechanism.

Since the C200 Series has 1,024 communication registers, its physical address mapping is located in the range 3C00 – 3FFF.

The breakdown of the physical address mapping and CIR division for the C200 Series machines is shown in the following figure:

Figure 2–7, C200 Series Physical Communication Register Address Mapping



The following table shows how a 16-bit virtual address is translated into a 10-bit RAM address for the C200 Series machines:

Table 2–1, C200 Series Communication Register Address Mapping

Virtual Address (hex)[1]	Virtual Address Bits Used[2]	Physical Address (binary)[3]	Restrictions
0000 - 001F	0000 0000 000a aaaa	ccc00aaaaa	Ring 0
0020 - 3BFF	N/A	N/A	Reserved[4]
3C00 - 3FFF	0011 11aa aaaa aaaa	aaaaaaaaaa	Ring 0
4000 - 401F	0000 0000 000a aaaa	ccc01aaaaa	Ring 0
4020 - 7FFF	N/A	N/A	Reserved[4]
8000 - 803F	1000 0000 00aa aaaa	ccc1aaaaaa	None
8040 - FFFF	N/A	N/A	Reserved[4]

1 Numbers in this column define a range of virtual communication addresses that are to be mapped to a valid physical communication address.

2 This column shows how virtual address bits are used to form the 10-bit physical address (where *ccc* is the 3-bit CIR and *a* bits are the relevant bits from each virtual address range).

3 this column shows how the bits in the 16-bit virtual address are decoded when the virtual communication address is translated to a physical communication address.

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

2.6.2 Primitive Communication Register Operations

This section describes the primitive operations that are performed on the communication registers. The C200 Series architecture includes instructions that execute these operations directly, as well as some instructions that use more than one of these operations to perform their function. The operations and instructions are the following:

- *put* — Write the communication register regardless of lock bit. The instructions are *put.w Ak,Ceffa* and *put.l Sk,Ceffa*.
- *get* — Read the communication register regardless of lock bit. The instructions are *get.w Ceffa,Ak* and *get.l Ceffa,Sk*.
- *send* — Write the communication register if the lock bit clear, and then set the lock bit. The send operation fails if the lock bit was already set indicating valid data was in the register. A carry status of 1 is returned if the send operation is successful (i.e., the lock bit was initially clear), and 0 if send operation fails (i.e., the lock bit was initially set, meaning data was already sent there). The instructions are *snd.w Ak,Ceffa* and *snd.l Sk,Ceffa*.
- *receive* — Read communication register if the lock bit is set, then clear the lock bit. The receive operation fails if the lock bit was clear which indicates no valid data was in the register to receive. A carry status of 1 is returned if the receive operation is successful (i.e., the lock bit was initially set), and a status 0 is returned if the receive operation fails (i.e., the lock bit was initially clear, meaning the register contained no valid data to receive). The instructions are *rcv.w Ceffa,Ak* and *rcv.l Ceffa,Sk*.

- *lock* — Set lock bit. The lock operation fails if the lock bit was already set. A carry status of 1 is returned if the lock bit is successfully set (i.e., the lock bit was initially clear), and a status of 0 is returned if the lock bit could not be successfully set and the operation fails (i.e., the lock bit was initially set meaning the bit was already locked). The instruction is *lck Ceffa.*
- *unlock* — Clear lock bit. The unlock operation fails if the lock bit was already clear. A carry status of 1 is returned if the unlock operation is successful (i.e., the lock bit was initially set), and a status of 0 is returned if the unlock operation fails (i.e., the lock bit was initially clear meaning the lock bit was already unlocked). The instruction is *ulk Ceffa.*
- *tst* — Read lock bit into the PSW <C> bit. The instruction is *tst Ceffa.*

The status returned for instructions such as *rcv Ceffa,Ak* are loaded into the PSW <C> or PSW <SC> bit and the flow of execution may change in the usual manner, using a branch instruction, as a result of the returned status. A receive/branch pair of instructions is typically used to pass information with semaphoring. For example, one CPU may wait for data from another CPU by executing the following sequence:

```
foo:  rcv.w  Ceffa,Ak
      bra.f  foo
```

Sequence execution will fall out of the loop with a successful receive when another CPU executes a *snd.w Ak,Ceffa* instruction.

An example of a communication register instruction that uses multiple primitive operations is *inc.w Ceffa,Ak.* This instruction increments the communication register at address *Ceffa* by the contents of *Ak* if the communication register is receivable (i.e., the lock bit is set meaning valid data has been sent). This instruction is implemented internally with a receive, add, and send combination, with the add and send primitives only executed if the receive primitive succeeds. Refer to the *CONVEX Architecture Reference* for the entire set of communication register instructions.

2.6.3 Memory Structures Locked with Communication Registers

The locking operations provided in the communication registers are used to synchronize structures located in memory. Since the communication register and memory pipes are disjoint, memory must be synchronized by software before the communication register lock can be manipulated. For example, the lock bits on two communication registers can be used when passing valid data between two or more CPUs operating in a producer-consumer relationship. The following code sequence, appearing to be correct, actually contains a memory synchronization problem, which will be elaborated further in the following text.

```
          Producer                          Consumer

P1:  lck   0x8000                    C1:  ulk   0x8001
     bra.f P1                              bra.f C1
     (store to memory)                     (load from memory)
     lck   0x8001                          ulk   0x8000
```

This code sequence was adapted from Dykstra [Dykstra, 1968]. L(8000) and L(8001) control two critical regions within which the communication occurs. Only one producer or one consumer critical region may be active at a time. When L(8000) is unlocked, the producer is free to store new data. The producer cannot store new data until the consumer has loaded the data previously stored by the producer. When L(8001) is locked, the consumer is free to load new data. The consumer cannot load new data until the producer completes storing the new data. The initial conditions are that L(8000) and L(8001) are unlocked. These conditions force the consumer to wait for the producer to store data to memory.

The producer enters its critical region by locking L(8000), produces (stores) its data, and locks L(8001) to indicate that data has been produced. The consumer, which may have been spinning at `C1` while waiting for the producer to store data to memory, now is free to enter its critical region and consume (load) the data. When the consumer completes loading of the new data, it unlocks L(8000) to signal the producer to produce new data.

However, in the code above, the consumer could see the lock set on the communication register located at L(8001) and load old data from the memory system before the producer's memory store reached the memory system. Likewise, the producer could see the lock clear on the communication register located at L(8000) and store new data to the memory system before the consumer's memory load was performed by the memory system. To remedy this problem, *msync* instructions must be used just after the memory operations. The *msync* instruction waits for the CPU to complete all store operations to memory, and for all data from load operations to arrive from memory. The correct code is shown in the following example:

```
        Producer                          Consumer

P1:  lck   0x8000                 C1:  ulk   0x8001
     bra.f P1                           bra.f C1
     (store to memory)                  (load from memory)
     msync                              msync
     lck   0x8001                       ulk   0x8000
```

If this was a single producer-consumer data transfer, the consumer does not need to synchronize memory (*msync*). When the consumer sees L(8001) set, then the producer's store has reached memory (the producer executed an *msync*) and the consumer can load the data the producer stored. If, however, the producer code is continually sending data to the consumer through the same memory locations whenever the lock bit at L(8000) is clear, the consumer code must perform an *msync* instruction before unlocking L(8000). By unlocking the communication register lock bit at L(8000), the consumer informs the producer that loading from memory is complete, so the producer may store more data in memory. The *msync* instructions after the memory store and load operations ensures the consumer loads valid data, i.e., the consumer does not load stale data or lose data.

2.6.4 Communication Register Modified Bits

Since the CIR defines an operating system process, the communication registers become a part of process state. Therefore, the communication registers are saved and restored by the operating system when the process is rescheduled, i.e., the process relinquishes its CIR (communication register partition).

The communication register hardware provides a structure, called *modified bits*, to accelerate these operations for the communication registers. A set of modified bits similar in function to the memory referenced and modified bits is maintained with the communication register hardware. The hardware uses these bits to save and restore only those communication registers that have been modified. In general, each register does not have a modified bit; instead, a modified bit covers a contiguous subregion of the communication register address space. Any time a communication register or lock bit in the particular region is modified with *put*, *lck*, *ulk*, *snd*, or *rcv* primitive operations, the modified bit corresponding to that region is set.

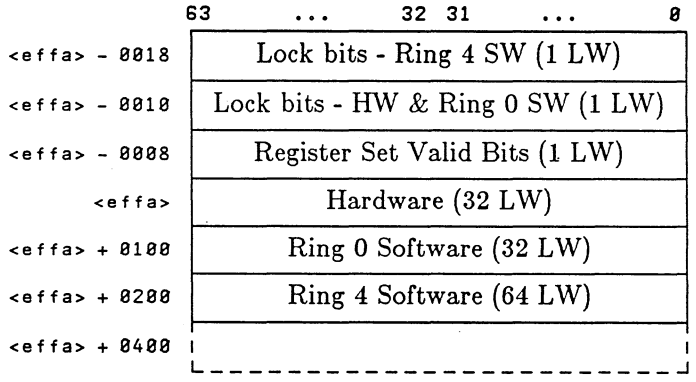
Two privileged instructions, *stcmr*, which copies communication registers for a specified CIR to memory, and *ldcmr*, which loads a specified CIR's communication registers from memory, implement operations for saving and restoring communication registers. The *stcmr* instruction examines these bits to store only the modified region of communication registers, and then stores the modified bits along with the communication registers. However, since a receive operation is used to implement the read operation, all of the lock bits will be cleared after the *stcmr* is complete. The subsequent *ldcmr* instruction that restores the communication registers will only load the registers that were actually saved by the *stcmr*. The memory copy of the modified bits are referred to as the *valid bits* to avoid confusion. The operating system may alter these bits in memory to force a *ldcmr* to restore more or less of the communication registers than the *stcmr* saved.

2.6.5 Communication Register Address Modified Bits

The C200 Series hardware implements 16 modified bits (2 bits per CIR). One bit for Ring 0 and the other bit for Ring 4.

Figure 2–8 shows the memory format of the communication register lock and valid (modified) bits, with respect to the *stcmr/ldcmr* instructions for the C200 Series machines:

Figure 2–8, C200 Series *ldcmr/stcmr* Memory Map



The longword of lock bits corresponding to hardware and Ring 0 software, from the MSB, has eight zeros (where the reserved would be). The longword then contains the lock bits for the hardware registers from virtual address 0000 down to 001F, in bits 55 to 32 of the longword, and the lock bits for Ring 0 software (virtual address 4000 to 401F) in bits 31 to 0 of the longword. The longword of lock bits corresponding to Ring 4 is arranged with the lower communication register address (8000) lock bit in the MSB and the highest communication register address (803F) lock bit in the LSB.

The valid bit longword contains the memory copy of the modified bit for Ring 4 registers in bit 32 and the modified bit for the Ring 0 registers in bit 0. The block of longwords that store the data portion of the communication registers are arranged with numerically lower addressed communication registers in numerically lower memory.

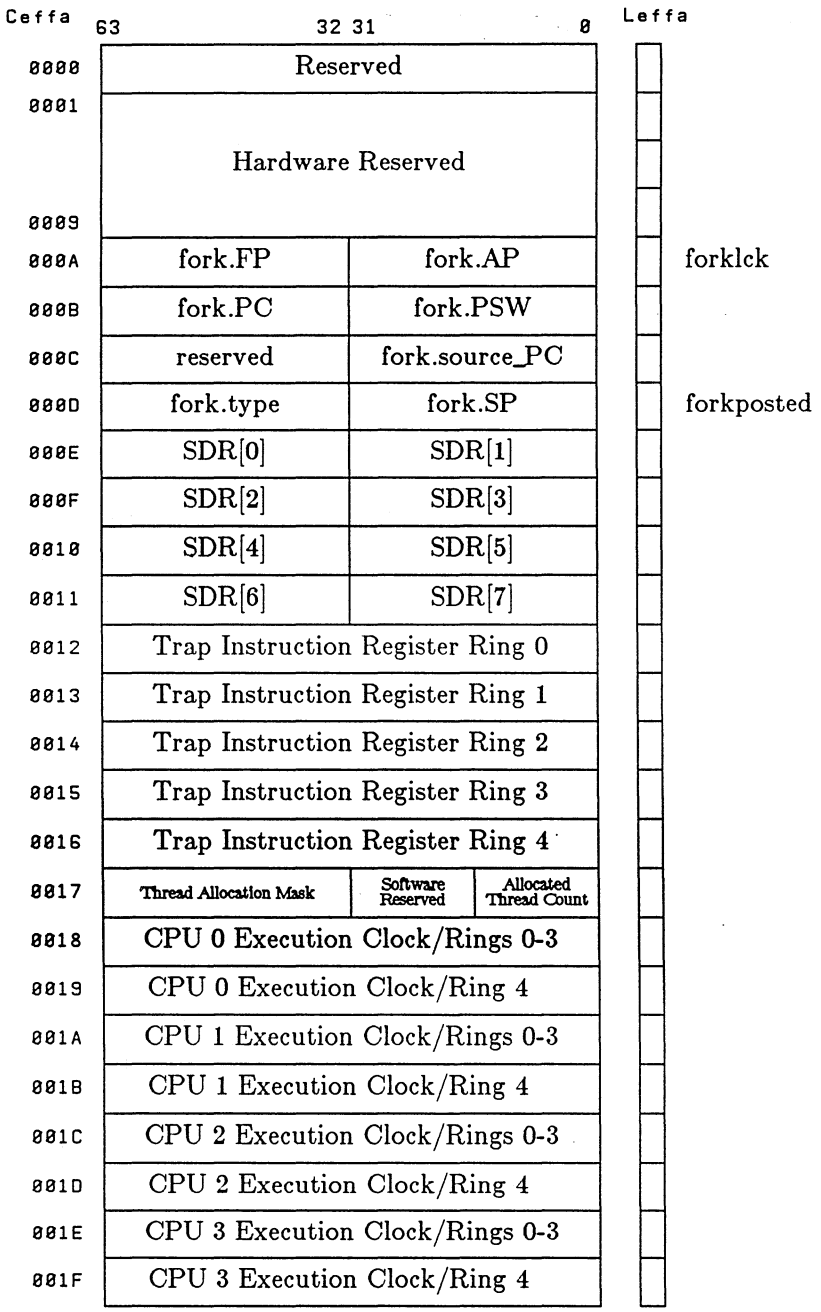
2.6.6 Hardware Communication Registers

As stated in the previous discussion on communication register addressing, half of the Ring 0 communication hardware. These registers are used by hardware and Ring 0 software to provide the multithread execution functionality to be described later. The registers are defined in the *CONVEX Architecture Reference*, and that discussion is reproduced here with some additional implementation specific information added. In the figures, the notation `Ceffa` denotes a communication register address **xxxx**, and the notation `Leffa` refers to its associated lock bit.

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

Figure 2–9 shows the hardware communication register address mapping for C200 Series machines:

Figure 2–9, C200 Series Hardware Communication Registers



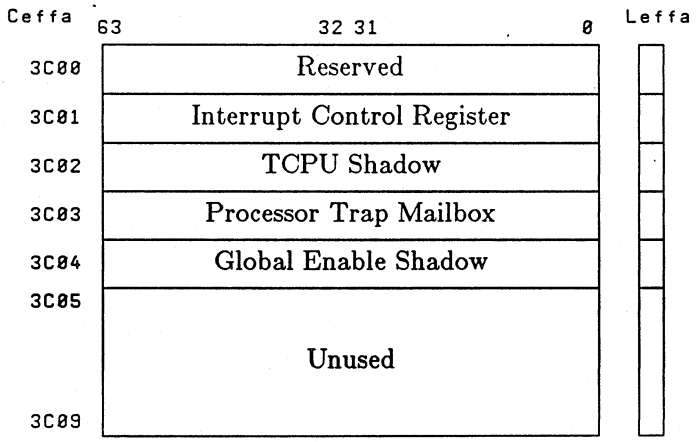
2.6.6.1 Hardware Reserved Communication Registers

The communication virtual address space from 0000 to 0009 is reserved for hardware. For the C200 Series hardware, the hardware reserved registers are used primarily as "shadow" copies of Complex-wide write-only registers. For example, the Target CPU (TCPU) Register, is a register in the interrupt logic whose purpose will be explained in the "Interrupt" section. There is a single TCPU in the Complex, located on the CPX board in the C210/C220. Each CPU has a path to write this register, but there is no returning read path.

However, in addition to the instruction to write this register (*mov Sk,TCPU*), there is also an instruction to read the register (*mov TCPU,Sk*).

The read function is implemented in the following manner. When the TCPU is written with *mov Sk,TCPU*, the register Sk is also written to one of the hardware reserved communication registers using the communication physical addressing mechanism; i.e., this shadow copy value is independent of the CIR. A *mov TCPU,Sk* instruction writes Sk from the shadow copy in the hardware reserved communication registers. The C200 Series hardware allocates these registers in the communication address range indexed by CIR = 0 as shown in the following figure. Recall from the discussion of addressing communication virtual address range 3C00 – 3C09 places the registers at the communication address range 0000 – 0009 that is indexed by CIR = 0.

Figure 2–10, C200 Series Hardware Reserved Communication Registers



The interrupt control register contains the interrupt mode and interrupt CIR. The interrupt mode portion is a shadow of a write-only hardware register. The interrupt CIR is not a shadow; it is entirely implemented in communication registers. The TCPU shadow and global enable shadow are copies of write-only hardware registers. The purpose of all these is described in the "Interrupt" section. The processor trap mailbox is used by the hardware to communicate between CPUs for instructions such as *patu* where one CPU executing the instruction must cause other CPUs to perform an action. This is implemented with a mechanism called a processor trap, explained in the "Page 0, Traps, and Exceptions" section.

2.6.6.2 Fork Event Communication Registers

The fork event registers are hardware communication registers used for holding the information required to create an independent thread of execution. Thread creation is described in more detail in the "Multiprocessing" section. Basically, one process executing on a CPU requests the addition of other CPUs by storing information in the fork event registers. An idle CPU then creates a thread and executes on behalf of the process by loading this information from the fork event registers into its own state registers, such as the program counter. The fork event registers are defined as:

- **fork.FP**—The initial frame pointer for the thread
- **fork.AP**—The initial argument pointer for the thread
- **fork.PC**—The program counter to begin execution of the thread
- **fork.PSW**—The initial PSW for the thread
- **fork.type**—A parameter passed from posting to acceptance of the fork. Refer to the "Multiprocessing" section for more details. For the C200 Series hardware, the possible hexadecimal values are:
  - *pforked*—0000 0000
  - *spawned*—0000 000A
  - *stopped*—0000 000B
- **fork.SP**—The initial stack pointer for the thread
- **fork.source_PC**—The PC for the thread posting the fork.

The lock bits on the fork event registers, called forklck and forkposted are used to convey the state of the fork during its transitions from cleared to posted to taken and to cleared. Refer to the "Multiprocessing" section for more detail.

2.6.6.3 Segment Descriptor Registers

The Segment Descriptor Registers (SDRs) define the extent of the virtual address space associated with a process. Locating The SDRs in the communication registers causes the entire address translation for a CPU to change whenever the CIR index is changed. Lock bits on these registers are ignored.

2.6.6.4 Trap Instruction Registers

There is one Trap Instruction Register (TIR) for each ring and CPU combination. The TIR is a 64-bit register used by the *trap* and *pbkpt* instruction. These instructions can set specific bits in this register to cause a process wide system exception. The TIR is primarily used for asynchronously trapping thread breakpoints and also for thread scheduling. The TIR is described in more detail in the "Page 0, Traps, and Exceptions" section.

2.6.6.5 Thread Allocation Mask and Count

The thread allocation mask is a 32-bit mask. Each bit position in the thread allocation mask represents a unique thread ID. This mask allows a process to create up to 32 unique threads. In order for a thread to be created (CPU transition from idle to allocated), a unique thread ID is generated by atomically clearing a single bit in the thread allocation mask. The CPU Thread ID (TID) register is then loaded with the allocated thread ID to identify the new thread throughout its existence. When a CPU transitions from allocated to idle, it automatically sets the bit associated with the CPUs TID register in the thread allocation mask.

The allocated thread count is a 16-bit integer which is a count of the number of thread IDs allocated from the thread allocation mask. When a thread is created, the thread count is incremented, and when a thread terminates, the thread count is decremented. Refer to the "Multiprocessing" section for more information concerning the thread allocation mask and count.

2.6.6.6 CPU Execution Clock Registers

The CPU execution clocks maintain a 64-bit microsecond counter per CPU, which provides the exact execution time per CPU within each ring. These timers are maintained by microcode in a scheme described in the "Timers" section.

2.7 Multithreaded Execution (Forking/ASAP)

This section explains the mechanisms used by the system to implement multithreaded execution, called forking or Automatic Self-Allocating Processors (ASAP) Scheduling.

A thread is any single instruction stream executing within a process. Multithreaded execution implies that more than one CPU is executing on behalf of the same process. The goal is to "divide and conquer" the workload. For example, a process that takes 10 seconds of CPU time will take 10 seconds of wall clock time on a single CPU, assuming is was allowed to run without intervention. Under ideal conditions, a process will execute in 5 seconds of wall clock time if two CPUs are available and the work is equally divided. Another goal is that programs written to exploit multithreading must be able to run on any number of processors (including one) with no software modification.

The mechanisms designed for this are closely associated with operating system. The hardware allows the operating system to observe and maintain some control of the thread extent of a process, since the operating system must also schedule threads to perform process multiplexing. The C200 Series CPU control instructions can create and terminate threads without operating system involvement, and vice versa. Therefore, the multithreading architecture is a software/hardware interface instead of a collection of instructions provided for the operating system to use. Both sides of the interface have rules that must be observed to insure that parallel execution functions correctly.

First it is necessary to define some terminology. The act of allocating another CPU to initiate an additional thread on behalf of a process is called forking. The fork event allows the user to post the need for a CPU to execute on behalf of a process (create a thread), clear the need for a CPU, or force the current CPU back into the idle state (terminate a thread).

The fork event has two possible states: posted or cleared. Posted means there is a current need for another processor to begin a thread. Cleared means there is no pending work to be done. A CPU posts a fork when more CPUs can assist with the parallel execution of a process. A posted fork means that a CPU requests, not demands, assistance from other CPUs. If there are no available CPUs, the posting CPU does not wait until another CPU becomes available; the CPU just continues with its thread of execution. This mechanism allows a parallel process to execute as a single thread if only one CPU is available.

A CPU can post two types of forks. The first type is a request for a single CPU to initiate a thread and is posted by executing a *pfork effa,Ak* instruction. The second type is a request for all available CPUs to initiate threads and is posted by executing a *spawn effa,Ak* instruction. These instructions load a group of communication registers (known as the fork event registers) with enough process state to start a thread. This state consists of a PC to execute from, an initial value of PSW, and stack, frame, and argument pointers to define local memory structures.

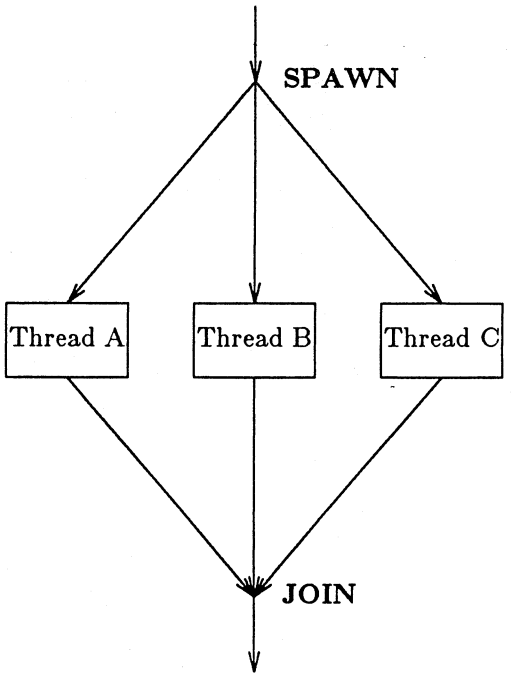
Communication registers addressing is based on a CIR index, so the fork is posted relative to a particular process. Idle CPUs scan through the fork event registers in each CIR, as a function of the CPU idle loop, looking for a posted fork. If a posted fork is found, the idle CPU binds itself to that CIR by loading the state in that CIR's fork event registers into its own CPU registers. If the fork was posted with *pfork*, the fork is cleared. If the fork was posted with *spawn*, it is left posted in the fork event registers for other available CPUs to take. The *cfork* instruction will explicitly clear a fork posted by a *pfork* or a *spawn* instruction. However, proper synchronization between threads may not be maintained if a fork is explicitly cleared with *cfork* and not *join*.

At the end of the thread execution, each CPU may terminate its thread (relinquish and deallocate the CPU). There are three instructions to do this — *wfork*, *join*, and the privileged *idle Sk* instruction. The *wfork* instruction terminates a thread begun with the acceptance of a fork posted through *pfork*, i.e., a single thread of execution. The CPU is returned to the idle state, where it looks for more posted forks in other CIRs. Forks posted through *spawn* should be terminated with *join*. If the processor is not the last thread to reach the *join*, the CPU is returned to the idle state. If it is the last thread, execution continues at the instruction following the *join*. Therefore, the process will continue executing as a single thread after the *join* instruction executes. The *idle* instruction is used by the operating system to reschedule the CPU, sending it to the idle state to look for forks beginning in CIR Sk. The action of these instructions is described in greater detail later.

The forking instructions provide for two types of parallel processing. The *spawn* and *join* instructions, when used together, implement symmetric parallel processing. A parallel process is symmetric when a thread executes a *spawn* instruction, which creates a thread for each available CPU. All threads in a symmetric process execute the same instruction stream. The compiler typically uses symmetric parallel processing to parallelize loops. For example, a loop that runs 10 iterations can be spread across two CPUs using *spawn* and *join* to use each CPU to compute 5 of the iterations. The loop iteration count would have to be shared between the two CPUs. A detailed example is given later.

Figure 2–11 shows an example of symmetric parallel processing:

Figure 2–11, Symmetric Parallel Processing



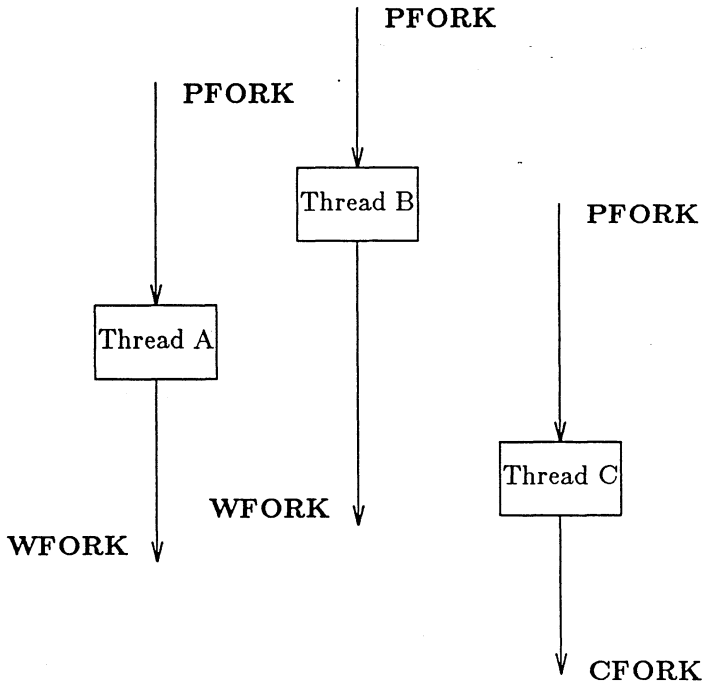
In the previous figure, a single process transitions from single threaded to parallel and back again. The initial single thread posts a need for more threads by executing a *spawn*. Each thread which enters the process in response to the *spawn* leaves the fork posted. When the first thread completes its job, it executes the *join* which marks the fork event registers so that no additional threads will accept the fork, then deallocates the CPU. When the last thread executes the *join* instruction, it clears the fork and continues on as a single threaded process.

The second form of multi-processing, called asymmetric parallel processing, occurs when multiple threads within a process execute different functions by creating a single additional thread of execution ananlogous to the *fork* system call construct under ConvexOS. Asymmetric processing differs from multithreaded execution in symmetric processing in that the posting thread usually requires another thread to accept the fork to perform the specific task, and then communicates with the created task. The compiler does not use asymmetric parallel processing; however, asymmetric parallel processing is used within the operating system.

Asymmetric parallel processing uses the *pfork* instruction to post a fork for a single CPU to execute, usually at another instruction stream that the posting CPU does *not* execute. The multiple threads within the process therefore execute different functions. The accepting CPU terminates the additional single thread with the *wfork* instruction.

Figure 2–12 shows a three threaded asymmetric process:

Figure 2–12, Asymmetric Parallel Processing



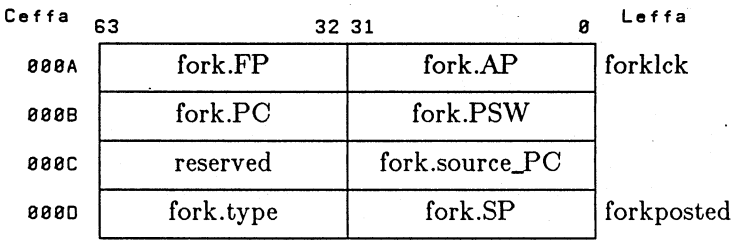
In the previous figure, thread B posts the need for an independent thread to execute concurrently. Thread A is started, and notifies B that it is executing. Thread B completes execution and terminates. Thread A posts the need for a new thread C to perform another task. Thread C starts, posts a fork, and notifies thread A that it is executing. Thread A completes execution and terminates in the same manner as thread B. Thread C determines all other threads have completed execution and examines the fork event registers to check the status of the fork that it posted. Thread C clears the need for another thread by clearing the fork and continues on as a single threaded process.

2.7.1 Forking Operations

This section will explain the forking instructions (*pfork, spawn, cfork, join, wfork, idle*) and the microcode idle loop in greater detail, providing verbal descriptions to augment the C-like pseudocode given in the *CONVEX Architecture Reference*. All of these instructions perform basic operations on the fork event registers.

Figure 2–13 shows the format of the fork event registers:

Figure 2–13, Fork Event Registers



The lock bits *forklck* and *forkposted* are used to semaphore the fork event registers between CPUs that may be attempting to post or accept forks at the same time. These two lock bits operate as a two-ended semaphore.

CPUs attempting to post forks must first successfully lock the "top", or *forklck*, then write fork state into the communication registers, then lock the "bottom", or *forkposted*. The locking operations performed by the *pfork* and *spawn* instructions are a *snd* operation to *forklck*, and if successful, a *snd* operation to *forkposted*. CPUs attempting to accept forks must first unlock the "bottom" (*forkposted*), read all the fork state from the communication registers, then unlock the "top" (*forklck*). Forks are accepted primarily by the microcode idle loop and in some cases by the *wforkR and idle* instructions, described shortly. The locking operations performed during fork acceptance are a *rcv* from *forkposted*, and if successful, a *rcv* from *forklck*. This protocol makes the fork event registers operate somewhat like a queue, i.e., They are written from the *forklck* end and read from the *forkposted* end. The four possible states of the two lock bits are:

- *forklck* = 0, *forkposted* = 0 — No fork posted, ready for forks to be posted
- *forklck* = 1, *forkposted* = 0 — In transition one of two ways:
  - CPU beginning to post a fork; any other CPU attempting to post will fail, any other CPU attempting to receive will fail until the fork is completely posted.
  - CPU beginning to accept a fork; any other CPU attempting to post will fail until the fork is completely accepted, any other CPU attempting to receive will fail.
- *forklck* = 1, *forkposted* = 1 — Fork posted ready for acceptance; any other CPU attempting to post will fail, any other CPU attempting to receive will succeed.
- *forklck* = 0, *forkposted* = 1 — Undefined state

Often the locking operation is combined with a conditional write (using the *snd* operation) of the fork event registers. Similarly, the unlocking operation is combined with a read (using the *rcv* operation).

The thread allocation register that contains the allocated thread count and thread allocation mask is manipulated as part of fork acceptance. The thread count is incremented each time a thread is created by fork acceptance, and decremented each time a thread is terminated. The thread count may range in value from 0 to 31. The thread count allows the microcode to determine whether the thread is the last in the process, which directly affects the operation of the *wfork* and *join* instructions. It also allows the operating system to determine how many threads are currently in existence.

The thread mask is a 32-bit mask with each bit corresponding to one of the possible threads in a process. The least significant bit corresponds to thread 0, the next to thread 1, and so on up to thread 31. If a bit is set in the thread mask, it means that thread does not currently exist and so it may be allocated by the microcode when it wants to create a thread. If the bit is clear, the thread is already allocated so microcode may not allocate it again.

The thread mask and count are governed by a *snd/rcv* locking protocol. This locking protocol forces the microcode or the operating system wishes to successfully receive the registers before manipulation and then send them back when the manipulation is complete as part of the process to examine or modify these registers.

The microcode performs the following actions when attemping to create a thread:

1. The mask/count is received.
2. The mask is searched for the least significant set bit, using a trailing zero count operation.
3. If there are no set bits, there are no threads available for creation. The remaining steps are skipped. If a set bit is found, it is cleared, marking it allocated.
4. The bit index of the set bit is written to the CPU Thread ID (TID) register.
5. The thread count is incremented.
6. The thread mask and thread count is sent back to the communication registers.

When a thread is terminated, the following occurs:

1. The mask/count is received.
2. The contents of the TID register are used as a bit index to set a bit in the thread mask.
3. The thread count is decremented.
4. The thread mask nad thread count is sent back to the communication registers.

Therefore, the operating system can block any further creation of threads in a process by receiving the thread mask/count in that CIR, clear the entire thread mask, and send it back. Since the thread mask has been cleared, no more threads can be allocatable.

Now that the communication registers dealing with forking have been described, each of the instructions that implement forking can be described in individual detail.

2.7.1.1 *pfork <effa>,Ak*

The *pfork <effa>,Ak* instruction posts a fork for a single CPU to accept if available. The new thread begins with a PC of <effa>, using a stack pointer specified in Ak, and inheriting the PSW, FP, and AP from the "parent" thread. First, the current frame and argument pointer are assembled into a longword and sent to the *fork.FP/fork.AP* fork event register, after attempting to lock *forklck*. If this send operation fails, a fork has already been posted, so PSW <C> is cleared indicating failure and the instruction is complete. This failure status allows the thread to determine that it was unable to post a fork.

If the *snd* succeeds, posting continues by assembling the <effa> from the instruction and the current PSW into a longword and *putting* it into *fork.PC/fork.PSW*. Next, the current program counter is *put* into *fork.source_PC*. This action sets the ring bits of the PC when the fork is accepted. Finally, the constant PFORKED is concatenated with Ak for assembly into a longword and sent to *fork.type/fork.SP*. The return status on this *snd* is not checked since it always succeeds. The success status from the first *snd* is returned in PSW <C> signaling that the *pfork* successfully posted the fork.

2.7.1.2 *spawn <effa>,Ak*

The *spawn <effa>,Ak* instruction works identically to the *pfork <effa>,Ak* instruction except the constant SPAWNED is put in *fork.type*. This lets accepting CPUs know the fork was spawned and is intended for multiple CPUs if any are available.

2.7.1.3 *cfork*

The *cfork* instruction will explicitly clear a posted fork, i.e., remove it from the fork event registers without accepting it and creating a new thread. This instruction first receives the *forkposted* register. If this *rcv* operation fails, there is no posted fork to clear so the *cfork* instruction returns a failure status of zero in PSW <C>. If the *rcv* succeeds, this success status is returned and the forklck is also received, making the status of the fork event register lock bits *forklck* = 0, *forkposted* = 0.

2.7.1.4 *wfork*

The *wfork* instruction terminates the current thread of execution and possibly return the CPU to the idle state. First, the *wfork* attempts to receive the thread mask/count. If the *rcv* operation fails, the *wfork* cannot continue since the mask/count is required to deallocate the current thread. The *wfork* instruction does not wait it, because interrupts would not be able to be taken on the CPU, since interrupts are only delivered to the CPU at instruction dispatch boundaries. Therefore, the microcode restarts the *wfork* instruction until the *rcv* operation succeeds.

When the *rcv* eventually succeeds, the thread count is checked for a value of 1. If so, this means the terminating thread is the last thread in the process, so the current CIR is checked to see if a fork is posted. If there is a posted fork of type PFORKED, it is taken. Although, strictly speaking, a thread has been deallocated and reallocated, but the microcode will still accept the fork in the current CIR as the current TID. The specifics of fork acceptance are discussed in the section covering the microcode idle loop.

If a posted fork of type SPAWNED or STOPPED is found, the thread was started with a *spawn*, and should have been terminated with a *join*. Since *spawn* and *pfork* instructions should not be mixed in the same process without proper synchronization, a deadlock is reported to the operating system through a system exception. Deadlocks are described in more detail in a later section. If there is no fork posted at all, the process has ended erroneously, so a deadlock trap is invoked.

If the thread count > 1, it means an added thread is terminating correctly. The thread count is decremented and the bit in the thread mask corresponding to the current TID is set, indicating the terminating thread is now available for allocation. The thread mask/count is sent back to the communication registers and the CPU enters the microcode idle loop, described shortly.

2.7.1.5 *join*

The *join* instruction reduces the process to a single thread of execution. All threads in the process that have accepted the spawned fork reach a *join* at the termination of their execution. Each CPU terminates its current thread of execution and either returns to the idle state or continues execution after the *join* as a single threaded process. The spawn/join mechanism may be viewed as a race from *spawn* to *join*, where the first N-1 threads to reach the *join* terminate, and the *N*th thread continues executing instructions after the *join* instruction. Once one thread has joined, any idle CPUs do not accept the fork, since the region of code between the *spawn* and *join* instructions has been completed by one thread, and should soon be completed by all threads.

First, the *join* instruction attempts to receive the thread mask/count. If the *rcv* fails, the instruction is restarted as described in the *wfork* section. When the *rcv* eventually succeeds, the CPU waits for all stores in the current CPU to reach the memory system by performing an *msync* operation. This is to insure that the single thread continuing after the join will see all operands stored by all "child" threads if the "parent" thread loads these operands. Next, the constant value STOPPED is sent to the *fork.type* fork event register. This value signals other threads within the process that at least one thread has reached the *join* instruction, meaning the process is reducing to a single thread which prevents other CPUs from accepting the fork.

Next, the thread count is checked for a value of 1. If the thread count > 1, then a "child" thread is is correctly terminating. The thread count is decremented and the bit in the thread mask corresponding to the current TID is set, indicating an allocatable thread. The thread mask/count is sent back to the communication registers and the CPU enters the microcode idle loop, described shortly. If the thread count is one, it means this thread was the last to reach the *join* so the thread mask/count is sent back to the communication registers and the thread continues after the *join* instruction.

2.7.1.6 *idle Sk*

The *idle Sk* instruction allows operating system to send a CPU to the idle state. As discussed in the next subsection, the idle state is a microcode loop that searches all CIRs for posted forks. The Sk parameter of the *idle* instruction specifies which CIR is searched first. The *idle* instruction is different than *wfork* or *join* because it does not terminate and deallocate the current thread before going idle, the thread is just left behind.

First, the CIR is switched to the value specified in Sk. Next, the *idle* instruction attempts to receive the thread mask and thread count. If the *rcv* operation fails, the idle loop is entered. The idle loop is described in detail in the next section. Next, the thread mask is searched for the leftmost set bit, indicating an allocatable thread. If no bits are set, the thread mask and thread count are sent back to the communication registers and the idle loop is entered.

If an allocatable thread can be found, the CPU attempts to accept a fork by first receiving *forkposted*. The specifics of fork acceptance are discussed in the section covering the microcode idle loop. If a fork of type STOPPED is found, then the process in that CIR is attempting to join so the fork is ignored, the thread mask/count is sent back, and the idle loop is entered. If a fork of type SPAWNED or PFORKED is found, it is taken in the new thread. The updated thread mask and thread count (thread allocated and count incremented) are sent back to the communication registers and execution begins at the PC of the fork.

2.7.1.7 Idle CPU Allocation

The idle state of a CPU consists of a microcode sequence referred to as the *idle loop*. This section describes the algorithms used by the C2 series hardware and microcode (CPU idle loop) implementing the C200 Series architecture to allocate idle CPUs to a process.

> **NOTE**
>
> These algorithms are specific to the C2 series hardware implementation and may not represent all implementations of the CONVEX C200 Series architecture.

The idle loop is the common end of the *wfork*, *join*, and *idle* instructions. The *wfork* and *join* instructions search the current CIR for a posted fork before idling. The *idle* instruction searches the CIR specified by Sk before idling, and does not deallocate the thread the *idle* instruction executed from.

The algorithm for the CPU microcode idle loop is shown in pseudocode in Figure 2–14:

Figure 2–14, CPU Idle Loop — C200 Series

```
(update CPU timer for current CPU and ring);
(purge instruction cache);
if (wfork || join) {
        threadcount = threadcount - 1;
    (set threadmask<TID>);
        snd(threadcount/mask);
}
CIR = ICIR;                                          /* enter the interrupt CIR */
for (;;) {        /* loop through CIRs round-robin using physical addressing */
    if (!rcv(threadcount)) {
        continue;
      }
    if (threadmask == 0) {                          /* no threads available */
        snd(threadcount);
        continue;
      }
    if (!rcv(forkposted)) {                                    /* no fork */
        snd(threadcount);
        continue;
    } else {                /* fork.type, fork.SP loaded in rcv(forkposted) */
        if (fork.type == STOPPED) {                              /* joining */
            snd(forkposted);                                /* repost fork */
            snd(threadcount);
            continue;
        } else {
            TID = tzc(threadmask);
            threadcount = threadcount + 1;
            threadmask<TID> = 0;
            snd(threadcount/mask);
            PC = get(fork.source_PC);                  /* establish ring */
            PC/PSW = get(fork.PC/PSW)                         /* in ring */
            if (fork.type == PFORKED) {
                FP/AP = rcv(forklck);
            } else {                                  /* spawned - repost */
                FP/AP = get(forklck);
                lck(forkposted);
            }
            if ((PSW<TTC> == 1) || (PSW<TIT> == 1)) {
                (trace trap);
            } else {          /* jump to fork.PC in ring of fork.source_PC */
                break;
            }
          }
        }
      }
  }
```

The idle loop algorithm loops sequentially through all CIRs in search of a posted fork, providing the idle CPU useful work to undertake. There are four conditions that must be met in order for a fork to be taken.

The current C200 Series implementation checks these conditions in the following order:

- The thread mask/count communication register must be receivable; i.e., have its lock bit set.
- The thread mask must be nonzero, indicating that threads are available for allocation.
- There must be a fork posted, i.e., *forkposted* must be locked.
- The posted fork must not be of type STOPPED.

If any of these conditions are not met, the CIR is skipped and the next CIR is searched. If these conditions are met, the fork is accepted in the following manner. First, the successful *rcv* of *forkposted* not only shows the availability of a fork but also fetches the *fork.type* and *fork.SP*. The CPU's address register A0 is loaded with *fork.SP*, and *fork.type* is checked to insure it it not of fork type STOPPED. If *fork.type* is of fork type STOPPED, then the process running in the candidate CIR is attempting to *join* to a single threaded state, so the fork is ignored.

Assuming the fork is not ignored, the *fork.source_PC* is fetched with a *get* and loaded into the program counter (PC). This sets the ring bits for the new thread of execution. Next, the *fork.PC* and *fork.PSW* are fetched with a *get* operation and loaded into the PC and PSW. The PC is loaded the second time with the least significant 29 bits only, i.e., the ring bits are not loaded.

To understand why PC is loaded twice, consider the case where a *pfork 0,sp* is executed in Ring 4. If the *fork.PC* of 0 written to the fork event registers by the posting thread were simply loaded by the accepting CPU, an illegal entry into Ring 0 would be implied. To avoid an illegal Ring 0 entry, the PC of the posting thread is written to the *fork.source_PC* to set the ring bits (establish a current ring of execution) for the accepting CPU. This makes the posting and acceptance of forks consistent with the *jmp* instructions.

If the fork was of type SPAWNED, the accepting CPU's frame and argument pointers are loaded with a *get* from *fork.FP* and *fork.AP*. Finally, *forkposted* is locked with a *lck* operation, leaving the fork posted. If the fork was of type PFORKED, the accepting CPU's frame and argument pointers are loaded with a *rcv* of *forklck* which also reads *fork.FP* and *fork.AP*. This also clears the fork.

Finally, a thread is allocated as described earlier, and the thread mask/count is sent back to the communication registers. The new thread begins execution at *fork.PC* in the ring of *fork.source_PC*.

Fork acceptance is one of two events that can make a CPU leave the idle state. The second event is an I/O interrupt. An idle CPU is always able to respond to interrupts. Interrupt processing is fully described in Chapter 6, "Exceptions and Interrupts."

The CPU microcode idle loop employs an equitable "round-robin" scheduling algorithm. CPUs take forks from different processes (CIRs) by binding to the interrupt service process context (setting the CPU's CIR index equal to the interrupt CIR index) and looping through the other communication register sets using physical communication addressing. If the idle CPU always began searching for forks in the fork event registers at CIR = 0 and progressed sequentially through to CIR = 7, the lower CIR index values would be treated more favorably. To circumvent this inequity, each time a CPU accepts a fork, it saves the CIR index from which it accepted the fork. The next time that CPU goes idle, it begins searching at the CIR index following the one it last found a fork in, and starts executing at the next sequential CIR index.

The CPU checks for interrupts after one complete pass of the eight CIRs. If there are no pending interrupts, another pass of the CIRs begins. The interrupt CIR (ICIR) must be entered during the idle loop to give the idle CPU enough context from which to take an interrupt or *trap* instruction trap.

2.7.1.8 CPU Deadlock Detection

The C2 series hardware implementing the C200 Series architecture is capable of detecting when the currently executing threads within a process have reached a deadlock condition. A deadlock occurs when all the currently executing threads of a process are doing a "synchronization" instruction followed by a branch back to that instruction. Synchronization instructions are any instructions which attempt to change the value of a lock and return status on the success or failure of the lock/unlock operation. Examples of such instructions are the *tas*, *snd*, *rcv*, and *inc* instructions.

Deadlocks are defined as system exceptions and are passed through page 0 of Ring 0 to the process deadlock handler. The system can then determine if any other threads within the process can be run and schedule them accordingly. Refer to Chapter 6, "Exceptions and Interrupts," for more information on process deadlock exceptions.

2.7.1.9 Process deadlock

A *process deadlock* usually occurs when all threads of an executing process are in a synchronization instruction sequence. When any of the deadlock detection instructions are followed by a branch back to the same instruction, these instructions have the potential of triggering a process deadlock.

The group of instructions listed in Table 2–2 are classified as *synchronization* or *deadlock detection* instructions:

Table 2–2, Deadlock Detection Instructions

Instruction	Description
snd	Send a value to a communication register
sndr	Send a value to a synchronized resource structure in memory
rcv	Receive a value to a communication register
rcvr	Receive a value to a synchronized resource structure in memory
inc	Increment a communication register
incr	Increment the data field of a resource structure
lck	Lock a communication register
ulk	Unlock a communication register
mat	Compare an address register with a communication register
tas	Test and set a memory byte
tac	Test and clear a memory byte
pshr	Push an address register onto a resource structure
popr	Pop an address register off of a resource structure

Synchronization instructions are any instructions which attempt to set the value of a lock and return status on the success or failure of the lock. All data representations of these basic instructions are implemented in the deadlock detection instructions, i.e., *snd* includes *snd.w* and *snd.l*.

NOTE

These instructions all perform some sort of semaphore or synchronization operation and return status in PSW<C> or PSW<SC>.

Whenever a CPU executes one of these instructions and ***immediately*** follows it with a branch back to the same instruction (i.e., the same opcode at the same PC where the branch displacement must be the negative of the size of the synchronization instruction), the thread is deadlocked. If all threads currently executing in a process are deadlocked, then the entire process is deadlocked.

When a deadlocked process is detected, each thread within the process immediately enters the Ring 0 process deadlock handler pointed to by location 0000 0010 of Ring 0 page 0. The process deadlock handler schedules other threads within the process to resolve the deadlock condition. An example code sequence would be the following:

```
foo: rcv.w  0x8000,a2
     bra.f  foo
```

If the *rcv* instruction fails in this code sequence, and returns a PSW<C> of 0, a backward branch is taken. The deadlock handler would be dispatched instead of retrying the *rcv* operation. Refer to Chapter 6, "Exceptions and Interrupts," for more information about the process deadlock exception.

The concept of deadlock also extends to certain cases of thread termination and fork acceptance. For example, the last thread in a process terminates or a thread that should have executed a *join* executes a *wfork* instead. The process deadlock mechanism is used with a separate qualifier code to signal these cases to the operating system.

Specifically, if the last thread in a process executes a *wfork* instruction (i.e., the entering thread count is 1) and a fork is not posted in the CIR, a *last thread termination* deadlock is signaled. However, if a fork is found posted by the last thread executing a *wfork* instruction, and the fork is a STOPPED or SPAWNED type, then the process executed a *spawn/join* instruction pair mixed with a *wfork* instruction without proper synchronization.

The last thread of a process should never execute a *wfork* since the process cannot continue. The acceptance of a fork in the current CIR is provided as a last opportunity to avoid deadlock, but if the fork is of the wrong type, it still causes deadlock.

2.8 Page 0/Exceptions/Traps

The new features included by the C200 Series architecture necessitated some changes in the operating system interface of the architecture. These are divided into three major areas:

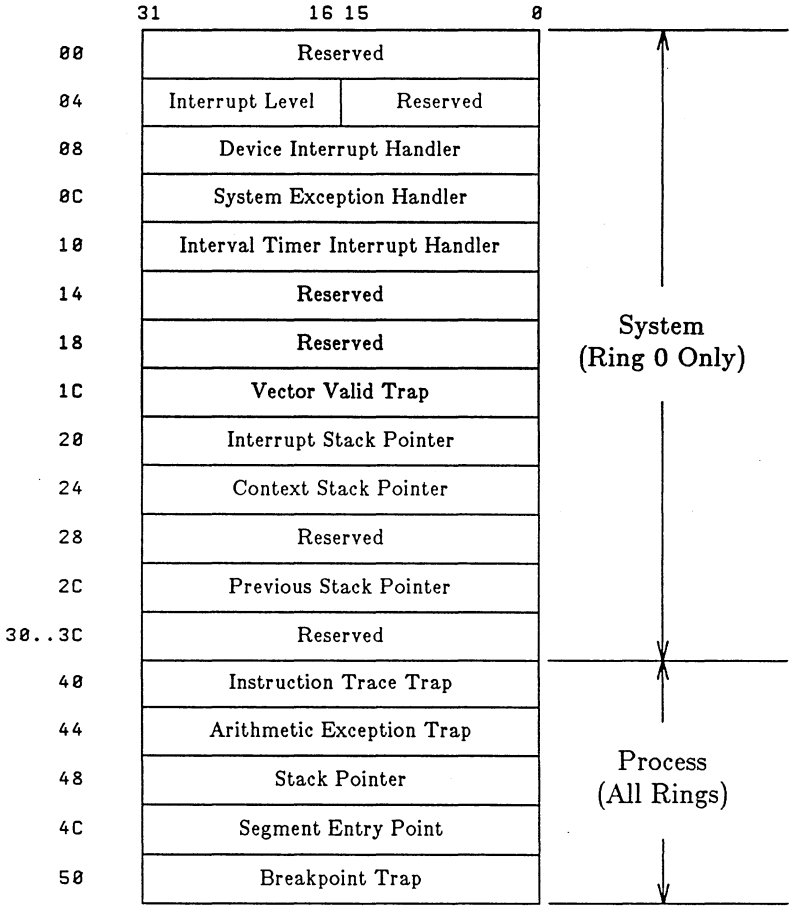
- Reserved memory formats, usually called *page 0*
- PSW bits and traps
- Exception handling procedures

This section will describe these changes and additions. The reader is assumed to be familiar with the C100 Series architecture and therefore similar items are not described here.

2.8.1 Page 0

Figure 2–15 shows the C200 Series architecture page 0 format for Ring 0 (system) and Ring 4 (process). System page 0 exists only for Ring 0, while process page 0 exists for all 5 rings.

Figure 2–15, Page 0 Virtual Memory Organization



The first major difference is regarding interrupt handling, which is discussed in detail in a later section. Basically, there is a single interrupt handler entry point, rather than have a separate handler entry for the interval timer. Software must examine the interrupt channel information (provided in a5) to determine the source of the interrupt. In addition, the Interrupt Level, Interrupt Stack Pointer, and Previous Stack Pointer have been moved to another reserved state area called the Interrupt Context Block. Page 0 now contains a pointer to this block. The format of this block is described in the "Interrupts"section.

The next major addition is the inclusion of an I/O Register Pointer which gives the virtual address of the base of a page in I/O space that includes the interval timer registers and the new Time of Century Clock (TOC). The microcode for the *mov TOC,Sk* instruction uses this pointer to access the TOC when it is read. The operating system uses this pointer when it initializes the TOC and references the interval timer.

There are a few new types of exceptions defined in the C200 Series architecture. Most exceptions enter Ring 0 through the system exception handler. Codes were added for the new types of exceptions. One new exception enters through a different path—the Process Deadlock Handler. Deadlocks are described in detail shortly.

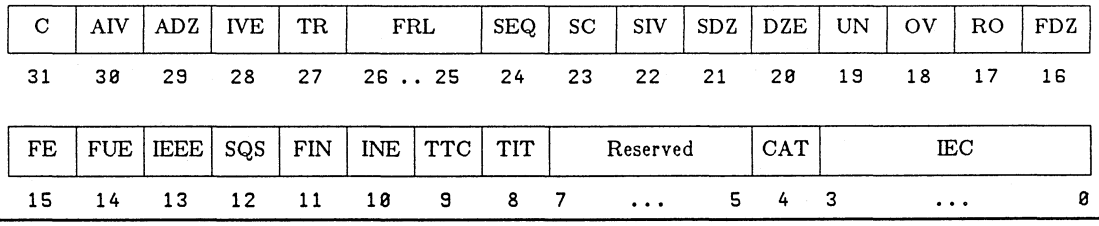
When a process enters Ring 0 (called a ring crossing) the state of that process is pushed onto a stack in Ring 0. This state is either an extended frame for interrupts, system calls, and system exceptions, or a context frame for page faults. The C100 Series architecture has a location in page 0 that contains the pointer to this stack. The CPU's stack pointer is loaded from page 0 during a ring crossing. In case a fault occurs in Ring 0, a context frame is pushed on a stack that is always available and is pointed to by a separate context stack pointer.

2.8.2 New PSW Bits and Traps

The C200 Series architecture adds some new types of traps which require additions to the Processor Status Word (PSW).

The format of the C200 Series PSW is shown in Figure 2–16:

Figure 2–16, C200 Series Processor Status Word (PSW)



The PSW bits that are new to the C200 Series architecture (i.e., not included in the C100 Series architecture) are:

- **Bit <12> — Sequential Store Enable (SQS)**
  If this bit is clear, stores to memory may occur in a nonsequential order. If this bit is set, all stores to memory occur in instruction execution order.

- **Bit <11> — Intrinsic Error (FIN)**
  This bit is set to indicate that an intrinsic instruction detected an error. The PSW<3..0> bits contain a code that specifies the type of error.
- **Bit <10> — Intrinsic Error Enable (INE)**
  If this bit is set, and PSW<11> is set, a floating point exception occurs. If this bit is clear, no exception occurs.
- **Bit <9> — Trace Thread Concurrency (TTC)**
  If this bit is set, an instruction trace trap occurs prior to a CPU entering the hardware idle state and after it leaves the hardware idle state. The *wfork*, *idle*, and *join* instructions can cause the CPU to enter the idle state. The hardware acceptance of a posted fork causes the hardware to leave the idle state. The PSW<9> bit causes a trace trap any time a thread is created or terminates. Refer to Chapter 6, "Exceptions and Interrupts," for more information.
- **Bit <8> — Thread Initialization Trap (TIT)**
  If this bit is set when a CPU picks up a fork, a trace trap will be taken to allow a handler to initialize the user-indicated code. A code of 0x800 (class 8, no qualifier) is placed in register A5, to distinguish this trap from the other trace traps.

  This trap is based on the PSW in the fork block in the communication registers. This is a user trap, i.e., occurs in the ring where it was executed. The CPU does not have to be in sequential mode for TIT traps to function correctly.
- **Bits <7..5> — Reserved (RES)**
  These bits are reserved for future system use.
- **Bit <4> — Communication Address Trap (CAT)**
  This bit is set whenever the processor detects an invalid communication register address, which causes a system exception (ring violation) to occur. This bit remains set until the trap is recognized in order to allow a trap to be "remembered" in the event of ring crossing (*sysc*, interrupt, etc.). This ensures that the trap is attributed to the correct ring of execution. The hardware clears this bit in the extended frame passed to the system exception handler when the trap is processed.
- **Bits <3..0> — Intrinsic Error Code (IEC)**
  When PSW<11> is set, the PSW<3..0> bits contain a code that specifies the type of error encountered by the intrinsic instruction. Each intrinsic instruction that encounters an error will first clear the PSW<3..0> (bits if these bits were set from a previous error that occurred with PSW<10> clear). The new code is entered in PSW<3..0> and PSW<11> is set. If PSW<10> is set, an arithmetic trap will occur. If PSW<10> is clear, no trap occurs, i.e., if PSW<10> is clear, only the last intrinsic trap is meaningful (other intrinsic traps may have occurred and were disregarded.)

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

The validity of these bits when there is no trap are:

00 — A square root operation with a negative operand (vector or scalar) was attempted

01 — An overflow occurred when an exponential operation (*exp.s, exp.d*) was attempted

02 — An argument to a logarithmic operation (*ln.s, ln.d*) was less than or equal to zero

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

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

2.8.2.1 New Arithmetic Exceptions

The new intrinsic instructions can detect errors in their calculations based on their input operands. For example, the square root of a negative number is an error condition. These conditions are handled by using the **FIN** and **INE** PSW bits described earlier. They are handled just like other arithmetic exceptions. When the condition is detected by hardware or microcode, the **FIN** bit is set. If **INE** is also set, enabling the exception, an arithmetic trap is dispatched at the next instruction boundary to the scalar processor. An extended frame is pushed in the current ring and the arithmetic exception handler address is fetched from page 0 of the current ring. After the PSW is pushed, all error bits in the PSW are cleared to prevent further dispatches of the exception. Execution continues with the exception handler.

The intrinsic instructions may also generate other arithmetic traps, such as reserved operand. These are handled in the same fashion that the C100 Series architecture did.

2.8.2.2 New System Exceptions

The addition of the new level T PTE and thread shared memory implies the addition of 2 new system exceptions — invalid level T PTE and nonresident page level 2 PTET page. These exceptions are similar to the C100 Series PTE2 violations and are handled in the same manner, with different class codes and qualifiers, as in the following list:

- 0000 0C08 — Invalid Level T PTE
- 0000 1002 — Nonresident Page Level 2 PTET Page

These codes are placed in address register A5 before entry to the system exception handler.

2.8.2.3 Invalid Communication Address Exception

If a communication register operation is supplied an invalid communication register address, the processor detects the invalid communication register address which causes a system exception (ring violation) to occur.

In general, an invalid communication register address can be one of the following types:

- Unimplemented address — An example would be the address 8040 on the C200 Series implementation (the implemented address range stops at 803f).
- Ring-protected address — An example would be a Ring 4 (user) program specifying an address of 0000 (only Ring 0 can reference this address).

For a description of communication register address mapping refer to the "Communication Registers" section.

When an invalid communication register address is detected, the PSW <CAT> bit is set. At the next instruction boundary, microcode is dispatched to a trap routine that pushes an extended frame, clears the PSW, and enters the Ring 0 system exception handler. The system exception is "deferred" through the PSW because of the pipelined nature of CONVEX machines.

For example, the communication address trap would be "charged" to Ring 4 if a Ring 4 program executed the following sequence:

```
get.l  0x0000,s0
sysc   #0,#1
```

In this example, if the *sysc* operation had been dispatched before the invalid communication register address had been detected, the crossing to Ring 0 may have already been completed. By placing the trap condition in the PSW, the *sysc* operation pushes the PSW with the **CAT** bit set, which defers the exception until the *rtn* from the *sysc*.

Invalid communication register address traps do not indicate the invalid communication register address. When the system exception handler is entered, the address register A5 contains the class code and qualifier 000000806, identifying the type of system exception, but the invalid communication register address is not provided.

2.8.2.4 Thread Concurrency and Thread Initialization Traps

The trace thread concurrency (TTC) bit can be used to monitor all thread creation and termination which occurs within a process as a result of *pfork* or *spawn* instructions. An instruction trace trap occurs after the execution of a *wfork*, *idle*, or *join* instruction prior to the CPU entering the hardware idle loop. The PC that is pushed on the stack references the next instruction to be executed, i.e., the instruction after the *wfork*, *idle*, or *join* instruction. The thread is not deallocated because the trap handler must have a thread identification in order to actually process the trap. When the thread returns from the trap handler, Ring 0 software (operating system) is expected to "back up" the PC to the immediately preceding instruction and execute the forking instruction with PSW<TTC> cleared to allow the thread to properly terminate. The TTC bit also causes an instruction trace trap to occur prior to the first instruction executed by a newly created hardware thread which accepted either a *pfork* or a *spawn* instruction. Refer to Chapter 5, "Multiprocessor Management," for a description of the CPU idle loop and related topics.

The trap class qualifier loaded into address register A5 may be used to decode the cause of the trap. Table 2–3 lists the class codes and qualifiers placed in register A5 for each exception.

> **NOTE**
>
> The test for trace thread concurrency is performed based on *fork.PSW* after the fork is taken. The PC in the trap frame will be *fork.PC*, i.e., the starting address of the new thread.

The *thread initialization trap* (TIT) allows a thread to have its state initialized as the thread begins execution. This trap is primarily for vector registers permitting each thread to be forced to start with a known vector register state. If the PSW<TIT> bit is set when a CPU picks up a fork, a trace trap is taken to allow a user-defined handler to initialize the desired state. A code of 0800 (class 8, no qualifier) is placed in address register A5. Refer to the section describing system exception processing for an explanation of the structure of the halfword containing these class codes and qualifiers.

> **NOTE**
>
> This trap is based on the PSW contained in the fork block *(fork.PSW)* that is located in the communication registers. The TIT is a user trap, i.e., this trap occurs in the ring where it was executed. The PC in the trap frame will be *fork.PC*, i.e., the starting address of the new thread. A CPU ***does not*** have to be in sequential mode for TIT traps to function correctly.

Table 2–3, Trace Trap Class Codes and Qualifiers

Exception Type	Class (Hex) Byte 2	Qualifiers Byte 3	Priority
Instruction Trace	00	None	Highest
Thread Concurrency Trace	04	0 – Thread Creation (*pfork/spawn* accepted) 1 – *join* instruction executed 2 – *wfork* instruction executed 3 – *idle* instruction executed	
Thread Initialization Trap	08	None	Lowest

The following conditions can cause each kind of Trace Thread Concurrency (TTC) trap:

- **Thread creation traps** — code 0400
  1. An idle CPU picks up fork and sees PSW <TTC> set in *fork.PSW*.
  2. An *idle* instruction is executed, a fork found in CIR specified in Sk, and PSW <TTC> is set in *fork.PSW*
  3. A *wfork* instruction is executed by the last thread, a fork is found in the current CIR, and PSW <TTC> is set in *fork.PSW*.
- **Thread termination traps** — codes 040x

  0401 The *join* instruction was executed by a thread other than the last thread, so the CPU traps instead of entering the idle loop.

  0402 The *wfork* instruction was executed by a thread other than the last thread, so the CPU traps instead of entering the idle loop.

  0403 The *idle* instruction finds no fork to take in CIR Sk and goes to the CPU idle loop, ***after*** determining that the CPU's PSW<TTC> is set, and traps rather than entering the CPU idle loop. Note that when a CPU cannot find a fork to take, this either means no fork is posted, the CPU cannot receive (*rcv*) thread count/mask, no allocatable threads exist in the thread mask, or the posted fork is marked *STOPPED*, i.e., another thread has joined.

In order for a thread termination trap to continue after all the threads of a process are joined in the trap handler, and to keep the PSW <TTC> set after the last thread continues the serial thread, the operating system should initiate the following trap processing sequence:

1. A *join* instruction is executed in the trace trap handler.
2. The trace trap handler returns from the TTC trap frame.

For example, consider a two-threaded trap process. The first thread executes a *join* instruction traps before terminating, so the trace handler is called ***without deallocating the thread.*** The second thread (that would normally be the last thread) encounters the *join* instruction, sees the threadcount of 2, and also trace traps. Since all threads have trace trapped, the serial thread continues execution after the *join* completes with the PSW<TTC> bit cleared.

A *join* instruction must always be executed to allow all but one thread to terminate ***with TTC clear***. All threads are joined in the trace trap handler and the trace trap handler returns with a serial thread. Only the last thread will execute the return. This thread continues after the join operation with the PSW <TTC> bit set which was reloaded when the extended frame is popped.

2.8.2.5 Process Deadlock Exceptions

A class code and qualifier for an exception are placed in address register A5.

The class codes and qualifiers for process deadlock exceptions are listed in Table 2–4:

Table 2–4, Process Deadlock Class Codes and Qualifiers

Exception Type	CLASS (Hex) Byte 2	QUALIFIERS Byte 3	Priority
Last Thread Termination	0	None	Highest
Hardware Detected Deadlock	4	0 – all threads branching to synchronizing instruction 1 – mixed *wfork* and *join*	Lowest

2.8.2.6 Process Trap and Process Breakpoint

The trap instruction, *trap #rm,#b*, and the process breakpoint instruction, *pbkpt*, provide the only means, other than deadlock, for gaining control of all threads within a process in a timely fashion. When a trap instruction is executed, each CPU which is currently executing on behalf of the process, and within the rings specified, will reduce the process to a single thread and immediately trap to the Ring 0 process trap exception handler. Any CPU which subsequently attempts to enter a ring which has a process trap exception still pending will enter the Ring 0 process trap exception handler.

Refer to the appropriate table that summarizes system exception class codes and qualifiers for the particular architecture. This table is located in the section describing "System Exception Processing," in this chapter.

The value of the trap instruction register containing the trap will be placed in scalar register S0 by the hardware. This allows the exception handler to determine the source of the trap. The *trap* and *pbkpt* instructions are detailed in Chapter 8, "Instruction Set."

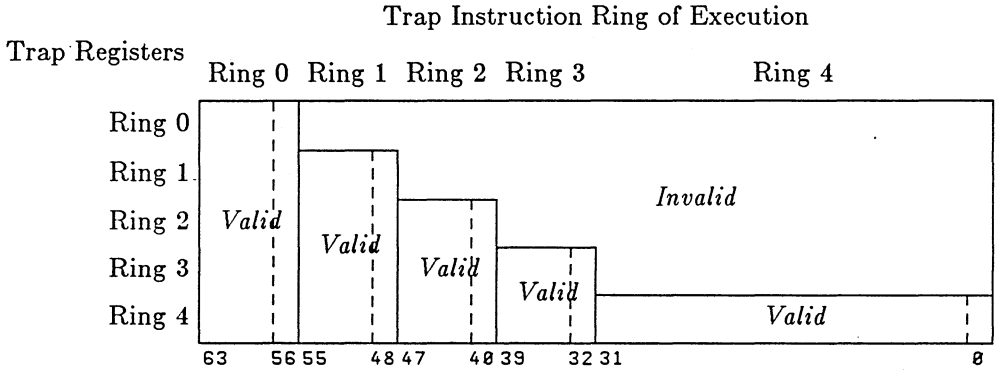
The *#rm* field is a 5-bit ring mask used to select which rings will be forced to trap. This mask defines which of the five 64-bit trap event registers in the hardware communication register set are to be modified. The most significant bit of *#rm* specifies Ring 4, down to the least significant bit which specifies Ring 0. Refer to Chapter 5, "Multiprocessor Management," for more information on the trap event registers.

The *#b* field is a bit number between 0–63 which is set in all the trap instruction registers specified by the #rm field.

All accesses to the trap registers by the *trap* instruction are protected by ring maximization as shown in Figure 2–17. The height of the boxes show the validity of the target ring references based on ring of execution. The width shows which bits may be set by the *trap* instruction within each ring of execution. The occurrence of any set bit in the trap register will cause a trap of all threads associated with, or entering, the ring shown on the left.

Figure 2–17 illustrates trap instruction register partitioning:

Figure 2–17, Trap Instruction Register Partitioning



The execution of a *trap* instruction can only change the trap instruction registers of greater than or equivalent rings. Each ring may only modify a fixed set of bits within any trap register. The bit positions associated with the dashed boxes are reserved for use by the process breakpoint (*pbkpt*) instruction (refer to the next section in this chapter).

For example, a trap instruction executed in Ring 3 can only set bits <33..39> in the trap instruction registers for Rings 3 and 4. Any trap instruction executed with an invalid ring or bit field will cause the executing thread to enter the system exception handler with an *invalid trap instruction* exception code.

By providing the protection in this way, Ring 0 is able to determine which ring executed the trap instruction by the position of the bit in the trap register.

When a valid trap occurs, the system exception handler is entered with a trap instruction class violation. The trap condition remains outstanding until the bit which caused the exception is cleared in the trap register. The program counter of the ring which trapped and the value of the trap instruction register (TIR) where the trap was recognized (returned in scalar register S0 by hardware) can be used to determine the trap register and permit clearing of the exception.

There is no locking protocol on the TIRs. In order to clear the TIRs in a communication register set, the exception handler reduces the process running in that communication register set to a single thread to ensure that no traps are missed. Therefore, the exception handler must wait until all threads have entered the exception handler before clearing the TIRs.

The Trap Instruction Registers (TIRs) set by the *trap* and *pbkpt* instructions are checked upon entering a ring. Whenever a ring crossing is executed, except for interrupts, e.g., a base-level interrupt that is taken in an outer ring enters Ring 0 for interrupt handler, the TIR for the target ring is checked for any set bits. If any of the target ring's TIR bits are set, the system exception handler is entered with a trap instruction qualifier (1400 from Table 2–5) instead of performing the original function of the ring crossing (enter exception handler, do *sysc*, etc.). The trap condition remains outstanding until all bits in the hardware communication trap instruction register are cleared. If a thread attempts to enter a ring that has any bits set in that ring's trap instruction register, the thread will immediately enter the Ring 0 system exception handler. The trap frame for this exception points to the intended function of the original ring crossing.

For example, consider a two-threaded process running with one thread in Ring 0 and one thread in Ring 4. The Ring 0 thread executes a *trap* instruction to trap Ring 0, but the other thread doesn't trap because it's in Ring 4. Next, the Ring 4 thread takes a page fault. On the way to the exception handler for the fault, the Ring 4 thread sees that the TIR bit is set, pushes a frame pointing to the exception handler (to handle the page fault), and enters the exception handler. After the TIR trap is processed, the return from the exception handler causes a second entry to the exception handler for the page fault. After the fault is processed, the context return sends the thread to Ring 4.

Although the "invalid" bits shown in Figure 2–17 cannot be set by a *trap* instruction, Ring 0 can set them with a *put* instruction. On the next ring crossing into that ring, a trap would be taken because the hardware checks the TIR on ring crossings. If the TIR is nonzero, the CPU traps.

> **NOTE**
>
> This should not be done as it subverts the intent of the *trap/pbkpt* mechanism.

2.8.2.7 Process Breakpoint

The process trap mechanism is also used to implement a process breakpoint facility. A 16-bit process breakpoint instruction *pbkpt* is defined which causes all rings greater than or equal to the current ring of execution to trap. This is done by setting the appropriate process breakpoint bits in the communication trap registers for all rings greater than or equal to the ring of execution. Figure 2–17 in the previous section shows which bit for each ring of execution is set by the process breakpoint instruction. For example, if Ring 2 executes a *pbkpt* instruction, bit <40> is set in the communication trap registers for Rings 2, 3, and 4.

2.9 Interrupts

Interrupts are asynchronously occurring events; they belong to the system and not to the executing process. They are processed on an *interrupt stack* in Ring 0. Interrupts are nested if additional interrupts occur during interrupt processing. When an interrupt occurs, the processor will vector to a particular interrupt handler as a function of the source of the interrupt.

2.9.1 Interrupt Channels

There are 256 interrupt channels within a processor. Each channel is either a *CPU virtual channel* or an *I/O virtual channel.* Eight channels are specifically allocated to the CPU. These eight channels are addressed as channels 0–7 of the 256 system-wide channels.

The remaining 248 interrupt channels are allocated to I/O processors. The number of I/O processors and the number of virtual interrupt channels allocated to I/O processors are specific to each particular implementation. The instruction set for each processor includes instructions to allow any one channel to interrupt any other channel.

All external devices and controllers, regardless of their local intelligence, interrupt the CPU on one of the eight CPU virtual channels. The number of CPU virtual channels is independent of the number of actual I/O channels. For example, a physical I/O controller may use only one I/O channel to initiate interrupts using more than one CPU virtual channel. In some cases, the processor may interpret one physical I/O controller as multiple I/O channels. Conversely, there may be up to 248 I/O virtual channels competing for eight CPU virtual channels. The processor can also individually interrupt any I/O channel by using the *xmti* instruction.

All virtual interrupt channels are *Complex virtual channels*. All external devices and controllers, regardless of their local intelligence, interrupt the Complex on one of eight Complex virtual channel ports. The *enal* instruction is used by any CPU in the Complex to enable interrupts selectively to a particular virtual channel. The *enag* instruction is used to enable interrupts selectively to a particular virtual channel from the entire Complex. This pair of instructions has the same function for the C200 Series architecture as the *mski* does instruction for the C100 Series architecture.

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

The number of Complex virtual channels has no relationship to the number of actual I/O channels. For example, one I/O channel may initiate interrupts using more than one Complex virtual channel. Conversely, as many 248 I/O channels may be competing for eight Complex virtual channels.

2.9.2 Interrupt Processing Arbitration

The following sections describe the interrupt control flow and interrupt service processing sequences for the C200 Series architecture.

The ION flag is a single CPU Complex interrupt enable flag that is controlled by the *eni* (enable) and *dsi* (disable) interrupt instructions. When ION is disabled, all interrupts sent to the CPU Complex are deferred until ION is enabled. The *dsi* instruction atomically disables interrupts and returns the previous interrupt state. This enables *dsi* to be used as a simple lock for protecting critical code sections within the entire CPU Complex.

2.9.2.1 Local and Global Interrupt Enable Registers

Each CPU has an 8-bit local enable register which selectively permits each channel (0 – 7) to interrupt that CPU. Bit <0> corresponds to virtual channel 0, bit <1> to channel 1, and so on. If a bit is set (1), that channel is enabled. If a bit is clear (0), it is disabled (masked out) from that CPU. The *enal* instruction is provided to manipulate this register.

In addition, the Complex has a single 8-bit global enable register that selectively enables or disables virtual channels from the entire Complex. The interpretation of this register is identical to the local enable. The *enag* instruction is provided to manipulate this register.

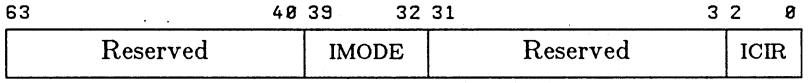
Two additional registers complete the interrupt control and arbitration scheme in the C200 Series architecture: Interrupt Control Register (ICR) and the Target CPU (TCPU) register.

2.9.2.2 Interrupt Control Register (ICR)

The Interrupt Control Register (ICR) within the CPU Complex defines the operating modes of each interrupt channel and the communication register set used during interrupt processing.

Figure 2–18 shows the format of the Interrupt Control Register:

Figure 2–18, Interrupt Control Register (ICR)



The Interrupt Mode (IMODE) in the Interrupt Control Register is an 8-bit field in which bit <0> corresponds to Complex virtual channel 0, bit <1> to Complex virtual channel 1, and so on. The Interrupt Mode controls the mode of operation for each Complex interrupt channel. Any Complex virtual channel can be selected to operate in one of the following modes:

- **Local Interrupt Mode** — A single CPU within the Complex is selected to receive the interrupt. The interrupt is delivered to the selected CPU when the interrupt occurs.
- **Broadcast Interrupt Mode** — All CPUs within the Complex are selected to receive the interrupt. The interrupt is delivered to all CPUs when the interrupt occurs.

If the bit associated with the virtual channel is clear (0), the channel is considered a local interrupt channel. If the bit is set (1), the channel is considered a broadcast interrupt channel.

Both broadcast and local interrupts can be selectively enabled or disabled with one of the two interrupt channel enable instructions. The global CPU enable instruction, *enag*, is used to enable or inhibit interrupt delivery to all CPUs within the Complex. A global Complex interrupt channel is enabled or disabled by setting or clearing, respectively, the appropriate bit in the *global enable* interrupt register.

For broadcast (global) interrupts, the global interrupt handler ensures that all CPUs have entered the interrupt handler before executing an *eni* instruction at the end of the handler and returning (or the CPU goes idle). Otherwise, a hardware race condition could exist where one CPU may execute an *eni* instruction before the other CPU(s) have received the interrupt causing the other CPU(s) to lose the interrupt.

The local CPU enable instruction, *enal*, is used to enable or inhibit delivery to a single CPU. A local Complex interrupt channel is enabled or disabled by setting or clearing, respectively, the appropriate bit in the *local enable* interrupt register. These two instructions allow any single CPU within the Complex to enable or disable interrupt reception locally or for the entire Complex.

The Interrupt Communication Index Register (ICIR) is a 3-bit field which defines the Communication register set that is mapped when servicing interrupts. This communication register set provides the process context necessary for an idle CPU to service an interrupt.

2.9.2.3 Target CPU Register (TCPU)

A single Complex register exists that contains the identification of the target CPU(s) that services all global interrupts. This register allows one or more CPU(s) to serve as the nesting point for interrupts. The interrupt target CPU (TCPU) register is a 64-bit register that contains the CPUID of the CPU where all interrupts must be delivered. If the target CPU register contains all binary 1s, any CPU within the CPU Complex can be selected as the target CPU for interrupt delivery.

Interrupt handler entry is accomplished by each CPU disabling the ION flag. No other interrupt delivery can occur until the ION flag is explicitly enabled with an *eni* instruction.

2.9.2.4 Interrupt Control System

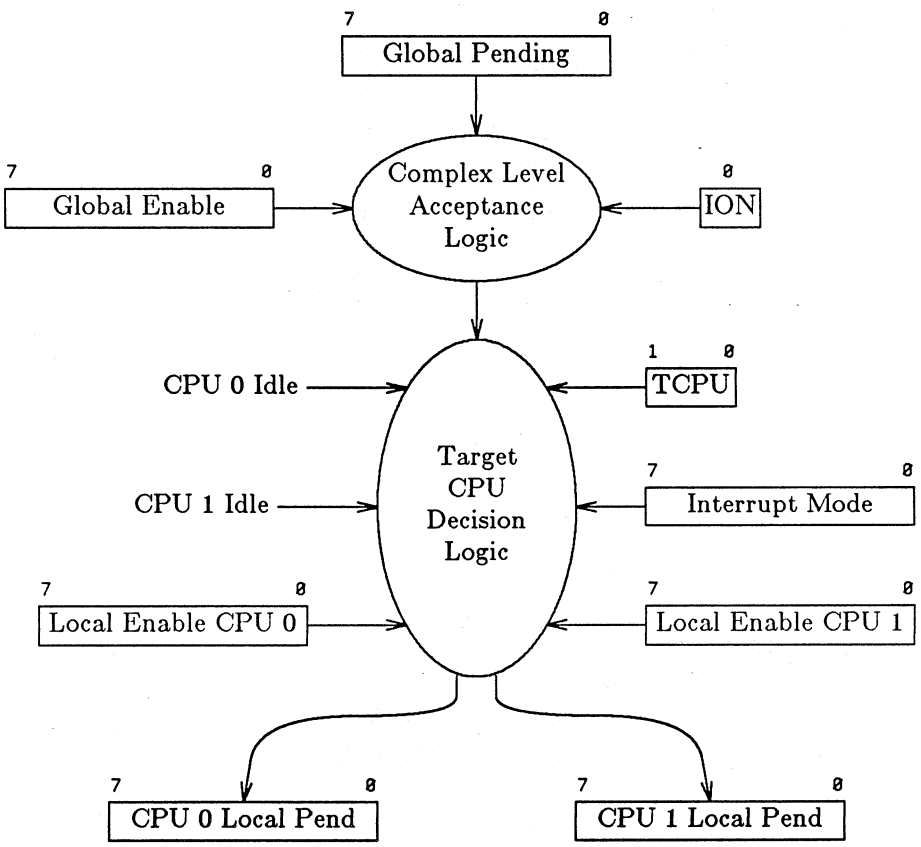
This section describes the particular implementation of interrupt flow on the C220 processor. The following description and figure uses the C220 as a representative multiprocessor to present the basic implementation of the interrupt system for a CONVEX multiprocessor regardless of the number of CPUs in a given CPU Complex.

Interrupts enter the CPU Complex in the 8-bit Global Pending register. Each bit in this register corresponds to one of the eight virtual channels that the CPU responds to. Before a CPU can recognize an interrupt in the Global Pending register, a series of enables and destination checks must be made by the target CPU. If all these checks are satisfied, the interrupt is registered in the Local Pending register for the CPU(s) that respond to the interrupt.

> **NOTE**
>
> The bit in the Global Pending register will remain set if it cannot be taken all the way to a Local Pending register. These checks are performed every system clock. If the interrupt is taken, the ION flag is reset to zero, disabling interrupts until they are explicitly re-enabled with the *eni* instruction.

Figure 2–19 shows the flow of interrupts in a C220 from the Global Pending register to the two Local Pending registers. Where CPU 0 is specified in the following discussion, it is equivalent to physical CPU A, and CPU 1 is equivalent to physical CPU B.

Figure 2–19, Interrupt Flow — C220



The first level of interrupt checking is at the Complex level, shown as the Complex Level Acceptance Logic in Figure 2–19:

1. If the ION flag is zero, interrupts are disabled for the Complex and the interrupt stays in the Global Pending register. The ION flag must be cleared (0) to allow software to modify the state of the subsequent interrupt control hardware.
2. The next check is the Global Enable register. Each bit in this register enables the corresponding bit in the Global Pending register.

Now that the Complex level checks are complete, the destination CPU is chosen by using the following decision logic:

1. If the bit in the Interrupt Mode register corresponding to the Global pending bit is set (1), the interrupt is considered a broadcast interrupt, and **must** be sent to both CPUs.
2. If the Target CPU (TCPU) register is set to 0, the interrupt **must** be sent to CPU 0; likewise if TCPU is 1, the interrupt **must** go to CPU 1. In this case, the target CPU must also have the corresponding Local Enable bit set to 1.

3. If TCPU is –1 (binary "11"), either CPU may be chosen. In this case, an idle CPU (designated by the CPU "n" Idle signal shown in Figure 2–19) will be chosen if the correct bit in the idle CPU's Local Enable is set to 1.
4. If both CPUs are idle and locally enabled, CPU 0 is chosen.

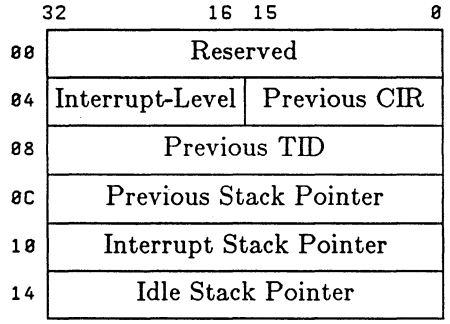
If all the preceding conditions are met, the corresponding bit in the selected CPU's Local Pend register is set, the bit is cleared in the Global Pending register and the ION flag is cleared.

2.9.2.5 Interrupt Context Blocks

While servicing an interrupt, each CPU locates its associated interrupt context block (ICB) by using unshared data segments. Each CPU indexes into memory based on Thread ID (TID) as described in this chapter. The pointer to this unshared data is located in the interrupt context block pointer contained in location 0000 0014 of Ring 0 page 0.

Figure 2–20 shows the format of an interrupt context block:

Figure 2–20, Interrupt Context Block



The 16-bit halfword located at bytes 4 and 5 of the ICB is fetched. If this halfword is 0, then the interrupt is the first interrupt processed. This condition is referred to as *base-level interrupt processing*. If this halfword is not 0, then the interrupt is not the first interrupt, and the processor is already at *interrupt-level*. (The current Ring 0 stack used is the interrupt stack. In effect, the Ring 0 process stack pointer has temporarily become the interrupt stack pointer).

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

In the following sections, the program counter (PC) that is pushed onto the stack references the instruction that would have been executed if the interrupt had not occurred. The interrupt handler is entered with all Complex interrupts disabled. The Complex virtual channel interrupt is always reset after a CPU responds to the interrupt.

2.9.2.6 Virtual Memory Mapping Restrictions

Some restrictions are placed on virtual memory mapping due to existence of a separate interrupt CIR. The interrupt service microcode must deal with both process context (the extended frame pushed on the Ring 0 stack) and ICB context (interrupt-level, etc.).

The ICBs **must** be resident at all times; a machine exception will occur if they are not resident. In addition, the ICBs **must** be mapped in all CIRs, so that while the interrupt service microcode is transitioning the CPU from the process CIR to the interrupt CIR, both process and ICB context may be accessed in the process CIR.

The interrupt CIR does not require access to all process context, i.e., each process may have a unique page 0. For the same reason, the interrupt handler with an address found in the process's page 0 must be mapped in all CIRs. Whenever the interrupt handler handler is entered, the stack pointer is always equal to the frame pointer.

These memory mapping restrictions will become apparent in the following sections describing CPU interrupt processing. Please note carefully where the transition from a process CIR to an interrupt CIR is made.

2.9.3 Idle CPU Interrupt Processing

When an idle CPU takes an interrupt, it must be at base-level (interrupt level 0). An idle CPU has no state, so the CPU state is not saved or restored. An "idle stack" provides the operating system a consistent entry/exit mechanism. The process context referenced by an idle CPU is mapped to the interrupt CIR. The interrupt service for an idle CPU performs the following interrupt processing sequence:

1. The interrupt CIR is fetched from the Interrupt Control Register (ICR) and loaded into the CPU's CIR. The TID is set to the CPUID. The CPU assumes that this thread is always allocated. The TID is set before any memory references to the ICB occur to allow the ICBs to use unshared memory for multiple CPUs entering the interrupt handler simultaneously in different threads.
2. The Interrupt Context Block (ICB) pointer is fetched from page 0 of the process's address space. The ICB must be mapped into the interrupt CIR because the CPU has already entered the interrupt CIR.
3. The interrupt handler address is fetched from page 0 of the process's address space. The interrupt handler must be mapped into the interrupt CIR because the CPU has already entered the interrupt CIR.
4. The stack pointer is loaded with the value contained in the idle stack pointer in the ICB. Whenever the interrupt handler handler is entered, the stack pointer is alway equal to the frame pointer.
5. The interrupt-level in the ICB is set to 1 (incremented from 0, or base-level).
6. The previous CIR in the ICB is set to –1 which means that an idle CPU serviced the interrupt. Setting the CIR in the ICB to –1 means that the previous TID in the ICB is meaningless; in this case, the TID is ignored on the subsequent *rtn* or *idle* instruction.

7. A PC of zero and a PSW with the FRL field containing a binary "01" (extended frame) is "pushed" on the stack (idle stack). The stack pointer is updated as if an entire extended frame had been pushed.
8. The stack pointer (reflecting the "push" of the extended frame) is stored in the previous stack pointer in the ICB.
9. The stack pointer and frame pointer are now loaded from the interrupt stack pointer in the ICB, establishing the interrupt stack.
10. The PSW is cleared.
11. The channel ID of the interrupting virtual channel is placed in address register A5.
12. The PC is loaded with the interrupt handler address that was previously fetched. Execution continues in the interrupt handler (still within the interrupt CIR) with a newly established thread.

2.9.4 Active CPU Interrupt Processing

An active CPU may respond to both base and interrupt-level interrupts. If the interrupt is base-level, the CPU is not already in the interrupt CIR and the process context referenced belongs to a different CIR. An active CPU may have to cross rings to enter Ring 0. Interrupt-level interrupts are already in the interrupt CIR in Ring 0. Both cases possess the following common processing sequence:

1. The Interrupt Context Block (ICB) pointer is fetched from page 0 of the process's address space.
2. The interrupt-level is fetched from the ICB and tested. If the interrupt level is 0, an active CPU continues with base-level processing. Note that the interrupt-level fetch occurred in the process CIR which requires that the ICB be mapped in all CIRs. If the interrupt-level was nonzero, execution continues with interrupt-level processing.

2.9.4.1 Active CPU Base-Level Processing

1. The interrupt CIR is fetched from the Interrupt Control Register (ICR) and loaded into the CPU's CIR. The TID is set to the CPUID. The CPU assumes that this thread is always allocated. Before any memory references to the ICB occur, the TID is set to allow the ICBs to use unshared memory for multiple CPUs that are entering the interrupt handler simultaneously from different threads.
2. The incremented interrupt-level is stored in the ICB (while still executing within the process CIR).
3. If the interrupted process was not executing in Ring 0, the CPU crosses to Ring 0 and allocates a Ring 0 stack from the Shared Resource Structure (SRS). If the interrupt occurred while in Ring 0, a Ring 0 stack already exists.
4. The address of the interrupt handler is fetched from page 0 of the process's address space.
5. An extended frame is pushed on the Ring 0 stack.
6. The updated stack pointer is stored in the previous stack pointer field of the ICB (still in process CIR).

7. The interrupt stack is established by initializing the stack and frame pointers from the interrupt stack field of the ICB (still in process CIR).
8. The CIR is loaded from the Interrupt CIR field of the Interrupt Control Register. A new thread is allocated and the thread count incremented in the new (interrupt) CIR. The TID is set to the CPUID.
9. The PSW is cleared.
10. The channel ID of the interrupting virtual channel is placed in address register A5.
11. The PC is loaded with the interrupt handler address that was fetched earlier. Execution continues in the interrupt handler with a newly established thread (while still within the interrupt CIR).

2.9.4.2 Active CPU Interrupt-Level Processing

During interrupt-level processing, an active CPU is already executing in Ring 0, within the interrupt CIR, with the thread that was established at the base-level interrupt.

1. The incremented interrupt-level is stored in the ICB.
2. The address of the interrupt handler is fetched from page 0 of the process's address space.
3. An extended frame is pushed on the Ring 0 stack.
4. The updated stack pointer is stored in the previous stack pointer field of the ICB.
5. The PSW is cleared.
6. The channel ID of the interrupting virtual channel is placed in address register A5.
7. The PC is loaded with the interrupt handler address that was previously fetched. Execution continues in the interrupt handler, while still within the interrupt CIR.

2.9.5 Returning from a Base-Level Interrupt

After an interrupt has been processed, executing the *rtn* or *idle* instruction will return from a base-level interrupt.

The software performs the following operations:

1. If the previous CIR field in the ICB is "–1", (the interrupt was serviced by an idle CPU), the CPU is returned to the idle state with an *idle* instruction. Since an idle CPU has no state, an extended frame is not popped.
2. If returning to a CPU which was previously executing, the CIR and TID is restored from the previous CIR and previous TID fields in the ICB. Address register A7 is then loaded from the previous stack pointer in the ICB. A standard subroutine performs an extended return sequence, because the FRL bits in the pushed PSW indicate that an extended return block was pushed.
3. Software always leaves N threads allocated in the interrupt CIR (where N is the number of CPUs in the Complex), to allow each CPU to set TID equal to the CPUID as it services an interrupt.

2.9.6 General Interrupt Processing Information

- The identification of the interrupting device loaded into address register A5 after the return block is a 32-bit value. This value takes the form of 29 "0" bits followed by a 3-bit encoding. This 3-bit encoding identifies which CPU Complex virtual channel initiated the interrupt.
- All Complex interrupts are disabled when the interrupt handler is entered. The interrupt handler must explicitly re-enable interrupts.
- The interrupt return sequence determines if the return is to base-level or interrupt-level as a function of the interrupt-level in the ICB.
- A CPU returning to base-level is returned to the idle state or the previous context based on the previous CIR in the interrupt context block.
- In order to return from an interrupt, the following steps must be taken by software:
  1. The interrupt-level is decremented by one.
  2. If the level is now 0, the frame pointer (A7) is loaded from the previous stack pointer in the ICB. The *rtn* or *idle* instruction is now executed.
  3. If the level is not zero, the *rtn* instruction is executed. Address register A7 is not restored since the Ring 0 stack must still be the interrupt stack.

2.10 Timers

A major addition to the C200 Series architecture are the many timers provided to permit fine grain, accurate accounting of execution time and to assist in process scheduling. There are four timers in the C200 Series architecture:

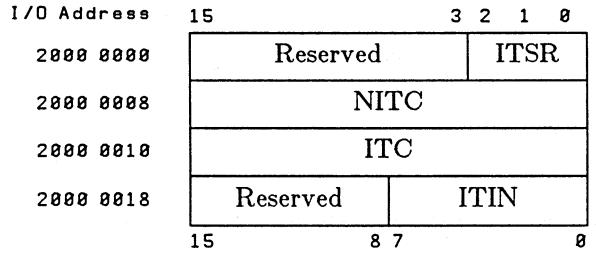
- **Interval Timer** — C100, C200 Series
  All CONVEX CPUs contain an interval timer; used to interrupt the processor at a programmable rate. The implementation of the interval timer is processor-specific, but the logical structure of the interval timer is the same on all processors. Each processor contains an interval timer counter and a next interval register. Each time the interval timer counter counts to zero, it is loaded from the Next Interval Timer Count (NITC) register.

  An Interval Timer Status Register (ITSR) controls the operation of the counter and controls the generation of interrupts in addition to the interval time. The ION flag enables and disables the interval timer ***only*** in the C100 Series architecture.
- **Time of Century clock (TOC)** — C200 Series only
  In the C100 Series architecture, the operating system software implements a TOC clock via the C100 interval timers. The C200 Series architecture implements a TOC clock in hardware that can be both read and written. This clock keeps "wall" clock time, not user time.
- **CPU Execution Timer** — C200 Series only
  See Chapter 5, "Multiprocessor Management," for more information on this architecturally-defined timer.
- **Thread Timer** — C200 Series only
  The C200 Series architecture has a thread timer that measures the elapsed CPU time for executing a thread. This thread timer can be both read and written. See Chapter 5, "Multiprocessor Management," for more information on this architecturally-defined timer.

2.10.1 Interval Timer — C200 Series

The interval timer on the C200 Series architecture is accessed via I/O address space. The interrupt frequency of this interval timer is programmable from 100 KHz to about 1.6 Hz. This timer is controlled by a set of four 16-bit registers, longword aligned at I/O addresses 2000 0000 to 2000 0018. The interval timer decrements by one at 10 microsecond (μsec) intervals. Because this timer is located in I/O address space, no special instructions are required to service it. Figure 2–21 presents the address locations and format of the interval timer for the C200 Series architecture:

Figure 2–21, Interval Timer Registers



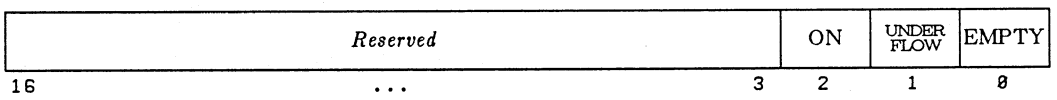
The following registers are located in the C200 Series interval timer:

- Interval Timer Status Register (ITSR)
- Next Interval Timer Counter (NITC)
- Interval Timer Counter (ITC)
- Interval Timer Interrupt Number (ITIN)

2.10.1.1 Interval Timer Status Register

The Interval Timer Status Register (ITSR) is a 16-bit register located at I/O address 2000 0000. Only the least significant three bits of the ITSR are valid. Figure 2–22 presents the format of the ITSR register:

Figure 2–22, Interval Timer Status Register



The three least significant bits are defined as:

- **Bit <2>** — On
  When this bit is set, the interval timer (ITC) is active and counts by decrementing in bit 0. When this bit is cleared, the interval timer is halted. Interrupts generated by the interval timer are disabled by disabling the virtual interrupt channel specified in the ITIN.
- **Bit <1>** — Underflow
  This bit is ***read only*** and is set by incrementing whenever the ITC underflows and the Empty bit has already been set by a previous ITC cycle. The ITC completes a cycle by reaching the terminal count value located in the NITC. The Underflow bit is cleared by reading the ITSR.
- **Bit <0>** — Empty
  This bit is ***read only*** and is set whenever the ITC underflows. This bit is cleared at the same time as the Underflow bit (when the ITSR is read as a normal part of the interrupt service.) This bit is set when the ITC completes a cycle by reaching terminal count and underflowing.

2.10.1.2 Next Interval Timer Count

The Next Interval Timer Count (NITC) is a 16-bit register located at I/O address 2000 0008. The NITC contains the count value to be loaded into the ITC upon underflow. It can be both read and written.

2.10.1.3 Interval Timer Counter

The Interval Timer Counter (ITC) is a 16-bit down-counter located at I/O address 2000 0010. The ITC is loaded from the NITC each time it decrements to zero. It can be both read and written.

2.10.1.4 Interval Timer Interrupt Number

The Interval Timer Interrupt Number (ITIN) is a 16-bit register located at I/O address 2000 0018. Only the least significant byte is valid, and it contains the virtual interrupt channel that receives timer underflow interrupts. The interval timer interrupts can be disabled by disabling the corresponding virtual interrupt channel.

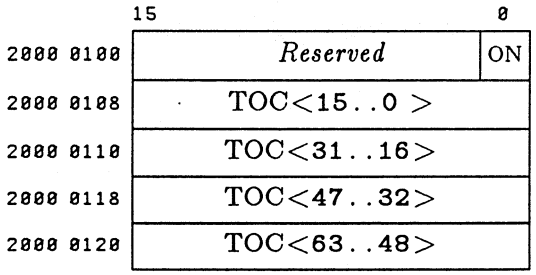
2.10.2 Time of Century Clock — C200 Series

The C200 Series architecture implements a 64-bit Time Of Century (TOC) clock that keeps time in 1-microsecond (μsec) increments and overflows every 500,000 years. The TOC can be initialized by writing four halfwords to a series of I/O addresses containing the counter, and by writing two halfwords to the counter status halfword. The TOC will count continuously thereafter by incrementing in TOC <0> in 1 μsec intervals. The TOC should be initialized immediately after the processor is powered up.

Note that this clock keeps *wall-clock* time, not user time. It is not saved and restored during context switches; the TOC is not altered by context switches and will keep time indefinitely.

The 64-bit TOC clock is implemented as four 16-bit I/O locations for the counter and one 16-bit I/O location for the counter status. All I/O addresses are developed from the page 0, Ring 0 I/O register pointer. Refer to the section in Chapter 4, "Memory Management," discussing page 0, Ring 0 for more information regarding the I/O register pointer. Figure 2–23 presents the logical structure of the TOC with the actual physical I/O addresses for the C200 Series architecture:

Figure 2–23, 64-Bit TOC Clock



The most significant bit of the clock is contained in the most significant bit of I/O address 2000 0120, and the least significant bit is located in the least significant bit of I/O address 2000 0108.

Before the TOC can be written, it must be turned off by clearing the ON bit to 0. All four remaining clock locations should then be written, and the TOC set to continue counting by setting the ON bit to 1.

The TOC should be written to only 16 bits at a time. To correctly load a 64-bit value in the TOC counter, the TOC should be written at four separate locations with four separate instructions. The TOC should be disabled by clearing the control register (which disables counting) until loading the TOC is completed. This action avoids having the TOC incorrectly loaded while it continues to increment.

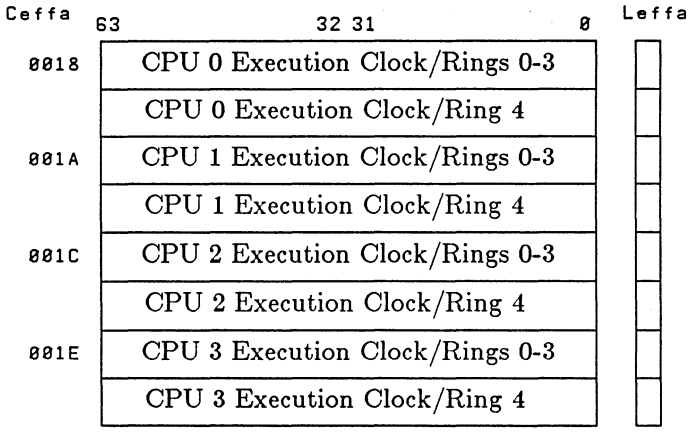
A special instruction, *mov toc, Sk*, simplifies the process of reading the TOC. This instruction will atomically read the TOC information by performing the actual I/O accesses in microcode. The *mov toc, Sk* instruction cannot be interrupted when the TOC is actually being read.

2.10.3 CPU Execution Timer

In order to provide accurate accounting information to the operating system, the C200 Series architecture includes CPU Execution Timers (CTR) for each process which maintain microsecond timing for each CPU's time spent in rings 0-3 (system time) and ring 4 (user time). These are visible to the operating system in the hardware communication registers, in a CIR-dependent manner (i.e., there is a set of these timers for each CIR).

For the C200 Series hardware, the format of these timers in the communication registers is shown in Figure 2–24:

Figure 2–24, CPU Execution Clock Registers



These timers are maintained by microcode on a "demand" basis. This means that if the process is running in ring 0 and a *get* of the ring0 CTR for that CPU is executed, it most likely is not an up-to-date time. The "demand" events that cause the microcode to update a particular timer are primarily ring crossings and execution of the *ctrsg* instruction. *ctrsg* is a privileged instruction used by the operating system to instruct all CPUs to update their CTR (i.e., the timer for the CPU they are on and the ring they are executing in). For example, on a C220, if one thread of a process is the operating system kernel executing on CPU0 (in ring 0) and the other thread is a user program in ring 4, the operating system thread can execute *ctrsg* and then read the ring 4 or ring 0 CTR and know that it is accurate. The C200 Series implements this global timer updating with the processor trap mechanism discussed in the "Page 0, Traps, and Exceptions" section.

The C200 Series machines implement the CPU execution timer with a single hardware delta timer on each CPU (located on the ASP board) that counts up by microseconds and is clearable. An update of the CTR is implemented by reading the current CTR value from the communication register, adding the current delta time to that value, writing the sum back to the communication register, and clearing the delta time. This delta timer is also used to implement the thread timer discussed in the next section.

2.10.4 Thread Timer

The C200 Series architecture has one 64-bit microsecond timer per thread implemented in microcode that is accessed by nonprivileged instructions. The thread timer allows each thread to determine the CPU execution time of any code region without the overhead of a system call. This register only reflects the CPU time on a ring-specific basis and cannot be used to time inner ring calls. This timer increments in bit 0 whenever a CPU is executing a thread. It can be read or written at any time by the currently executing thread.

The C200 Series Complexes implement the thread timer on each CPU, and update it using same hardware delta timer that is used to implement the CTRs. The delta timer is a microsecond timer that exists on each CPU. It is used to time intervals between accesses to the TTR or CTR. The thread timer is updated by adding the delta timer to the current TTR's value and clearing the delta timer.

The *mov TTR,Sk* instruction reads the thread timer by updating the TTR's value and copying the updated value to Sk. The *mov Sk,TTR* instruction writes the thread timer by copying Sk to the current TTR's value and clearing the delta timer. Refer to Chapter 8, "Instruction Set," for more detailed information on the instructions used to access the thread timer register.

This timer is primarily used for timing sections of code running in Ring 4, without including time spent in asynchronous events such as interrupts and page faults. The thread timer is not as effective in Ring 0, since there are many events that can change their own CIR and TID in Ring 0 and not affect the thread timer. These events do not affect the TTR since the old CIR or TID's thread timer is not saved, as it is in the extended frame on ring crossings.

The thread timer register is saved on the stack on all cross-ring calls, and restored from the stack on all cross-ring returns. This enables the timer to track a particular thread's context if the thread migrates between CPUs during its execution.

On entry to the inner ring, the thread timer is cleared to 0, although the CPU maintains the same TID. The outer ring timer value is saved in the return block, and is restored when control is returned to the outer ring. If an extended frame is pushed is pushed on the stack without a ring crossing (i.e., a system call to Ring 0 from Ring 0, etc.), the thread timer value in the extended frame is undefined. The subsequent *rtn* instruction examines the PC and determines no ring crossing occurred, so the thread timer is not popped from the extended frame. For example, if the thread takes an interrupt, the TTR is saved on the extended frame before entering the Ring 0 interrupt handler. It is restored on the subsequent return from the interrupt handler.

2.10.4.1 CTR and TTR Manipulation

The CTR and TTR timers are closely related since both timers are maintained with the same delta timer. When an event forces the CTR to be updated, the delta timer is cleared so the TTR must be updated at the same time. Similarly, when an event forces the TTR to be updated, the CTR must be updated also. Because these timers are so closely related, the specific events that manipulate the CTR and TTR are organized by event rather than by each timer and are as follows:

- Power-up (cold start) — The delta timer and the TTR are cleared
- Push extended frame (interrupts, exceptions) — If a ring crossing occurs, then update the CTR, update the TTR, push the TTR, and clear the delta timer
- Pop extended frame (*rtn*) — If a ring crossing occurs, then pop the TTR from the extended frame, update the CTR, and clear the delta timer
- Page fault — The TTR is pushed in the context block. In addition, if a ring crossing occurs, then the CTR is updated, the TTR is cleared, and the delta timer is cleared.
- *rtnc* — If a ring crossing occurs, then the TTR is popped from context block, the CTR is updated, and the delta timer is cleared

- *ctrsg* — The CTR is updated, the TTR is updated, and the delta timer is cleared (all CPUs)
- *mov Sk,TTR* — Writes the TTR, updates the CTR, and clears the delta timer
- *mov TTR,Sk* — Updates the TTR, updates the CTR, and clears the delta time.
- *mov Sk,CIR* — Update the CTR (in old the CIR) and clear the delta timer. The TTR is ***not*** modified.
- *mov Sk,TID* — ***No action.*** The TTR is ***not*** modified.
- *stcmr* — Update CTR (so the CTR is stored correctly), update the TTR, and clear the delta timer
- *ldcmr* — Update the CTR (after loading, in case current CIR is loaded), update the TTR, and clear the delta timer
- *idle Sk* — Update CTR (in the old, pre-Sk CIR), and clear the delta timer. If the fork accepted in CIR Sk, then clear the TTR.
- *wfork* accepts a fork in the current CIR — ***No action***
- Enter the CPU idle loop (thread termination) — Update the CTR (the event relies on subsequent thread creation to clear delta timer)
- Idle CPU takes interrupt — Clear delta timer
- Idle CPU accepts fork — Clear TTR, and clear delta timer
- Base level interrupt — Update CTR, and clear delta timer (regardless whether or not a ring crossing occurred)

References

Dykstra, E. W. "Co-operating Sequential Processes" *Programming Languages*, Edited by F. Genuys. 1968. Academic Press. pp. 43-112

THIS PAGE INTENTIONALLY LEFT BLANK

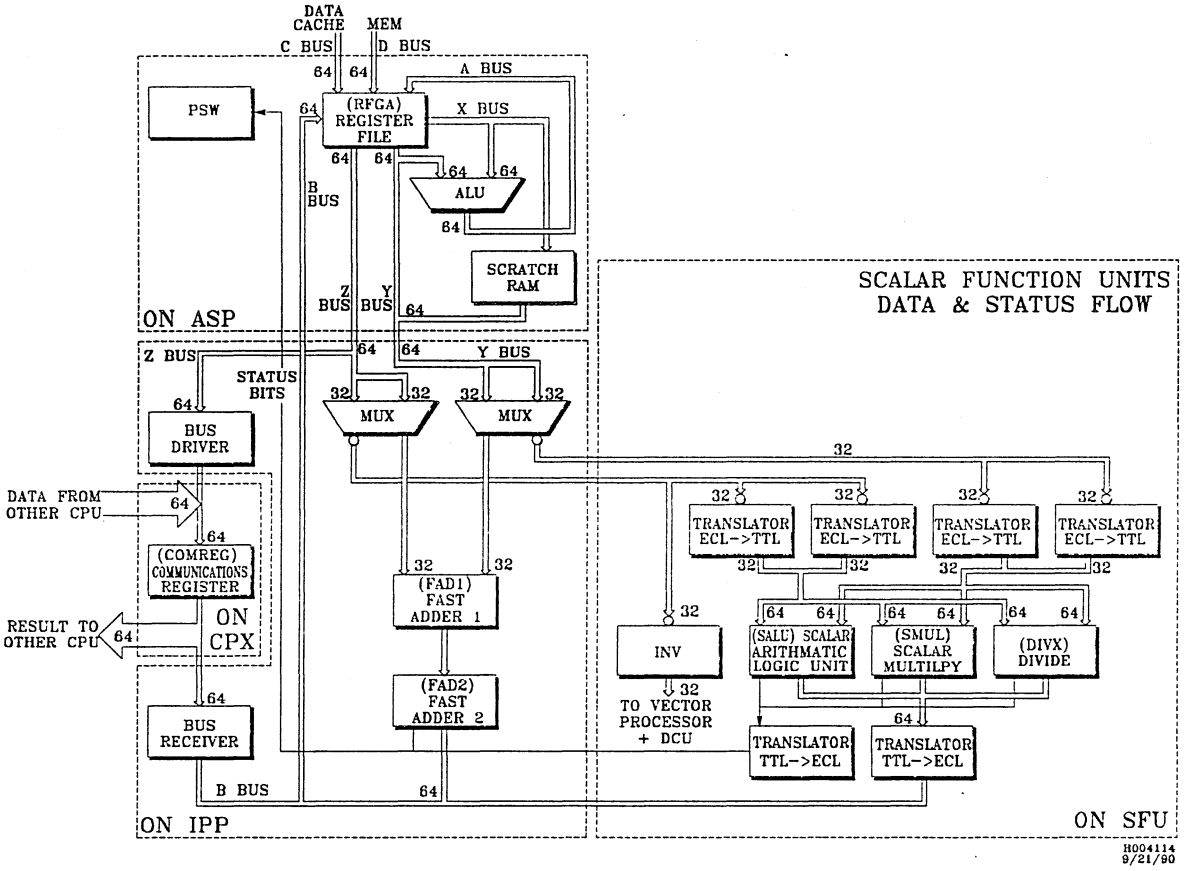
Chapter 3
Scalar Subsystem

3.1 Overview

The Scalar Processor (SP) executes the addressing functions for the SP, some addressing functions for the Vector Processor (VP), and all scalar arithmetic. Three boards complete the functions that the SP executes. These include the Address and Scalar Unit (ASP), the Instruction Pre-Processor (IPP), and the Scalar Function Unit (SFU). The SP processes scalar arithmetic operations, executes logical functions, generates memory addresses, and shifts data within the SP.

Figure 3–1 shows the functional block diagram of the C200 Series Scalar Processor Subsystem:

Figure 3–1, Scalar Processor Subsystem Functional Block Diagram



The following subsections review the SP operations and the circuits that execute them. Each subsection title includes the name of the circuit and the 2- or 3-character designator (in parentheses) of the board where it is located. See the card cage drawings at the beginning of chapter 1 for a list of the designators and their corresponding boards.

3.1.1 Register File (ASP)

The register file (RFGA), on the ASP board, includes the following registers:

- Eight address registers that are 32 bits wide
- Eight scalar registers that are 64 bits wide
- Eight temporary registers that are 32 bits wide

The register file receives 64-bit input data from four sources, the A, B, C, and D buses.

- The **ABUS** input to the register file carries a copy of the output from the ALU.
- The **BBUS** input to the register file shares data from the communication registers, the FAD gate array on the IPP board, and the SALU, SMUL, and DIVX gate arrays on the SFU board. The back door queue on the Logical Function Unit (LFU) tracks the information for the file, including the file location and the size of the result.
- The **CBUS** input to the register file carries the output from the data cache (Dcache). The parity of the Dcache output data is checked as it exits the Dcache.
- The **DBUS** input to the register file is derived in the following manner. Data from a memory read operation and some data from the VP return to the Data Flow Gate Arrays (DFW). The DFW gates the input from these sources, and checks the parity of the output. The DBUS then transmits this value to the register file.

The register file outputs information on the XBUS, YBUS, and ZBUS.

- The **XBUS** feeds into the ALU and the scratch RAM.
- The **YBUS** information combines with the input from the IP Dispatch to input data into the ALU.
- The **ZBUS** transfers information from the register file to the Fast Adder (FAD), on the IPP board.

3.1.2 Processor Status Word (ASP)

The Processor Status Word (PSW) is a 32 bit status register which contains flags to enable or disable exception processing and to show the results of numerical operations. The PSW register indicates the status of operations performed on both scalar and vector processors. Refer to the Reference section in this manual for a definition of the PSW bits.

3.1.3 BBUS Control (SFU)

Five sources can output data to the Register File along the BBUS. The communication registers and FAD, SALU, SMUL, and DIVX gate arrays share the BBUS to transport data to the register file. The control for the BBUS is part of the Logical Function Unit (LFU). The LFU also maintains the queue logic that maintains the information for the register file.

The SALU, SMUL, and DIVX circuits, on the SFU board, go through two levels of arbitration. These three circuits are CMOS gate arrays that share a common output bus which goes through the IPP board to the register file on the ASP board. The gate array controllers request use of the bus. An arbitration state machine tracks which gate array may use the bus. The output to the bus must be stable for at least two clocks; therefore, the arbitration state machine must enable output a clock before the results can enter the register file.

The output from the first arbitration must then contend with the FAD and the communication registers for the BBUS. Programmable Array Logic (PAL) determines which circuit can output data to the register file. Both arbitration schemes are based on priorities.

Table 3–1 lists the sources that share the BBUS to the file, in order of their priorities:

Table 3–1, BBUS Arbitration Control

PRIORITY	DATA SOURCE
1	Communication Register
2	DIVX gate array
3	SMUL gate array
4	SALU gate array
4	FAD gate array

The LFU also maintains the back door queue that contains the register file location and result size. The back door queue may hold nine operations for the register file while the file is accepting one. The communication register read operations take two clocks to complete. Therefore, the register can have one item in the queue. The FAD and DIVX gate arrays have an input pointer to their next operation, and an output pointer to the queue information. So, they may each have two items in the back door queue. The SALU and SMUL gate arrays have a FIFO arrangement with the queue. When one of those controllers opens the input latch, it also loads the one register of the queue. When it opens the output latch, it loads another register. This back door queue allows faster operations, with a higher priority, to bypass slower operations, with a lower priority. The lower priority unit then continues to request the bus to output its data.

3.1.4 Scratch RAM (ASP)

The scratch RAM, on the ASP board, has 1024 locations, 64 bits wide plus parity, that stores three types of data. The ASP board precomputes masks and constants and stores them in scratch RAM. For example, system calls require certain bits to be masked. The scratch RAM provides the constants for these operations. It receives its input from the register file, through the XBUS.

The scratch RAM also stores some of the hardware fault context during page faults. The hardware context contains the contents of all registers, including PC. This hardware context is stored to memory when a fault occurs. Thus, the context saved during a fault is larger than for a regular system call or ring crossing, since a fault usually occurs in the middle of an instruction.

These context values are transferred to the scratch RAM from the context registers. Shift rings 32 bits long and containing 32 bits save the ASP board's group of bits. They are shifted to scratch RAM, then stored to memory as part of the machine state.

To resume execution of a faulted program, context is loaded from memory into scratch RAM, then shifted back into the context registers so the registers are exactly the same as before the fault. The ASP board signals a jump address with the saved PC. The instruction stream then resumes execution.

The scratch RAM also holds intrinsic constants for microcoded intrinsic operations. The microcode handles intrinsic operations and intrinsics require a large amount of microcode to complete. No hardware array executes these operations directly. They are, therefore, of a higher level than add/sub/mult/div. Intrinsics include sine, cosine, and tangent operations. Parity is generated and checked for scratch RAM data.

3.1.5 Arithmetic and Logic Unit (ASP)

The Arithmetic and Logic Unit (ALU), on the ASP board, is implemented through ALU gate arrays. Input arrives from the register file along both the XBUS and the YBUS. The ALU then executes the integer arithmetic operations including addition, subtraction, and conversions, and logicals including ANDs, ORs, and EXCLUSIVE ORs. Output transfers from the ALU, along the FBUS, to become input to the register file.

Some memory addresses also pass through the ALU. The ALU generates both Alpha (Alpha) and Alpha plus eight (Alpha8) addresses for the Memory Interface (MI) processes. The Alpha address is the base address of an operand. The ALU adds eight to the base address (Alpha) to derive the Alpha8 address, and transmits both the Alpha and Alpha8 addresses along the FBUS to the Swapper. Bits <63..32> hold the Alpha8 address, and bits <31..0> hold the Alpha address. If the first operand in a longword accesses odd memory, the Alpha8 bits of the FBUS hold the address for even memory. Otherwise, the MI discards the Alpha8 address. Refer to the Alpha Addressing section of the Memory Interface section.

3.1.6 FAD Fast Adder (IPP)

The FAD fast adder, on the IPP board, consists of a pair of gate arrays, FAD1 and FAD2. The gate arrays pipeline data from the register file to allow one floating point add or compare operation per clock. It also is able to hold two results while the FAD controller requests the use of the BBUS to output the FAD results.

Table 3–2 lists the state of the controller, which latches in the gate array are active during that state, and the overall condition of the FAD during that state:

Table 3–2, FAD Control States

STATE	FAD REQUEST FOR BBUS	MIDDLE LATCH	INPUT LATCH	BUSY
S0	0	0	0	0
S1	1	0	0	1
S2	1	1	1	1
S3	1	1	0	1

3.1.7 SMUL Multiplication (SFU)

The SMUL controller, on the SFU board, handles scalar multiplication. It controls the SMUL gate array, and controls the internal latches during longword and double multiplication. The controller also tracks the time of an operation as it propagates through the gate array. The SMUL can hold two results while it requests use of the BBUS to output results. The SMUL receives input from both the YBUS and the ZBUS from the register file.

3.1.8 DIVX Divide and Square Root (SFU)

The DIVX controller, on the SFU board, contains two dividers and one opcode latch. A register external to the DIVX holds the last opcode transferred to the DIVX, and a comparator determines if the current opcode is the same as the previous opcode. If the opcodes are the same, the second divider can begin an operation before the first divider completes its operation. The DIVX receives input from both the YBUS and the ZBUS from the register file, and outputs data to the register file by the BBUS.

The DIVX gate array runs at half the clock speed (80 ns) as other system logic. Two external state machines synchronize the DIVX status with the rest of the machine. One state machine notes that DIVX is either busy or can execute a new operation. To begin an operation, the DIVX controller must signal a start to the DIVX, two system clocks before the DIVX clock edge. The second state machine tracks when the DIVX outputs results on the BBUS. The DIVX controller signals that DIVX output is valid, two system clocks before the DIVX clock edge, and shows, two signals later, that the contents of the circuit is no longer valid.

3.1.9 SALU Conversions and Floating Point Subtraction (SFU)

The Scalar Arithmetic and Logic Unit (SALU), on the SFU board, controls the gating into and out of the SALU gate array. It also tracks the time an operation requires to propagate through the SALU. One operation passes through the gate array at a time, but the SALU can hold two results while it requests the use of the BBUS. The SALU receives input from the YBUS and the ZBUS, from the register file.

3.2 Instruction Processor

The Instruction Processor (IP) retrieves instructions from memory, decodes those instructions, and dispatches them. The IP uses lookahead logic to request instructions from memory. Instructions transfer from memory through the memory data register to the pre-crack, which begins to decode the instruction. The Icache holds these instructions until the IP needs them.

The IP also determines whether the SP, the VP, or both, process the instruction. The align circuit positions the halfwords of an instruction for the post-crack and dispatch circuits to decode. The dispatch circuit then sends that instruction to the correct processor(s). The information which the IP transfers includes the instruction and its microcode starting address.

The IP directly executes only no-op, jump, and conditional and unconditional branch instructions, using circuits on the ASP and IPP boards. It maintains and updates the program counter based on registers that contain jump and branch addresses.

The following subsections review the IP operations and the circuits that execute them. Each subsection title includes the name of the circuit and the 2- or 3-character designator (in parentheses) of the board where the circuit is located. See the card cage drawings at the beginning of Chapter 1 for a list of the designators and their corresponding boards.

3.2.1 Instruction Cache (IPP)

The Instruction Cache (Icache), on the IPP board, contains 8 Kbyte locations of memory. Each location maintains 108 bits of information. The Icache holds instructions until they transfer to the align and pre-crack circuits.

Table 3–3 lists the information and corresponding number of bits in each Icache location, and the register that adds that information to the Icache:

Table 3–3, Icache Memory Allocation

NUMBER OF BITS	ICACHE INFORMATION	REGISTER
64 8	Data from memory Parity bits	Memory data register
19 3	TAG TAG parity bits	Write address register
12 2	Pre-crack output Parity bits	Pre-crack

The virtual address bits <12..3> addresses the lookahead cache at the same time it addresses the Icache. The virtual address bits <31..13> are the tag value and identical copies are placed in the Icache and the lookahead cache. Refer to the Lookahead Cache in this chapter for more information.

Either the jump address register, the next address register, or the branch address register can contain the source address for the Icache during a read.

The address fields are tested by the lookahead cache to find if the instruction is in the Icache. If the Icache does not contain the proper instruction, it must wait until the lookahead logic can bring the instruction from memory.

Once the instruction is in the Icache, the 108 bits of the Icache location are read and latched to a register on the Icache output. The Icache is read twice if the instruction crosses a longword boundary. The lookaside register holds the information from one read instruction. All output is checked for parity errors.

When a process changes, the information in the Icache becomes invalid and any new data writes over the cache locations. The Icache Valid columns show whether data in the cache is valid or invalid.

3.2.2 Instruction Cache Operations

Each clock, the Instruction Cache (Icache) is read twice and possibly written once. To allow these three operations to take place in one clock, the Icache data RAMs are double cycled. On the first half clock of a system clock the RAMs are read. On the second half clock the RAMs are written if it is necessary to write the RAMs. The two reads are necessary to allow instruction dispatch and instruction lookahead to both occur at the rate of one per clock.

To allow two reads in the first half clock, two copies of tag and validity are maintained. Each copy is independently addressable so that in the first half clock each copy can be used to read a different location.

3.2.2.1 Instruction Dispatch

On the first half clock of each clock, one copy of Icache tag and validity and the Icache data is addressed with the address of the next instruction to be executed. This address is selected from the branch or jump address if a branch or jump instruction was just executed or is based on the previous instruction executed (PC).

The data read on the first half clock is then latched at half clock so that the data is available for the rest of the clock cycle since the addresses to the Icache RAM change during the second half clock to allow the cache to be written. This data is checked for correct parity and a hard error generated if the parity check fails. The tag read from the cache is compared to the upper bits of the address of the current instruction. The validity is also checked for this cell. If the tags match and the cell is marked valid, then the data read out of the cache corresponds to the data needed to execute the instruction.

The Icache actually contains an A and a B column of validity, only one of which is in use at any one time. When one column is in use, the other column is either being written all invalid or has already been written invalid. When the ASP purges the Icache, the current validity column is switched so that the column in use is the entirely invalid column and the old column is cleared in preparation for the next purge. It takes 1024 clocks to purge all the cells in a column. Therefore, if purges are issued within 1024 clocks of each other, no validity columns will be available and the Icache will be unable to operate until a column is ready.

The actual instruction to be dispatched is selected from the instruction data read from the cache on the current clock and the data in the look-aside-buffer (LAS). The Icache can only be addressed in aligned longword blocks. If an instruction crosses a longword boundary, it is necessary to take the instruction data from two aligned longword reads. The LAS preserves the contents of the previous longword read so that such instructions can be properly dispatched.

The halfword containing the instruction is selected by the low order bits of the current instruction's address and is selected from the four halfwords in the LAS and the four half words in the Icache read latches. This selection is done by the aligner. The selected instruction is then processed to produce an entry point, register fields and displacement field (if any). This information is then registered in the instruction dispatch registers on the IP.

An instruction can be dispatched to the ASP, the VP or both. Instructions dispatched only to the VP deal exclusively with vector register operations. Those instructions dealing with vector and memory or that pass data between the VP and scalar processor are dispatched to both. The remainder of the instructions are dispatched only to the scalar processor. The VP and ASP each send a ready for dispatch signal to the IP. The IP, in turn, sends a dispatch signal to the ASP and VP when it has dispatched an instruction. If an instruction is to be dispatched to both VP and ASP and one is ready while the other is not, the IP will dispatch the instruction to the ready unit immediately and dispatch it to the unready unit when it becomes ready.

The microcontroller can inhibit dispatches from the IP.

3.2.2.2 Branches and Jumps

Branches and jumps alter the sequential flow of the instruction stream. Both branches and jumps can be conditional or unconditional. Branches and jumps each use the same conditions. The two differ in the manner in which they are executed and in the computation of target address.

The following discussions of branch and jump timing assume the target instruction is found in the Icache. If the target is not in the cache, it must of course be fetched from memory.

The target of a jump address is computed in the effective address computation used for loads and stores. The contents of a register (or zero) are added to a 16- or 32-bit displacement. The microcontroller examines the jump condition to determine whether or not the jump is to be taken. If it is, it tells the IP to take a jump and passes it the jump address. The IP then loads the jump address as the new PC and then continues.

Jumps take two to five clocks to execute depending on whether the jump is conditional or unconditional and the address of the target of the jump. Unconditional jumps take three or four clocks. Conditional jumps that are not taken take two clocks. Conditional jumps that are taken take four to five clocks.

Unconditional jumps progress as follows:

1. The IP dispatches the jump instruction to the ASP.
2. The microcontroller determines that the jump is to be taken. It tells the IP to take a jump and puts the jump address on the AS-IP.JMP_ADDR bus.
3. The IP loads the jump address at the beginning of this clock and applies the new address to the Icache.
4. If the jump address indicates an instruction which is completely contained within the longword read on the previous clock, the IP dispatches the instruction at the target of the jump. In either case, it also reads the next longword following the jump on this clock.
5. If the target of the jump crossed an aligned longword boundary, the target instruction is dispatched on this clock.

Conditional jumps progress in this manner:

1. The IP dispatches the jump instruction to the ASP.
2. The microcontroller executes a conditional microbranch which will test the condition required. If the jump is not to be taken, no further action is taken. The IP is inhibited from dispatching by the ASP during this clock.
3. If a jump is to be taken, the IP is still inhibited, the IP is told to take a jump and the jump address is put on the AS-IP.JMP_ADDR bus. If the jump was not taken, the IP will dispatch the instruction following the jump on this clock.
4. The IP loads the jump address at the beginning of this clock and applies the new address to the Icache.
5. If the jump address indicates an instruction is completely contained within the longword read on the previous clock, the IP dispatches the instruction at the target of the jump. In either case, it also reads the next longword following the jump on this clock.
6. If the target of the jump crossed an aligned longword boundary, the target instruction is dispatched on this clock.

For jumps that are not taken, two clocks elapse from the dispatch of the jump to the next instruction dispatch. For jumps that are taken, four to five clocks elapse before the next instruction dispatch.

Unlike jumps, in which determination of next address is done by the ASP, branches are computed internally to the IP. For this reason they take less time to execute. Branch targets are computed by adding a signed eight bit displacement to the current PC. Therefore, they are limited to changing the PC by +127 to -128 halfwords. Jumps can go directly to any logical address.

When the IP detects a branch instruction, instead of dispatching that instruction, it puts out a branch condition select code on the entry point to the ASP. The entry point normally selects the starting address of the next microinstruction sequence; in the case of a branch, it selects which condition is to be tested for the branch. At the end of the clock that the branch select was placed on the entry point, the selected condition is registered along with a hazard signal from the ASP which tells the IP whether it can use the branch condition. If a hazard is asserted, the IP must wait until the hazard goes away. The IP computes the branch target address by adding the signed, eight bit displacement to the current PC.

The branch will take two or three clocks, as follows:

1. Instead of dispatching the branch instruction, the IP puts out the branch condition select code on the entry point bus to the ASP. At the end of this clock, the IP registers the branch condition and a branch condition hazard. The IP also registers the branch target address at the end of this clock which it has internally generated.
2. Regardless of whether the branch is to be taken, the IP reads the aligned longword containing the address of the branch target. The LAS is held and will contain at least the first halfword of the instruction to be executed if the branch is not taken. After the Icache read, the Icache read latches will contain at least the first halfword of the instruction to be executed if the branch is taken.
3. If the branch is taken and the branch target instruction falls completely within the previously read longword, the branch target instruction will be dispatched. If the branch is not taken and the next instruction is contained completely within the LAS, it will be dispatched. The aligned longword following the current instruction is now read from the Icache.
4. The branch target is dispatched if it was not contained in an aligned longword. If the branch was not taken, the following instruction is now dispatched if it was not contained entirely within an aligned longword.

From "dispatch" of the branch (branch condition selection) to dispatch of the next instruction either two or three clocks elapse regardless of whether the branch is taken.

3.2.2.3 Traps and Interrupts

The IP dispatches traps and interrupts in a manner similar to instructions. It detects the occurrance of traps and interrupts from the various sources and then dispatches an entry point corresponding to the appropriate trap or interrupt. Since the ASP must accept a trap or interrupt as it would any normal instruction, the ASP will not take them until it is finished with the current instruction. Therefore a trap or interrupt will not interrupt the micro-sequencer as a fault from the DCU can.

Traps are generated by arithmetic exceptions in the scalar processor or vector processor, deadlock or idle conditions, among other conditions. Interrupts are detected by the CPX and dispatched as a trap by the IPP.

Interrupts are registered by the CPX in an interrupt pending register. When the CPX has an interrupt to send to the IP, it will assert a request signal and wait for the IP to give it an acknowledge before clearing the interrupt from its pending register.

3.2.2.4 Instruction Lookahead

The look-ahead function reads the second copy of Icache tags and validity. All copies of tag and validity are identical. The two copies exist so that the instruction dispatch and the look-ahead can each read a copy of tag and validity on the first half clock of each system clock. No copy of instruction data is maintained for the lookahead read since it only needs to determine whether the data is in the cache but does not actually do anything with the Icache data.

The address used for the lookahead read is the R.LA address. This address is closely related to the current program counter. The lookahead process is intended to examine the eight or nine longwords in the Icache following the current PC and request any longword not in the cache. Each clock, the cache is checked to see if the current lookahead address is valid in the Icache. If it is the lookahead address is incremented. If it is not, that address will be requested from memory. The IP will not increment the look-ahead address until its request is accepted.

Lookahead requests are not allowed to cause a fault since the requested data may not actually be needed by a program. If the memory interface determines that the IP's request would fault, the request is instead ignored.

When the memory interface accepts IP requests, it saves the logical address of the request in the return control queue. When the return controller informs the IP that memory data is ready for it, it also gives it this logical address so that the IP knows where to put the data.

If the lookahead address is eight longwords ahead of the PC, the address is not incremented. If it is eight longwords ahead and the instruction data needed to execute the current instruction is not in the Icache, the lookahead address is reset to the current PC which will lead to a request for that address being made. If the IP has twice asked for the data necessary to execute an instruction with out receiving that data, it will make a request which is allowed to fault.

> **NOTE**
>
> The lookahead must run to eight longwords ahead of the PC, twice, before issuing the faulting request.

On a jump or branch, the lookahead address is immediately loaded with the new address.

Memory requests from the IP can be inhibited by the ASP.

3.2.2.5 Writing the Icache

Each clock, the IP registers AS-BP.DATA, FU-IP.MMQA1, and FU-IP.MXI_RDY. Registered MXI_RDY tells the IPP that read data has returned for it. If MXI_RDY is true than AS-BP.DATA held return data from memory and MMQA1 was the logical address of the return data. Using the registered MMQA1 address to address and tag the Icache the return data will be written into the Icache and marked valid. Both copies of tag and validity are written identically; they must always be equal.

The return data is blocked into halfwords and each halfword is precracked to decode size and extended information. This information is decoded regardless of whether each halfword is actually the beginning of an instruction since that information is unknown at the time the data is written.

There is no synchronization between IP requests and the IP return data. Since the normal IP request is not allowed to fault, it may never get return data for some of its requests. Therefore, it simply accepts all data which the return controller on the SFU indicates is for it and writes it into the Icache. Since the request logic maintains no history of previous requests, it is possible for it to have two requests for the same piece of data outstanding, although this event is unlikely.

Only the validity column currently in use is written valid. The other column is the one that is either being purged or has already been purged. If it is being purged, the validity cell addressed by the current purge count is marked invalid.

Both return data write and purge write are done on the second half clock of each system clock.

3.2.3 Icache Valid A and Valid B (IPP)

The Icache valid A and valid B are two single-bit columns that determine if the information in the Icache is valid information for the instruction that is being processed. When the IP writes information to an address in the Icache, it also sets the value in the corresponding valid column to one. For example, when the IP writes to the first address in the Icache, it places a one in bit <0> of the valid column; when the IP writes to location two, it also places a one in bit <1> of the valid column, etc. If the system has not accessed the IP since the system booted, or if the information in the cache is invalid, the valid bits are zero; if the information is valid for the instruction, the values are one.

Alternating sets of processes switch to alternating valid columns. If the system begins executing operating system code, or if a process completes execution and another process begins, then the data in the cache is no longer valid, and the control switches from one valid column to the other. However, if an instruction requests data from a disk, the system faults until that data is available. The IP waits until the data returns from the disk. System faults do not cause a change of process and therefore the IP continues to use the same valid column.

When the control transfers, purge circuits clear to zero all values in the previous column. If control switches again before the column is clear, the system must wait until the purge circuitry clears all values in that column. Purge circuits clear one value per clock. The IP requires 1024 clocks to clear a column to zero.

3.2.4 Lookahead Cache (IPP)

The lookahead cache, on the IPP board, ensures that instructions are in the Icache for execution. It eliminates the possibility of an instruction waiting for data to return from memory. This cache receives a duplicate copy of the tag and tag parity bits of an instruction that the IP places in the cache.

The lookahead cache uses the address from the lookahead address register. Bits <31..13> of the address field are compared to the tag value of the lookahead cache; if the bits match the tag value of the lookahead cache, the information in the Icache is the proper data.

If the longword is in the Icache, the lookahead address register increments by eight bytes and the process repeats. If the instruction is not in the Icache, the Data Cache Unit board (DCU) requests from memory the longword the lookahead address register addresses. If memory is not ready for a request, the process repeats without loading the lookahead address register until the longword is in the Icache or memory becomes ready for the request.

The lookahead address register may access up to eight longwords from memory. The IP can track the number of instruction by subtracting the value of the register from the value of the program counter. The range may increase to 16 longwords or decrease to 4 longwords if the pal 1482 is reprogrammed.

3.2.4.1 Deadlock

Deadlock traps are sourced on the IPP and sent to the ASP. A deadlock condition is detected when all CPUs with equal CIR values are in a deadlock loop. The purpose of a deadlock trap is to keep a CPU from hanging while running a process that is waiting on a condition dependent on another CPU which is running the same process, and waiting on a condition dependent on another CPU, etc.

There is a set of instructions that are called deadlock instructions. A loop containing one of these can cause a deadlock trap. A deadlock loop is a two instruction sequence of code containing a deadlock instruction immediately followed by a branch back to the deadlock instruction.

Figure 3–2 shows a deadlock loop:

Figure 3–2, Deadlock Loop

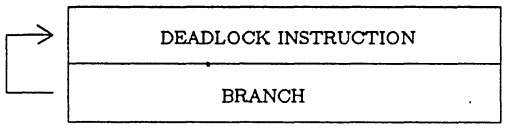
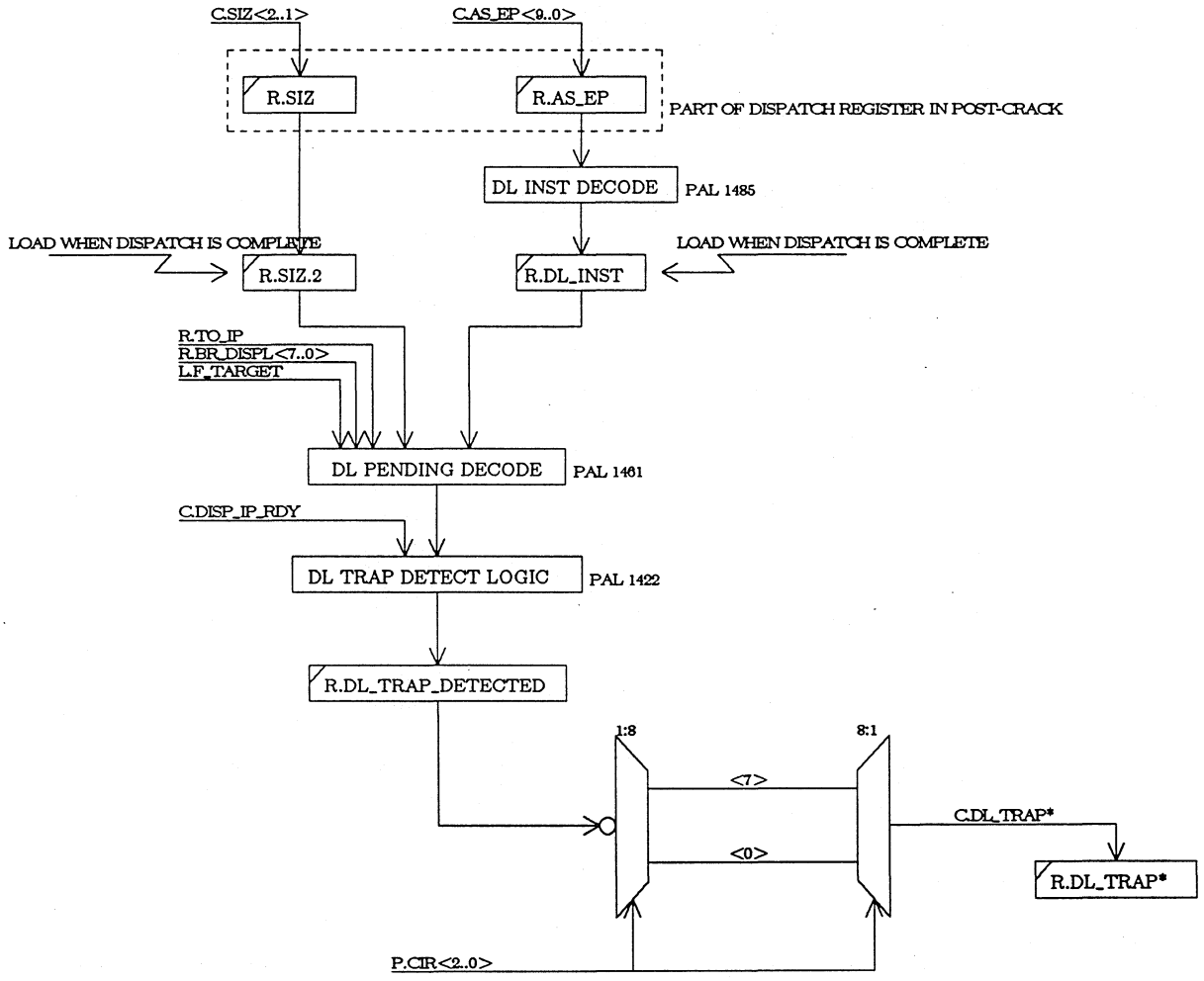


Figure 3–3 shows the deadlock block diagram:

Figure 3–3, Deadlock Block Diagram



3.2.4.2 Traps and Interrupts

The IP dispatches traps and interrupts in a manner similar to instructions. It detects the traps and interrupts from various sources and dispatches an entry point corresponding to the appropriate trap or interrupt. Since the ASP board must accept a trap or interrupt as it would any normal operation, it does not take them until it completes its current instruction. Therefore, a trap or interrupt does not interrupt the micro-sequencer as a fault from the DCU board can.

Two possible causes of traps include SP/VP arithmetic exceptions and deadlock/idle conditions.

The CPX detects interrupts and dispatches them as a trap by the IPP board. It registers the interrupts in an interrupt pending register. To send and interrupt to the IP, CPX asserts a request signal and waits for an acknowledge from IP before clearing the interrupt from its pending register.

3.2.4.3 Faults

When the DCU board ignores a memory request from the IPP board, though there is a proper handshake, the following occurs:

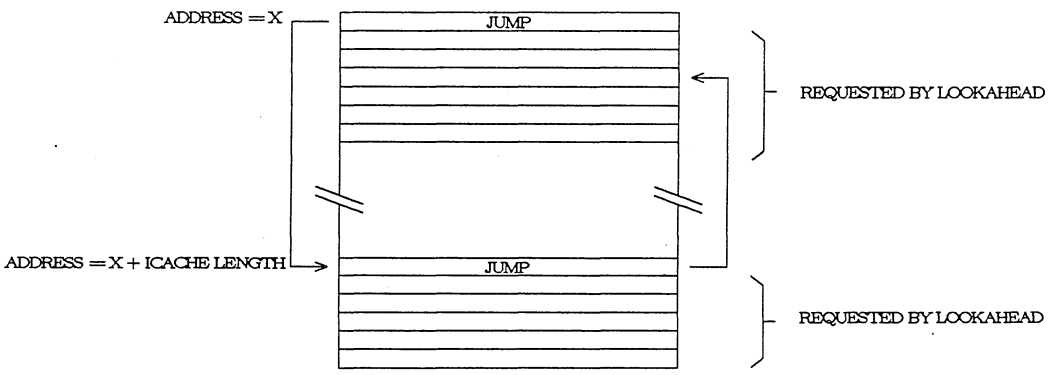
- The lookahead cache can not request another longword.
- The Icache does not contain the instruction.
- The DCU board's memory queue is empty.

Now the IPP board dispatch becomes locked and enables a fault state which remains active until the IPP board makes a faulting memory request to DCU, or the instruction becomes ready.

The IPP board dispatch also may lock up if a jump has an offset greater than the Icache length followed by a jump at least one longword from the original location. After the first jump the lookahead controller requests data associated with the new program counter. This new data may write over the Icache values that the IP needs after the second jump.

Figure 3–4 shows the jumps that could cause the dispatch to lockup:

Figure 3–4, Jumps That Cause Dispatch Lockup



If the second jump executes quickly after the first, the lookahead address register may read data before the new data it requested writes over what was in the cache. The lookahead address register would believe that the values in the cache are correct. The IPP board would make a faulting request as listed above. The lookahead address register must reset its value to equal the program counter, and the IP must rerequest the data from memory.

3.2.5 Lookahead Valid A and Valid B (IPP)

The lookahead valid A and valid B are two single-bit columns that determine if the information in the Icache is valid information for the instruction that is being processed. When the IP checks the tag value in the lookahead cache for the proper data, it also compares the value in the Lookahead valid column for a one.

For example, when the IP reads the first lookahead cache location and finds that data in the Icache is proper data, it also compares the value in the valid bit <0> for a one. If the system has not accessed the lookahead cache since the system booted, or if the information in the cache is invalid, the valid bits are zero. If the information is valid for the instruction, the values are one.

Alternating sets of instructions switch to alternating valid columns, e.g., if a process completes execution and another process begins, then data in the cache is no longer valid, and the control switches from one column to the other. However, if an instruction requests data from a disk, the system faults until that data is available. The IP waits for the data to return from the disk. System faults do no cause a change of process, and therefore the IP continues to use the same valid column.

When the control transfers, the purge circuits clear to zero all values in the previous column. If control switches again before the column is clear, the system must wait until the purge circuitry clears all values in that column. Purge circuits clear one value per clock. The IP requires 1024 clocks to clear a column to zero.

3.2.6 Pre-Crack (IPP)

The pre-crack is a circuit which decodes instructions before the Icache stores them in one of its locations. Before information can transfer to the pre-crack, the return queue on the DCU board must send that data to the memory data register.

The pre-crack then receives the data from the memory data register and decodes each halfword as if it were the opcode of the instruction. For each halfword, the pre-crack adds three bits, two bits for the instruction size and one for the extend prefix.

For the four halfwords in an instruction, the pre-crack adds two parity bits. The pre-crack then sends the data it received plus the 14 bits it generated to the Icache for storage before processing.

3.2.7 Memory Data Register (IPP)

The Memory Data Register (MDR), on the IPP board, receives 64 bits of data plus eight parity bits from memory. This data transfers from the data flow gate array when the memory control sends the instruction address from the return queue to the write address register. The MDR then holds this information for the pre-crack, which decodes the longwords. The return queue must send data through the MDR for the pre-crack to decode. The IP tests the parity of the data leaving the MDR.

Table 3–4 shows the layout of the data and the parity bits:

Table 3–4, Memory Data Layout

HALFWORD	DATA BITS	DATA BYTE	PARITY BIT
0	<63..56>	0	0
0	<55..48>	1	1
1	<47..40>	2	2
1	<39..32>	3	3
2	<31..24>	4	4
2	<23..16>	5	5
3	<15..08>	6	6
3	<10..00>	7	7

NOTE

Any request by the Memory Interface is also gated by the Data Flow Gate Arrays to the Memory Data Register. It is placed in the Memory Data Register, on the IPP board, whether it is intended for the Instruction Processor (IP) and parity is checked. Therefore, an IP parity error may be displayed for any incorrect data from memory.

3.2.8 Write Address Register (IPP)

The write address register, on the IPP board, contains 32 bit virtual addresses corresponding to the address of data in memory. The register receives these addresses from the return queue on the SFU board. The addresses transfer to the write data register when data returns from an IP request to memory.

Bits <12..3> are used to access locations in both the Icache and the lookahead cache. Bits <31..13> are the tag bits that these cache store to compare during an Icache read operation.

3.2.9 Jump Address Register (IPP)

When the ASP board dispatches a jump instruction to the IPP board, the ALU combines the instruction with output from the register file to create a jump address. The SP then loads this value into the jump address register, on the IPP board. The jump address register then holds the address of the next instruction. The SP forces the jump address register to be chosen as the source to update the contents of program counter.

If the jump address indicates an instruction which is completely contained within the longword read of the previous clock, the IP dispatches the instruction at the target of the jump. In either case, it also reads the next longword following the jump on this clock. If the target of the jump crossed an aligned longword boundary, the target instruction is dispatched on this clock.

Table 3–5 lists the forms of a jump instruction and the time each form takes to execute. The times to execute depend on the address of the target jump.

Table 3–5, Jump Instruction Execution Times

JUMP INSTRUCTION	EXECUTION TIME
Conditional jump not taken	Two clocks
Conditional jump taken	Four to five clocks
Unconditional jump	Three to four clocks

3.2.10 Branch Address Register (IPP)

When the post-crack decodes an instruction as a branch instruction, the IP loads the branch instruction address into the branch address register. The branch address register becomes the source to update the contents of the program counter. If the branch is conditional, the ASP board determines whether to branch.

When the IP detects a branch instruction, it places a branch condition select code on the entry point to the ASP board. The entry point normally selects the starting address of the next microinstruction sequence; for a branch, it selects which condition to test for the branch. At the end of the clock that the IP placed the branch select on the entry point, the selected condition is registered along with a hazard signal from the ASP board which tells the IPP board whether it can use the branch condition. If a hazard is asserted, the IP must wait until the hazard goes away. The IP computes the branch target address by adding the signed, eight bit displacement to the current program counter.

Since the IPP board computes the address of the next instruction, branch instructions take less time to execute. Since the IPP board adds a signed eight bit displacement to the current program counter to compute the next address, the operations are limited to changing the program counter by +127 to -128 halfwords.

3.2.11 Next Program Counter (ASP)

During a change of processes, the Next Program Counter (NPC), on the ASP board, is the source to update the contents of the program counter. The NPC generates most instruction addresses for all processes.

3.2.12 Lookaside Register (IPP)

When an instruction spreads over two longwords, the IP must access the Icache twice. The Lookaside Register (LASR), on the IPP board, holds the output from one read. The information in the LASR includes the opcode halfword of the instruction being fetched and possibly all of the instruction. If the instruction is a branch, the LASR holds the longword of the instruction immediately following the branch instruction.

The latched output of the Icache and the information from the LASR enter the Align circuit to have their data extracted. If the align circuit does not use all bytes of a longword for an instruction, the LASR reloads the longword for future processing.

3.2.13 Align (IPP)

The align circuit receives longwords and organizes the bytes into an executable instruction. An instruction is unaligned if its address does not fall on a word boundary. If the Icache contains the instruction within one longword, the align circuit accepts the longword and separates any unnecessary information.

If an instruction extends over two longwords, the align circuit receives one longword from the Icache and one from the lookaside register. It then removes the halfwords that belong with another instruction and aligns the remaining information into one longword for execution.

3.2.14 Post-Crack and Dispatch Register (IPP)

The post-crack body uses the displacement decoder to complete decoding the output, four halfwords, from the align circuit. It then registers the information to dispatch the instruction.

To dispatch an instruction, the IPP board must know the following:

- Which processor should receive the instruction
- Whether the instruction could cause a vector valid trap
- If the processor status word, on the ASP board, should have a hazard active when executing the instruction

3.2.15 Hazards

Hazard locking is a block in the flow of execution. The ASP board reserves a register until the hazard no longer exists. The system can not go past a certain point of execution until the hazard ends.

The ASP board executes hazard locking for any instruction that takes longer than one clock, which leaves the ASP board to go to the SFU board, memory, or the Data Cache. The ASP board reserves a register with a set of bits and when data returns, clears the hazard by clearing the bits to zero.

If the next instruction requires data that is not in a register, the ASP board stops until the SFU board writes the data there. If the ASP board removes control lines, the blocking can reflect throughout the system.

Hazard locking prevents the ASP board from jumping ahead of itself while it executes instructions. It can stop processes that are running, until the data is available. This also means that the ASP board can continue processes that do not depend on the data. Still, the ASP board spends 25% to 50% of its clocks waiting for data to return so it can execute the next instruction.

Each thirty two bit register has a hazard bit associated with it (for the sixty four bit scalar registers, a hazard bit is maintained for both the upper and lower thirty two bits). These hazards are set when a microinstruction executes an operation which causes data to be written to that register at some point in the future. For instance, when the ASP board executes a load instruction, it makes a memory request to the memory interface. Eventually the memory data returns from the memory.

While the data has not returned, the ASP board can execute other instructions as long as it does not use the contents of the register where the memory data goes. The hazard prevents the register from being used until the data for it has returned. When the data returns, the hazard is cleared.

One register file gate array runs the hazards. Therefore, the ASP board has nine gate arrays, eight for the register data and one for hazards.

3.2.16 Context Bits

The IPP board controls four context bits and the program counter. The ASP board saves the value of the four bits and the program counter on the next rising system clock edge. To restore the values, the IPP board load the four control bits. The ASP board restores the program counter by giving a take jump command with an address equal to the previous program counter value.

3.3 Memory Interface

The Memory Interface (MI) controls transferring instructions and data between memory and the Instruction Processor (IP), the Scalar Processor (SP), and the Vector Processor (VP). For the IP to request an instruction from the memory, it must transfer an instruction address from the lookahead cache, on the IPP board, to the address Swapper. The IP only requests aligned longword instructions from even memory.

Before the SP sends a request to memory, the ALU must create the Alpha and Alpha plus eight addresses. The ASP board transfers these values on the FBUS to the address swapper. The ASP board also provides the starting address for the first vector memory request and the Vector Address Generator (VAG) makes later requests.

> **NOTE**
>
> Any request by the Memory Interface is also gated by the Data Flow Gate Arrays to the Memory Data Register. It is placed in the Memory Data Register, on the IPP board, whether it is intended for the Instruction Processor (IP) and parity is checked. Therefore, an IP parity error may be displayed for any incorrect data from memory.

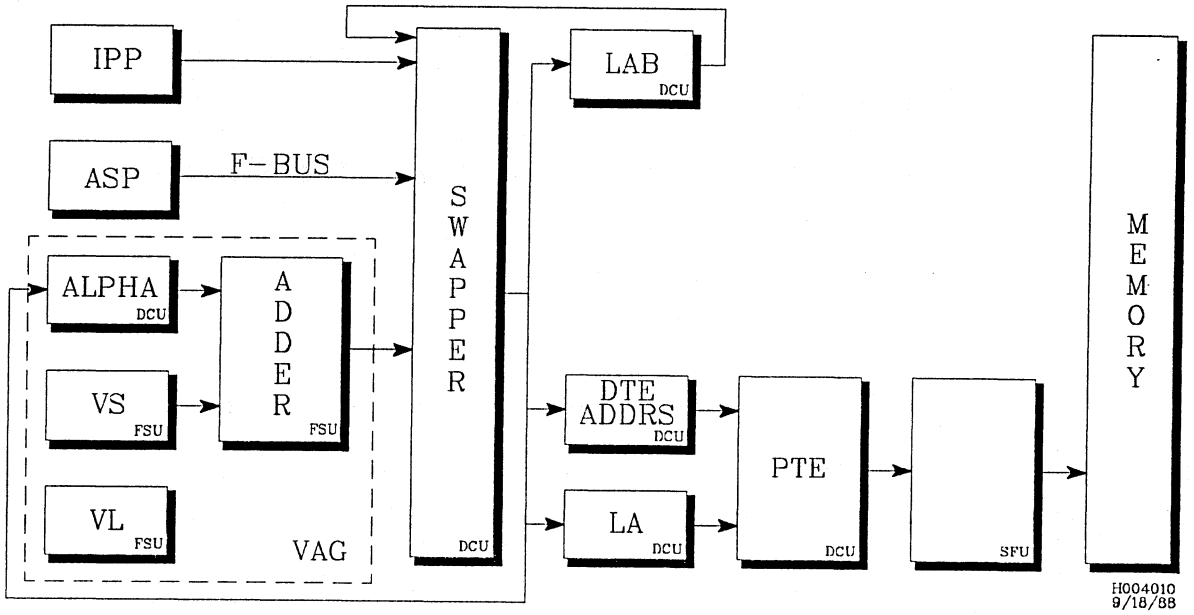
If the MI is processing a request, the next request is stored in the Look-Aside Buffer (LAB). On the rising clock the LAB returns the address to the swapper as the current request.

The swapper selects a request source address and sends a copy to the Look-Aside Register (LA), the Page Table Entry Cache (PTE), the alpha, and alpha8 registers in the vector address Generator. The LA register holds the logical address of the instruction.

After translating logical address to physical address, the PTEs transfer the data to the backplane to access the memory board.

Figure 3–5 shows the address paths to memory from each of the processors:

Figure 3–5, Memory Address Paths



The Memory Interface (MI) consists of the following:

- Nine data flow gate arrays—The Data Flow Gate Arrays (DFW) steer the data between the memory boards and the processors.
- Data cache—The Data Cache (Dcache) is a 4K cache (two 2K banks, one for even and one for odd memory). It holds a copy of data from most scalar read requests, so each memory request does not have to reaccess the memory boards. Data still maps from the memory boards to see if it is in the cache. When the SP makes a read request to memory, the data flow gate arrays place a copy of the data into the cache. Then, if the SP makes a second request for data at the same address, it can retrieve the data from the cache.
- Logical-to-physical address translation—Addresses entering the swapper and LA register arrive in a logical form. The logical address registers and the PTEs map these addresses to their physical form, which the memory can recognize. The vector address generator, on the DCU and SFU boards, generates addresses for the VP.
- Memory Control—The memory control, on the DCU and SFU boards, coordinates the access of memory by the vector, scalar, and instruction processors. Both the VP and the SP read from and write to memory, while the IP requires only read memory access. The return controller handles the handshakes between the memory system and the unit receiving the data. Data actually goes to the data flow gate array.

The following subsections review the MI operations and the circuits that execute them. Each subsection title includes the name of the circuit and the 2- or 3-character designator (in parentheses) of the board where the circuit is located. See the card cage drawings at beginning of Chapter 1 for a list of the designators and their corresponding boards.

3.3.1 Data Flow Gate Arrays (ASP)

The nine Data Flow Gate Arrays (DFW), on the ASP board, contain the main MI data path. There is one DFW gate array for each data byte. The ninth DFW gate array handles parity for the eight data bytes.

The DFW does all memory data steering in the SP. All data transfers to and from memory pass through the DFW. The memory control decodes, from the memory request, which data to select from the store data queue on the DFW and how much to rotate that data. The DCU board controls the DFW as it selects the data, then rotates it. The rotator ensures that data bytes align correctly. Each byte always goes to a certain address on the memory cards.

> **NOTE**
>
> Any request by the Memory Interface is also gated by the Data Flow Gate Arrays to the Memory Data Register. It is placed in the Memory Data Register, on the IPP board, whether it is intended for the Instruction Processor (IP) and parity is checked. Therefore, an IP parity error may be displayed for any incorrect data from memory.

The gate arrays include a connection that sends data to the vector and the instruction processors; one bus is shared by both of the processors. The DFW outputs data to memory on a separate bus and inverts a copy of this data to transfer on another bus to the Dcache.

An additional bus returns data from the VPD board on the vector processor to the ASP board and this data can then be output either to memory or to the instruction processor. Data from the odd and even memory boards also share a bus to the DFW. The information on these buses includes 32 data bits plus parity bits.

One final bus transfers data that has returned from the VP along the DBUS to the register file, also on the ASP board. The DFW checks the parity of the data before it is transferred to the register file. This is the only bus connected to the DFW that is checked for parity.

> **NOTE**
>
> Data is not checked for parity after it arrives from memory, and before it outputs to the IPP board, the vector processor, or the Dcache.

3.3.2 Data Cache (ASP, DCU, SFU)

One half of the 4K byte Data Cache (Dcache) maintains locations for even addresses, while the other half has locations for odd addresses. Each half of the Dcache has 512 words (2K bytes). The data RAMs for the Dcache are physically on the ASP board though the registers on the DCU board address them. This design keeps the RAMs near the data path structure of the SP.

3.3.2.1 Dcache Data (ASP)

The Dcache data RAMs are 72 bit locations, eight bytes with one parity bit per byte. Though they are functionally part of the DCU board, the data RAMs are located on the ASP board. This allows the DCU board to transfer 18 address lines to the ASP board, rather than 72 data bits for each instruction. Nine address lines are for the even side of memory, and nine address lines are for the odd side of memory.

The Dcache is a directly mapped cache. Physical address bits <11..3> select the location for the Dcache. Any physical addresses with identical bits <11..3> access the same location. The Dcache data includes the physical address of the encached word. A read must first access the PTE cache to obtain the physical address of the request. Though the logical address may change, the physical addresses of memory do not change, so the Dcache does not have to invalidate every location when the mapping of the logical address changes.

3.3.2.2 Dcache Tags

Every word in the Dcache has a 20-bit tag (bits<31..12> of the physical address), plus 3 bits of parity, and physically includes the parity checkers and the comparators to detect whether data hits in the cache. This tag, along with the location address of the word in the cache, defines the address of that word. The remaining bits <2..0> are the odd and even select and the byte select values. This allows the Dcache to address independently even and odd data.

The Dcache maintains three identical copies of the tag to allow enough bandwidth for all Dcache activities. The three odd and even copies require 36 tag RAMs in the Dcache. An A copy and its address registers are on the DCU board. The remaining copies, B and E, and their registers and address multiplexers are on the SFU board. The local processor uses the A copy, while a remote CPU accesses the B copy, and a remote I/O system uses the E copy. The cache writes all three copies when it writes the data. Three copies gives the system the option to read three different locations simultaneously. Different tag data for the same cell is an illegal condition.

3.3.2.3 Validity Tags

The valid bits tell the system whether the data in the Dcache is a real copy of a word in memory. The validity RAMs, on the DCU board, are similar to the Dcache data RAMs. Both even and odd halves of the Dcache maintain sixteen RAMs which the processor addresses in groups of four bits. The processors may write independently on the each valid bit. Each valid bit has a parity bit.

Two bits mark the validity of each data byte. Both bits must be valid for a data byte to be valid. When the local processor writes data into the cache both bits receive the same value. When the processor attempts to read data from the cache it also reads both validity bits. A remote invalidate operation may mark either or both bits invalid.

3.3.2.4 Update Tags

The update tags determine if data returning from memory should update the cache. The cache maintains one four bit update tag plus four parity bits for each even and odd location within the cache. Each tag requires four RAMs, on the SFU board, which the processor can address independently.

The update tags prevent the cache being inconsistent. The inconsistency can occur if the Dcache allows old data to enter the cache. When a write operation occurs after the cache makes a read request but before the data returns, the data is obsolete. The system must assume that the write operation changed the data before it could reach the cache. They also prevent an update if two read operations which map to the same location occur before any data can return from memory. The data returning for the second read is the only data which can update the cache.

3.3.2.5 Dcache Operations (ASP, DCU, SFU)

The Dcache can perform four operations: read data, write data, update the data, remote invalidate. The read, or load, operation applies to scalar reads from the Dcache. The write, or store, operation automatically writes to memory when it writes to the Dcache. This is to keep the values in the Dcache and memory current. The update operation occurs when a read operation misses the Dcache. All scalar read and all write requests write the update tags during the first half of the clock. Any update reads the tags on the second half clock. A write (during the second half of a clock), or a scalar read and a remote invalidate (using both half clocks) postpone update operations.

Two operations may occur during a single system clock (40 ns) since the Dcache is clocked at twice the system clock rate (20 ns). Plus, the Dcache may execute three operations if two of them are remote invalidate operations. If two invalidates occur during the same clock as a read operation, the read occurs first; if they occur with a write, the write occurs last. Since there are three copies of the tag, the two invalidates and the request may read the tags simultaneously without interfering with each other.

Table 3–6 lists the combinations of Dcache operations and the order within the combination:

Table 3–6, Dcache Operations

FIRST CLOCK 20 NS.	SECOND CLOCK 20 NS.
Scalar read	---
---	Write data
Write update tag	---
---	Read update tag
---	Write update data
Remote invalidate read	---
Remote invalidate write	---
Scalar read	Remote invalidate write
Remote invalidate write	Scalar write

Since write operation uses the second half clock of the Dcache, they postpone current update operations until the next clock. A scalar read and a remote invalidate together use both half clocks of the Dcache and also postpone current update operations. The Dcache does not encache any data that returns from memory while an update operation is waiting.

A read operation begins when the memory control receives a request with a read memory opcode. During a Dcache read, the memory control compares the location tag values with the physical address of the request and checks the validity of the bytes which the request requires. If the tags match and the validity is true for the data, the memory control writes the data from the Dcache into the SP register file. It does not have to access memory.

All write operations issue directly to memory. For any scalar write operation, the Dcache encaches the data. However, for a vector write operation, the SP checks to determine whether the data is already stored in the cache. If the vector data is in the cache, the processor marks that location invalid.

The update operation is the process of placing return data in the Dcache if the update tags allows it. An update occurs when the result of a read request misses the Dcache. When a request is eligible to come from the Dcache but is not in the cache, the SP issues the request to memory. The data that returns is updated into the cache so the next read request for that data does not have to access memory.

The process of updating the Dcache with return data is a two or more clock process. The update holding registers hold all the information to encache the return data. The DFW holds the return data while the SFU board holds the additional information.

A remote invalidate occurs when a remote processor makes a write request to a memory location in the local processor's Dcache. The local cache invalidates the entry to maintain consistency between itself and memory. The operation uses the physical addresses of remote processors without mapping the address to the local processor's logical mapping.

When another head executes a write operation, the local processor checks if the write corresponds to a value in its cache. The local processor uses the B and E tag copies to compare Dcache tags with the physical address of the write. If the address of the write and the tags of the cache match, the local processor marks the location invalid. The local processor does not examine the validity of the location.

3.3.2.6 Byte Validity (DCU)

The Dcache can not handle two processors accessing data at the same time. The processors use a semaphore to transfer control of the data. Still, several processors may use separate bytes within an aligned word. One processor may own some of the bytes of a word while another processor owns the remaining bytes. Though the memory system returns entire even and odd words from a read operation, the Dcache encaches only the bytes that an instruction requested. The processor uses byte validity to encache non-word and unaligned word operands.

If a read or update operation requested an aligned word or longword the memory control writes the data and its tag to the cache, and marks all bytes of the appropriate word valid.

For unaligned word or longword data, bytes and halfwords, the memory control does not write all bytes of a word in the cache. The memory control reads the A copy of the Dcache tags to determine if the word is in the Dcache. If the word is in the cache, the memory control writes the operand of the word and does not change the validity of the word. If the tags did not match, the control writes all the validity bytes for the cache, and marks those that correspond to the operand valid and marks the other bytes as invalid.

3.3.2.7 Dcache Control (DCU)

The memory control and WRSIG handle all the control for the Dcache. Memory control selects all the addresses and data for each data cache RAMs while WRSIG generates all the write enable signals. The control also determines when data returning from memory hits the Dcache and what to do with update data and remote invalidates.

The physical address multiplexer, on the SFU board, generates the tag data which writes all the tag RAMs, including those for remote invalidate operations. It also generates the compare tag which detects a tag read hit. The multiplexer generates the addresses for the copies of the data tags, and the local processor uses it to compare the A tag for an address in the Dcache. The memory control runs this mux.

3.3.3 Logical-to-Physical Address Translation (SFU, DCU)

Memory requests arrive from the processors with a virtual address. The memory, however, stores addresses in their physical form. Two PTEs, on the DCU board, map these virtual addresses. Each cache translates addresses for one side of the memory. Since the system provides 4 gigabytes of logical address space, and maintains 4K byte pages, it can translate addresses over three levels more efficiently than through one table.

Bits <31..29> of the logical address are the segment bits, which point to eight segments. Each of the segments has a segment descriptor register, pointing to the first level within the PTE table.

3.3.3.1 Swappers

The swappers, on the DCU board, determine the destination of a memory request. These circuits test address bits to decide where in memory the separate data bits will transfer. The FBUS transfers the alpha and alpha8 addresses to the swapper from the ALU. Bits <31..0> of the FBUS hold an alpha address, while bits <63..32> contain an alpha8 address.

Table 3–7 shows a breakdown of the address bits of the bits in a longword as that longword enters the swapper. Refer to the Alpha Addressing section in this chapter.

Table 3–7, Longword Address Bits

LONGWORD BITS	ORGANIZATION
<63..35>	Alpha8 address of the operand
<34>	Bit to select a side of memory
<33..32>	Bytes of the word
<31..3>	Base address of the operand
<2>	Bit to select a side of memory
<1..0>	Bits to access memory bytes

Address swappers separate the addresses for even and odd sides of memory based on alpha addressing. The swapper then produces the SA0 and SA1 buses to even and odd LABs, respectively. The address of the data travels on the buses to load the logic for the LABs. These buffers then begin translating the virtual addresses to physical addresses.

3.3.3.2 Alpha Addressing

Odd memory receives alpha addresses, while even memory loads either an alpha or the alpha address plus eight (alpha8). The swapper determines which address transfers to odd and even memory. If FBUS bit <2> contains 0, the swapper loads both sides of the LAR with FBUS bits <31..0>. Otherwise, the swapper loads the LAR for even memory with FBUS bits <63..32>, and the LAR for odd memory with FBUS bits <31..0>. The LAR for even memory uses the address in those bits and the LAR for odd memory adds four (#bytes in a word) to the address.

When an operand begins at an odd address, bits <31..3> of the address can not access even memory, so the alpha8 address provides the correct address.

The following tables give an example of how the swapper examines data to separate addresses to even and odd memory. Note that in the first table bit <2> is one, and in the second table bit <2> is zero.

Table 3–8 shows an example of alpha addressing:

Table 3–8, Alpha Addressing

ADDRESS	BINARY VALUE <15..0>	HEX VALUE
Alpha, or base	0010 0000 0000 0100	2004
Even memory	0010 0000 0000 0xxx	400
Odd memory	0010 0000 0000 0xxx	400

NOTE

An "x" denotes bits which have shifted out of the address.

Table 3–9 shows an example of alpha8 addressing:

Table 3–9, Alpha8 Addressing

ADDRESS	BINARY VALUE <15..0>	HEX VALUE
Alpha, or base	0010 0000 0000 0000	2000
Odd memory	0010 0000 0000 0xxx	400
Alpha8	0010 0000 0000 1000	2008
Even memory	0010 0000 0000 1xxx	401

NOTE

An "x" denotes bits which have shifted out of the address.

3.3.3.3 Logical Address Registers (DCU)

Two Logical Address Registers (LARs), on the DCU board, hold the logical address for the current request. One register contains the address for even memory, while the second holds the address for odd memory. If the current request can complete on the present clock, the LA and PTE are set to load. When the swapper generates the SA0 and SA1 buses, it loads the LA. The PTE cache receive a copy of the LAB bits <21..12> which enables the PTE cache RAMs to generate a physical address. If, however, the system is handling a fault, the LAs and PTE cache can not load any values.

3.3.3.4 Vector Address Generator (SFU)

The Vector Address Generator (VAG), on the SFU board, generates the memory requests for the VP using the Vector Length (VL) and Vector Stride (VS) registers. The SP, however, provides the starting address for vector memory requests. It handles both load and store requests, and also manages dual load requests on aligned word vectors.

> **NOTE**
>
> Any request by the Memory Interface is also gated by the Data Flow Gate Arrays to the Memory Data Register. It is placed in the Memory Data Register, on the IPP board, whether it is intended for the Instruction Processor (IP) and parity is checked. Therefore, an IP parity error may be displayed for any incorrect data from memory.

During a normal vector load, the ASP board makes the first request and the VAG makes later requests if the value in the VL register is greater than one. When the ASP board requests data for the VP it provides address information on the FBUS to the DCU board. This information includes the starting address, and the starting address plus eight. The DCU board uses these addresses to request data from memory. The QVA and QVA8 registers also receive a copy of the addresses for nonaligned addresses. The values of these registers are added to the VS register to create the VAG addresses.

When the VAG detects the first request transferred from the ASP board to the swapper it loads the value of the VL register bit <1> into the VLC. It also registers the memory opcode and the size of the request. The VAG state becomes busy, and the VAG sends a request to the memory controller. The VAG decrements the VLC each time the Memory Controller grants a request, and continues to send requests to the DCU board. When the VLC becomes zero, the VAG state changes to idle.

A dual vector load is the overlapping of two independent vector loads of different vector registers. Dual loads must access different sides of memory during a single request and must access the PTE Cache so the loads use separate cells. The first load occurs like any other VAG operation.

For a second load, the ASP board checks the following conditions to ensure that a dual load can occur:

- The VS register is not zero, and is a multiple of four bytes.
- The current operation is a vector read of words and the current addresses are word aligned.
- The VAG is processing a single load operation.
- The first vector is not about to complete.

3.3.4 Memory Control (DCU)

Memory control waits for the system memory to signal READY from each board, then selects which memory board should receive a request. Memory control distributes memory requests across memory boards based on the memory address. If the system memory is not busy, it services the next request. If the system memory is busy, however, memory control settles contentions through a fixed arbitration scheme.

Table 3–10 lists the four sources of a memory request based on their priority, and the board(s) where the source can be located:

Table 3–10, Processor-to-Memory Control Priorities

PRIORITY	SOURCE	BOARD(S)
1	Look-Aside-Buffer	DCU
2	Vector Address Generator	SFU, DCU
3	Scalar Processor	ASP
4	Instruction Pre-Processor	IPP

NOTE

SP fault requests have a higher priority than VAG requests. This is the only case when the SP has a higher priority than the VP.

After selecting a request to memory, memory control sends a REQUEST signal to the processor to begin transmitting information. It gates the store data from that processor, through the Data Flow Gate Array, to memory.

3.3.4.1 Memory Control Multiplexer (SFU)

The memory control multiplexer selects the memory operation code, size, and return code information. The SFU board processes this information. Memory Control uses the operation code to generate the proper memory activity, e.g., IREAD is a read request from the IP and requires that data returns to the IPP board, while READ is a request that brings data out of the Data Cache. The size code specifies the operand size as either a byte, halfword, word, or longword, and during a write, determines which bytes will be written. During a scalar read operation, the return code becomes the destination address in the register file on the ASP board.

The memory control multiplexer maintains a separate LAB. The multiplexer also writes to the Data Cache.

3.3.4.2 Look-Aside-Buffer (DCU, SFU)

Two Look-Aside-Buffers (LABs), on the DCU and SFU boards, act as overrun buffers to hold requests that the memory control is unable to process. Each LAB inputs data to one side of memory, even memory or odd memory. These buffers hold requests that are waiting to access memory. The vector, Scalar, and instruction processors are the sources for a memory request. Each LAB can hold a single request; the processors must hold any additional requests to memory. Information in a LAB has a higher priority for memory requests than any of the processor sources.

The LABs save the complete state of a request. That information includes the memory opcode, the size of the data, any return code, and alpha and alpha8 addresses for even and odd memory. Once a request enters the LAB, it waits there until memory completes the previous request.

3.3.4.3 Memory Return Control (SFU)

The Memory Return Control tracks data as it returns from memory. It receives data from memory and must combine this data with the return address and information from the first item in the queue. The queue contains the return address for the data, which memory board received the request, how to handshake with the processor that sent the request, and the size of the data.

The return control pulls the item from the top of the queue to find what to do with the data that returns from memory, and where to look for that data when it returns. It must distinguish between which processor requested data, and where the data actually is to go. One processor may have requested data for another processor, as in the first load of a vector load. The memory opcode determines the source of the store data request.

The control finds which read ready signal to look at and which read enable is set. When the read ready signal is true, the control issues a read valid signal to the processor that accessed memory. The data is accepted in order as it returns from memory and the return control combines it with the information that was on the top of the queue.

The IPP board addresses the Icache and lookahead cache with its data then tags the data as valid data. The ASP board uses the return code to access one of its registers. The VP tracks where data returns so the Return Control must tag the data with a signal to issue a handshake to the VPD board.

3.3.4.4 Store Data Queue (ASP)

Any data returning from memory must pass through the Store Data Queue. The Store Data Queue within the Data Flow Gate Arrays (DFW) holds the Memory Control Current State and the LAB state. During a memory request the ASP board stores write data information in the queue, which the DCU board accesses. When the ALU generates an address, the ZBUS transfers the contents of the gate array to the DFW. The DCU board then places the information on a write data bus to transfer to memory. One longword of data may pass through the queue per clock.

The register memory opcode tells the memory controller how to store data. It also lists the size and the least significant bits of the address which describes how the DFW should rotate the data. For a request from the ASP board, the queue also contains a return code. For an IP request, the IPP board expects a logical address for the data. The request state would be either in the Memory Controller or the LAB.

3.3.5 System Faults

The memory interface generates memory faults to show that a memory access requires the ConvexOS operating system to intervene.

A fault is an attempt to jump to an memory page address that is either protected, or not resident in the user's virtual address space. The memory reference can not complete. The system uses the following steps to gain access to that page in memory:

- The machine state is frozen and dumped out.
- The ConvexOS checks that the user may legally touch the page in memory. If the page is allowed in the user's memory space, ConvexOS does the following:
  - Reads the page from the disk, via the Event Governed Operating System (EGOS).
  - Sticks the page into the memory.
  - Changes the Page Table Entry so the page is in the user's virtual address space.
  - Returns control to the user's program.
- The user program retries to address the memory page, finds the page, and completes its process.

During a fault the ConvexOS saves the set of processor states, or *context.* The context divides into hardware context, and software context; the hardware context contains the contents of all registers, while the software context includes all program variables and data structures of the program. The context during a fault is larger than for a regular system call or ring crossing since a fault usually occurs in the middle of an instruction.

The board's group of bits is saved on 32 shift rings, each 32-bits long. They shift to memory as part of the machine state. Memory stores the fault contexts so the faults can execute. After the OS returns the control, the bits shift back so the registers are exactly the same as before the fault. The instruction set then resumes the process.

3.3.5.1 Saving Fault State

When a fault interrupts the micro controller, it allows the memory return data pipe to empty, and save the state of the memory request that caused the fault in the memory interfaces internal state. During this time, the IP can not dispatch and the VAG and the IP can not make memory requests. The microsequencer then tells the VP that a fault has occurred and waits for the VP to acknowledge the fault.

The state on the exception stack is the operating system needs to restore the process that the fault interrupted. The exceptions may include the following:

- The vector register data
- The vector length register
- The vector stride register
- The vector merge register

The ConvexOS operating system keeps track of which process's state is currently in the vector registers and leaves it there until another process must use this registers or until it restores the original process and must resume using one of the registers.

Chapter 4
Vector Subsystem

4.1 Overview

The C200 Series Vector Processor uses three pipe lined function units to operate on data. The three function pipes are the load/store function pipe, the add function pipe, and the multiply function pipe. Each function pipe has its own micro controller to control the sequence of actions required to perform the instruction dispatched to it.

The load/store function pipe is used to perform all load and store operations. These include vector register load and store operations, vector length register loads, moves between a scalar register and a vector register element, and load and stores to the vector merge register. All load/store data paths are staged so that a vector register element may be loaded/stored per cycle.

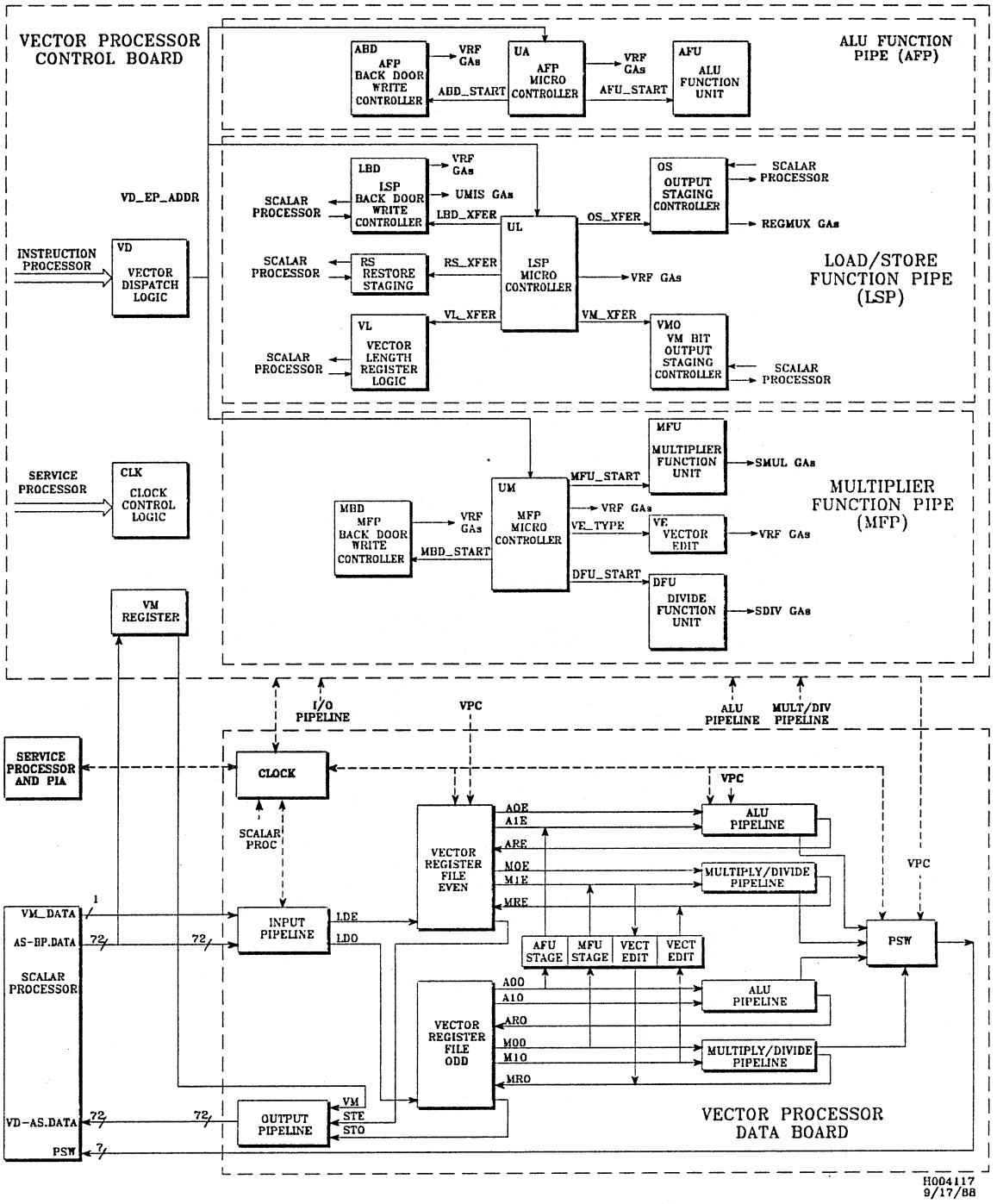
The ALU function pipe is used to perform all types of adds, population counts, shifts, logical functions, and data type conversions. The data paths of the ALU function pipe are staged so that a result is generated every cycle. The number of cycles required to complete an add of two operands and store the result is ten cycles. The first result is not valid until ten cycles after the operands are read from the vector register file, but each cycle thereafter a new result is valid.

The multiply function pipe is used to perform all multiplies, divides, square roots, and vector edits. The multiply function pipe data paths are staged similar to the add function pipe, thereby generating one result per cycle.

The three function pipes may execute different instructions concurrently. Once the result of an instruction has been written into the vector register file, another function pipe may select that vector register to be used as an operand for the instruction it is performing. This mechanism is called operand chaining.

Figure 4–1 shows the functional block diagram of the C200 Series Vector Processor Subsystem:

Figure 4–1, Vector Processor Subsystem Functional Block Diagram



Data Paths. The data paths are split into even and odd halves with each half operating at half the system clock rate. This configuration allows the ALU and multiply function pipe to generate two results every two system clocks.

4.2 Vector Processor Interfaces

The vector processor interfaces with three other processors, the scalar processor, the instruction processor, and the service processor. The service processor handles clock control and error detection.

The instruction processor dispatches instructions to the vector processor. A set of handshake control signals control the transfer of instructions to the vector processor's Vector Dispatch Logic (VDL). The information transferred includes the instruction type, registers to be used, and whether to operate under mask.

The scalar processor interfaces are used to transfer data to/from the vector processor. This includes transfers from the memory system or the scalar processor to the vector processor, transfers from the vector processor to the memory system or scalar processor, and transfers to the vector length register.

4.3 Vector Processor

The SP passes all vector operations to two boards, the Vector Processor Control board (VPC) and the Vector Processor Data board (VPD), which process the instruction. A 64-bit bus from the Data Flow Gate Array (DFW) on the Address and Scalar Processor board (ASP) parallel loads the Vector Merge Register (VM), the Vector Processor Control (VPC) state, and all vector elements. A second 64-bit bus from the VPD board stores the value of the VM register, the VPC state, and all vector elements.

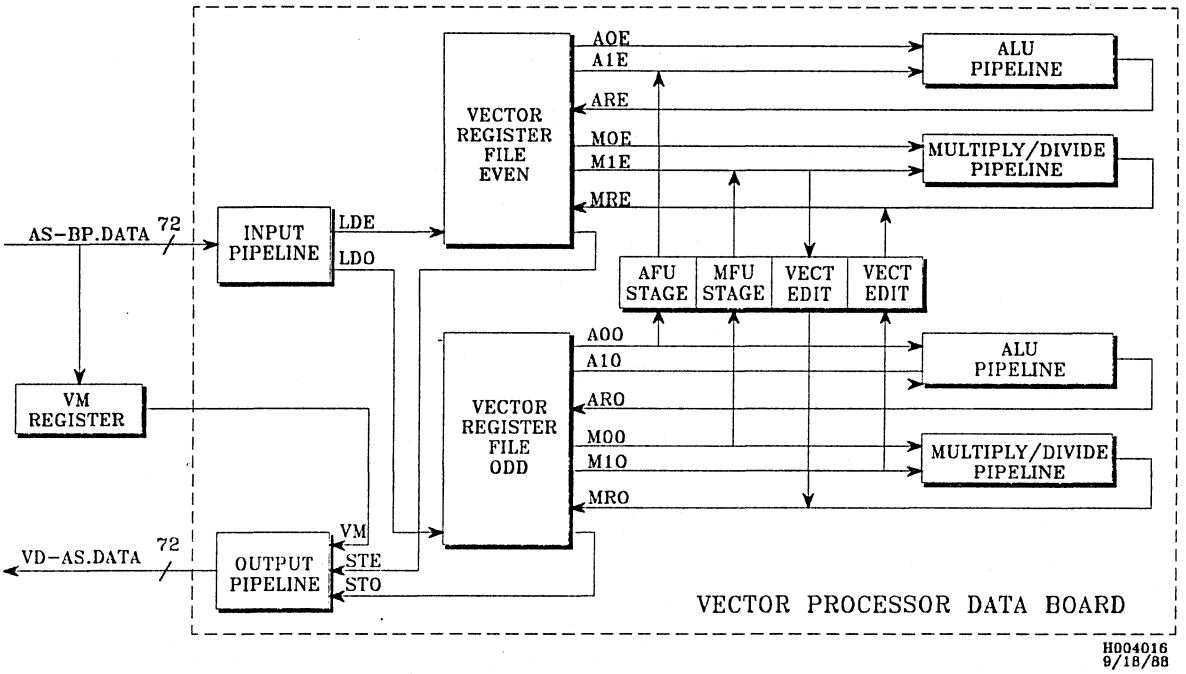
The VPD board contains the data path logic and some clock generation logic for the processor. The data flows through the following circuits:

- Vector Register Files
- Memory and Scalar Processor Interfaces
- Function Units
- Vector Merge Register

Though functionally part of the data path logic, the vector merge register is on the VPC board.

Figure 4–2 shows the data pathways of the VPD board for the vector processor:

Figure 4–2, Vector Processor Data Pathways

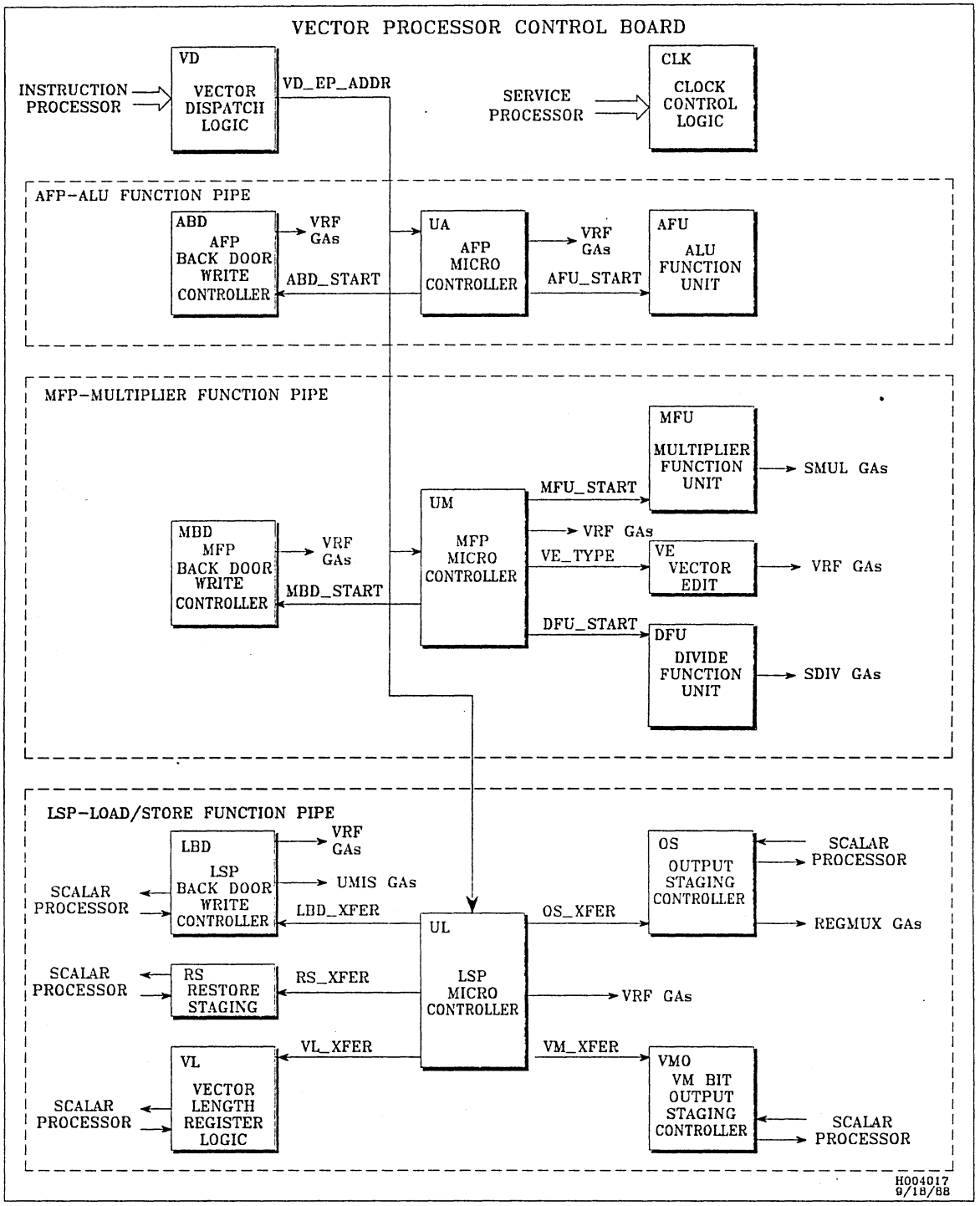


The VPC board contains the control logic and some clock generation logic for the vector processor. It maintains control of the following:

- The Instruction Dispatch Logic
- Load and Store Function Pipelines, including micro control and write control
- The Vector Length Register
- Output Staging
- VM Staging
- ALU Function Pipelines, including unit control, micro control and write control
- Multiplier Function Pipelines, including unit control, micro control and write control
- The Divide Function Unit Pipeline

Figure 4–3 shows the control lines of the VPC board for the vector processor:

Figure 4–3, Vector Processor Control



The Scalar Processor (SP) begins all Vector Processor (VP) memory requests with a scalar memory request. The Vector Address Generator (VAG) then makes any additional VP memory requests.

A request from the VAG consists of a memory opcode, a size code, and a pair of addresses. Since return data goes directly to the Vector Processor, the VAG does not generate a return code. The pair of addresses, VA and VA8, are the Alpha and Alpha8 addresses which the ALU generates. However, during a Dual Vector Load operation, the two values are the addresses of each of the two vectors.

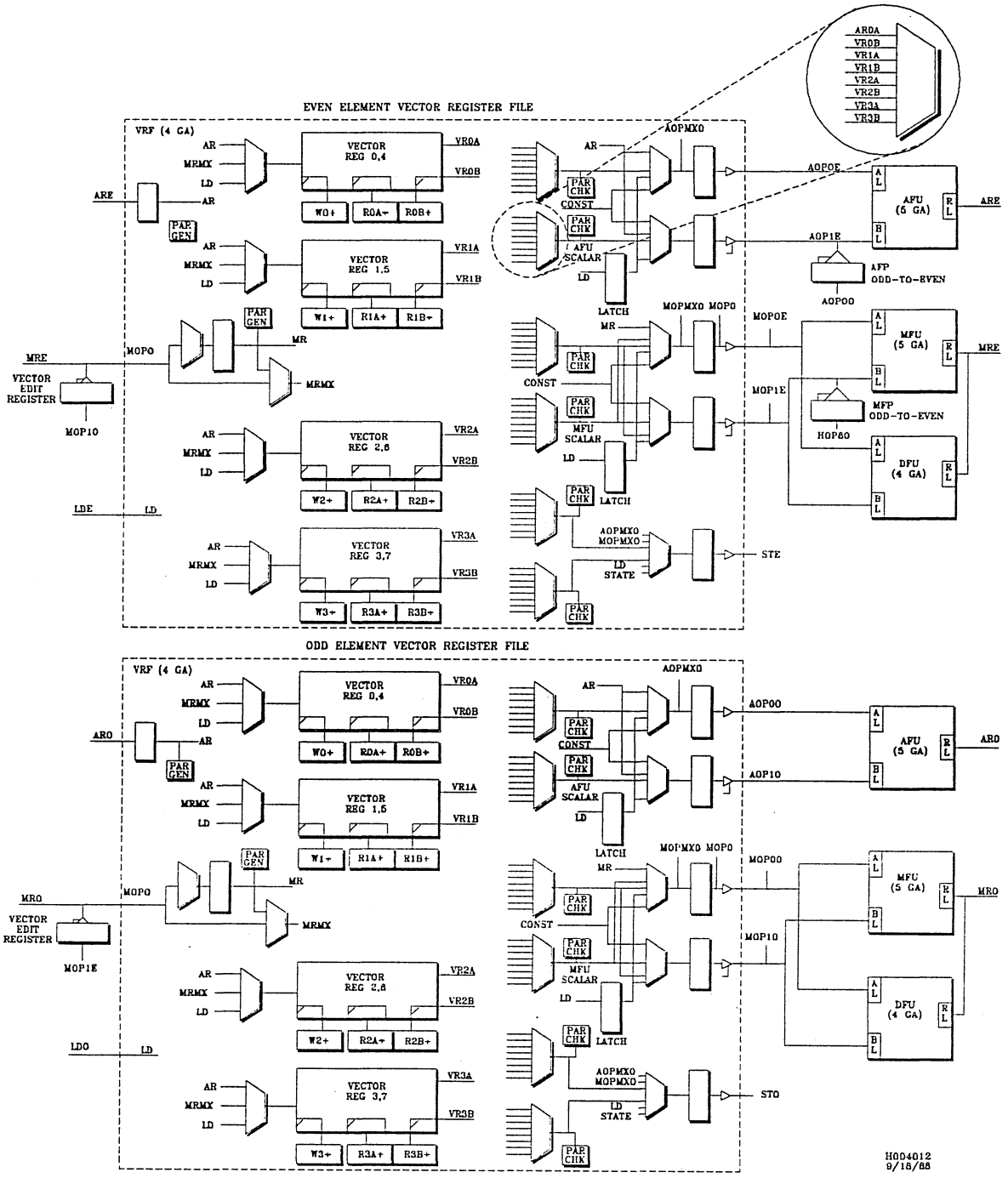
The VP has odd and even data paths to meet the performance goals of the machine. This design allows the VP to reduce the speed of operations without impairing their performance. The VP clock (80 ns), therefore, runs at half the speed of the system clock (40 ns). Design contraints made it necessary to execute divide operations more slowly than other operations.

4.3.1 Vector Register Files

Two Vector Register Files (VRF), one even and one odd, each contain elements of the eight vector registers, V7..V0. Each vector register has 128 elements with 64 data bits and 8 parity bits in each element. The even vector register file contains all the even elements [0,2,4,..,126] for all vector registers. The odd vector register file contains all the odd elements [1,3,5,..,127] for all vector registers. Eight gate arrays, four even and four odd, implement the vector registers.

Figure 4–4 shows the even and odd elements of the VRF:

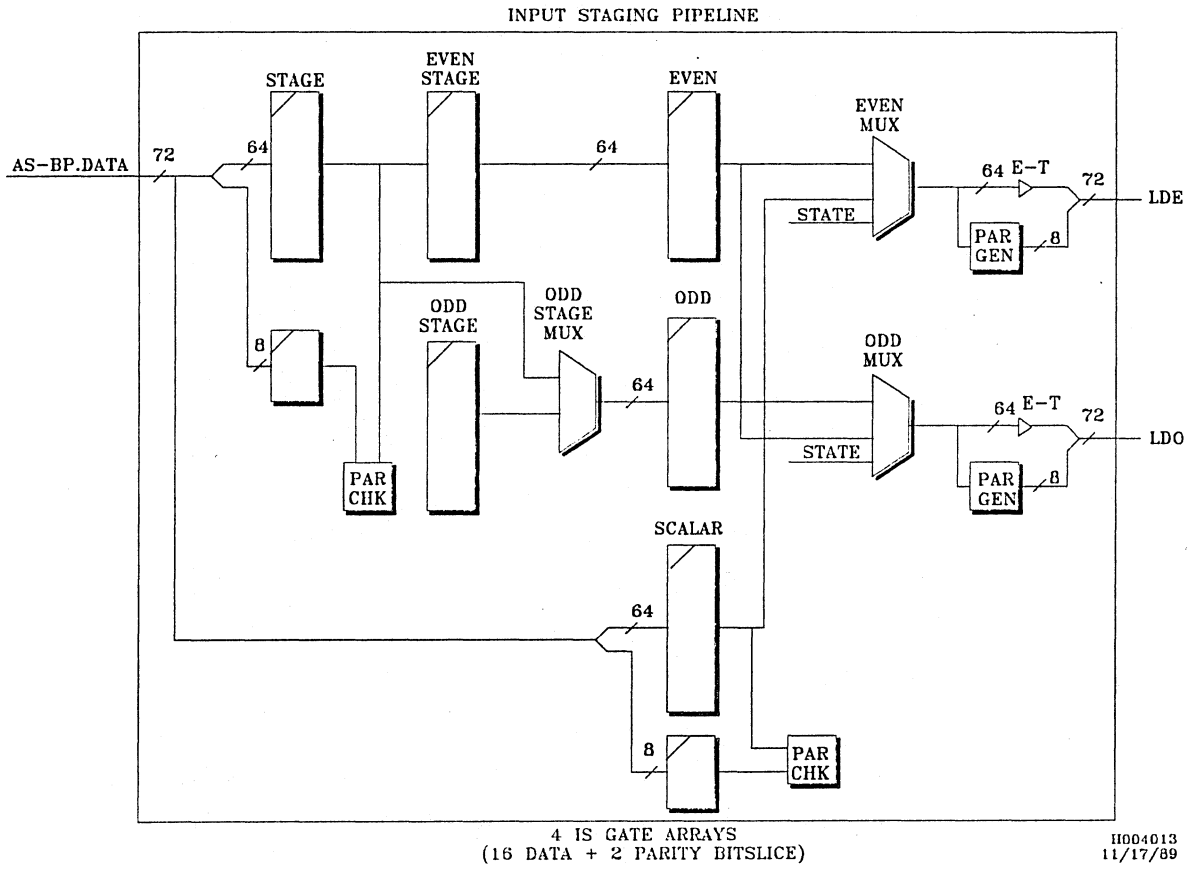
Figure 4–4, Vector Register File Gate Arrays



The input stage receives a set of 72 bits, 64 data bits and 8 parity bits, from the SP. It alternately transfers sets of bits to the even VRF and the odd VRF.

Figure 4–5 shows the input staging pipeline for the vector processor:

Figure 4–5, Input Staging Pipeline



Each vector register gate array in the file receives 18 bits, 16 data bits and 2 parity bits.

Table 4–1 lists the pairs of gate arrays, and the bits that either the odd or even gate array receives:

Table 4–1, VRF Input Bits

VRF EVEN GATE ARRAY	VRF ODD GATE ARRAY	DATA BITS	PARITY BITS
0	0	<63..56> <31..24>	<0> <4>
1	1	<55..48> <23..16>	<1> <5>
2	2	<47..40> <15..8>	<2> <6>
3	3	<39..32> <07..00>	<3> <7>

The clock for the VRF can not go low for more than one microsecond, or the information in the register could be lost. To prevent this, when the run is taken away, the VRF clock stays high. An extra cycle occurs if the VRF clock should have been low but stayed high. A halt signal causes the VRF clock to go high.

Each gate array memory macro has two independent read ports capable of accessing either vector register. For example, on the V0, V4 macro the ports may access either two elements of the V0 register, two elements of the V4 register, or one element from both.

The pipelines from the gate arrays include the ALU Function Pipe (AFP), the Multiply Function Pipe (MFP), and the Load and Store Function Pipe (LSP). Each pipe has an independent mux to select the proper operand. These muxes have both a vector register mux and an operand source mux. Sources for the operand include either a scalar latch, a feedback path for reduction operations, or a constant for operations under mask. The function controller for each pipe uses bits from the vector merge register to determine which operand source mux to select. The VRF contains additional staging registers to handle large CMOS output buffer delays and heavy loads on function unit gate arrays.

Results from the operations enter the VRF through the AR and MR inputs. Since the function units have CMOS output buffer delays, an output staging gate array immediately places the input into registers. The vector register input mux then sends the results to the proper half of the VRF to store in the memory macro location.

4.3.1.1 Vector Edit Logic

The vector edit logic steers the data for the vector edit instructions. It increments the read port counters, selects whether the even or odd data goes to the VRF, and decides whether to increment the write port counter. The logic uses the even and odd VM register bits and an internal register to track the least significant bit of the VM register counter.

The signals which read the values from the VRF gate arrays are applied directly to the gate arrays. Those signals which control writing values into the gate arrays stage one cycle so they can enter the gate arrays when the data arrives from either memory or the SP.

4.3.2 Memory and Scalar Processor Interface

The Vector Processor (VP) passes data to and from both the memory and the Scalar Processor (SP) over two unidirectional buses. The SP is the source for the bus that transfers data to the VP from the DFW gate arrays on the ASP board. The VP is the source of the data return bus to those same arrays.

Handshake signals control when data transfers between the SP and the VP; two signals controls each bus. When the VP requires data and a data transfer has not completed, the VP extends its data path clocks until either the transfer completes or memory issues a fault. Likewise, when the VP can not transfer data to the SP or memory, the VP data path clocks extend.

> **NOTE**
>
> Any request by the Memory Interface is also gated by the Data Flow Gate Arrays to the Memory Data Register. It is placed in the Memory Data Register, on the IPP board, whether it is intended for the Instruction Processor (IP) and parity is checked. Therefore, an IP parity error may be displayed for any incorrect data from memory.

4.3.2.1 Load and Store Function Pipe Micro Control

The load and store micro controller handles all data transfers between the SP and the VP, the Vector Merge Register (VM), and the Vector Length Register (VL). It handles the following operations:

- Selects operands for the output staging data paths, and starts the Output Staging Controller (OS)
- Requests Vector Length Register (VL) values
- Starts the VM Bit Staging Controller (VMO)
- Starts the Load and Store Back Door Write Controller (LBD)
- Specifies the next address for the ALU and MFU micro controllers during state save and restore functions

The load and store output controller instructs the output staging controller to transfer zero, one, or two pieces of data to the SP or to memory per VP data path clock cycle. The output staging controller then extends the data path clock until all the data has transferred. Refer to the Output Stage Controller section in this chapter.

The load and store micro controller enables the vector length register logic to request a new VL value. The VL logic stores the vector length value into the VL register. A separate eight bit data path between the scalar processor VL register and the vector processor VL register allows the SP to load the vector processor VL register without synchronizing the instruction streams between the two processors. The micro controller instructs the vector processor VL to load its code before the first cycle of a micro routine.

The load and store micro controller informs the VM bit staging controller of the number of data transfers per system cycle. The VM bit staging controller transfer VM bits to the SP for load and store under mask operations. The control stages one less cycle to the VM controller than to the output staging logic. Therefore, the SP receives the VM bits two system cycles earlier than the corresponding data, and can conditionally start memory. Refer to the VM Bit Staging Controller section in this chapter.

The micro controller starts the Load and Store Back Door Controller (LBD) to initiate a transfer from either memory or the SP to the vector register file. When only one or two data transfers are to parallel load the VM register, the LBD controller starts and extends the VP clocks until the data arrives. If the vector length is odd the last operating system and VMO start transfers only one element. The micro controller then guides the data to its destination.

The load and store micro controller also controls the next address it will execute. It generates the next address by one of the following methods:

- Conditional branch or dispatch
- Conditional stay at the current address
- Two-way branch

The test conditions may be either true or inverted. The micro controller reads the control store, loads the address for the control store into a micro instruction address register, and indicates that the controller is active.

4.3.2.2 Load and Store Function Pipe Write Control

The load and store function pipe write controller, or Load and Store Back Door Write Controller (LBD), generates signals to write any transfer from either memory or the SP to the Vector Register File (VRF). The load and store micro control and the vector dispatch logic start the LBD. The vector dispatch logic requests one SP transfer when the IP transfers a scalar or vector instruction. If an instruction which requires scalar transfers is executing, the vector dispatch logic waits until the instruction completes before it makes a request.

The micro controller can force the LBD Controller to extend the VP data path clock until data arrives. When the micro controller is active, the LBD controller extends the VP clock so the input staging register receives data within two cycles. The load and store micro code requests data in three forms:

- A vector load which is not under mask
- Load the Vector Merge (VM) register, load vector elements, or restore state
- A vector load under mask or a load vector with a vector of indexes

A vector load operation which is not under mask begins by requesting Vector Length Register (VL) value elements from memory. The VP clock continues to cycle until the first element of the load arrives from memory. Once the first element arrives, if the transfers do not continue each cycle, the VP clock cycle extends by a cycle (40 ns). This method allows operations which are executing to continue until that first element arrives from memory. The data path clock cycle extends so an even and odd element loads into the VRF each cycle. Data is then available for an instruction which chains to the load vector instruction. Once the first value has arrived, the vector dispatch logic releases the chaining hazard.

Load instructions which do not operate under mask do not send a VM bit to the scalar processor. However, the scalar processor forces the VM bit which is returned with each data element to be set. This allows the load back door controller to treat the masked and unmasked load operations the same. The load back door controller forces a zero VM bit at the end of an odd vector length load operation to inhibit writing an extra vector register file element.

Load vector merge register, load vector element, and state restore instructions request either one or two data transfers from the SP or memory. Once the data arrives, the micro code stores it. The controller issues this form of request with the force asserted. Therefore, the data returns to the input staging data registers two cycles after the controller makes the request.

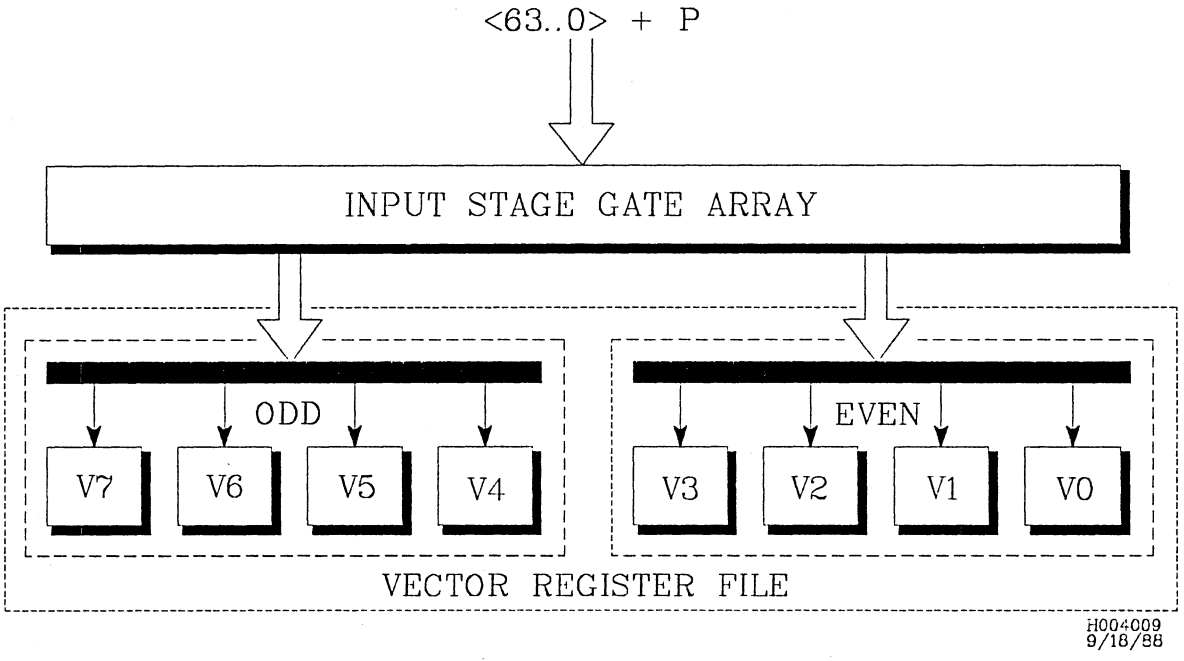
The final form of load and store requests is load vector register under mask or load vector with a vector of indexes instruction. Since data transfers to and from the VP, the first data element must arrive within a window. This first element sets the number of outstanding memory requests. It should be larger than the pipe length of the memory system, but small enough for the SP to accept all bits and index transfers and not to cause VP clocks to extend. VM bits or indexes transfer to fill the memory system pipe. The LBD controller is started by requesting vector length value elements without asserting force. The force issues a few cycles later to ensure the first element arrives before the end of the window.

4.3.2.3 Input Staging

The input staging pipeline can accept a 64-bit element from for the Vector Register File (VRF) during every VP clock. It stages these into 64-bit elements which then transfer to the VRF every system clock.

Figure 4–6 shows the transfer of bits from the input stage to the VRF:

Figure 4–6, Input Staging to the VRF



The SP can interrupt a vector load operation to transfer the scalar data that a vector instruction requires. The VP extends the controller clocks to ensure proper chaining. During vector reduction operations some data must transfer from the VP to the SP while the load and store micro controller is not using the output staging logic.

During a single vector load operation, the VP can obtain data from the SP every clock cycle if the memory system has no contention. When two vector load operations execute simultaneously, the SP transfers one element of each vector per system clock cycle. For a load under mask operation, the VM register presents a bit to the SP for each vector element. The SP carries the bit through its memory queue and returns it to the VP with the load data.

4.3.2.4 Output Staging

The output stage controller generates the signals for the output staging gate arrays. All three micro controllers can request data.

The following table lists the micro controllers with a description of the requests each controller can issue:

Table 4–2, Requests to the Output Stage Controller

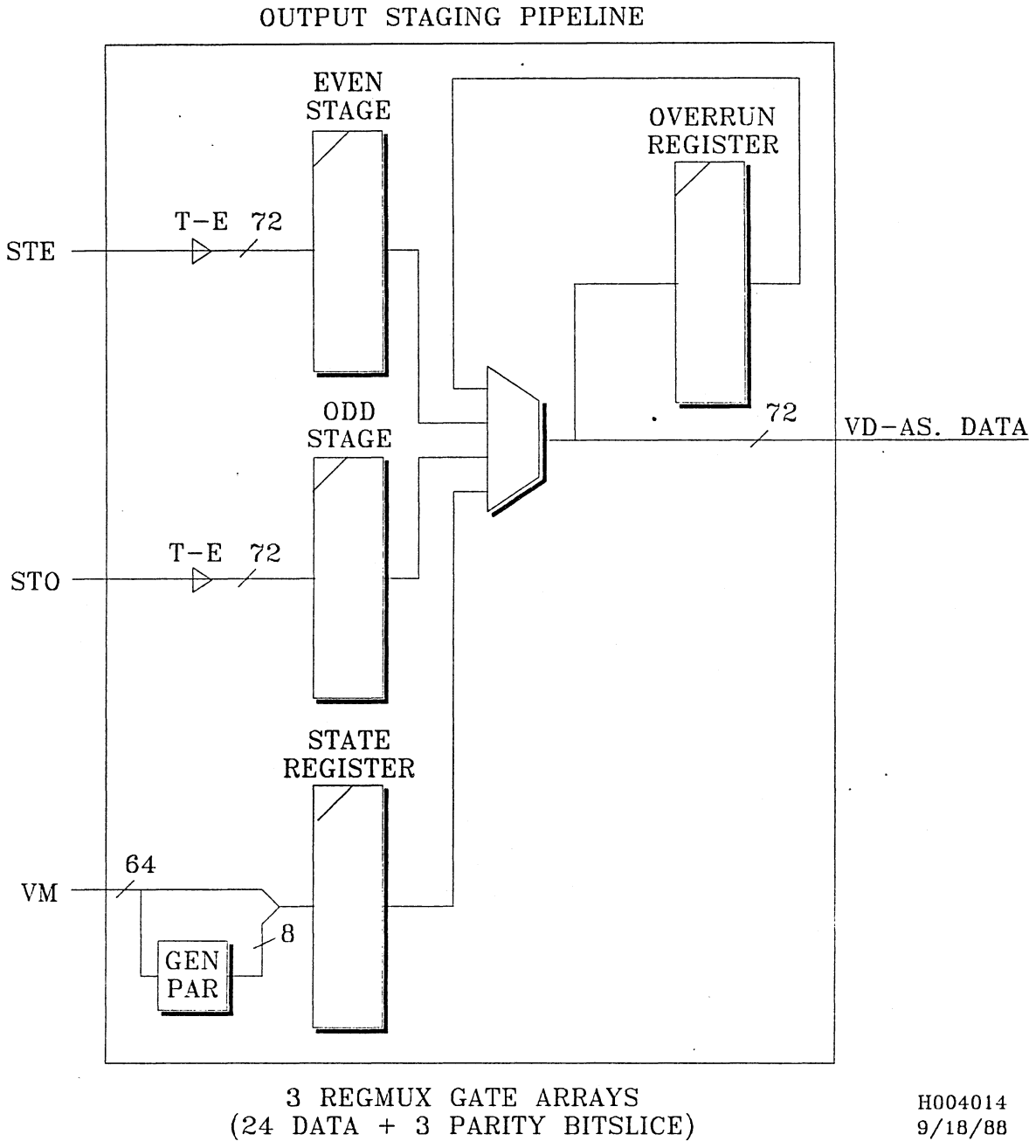
CONTROLLER	NUMBER OF REQUESTS	DESTINATION
AFP	1 or 2	Scalar Processor
MFP	1	Scalar Processor
Load and Store	1 or 2	Scalar Processor or Memory

The vector dispatch logic ensures that only one instruction that requires the output staging logic executes at one time. The controller decodes the requests and stages the data with the request. The output staging state machine then uses this information to transfer the data.

The output staging pipeline receives two 64-bit elements from the vector register file every VP clock (80 ns) and stages these into 64-bit elements to transfer to either memory or the SP during each system clock (40 ns).

Figure 4–7 shows the output staging pipeline for the vector processor:

Figure 4–7, Output Staging Pipeline



During the first half of a clock cycle, an even vector element may transfer along the bus to the SP; an odd vector element may transfer during the second half of the cycle. An overrun register holds data from the previous cycle if the transfer did not complete. Another register allows the VP to select an external register state.

4.3.2.5 Vector Length Register

The Vector Length Register (VL) loads directly from the Scalar Processor VL register. A pair of signals synchronizes the load process.

The SP asserts the first signal to inform the VP that the VL register loaded the value. The VP then asserts a signal to request a new value. When both processors have asserted their signals, the scalar processor VL register transfers the new VL value. The SP may not reload its VL register before the VP uses the previous value. Likewise, the VP does not reload its VL register until the SP signals that its processor VL register is holding a new value.

The load and store micro controller tests the signals in a micro code loop rather than stopping the VP clock. A load vector from memory operation followed by a load VL register from memory could cause a dead lock condition. If the load VL register instruction extended the clocks until a new value arrived, the load vector operation could not complete. This problem would inhibit the VL register value from storing correctly in the register.

4.3.2.6 Processor Status Word Register

The Processor Status Word Register (PSW) holds the processor status word bits, or flags, from the previous cycle. These flags enable or disable exception processing and shows the results of numeric operations. If any bit in the vector processor PSW register is one, it sets the scalar processor PSW bit to one. The vector processor PSW register ORs the PSW bit from each function unit. This result is the value that transfers to the SP.

The PSW bits are transparent to the user, so the processor must save them during a system fault. During a state save operation, the top bit of the VP state receives the bit, and shifts that bit back during a restore operation.

4.3.2.7 Faults

The vector processor handles all memory faults identically, independent of the vector instructions executing. Two routines execute when a fault occurs, state save and state restore. The state is saves and restored whether the state is meaningful when the fault occurs.

Fault processing begins when two signals are both asserted. The signals indicate that a memory fault has occurred and that the memory system has returned all outstanding read requests.

When the VP recognizes a fault, the VP clock logic micro interrupts the function pipe controllers to begin executing at location zero. The load and store micro controller generates control store addresses for the three micro controllers. This allows special state save and restore functions to use the micro fields, which normally control address generation for the add and multiply micro controllers. The three micro controllers of the VP load a new micro word while the VP extends its clocks.

The load and store micro controller asserts a restore signal to force the AFU and MFU micro controllers to choose the load and store micro address for their next micro address. After the state shifts to memory, the Load and store micro controller removes the restore signal to release the two controllers.

The vector processor resumes its functions after the operating system has corrected the memory fault condition. The return control signal (rtnc) begins by dispatching the Load and Store Function Pipe. The micro controller then asserts a restore bit to force all three micro controllers to execute from the Load and Store Micro Controller address.

After the state returns, and after the processor completes executing, the load and store micro controller becomes inactive. The Load and Store Micro Controller completes the fault operation by dispatching to the next operation or becoming idle. All three micro controllers then go idle and all interface signals operate as normal. The last micro word issues a dispatch signal to force the micro controllers to do one of the following:

- Resume executing an interrupted operation
- Begin a new operation
- Become idle

4.3.2.8 Instruction Dispatch Control

The Instruction Pre-Processor board (IPP) loads the instruction dispatch registers when a vector instruction is parsed. The VP dispatch logic stages all instructions which the IPP board dispatches to the VP, and uses the opcode from the IPP board to index an entry table to obtain specific information about the opcode. The fields of the table include the following:

- Function pipe reservation logic—The function pipe reservation logic checks if the instruction can chain to the current executing instruction. It also tells when the function pipe is available.
- Chaining hazards—This field determines if the instruction uses results from the current executing instruction.
- Port reservation logic—The port reservation logic checks if vector register ports are available for the operands that the instruction requires. It also shows which port the vector register allocated to the instruction.

For instructions that must obtain a scalar value to begin, the dispatch control requests the data from the SP.

The logic holds the instruction until the hardware resources that it needs are available. The dispatch information loads into the appropriate function pipe dispatch registers. The function pipe controller then begins at the entry point address in the dispatch table and controls the operation.

4.3.3 Function Units

Three function units execute all vector operations. The following lists the function units and the operations that they execute:

- ALU Function Unit (AFU)—add, subtract, compare, convert, logical, shift, and population count operations
- Multiply Function Unit (MFU)—multiply operations
- Divide Function Unit (DFU)—divide and square root operations

Each function unit takes the form of two sets of gate arrays, one even and one odd. A data path connects the gate arrays within each set.

Table 4–3 presents the function units with the gate arrays and the time to output operations:

Table 4–3, Function Unit Gate Array Operations

FUNCTION UNIT	FUNCTION PIPE	GATE ARRAYS	NUMBER OF GATE ARRAYS	OPERATION OUTPUT TIME
AFU	AFP	SALU	4 even 4 odd	1 operation per pipeline per VP clock
MFU	MFP	SMUL	5 even 5 odd	1 operation per pipeline per VP clock
DFU	MFP	DIVX	4 even 4 odd	4 bits per pipeline per system clock

NOTE

A system clock is 40 ns, while a VP clock is 80 ns.

4.3.3.1 ALU Pipeline

The ALU pipeline consists of the clock generation logic and the Scalar Arithmetic and Logic Unit gate arrays (SALU). Four gate arrays execute operations on data of the following lengths:

- bytes
- halfwords
- words
- longwords
- single precision
- double precision

Table 4–4 lists the functions of the SALU and the time to process each function:

Table 4–4, ALU Pipeline Functions

FUNCTION	PROPOGATION TIME
Addition Subtraction Compare Logical	195 ns
Convert Population counts	255 ns

If the even and odd pipelines are both calculating a partial sum, the AFU staging register transfers partial results from the odd pipeline to the even pipeline. The register places the odd element onto a bus and into the first latch of the SALU.

In an ALU pipeline operation, the VRF gate arrays send two operands on two even buses while the VPC board sends the operation code on the buses. The AFU clock and the operand latch clocks for the SALU drop low then go high to hold the operation code and the operands within both the even and odd latch SALU 0. The VRF place the next two operands on the even bus, two on the odd bus, then the next latch clocks hold the opcodes and operations in the even and odd latch SALU 1. When latch SALU 3 is loaded, the opcodes and operands are stored in SALU 0 while the result is enabled to the VRF.

4.3.3.2 ALU Function Pipe Unit Control

The controller for the ALU Function Pipe Unit (AFU) supplies the signals to the even and odd SALU gate arrays to start an operation and read the results from the gate arrays when that operation is complete. The logic of the controller maintains a counter to specify which odd and even gate arrays to start. Load and start signals effect only the SALU gate array that the counter enables. The control logic then selects an even and odd pair of gate arrays which drives the results onto buses to transfer to the vector register files.

The unit also contains a conditional load A or B latch and a conditional start Pass A or Pass B operation for min/max reduction instructions. The gate arrays must conditionally load and pass either the A or B operand for these operations.

The AFU state consists of the operand latches and the opcode latch. To save the AFU state, the controller shifts the registers to a state bus; to restore the state, the controller shifts the bits from that bus to the registers. Before any bits transfer to the bus, a signal disables the function unit latches to prevent the registers losing the state. If a fault occurs, the controller must save the state of the gate arrays to restore when the fault clears.

4.3.3.3 ALU Function Pipe Micro Control

The ALU micro controller controls reading the operands for the instruction executing. The vector dispatch logic maintains a table which lists the operations that the SALU gate arrays perform.

The ALU micro controller selects operands for the ALU function unit, starts the function unit, and starts the ALU Back Door Write Controller (ABD). It must stay active until it sends each operand to the function unit and the operation completes. The micro controller also transfers the results from reduction instructions to the scalar unit. Once all elements have been started in the ALU Function Unit, and either the ABD starts or the instruction transfers to the scalar unit, the micro controller goes idle or is dispatched.

For instructions other than state save and restore operations, the micro controller generates the address of the next instruction to execute. The following lists the methods which the controller may use to generate that address:

- Conditional branch or dispatch
- Conditional stay at current address
- Branch based on certain test conditions, true or inverted
- Branch based on the four least significant bits of the Vector Length Register

Two counters, the Vector Length Register (VL) and the Pipe Length Counter (PL), control the flow of the micro code. The code may either hold or decrement by two the value in the VL register. It may also hold, decrement, or load the PL counter with a value from the micro code. A field in the micro code tests the counters by comparing either the VL register to one or the PL counter to zero. The micro controller may also compare the contents of the VL register either to a value twice the contents of the PL, or to 127 for a dispatch condition.

For state save and restore instructions the load and store micro controller addresses the AFP control store to allow the branch field to be enable signals. The control store includes the VL counter and the PL counter, the micro instruction address register, and the activity register. The registers use scan paths to save and store the state of the AFP micro controller.

4.3.3.4 ALU Function Pipe Write Control

The ALU function pipe write controller, or ALU Function Pipe Back Door (ABD), generates signals to write either the ALU function pipe results to the Vector Register File (VRF), or the AFU compare status to the Vector Merge Register (VM). To determine when to write results, the ABD stages the VM bits until the AFU releases the corresponding results. A write instruction begins with a signal indicating the destination of the write. The controller then obtains from the vector length register the number of results to write. It continues to write a result to the vector register file every clock cycle until the value in the VL register decrements to zero.

The controller also clears hazards for either a VRF write or for a serial VM register write operation. Operations under mask must conditionally write results to the VRF. For a compare instruction the controller signals the VM gate array to serially write the VM bits.

4.3.3.5 MFU Pipeline

The MFU pipeline consist of multiply clock generation logic, divide clock generation logic, Scalar Multiply gate arrays (SMUL), and the Divide gate arrays (DIVX).

Table 4–5 lists the data lengths of the multiply operations, and the time for the SMUL to process the instruction:

Table 4–5, Multiply Pipeline Operation Times

INSTRUCTION LENGTH	PROPOGATION TIME
Byte Halfword Word Single Precision	175 ns
Longword Double Precision	290 ns

The longer operations require signals to latch intermediate results and steer internal data paths. The SMUL state machine pals produce the signals. Control signals select either even or odd SMUL gate arrays to output results or load opcodes and operands. The signals also determine which latch clock holds the 64-bit opcodes and operands.

The DIVX gate arrays, which execute the divide and square root functions, are on the same operand and result bus as the SMUL gate arrays. Each DIVX contains two divide and square root function units. Each function unit gate array produces two bits of results per clock cycle. The DIVX function unit differs from the other pipes since it generates a certain number of bits per cycle and the number of bits for the output differs with each operation type.

Table 4–6 lists the forms of the DIVX operations and the number of cycles to complete each operation:

Table 4–6, DIVX Function Throughput Times

DIVX FUNCTION	OUTPUT (80 NS PER CYCLE)
Divide and Square Root	16 bits per cycle
Single precision floating point	1 result per two cycles
Double precision floating point	1 result per four cycles

Since the VPD board has space for only four gate arrays per pipe, longer functions can not proceed as quickly as the other types of operations. The rest of the vector units must wait until the DIVX operations complete.

During a context save operation, the VP waits until the DIVX gate array completes the divide, then reads the result and the status from them. The DIVX gate arrays assert a busy signal when they are busy. The signals generate an idle and busy signal for the DFU clock.

The dispatch instruction logic starts the controllers for the AFU, the MFU, and the DFU when no hazards exist for an instruction. The following sections describe the controllers for the function unit pipelines, their micro control, and their write control.

4.3.3.6 Multiply Function Pipe Unit Control

The controller for the Multiply Function Unit (MFU) signals to the SMUL gate arrays to begin an operation and read the results from the SMUL when that operation completes. The control logic contains a counter to specify which pair of even and odd gate arrays to start. A load and start signal from the controller affects only the SMUL gate array that the counter enables. The control logic then selects an even and odd pair of gate arrays to drive the results onto buses to the Vector Register File gate arrays.

The MFU state consists of the operand latches and the opcode latch. To save an MFU state, the controller shifts the registers to a state bus; to restore the state, the controller shifts the bits from that bus to the register. Before any bits transfer to the bus, a signal disables the function unit latches to prevent the registers losing state. If a fault occurs, the controller must save the state of the gate arrays to restore when the fault clears.

4.3.3.7 Multiply Function Pipe Micro Control

The multiply micro controller selects operands for both the Multiply Function Unit (MFU) and the Divide Function Unit (DFU). It starts those function units and the multiply write controller, or Multiply Function Unit Back Door Controller (MBD). It also guides a vector edit operation. The controller must stay active until it sends each operand to a function unit and the operation completes.

For instructions other than state save and restore operations, the micro controller generates the address of the next instruction to execute. The following lists the methods which the controller may use to generate that address:

- Conditional branch or dispatch
- Conditional stay at current address
- Branch based on certain test conditions, true or inverted
- Branch based on the four least significant bits of the Vector Length Register

Two counters, the Vector Length Counter (VLC) and the Pipe Length Counter (PL), control the flow of the micro code. The code may either hold or decrement by two the value in the VLC. It may also hold, decrement, or load the PL counter with a value from the micro code. A field in the micro code tests the counters by comparing either the VLC to one or the PL to zero. The micro controller may also compare the contents of the VLC either to a value twice the contents of the PL, or to 127 for a dispatch condition.

For state save and restore instructions the micro controller addresses the MFP control store to use the branch field as the enable signal. The control store includes the VLC and the PL, the micro instruction address register, and the activity register. The registers use scan paths to save and store the state of the MFP micro controller.

4.3.3.8 Multiply Function Pipe Write Control

The multiply function pipe write controller, or Multiply Function Pipe Back Door (MBD), generates signals to write either the MFU, the DFU, and sends the results from a vector edit operation to the Vector Register File (VRF). The MBD begins with a signal indicating the source of the write. The controller then obtains from the vector length register the number of results to write. The controller then continues to write a result to the vector register file every clock cycle until the value in the VLC decrements to zero.

Operations under mask must conditionally write their results to the VRF. To determine when to write bits, the MBD controller stages the VM bits until the MFU releases the corresponding bits. The MBD controller also clears hazards before writing to the VRF.

4.3.3.9 Divide Function Unit Control

The controller for the Divide and Square Root Function Unit (DFU) executes divide and square root operations for all CONVEX operands. The DIVX has four pairs of even and odd gate arrays which execute instructions; each DIVX gate array can perform two operations on different operands simultaneously.

Operations with operands greater than one byte require more than eight data path clocks to complete, and therefore require more time to execute than one system clock. For those results that require more than one clock cycle, the controller extends the VP data path clocks while it continues to clock the gate arrays.

Table 4–7 lists the forms of the DIVX operations and the number of cycles to complete each operation:

Table 4–7, DIVX Function Throughput Times

DIVX FUNCTION	OUTPUT (80 NS PER CYCLE)
Divide and Square Root	16 bits per cycle
Single precision floating point	1 result per two cycles
Double precision floating point	1 result per four cycles

Three counters start and monitor the divide gate arrays. An additional stage register indicates when the DIVX should complete an operation. Each counter executes one of the following:

- Select which DIVX gate array to start
- Check if the gate array is busy with a previous operation
- Select which DIVX outputs the results

The start unit counter selects the DIVX gate array while the first stage of a shift register is set to show that a divide operation has begun. The stage register shifts once each data path clock cycle for eight shifts. After eight shifts of the stage register, the gate arrays should have completed an operation. If the divider does not have an operation completed, the clocks extend until the result is available. Last, the output unit counter selects the result to place on the result bus.

4.3.4 Vector Merge Register

The Vector Merge Register (VM) is a 128-bit register with four independent bit read ports and one write bit port. The VM register parallel loads and stores data. Any bit read or write operations occur in pairs with one bit for both even and odd elements.

Each function pipeline can access independently the VM register contents. The AFP and MFP each have one mux which connects to the VM register, while the load and store pipe has two muxes. The load and store pipe must be able to load two word vectors simultaneously.

All bits of the VM register transfer to or from the SP for load and store under mask operations; 64 bits transfer for move instructions. Either the instruction or a scalar value from the SP determines which 64 bits to load. The micro controller controls the data transferring to the vector merge register. It also starts the load and store back door controller to request the transfer to or from the SP. The output staging controller controls the data transferring from the vector merge register to the SP.

Vector compare instructions clear to zero all bits of the VM register that are higher than the current vector length. When the the last compare operation is ready to write to the VM register, it does a parallel load of all higher bits. This method allows chaining into a compare instruction. Also, the compare under mask instructions require the VM bits to operate conditionally.

4.3.4.1 VM Bit Output Staging Controller

The register has three sources for data which transfer between the VP and the SP:

- An even VM bit
- An odd VM bit
- The value saved from the previous cycle transfer

The VM Bit Output Staging Controller (VMO) shifts the VM bits serially to the SP for load and store under mask operations. During each VP data path clock a pair of VM bits transfers to the controller. The value of the VM bits determines the number of bit transfers to the SP. The VMO maintains a simple state machine to accept transfer requests from the load and store function micro controller each VP clock cycle, and transfer a VM bit to the SP each system clock cycle. The VMO extends the VP clock cycle if a VM bit is ready to transfer and the SP is not requesting the transfer yet.

4.3.4.2 State Save and Restore Data

The SP informs the VP that a fault has occurred. The VP ignores this fault condition until the SP memory queue is empty. When both conditions exist, the processor recognizes the fault on the next clock cycle. Refer to the Faults section in this chapter.

Once the initial micro word loads into each micro controller, the Load and Store micro controller uses the sequence through the state save micro code. The sequence code holds data while the control state shifts. For a state save, all external registers scan out to a state save bus and transfer to memory. Next all data paths transfer to memory. To restore the code, the bus shifts first the data from memory, then the control state scans into external registers. This method provides a single section of state save and restore micro code for each controller though for each context swap the controller must save that additional state.

The restore staging logic holds data for function unit opcodes during a return from a context sequence. The input comes from the VM register gate arrays and remains stable for only one VP clock period (80 ns). The SALU and SMUL gate arrays require the data to be stable longer than one system clock period.

4.3.5 Clock Generation Logic

The clock generation logic of both the VPC and VPD boards controls all clocks for the VPD board and controls the phase of the VP 80 ns clocks. The clocks extend if the memory system can not complete or accept a load and store request. If the DIVX gate arrays can not complete a divide and square root operation in one cycle, the DFU controller extends the VP clocks. The clock control logic also initiates a fault cycle when a fault occurs.

The VP has two data path clocks. These have a minimum period of two system clocks but can extend indefinitely in increments of the system clock. clocks on the VPD board include the following:

- ECL clocks
- TTL clocks
- Hold clocks

Table 4–8 lists the clocks of the VP by their logic, and includes their duty cycle, cycle time, and explanation:

Table 4–8, Vector Processor Clocks

CLOCK TYPE	DUTY CYCLE (percent high/low)	CLOCK NAME	CYCLE EXPLANATION
ECL	Free running	CLK_FREE	40 ns Free running system clock
	75 / 25	CLK_PAR	40 ns Clocks on HARDERR while other clocks become extended
		CLK_SYS	40 ns Free running system clock
	87.5 / 12.5	CLK_VP	80 ns VP data path clock—minimum period of two CLK_SYS cycles Can extend in increments of CLK_SYS periods
		CLK_SYS_FLT	40 ns System clock which extends for state save and restore context shifting During memory fault, scans state from VPC board
TTL	50 / 50	CLK_VRF	80 ns Clock for the Vector Register File Delayed for several (how many) ns
		CLK_AFP_SCALAR	80 ns Clock scalar latches in VRF for ALU. Can run when CLK_VRF extends, to allow scalar value transfer to ALU
		CLK_MFP_SCALAR	80 ns Clock scalar latches for MFU Can run when CLK_VRF extends, to allow scalar value transfer to MFU
		CLK_DFU	80 ns Clocks the DIVX gate array Continues to run when a result is not complete
Hold	50 / 50	CLK_HOLD CLK_HOLD2; CLK_HOLD3; CLK_HOLD4; CLK_HOLD5;	Latch control signals coming into the VRF from the Vector Control Unit to provide the setup and hold to the VRF

The AFU clock generation logic, on the VPD board, consists of latches and ECL to TTL transistors that send operation codes, output enable signals, and operand latch clocks to the SALU gate arrays. The latch clocks hold the operands and opcodes within the SALU. The clocks have rising clock edges identical to the VRF clock.

The MFU and DFU clock generation logic, on the VPD board, translates ECL control signals to TTL levels and provides the clocks that load the operands and opcodes into the gate arrays.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5
Memory Subsystem

5.1 Overview

The C200 Series Memory Subsystem uses 32-bit (4-byte) words, instead of the 64-bit (8-byte) longwords used in the C100 Series Memory Subsystem. This improves memory system performance for word operations. Word-aligned word operands can be written using a simple write cycle instead of a longer read-modify-write cycle. In addition, memory contention is reduced since word operands only involve one of the two independent halves of memory.

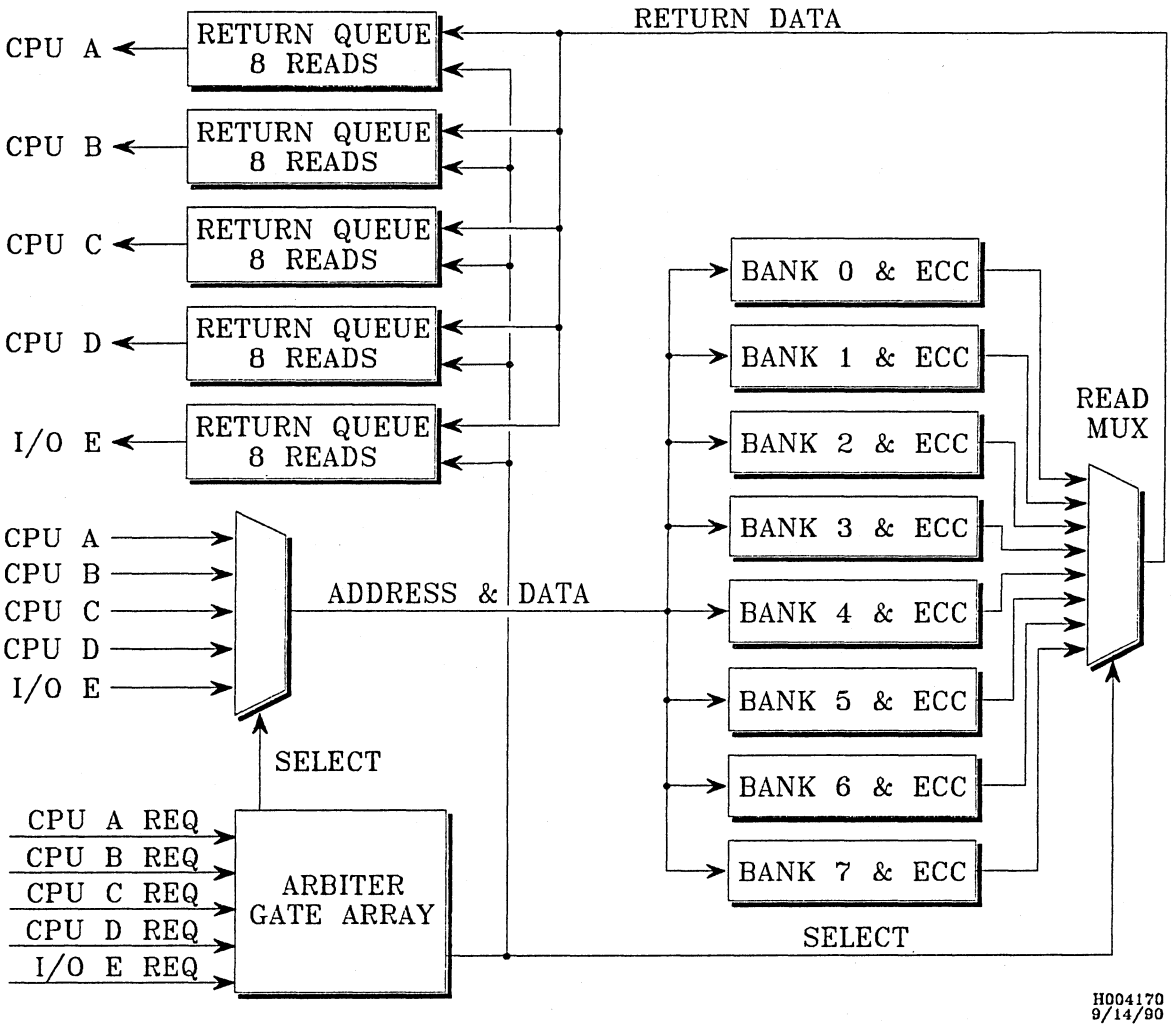
The Memory Subsystem contains multiple integrated memory controllers that allow all types of memory cycles to be pipelined in any order. It also allows more memory operations to begin and to proceed in parallel. This improves memory system bandwidth and reduces the memory system to a single board type. Adding memory board pairs to a C200 Series Memory Subsystem expands both capacity and bandwidth.

The C200 Series Memory Subsystem contains five ports into memory allowing the Memory Subsystem to be simultaneously accessed by up to four CPUs plus the I/O Subsystem. Arbitration is performed independently on each memory board. This allows memory accesses from multiple ports to proceed in parallel without any one processor blocking the entire system. The Peripheral Interface Adapter (PIA) and the Service Processor Unit (SP2/SP4) both access memory via the EBUS (memory port E).

The Memory Subsystem consists of from one to four pairs of Memory Control Modules (MCMs) that provide up to 2 Gbytes of physical memory. Each MCM contains a five port crossbar, arbitration logic, and four Memory Array Modules (MAMs). Each MAM contains two memory banks. Therefore, each MCM contains eight independent memory banks. A memory bank can consist of from one to four 32-bit wide rows of memory with each row containing as many as 4,194,304 locations.

Figure 5–1 shows the functional block diagram of the C200 Series Memory Subsystem:

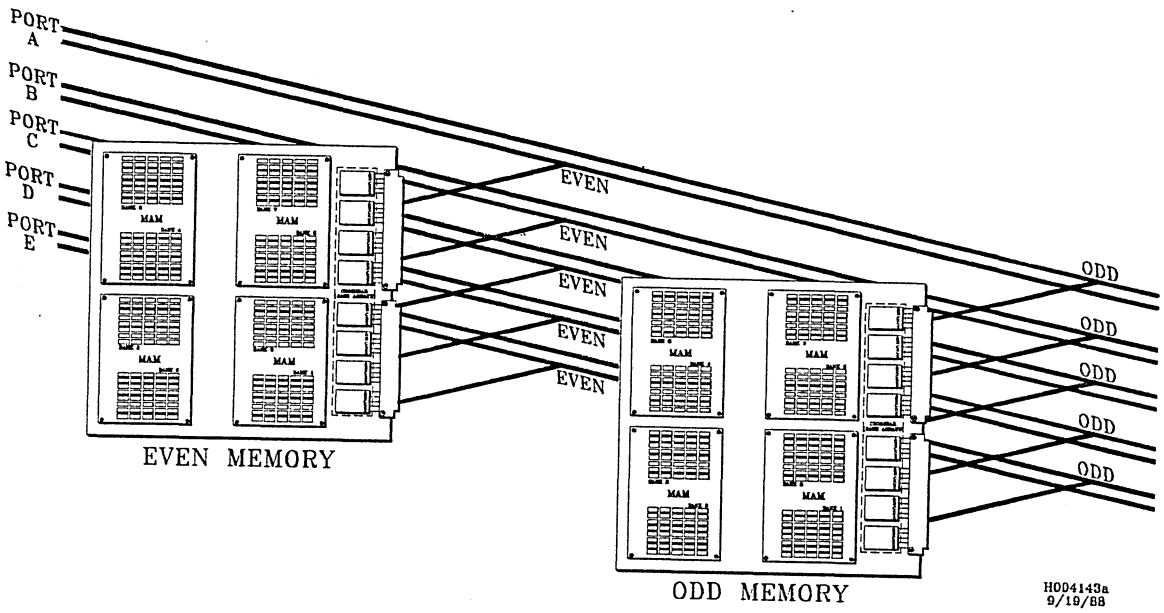
Figure 5–1, Memory Subsystem Functional Block Diagram



5.2 Organization

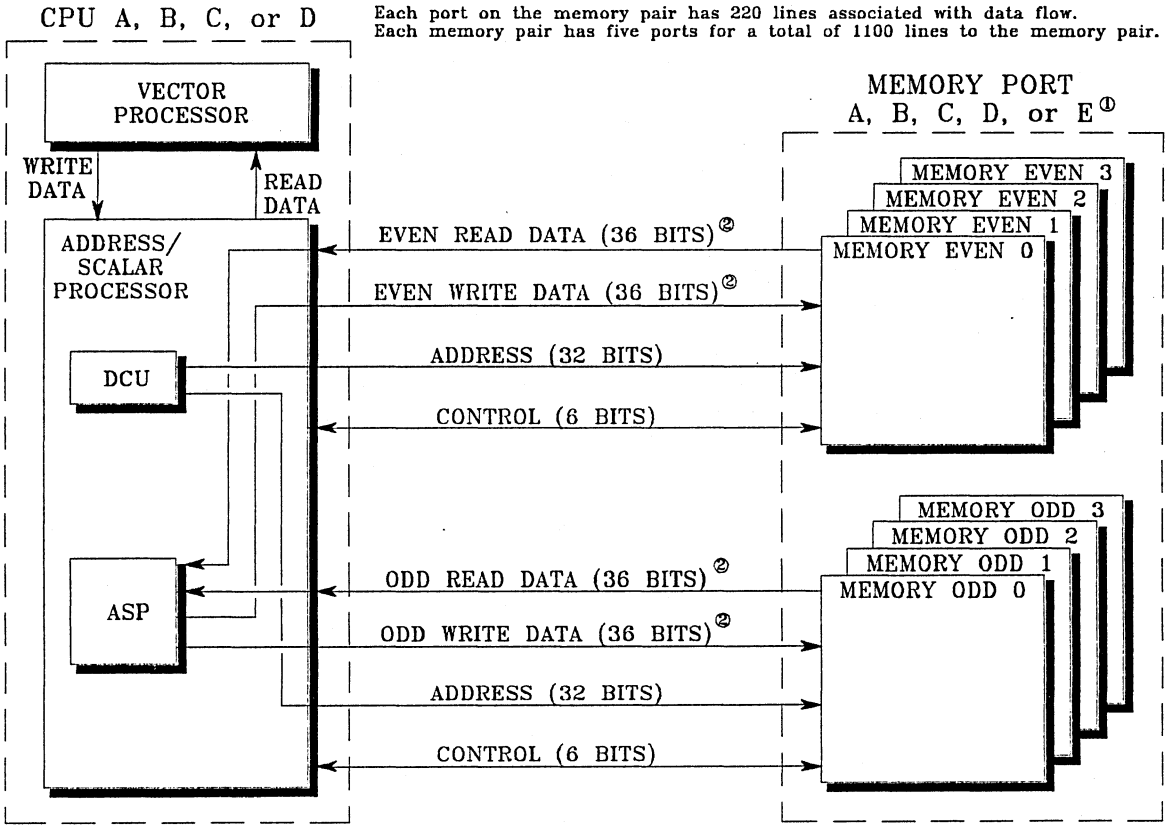
In order to support single-clock longword (8-byte) accesses, the Memory Subsystem is configured into two independent halves. One half contains the even address words (even memory) and the other half contains the odd address words (odd memory) as shown in figure 5–2:

Figure 5–2, Even and Odd Memory in a C200 Series System



The even and odd halves of memory each have their own set of addresses and data and control buses. This allows each processor independent access to each half, as shown figure 5–3:

Figure 5–3, Even and Odd Memory Buses in a C200 Series System



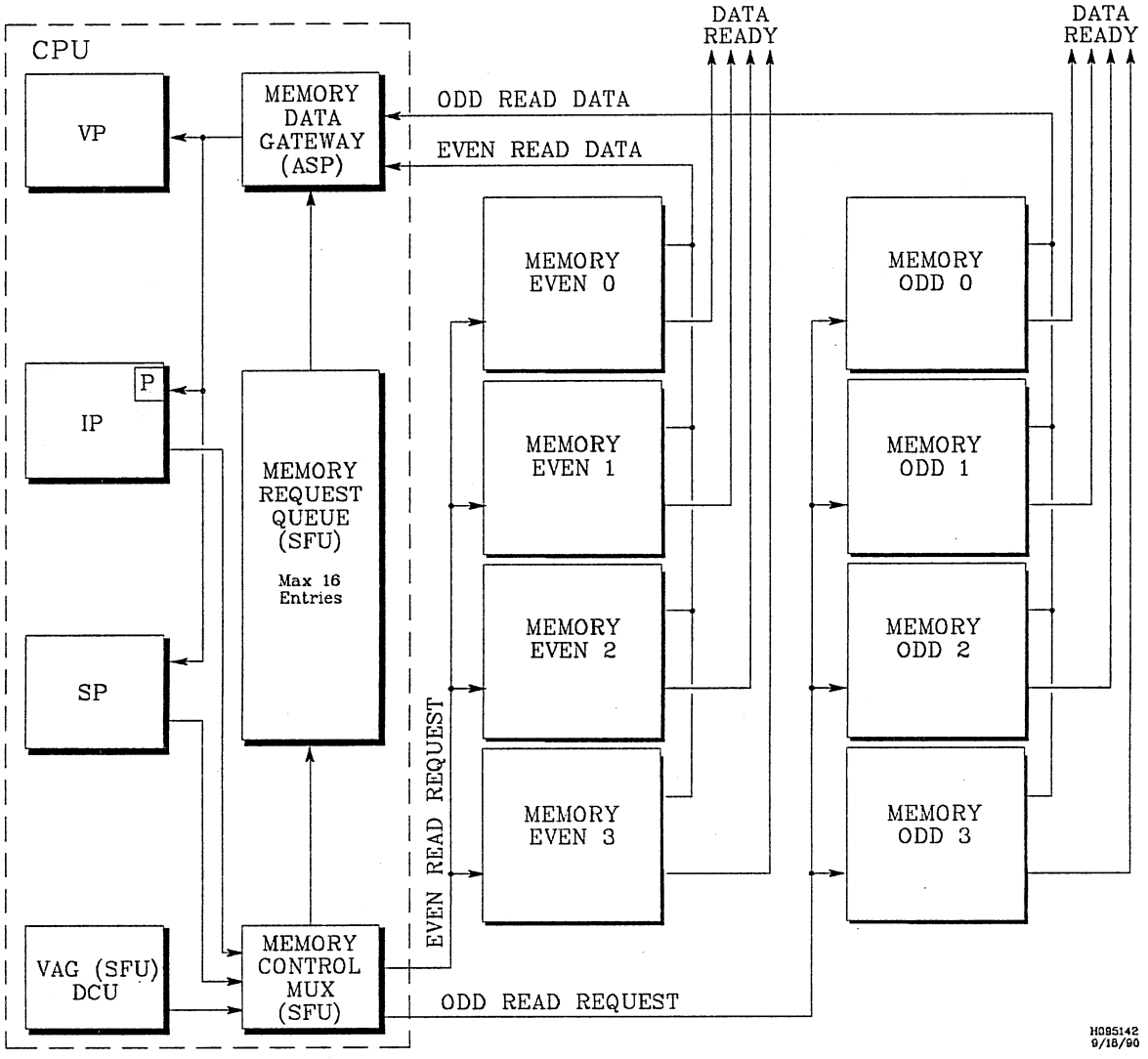
① Port E is reserved for I/O and the Service Processor.

② There are 32 data bits and 4 parity bits.

H004140
9/18/90

A single-memory access may involve the even half, odd half, or both halves of memory, depending on the size and the alignment of the operand. Figure 5–4 shows a block diagram for a CPU read memory access through its corresponding memory port. (CPU A uses Memory Port A, CPU B uses Memory Port B, etc.):

Figure 5–4, Diagram of a CPU Read Request to Even and Odd Memory

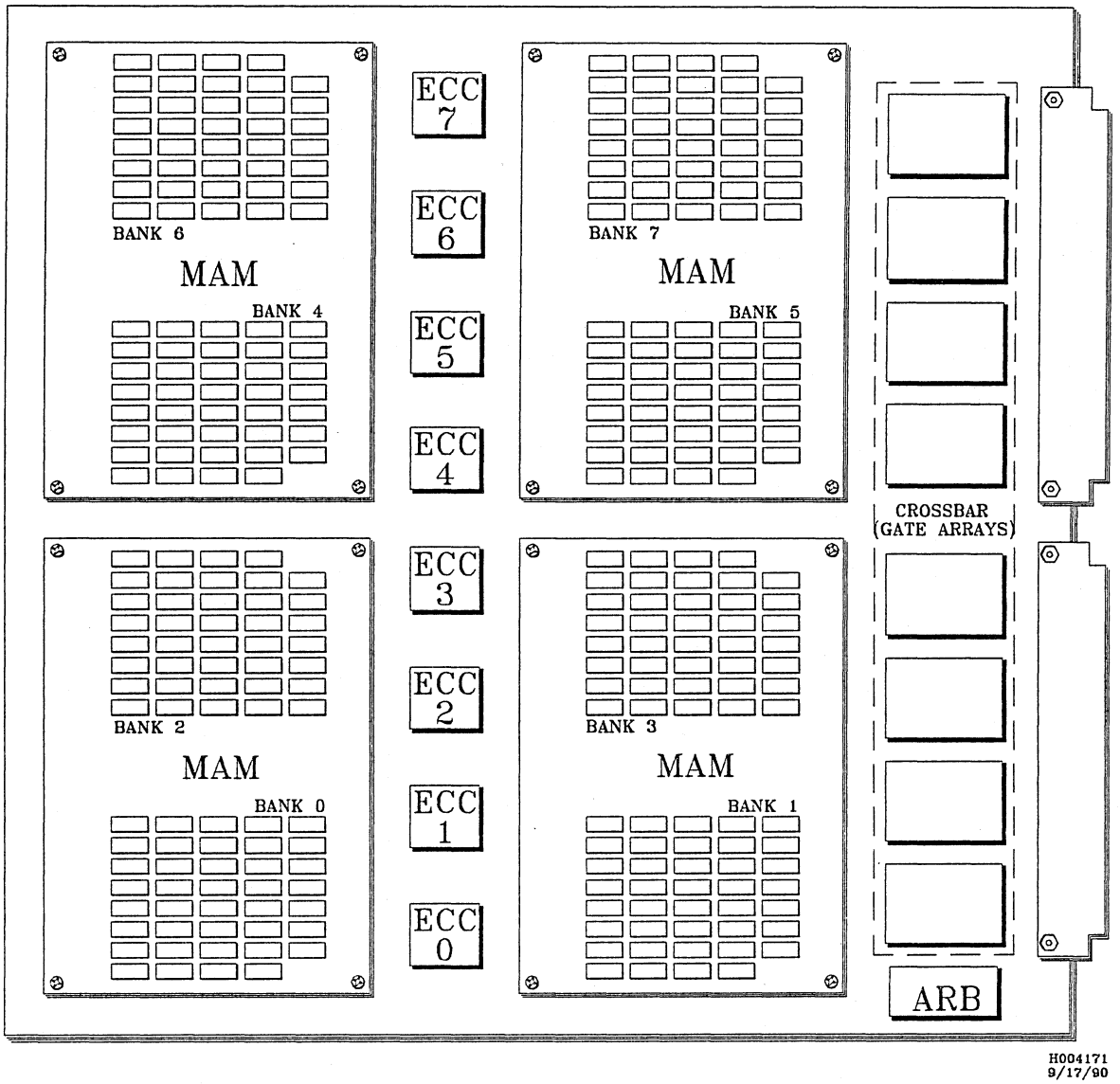


5.3 Memory Control Module

Each MCM has five ports (A, B, C, D, and E) that allow access to any of eight identical banks. The CPUs (up to four) use ports A through D, and the PIA and SPU use port E. CPU A uses port A, CPU B uses port B, etc.

Figure 5–5 shows an MCM board containing four MAMs and eight banks of memory:

Figure 5–5, An MCM Containing 4 MAMs and 8 Banks of Memory

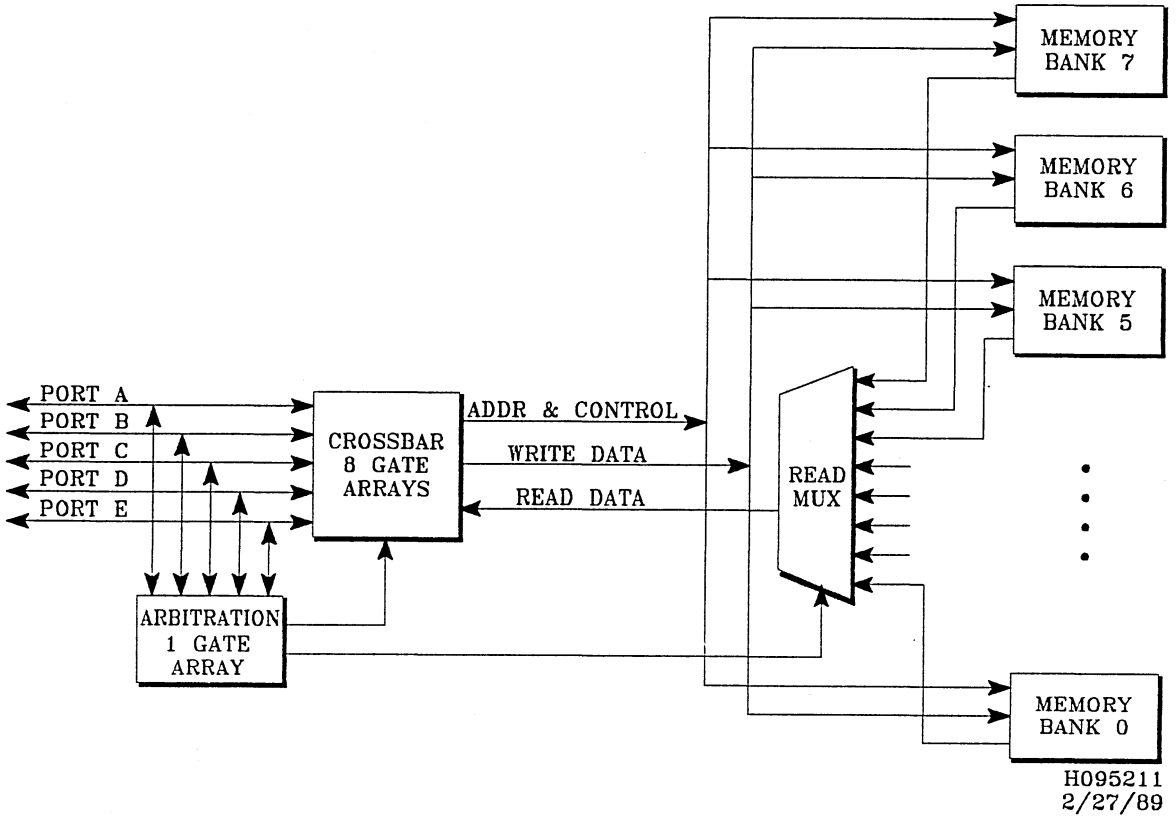


Ports A through E are identical, except that in the priority scheme, requests to port E have the highest priority. The SPU, in conjunction with scan chains, uses port E to handle memory Error Checking Correction (ECC) errors. Additional functions performed by the service processor include initialization of the memory boards and initiation of refresh cycles.

The MCM reports single-bit (correctable) and multi-bit (uncorrectable) ECC errors to the SPU. The memory bank that detected the error loads error identification information into the memory error logger scan chain. If the error was a multi-bit ECC error (fatal), the MCM sends a fatal error signal to the SPU, and the SPU halts the system.

Figure 5–6 shows the functional block diagram of a memory control module for the C200 Series Memory Subsystem:

Figure 5–6, Memory Control Module Block Diagram



A description the Memory Control Module (MCM) hardware follows:

- **Arbitration controller** — The arbitration controller determines which processor can access a memory board. Each memory board has an independent controller. The I/O system has the highest priority, but the controller allows the E port (I/O) to receive only one request if a CPU is waiting with a valid request. However, it is possible for the E port to win access every other clock. If more than one CPU requests memory in a given system clock cycle, the controller gives access to the CPU that won access to memory least recently. The service processor instructs the arbitration controller to generate a refresh cycle.
- **Address and data crossbar** — The address and data crossbar latches the address and control information plus any write data, and passes this information to the memory banks. The crossbar has five processor interfaces, and one memory interface. When the arbitration controller grants access to memory, the address, data, and control moves from the processor interface to the memory interface. The crossbar also holds any read data returning to a processor until that processor accepts the data.
- **Read multiplexer** — The read multiplexer selects read data and parity from one of the eight memory banks. The read multiplexer returns this data to the crossbar, which holds the data until the processor reads it. The multiplexer consists of a 36-bit wide 8-to-1 mux and a bus buffer to prevent unstable data (which could cause parity errors) from reaching the crossbar.
- **Eight memory banks** — The memory banks consist of from one to four 39-bit wide rows of memory (32 data bits and 7 ECC bits). Each row may contain as many as 4,194,304 locations. Two banks, one even and one odd, reside on each removable Memory Array Module (MAM).
- **Control logic** — The control logic supplies signals for the MCM. One signal generates and distributes the various clocks which the MCM uses. Another signal serves as the scan data and control interface to the MCM scan rings (this signal stages the incoming scan control signals, from the service processor, that control the operating mode of the MCM). Another signal reports memory errors to the service processor.

The following sections define the MCM interfaces and hardware.

5.3.1 MCM Processor Port Interface

A processor requests access to memory by asserting a port request signal while driving port cycle type, row select, address, bank select, byte write zone, write parity, and write data signals with the desired information. Independently, if the MCM is ready to accept a request, it asserts a port ready signal. When both signals are asserted during the same clock period, the MCM will latch the control information from the processor and initiate a memory access. All control information must remain valid through the end of the clock period in which the request and ready handshake is completed. If the latched request is a write access, the MCM will assert a port store pending signal until the arbitration controller selects that request for execution.

Once initiated, memory write accesses complete with no further action required on the part of the processor. Memory read and test-and-modify accesses, however, only return data as far as the crossbar without further handshaking from the processor. The processor must assert a port read enable signal to enable the MCM's read data path (port read parity and port read data signals) onto the backplane. When the MCM has return data ready for the processor, it will assert a port read ready signal. The processor must then respond with a port read acknowledge signal to signal acceptance of the read data.

5.3.2 MCM Control I/O

The free running master clock and clock phase signals are are used as time bases for generating the MCM clocks. The board run signal, log run signal, and board halt signals are used to gate all clocks except a memory clock, which runs continuously to allow the memory devices to be periodically refreshed. The board run signal is a low active signal that enables all gated clocks except the clock controlling the memory error logger, which is enabled by the log run signal. The board halt signal is a system-wide signal that may be driven by any board to halt the system in the event of a non-recoverable error. Gated clocks will be stopped high if either the board halt signal or the associated run signal are driven high.

The MCM will report single-bit (correctable) and multi-bit (uncorrectable) ECC errors to the service processor by asserting the soft error detected signal. At the same time, the memory bank that detected the error will load information to identify the cause of the error into its portion of the memory error logger scan chain. If the error was a multi-bit ECC error, the MCM will also assert the hard error detected signal to tell the service processor the error was fatal and will assert the board halt signal to halt the system. These signals (soft error detected plus hard error detected and board halt signals in the case of a multi-bit error) will be held true until the service processor clears the error from the memory error logger scan chain. Parity errors on the write data bus from the processors also cause the MCM to assert the hard error detected and the board halt signals; however, in this case, information describing the error is loaded into the main diagnostic scan chain. Both signals will be held true until cleared by the service processor.

The diagnostic mode signal (DMODE) and scan control (SCTL) signal control the operation of the main diagnostic scan chain while the log diagnostic mode signal and log scan control signals control the operation of the memory error logger scan chain. Both scan chains operate identically as demonstrated by the scan control modes shown in the following table:

Table 5–1, Scan Control Modes

DMODE	SCTL	FUNCTION
0	00	Normal (run) mode
1	00	Parallel load (overrides normal load control)
1	01	Scan left (LSB shifts to MSB)
1	10	Scan right (MSB shifts to LSB)
1	11	Hold (preserves state)

Mode changes are allowed only while clocks are stopped high. Both scan chains use the scan data signal (SCNDAT) to move data on and off the board; therefore, only one of the scan chains may be enabled at a time. Selection is controlled by DMODE: whenever it is set to diagnostic mode, the main diagnostic scan chain is selected; otherwise, the memory error logger scan chain is selected. The scan output data enable signal (ODENA) may be asserted to drive the output of the selected scan chain onto SCNDAT. This signal is also the source of input scan data; thus, when ODENA is active, any data shifted off the board is also shifted back into the chain. New data may be shifted into the scan chain only when ODENA is inactive.

The leading edge of the refresh request signal tells the arbitration controller to set its eight (one per memory bank) refresh request registers. As long as the refresh request signal is high, the arbitration controller will give priority to memory access requests from the processor ports (hidden refresh). When the refresh request signal goes low, however, outstanding refresh requests will be given priority (forced refresh). Once the forced refresh mode has been entered, all outstanding refresh requests will complete in no more than:

(M - 1) + R clocks

where:

M is the number of clock periods required to complete a read-modify-write cycle (the longest memory cycle)

R is the number of clock periods required to complete the refresh cycle

(For a 25MHz clock rate, M = 11 and R = 8)

Once a refresh cycle for a memory bank has been initiated, the refresh request register associated with that bank will be reset. As long as any refresh request remains uncompleted, the arbitration controller will assert the refresh pending signal.

5.3.3 Memory Cycle Types

A Memory Control Module (MCM) supports no-op, read, write, and Test And Modify (TAM) memory cycle types. The basic types of memory cycles are listed in the following table:

Table 5–2, MCM Cycle Types

CYCLE	NAME	DESCRIPTION
00	No-op	No operation performed
01	Read	Read word (4-bytes)
10	Write	Write byte(s) per ZONE bits
11	TAM	Test-and-modify byte(s) per ZONE bits

The type of cycle is determined by the port cycle type signal, and in the case of a write request, by the port byte write zone signal. Read requests always return four bytes (32-bits) of data. The write data and zone bits transferred with a read request are ignored, although the data is checked for correct parity. Write requests can write from one to four bytes of data depending on the number of zone bits that are set.

If all four zone bits are set, the MCM performs a simple write cycle, writing all four bytes at once. Otherwise, it performs an indivisible read-modify-write cycle in order to generate the correct ECC bits for the partially written word. Write operations never return data to the requesting processor, even in the case of a partial write.

Test-and-modify requests cause the MCM to perform an indivisible read-modify-write cycle to update the bytes indicated by the zone bits, as in the case of a partial write. Test-and-modify requests, however, also return all four bytes of the unmodified data. It is generally expected that the write data associated with a test-and-modify request will either be all zeroes (test-and-clear) or all ones (test-and-set); however, the MCM will function as expected regardless of the write data pattern. A write request or a test-and-modify request with no zone bits set will still perform a read-modify-write cycle but without modifying the data. Since the MCM always corrects single-bit ECC errors on the read portion of a memory cycle, a write request with no zone bits set makes an ideal memory scrub operation.

5.3.4 Arbitration Controller

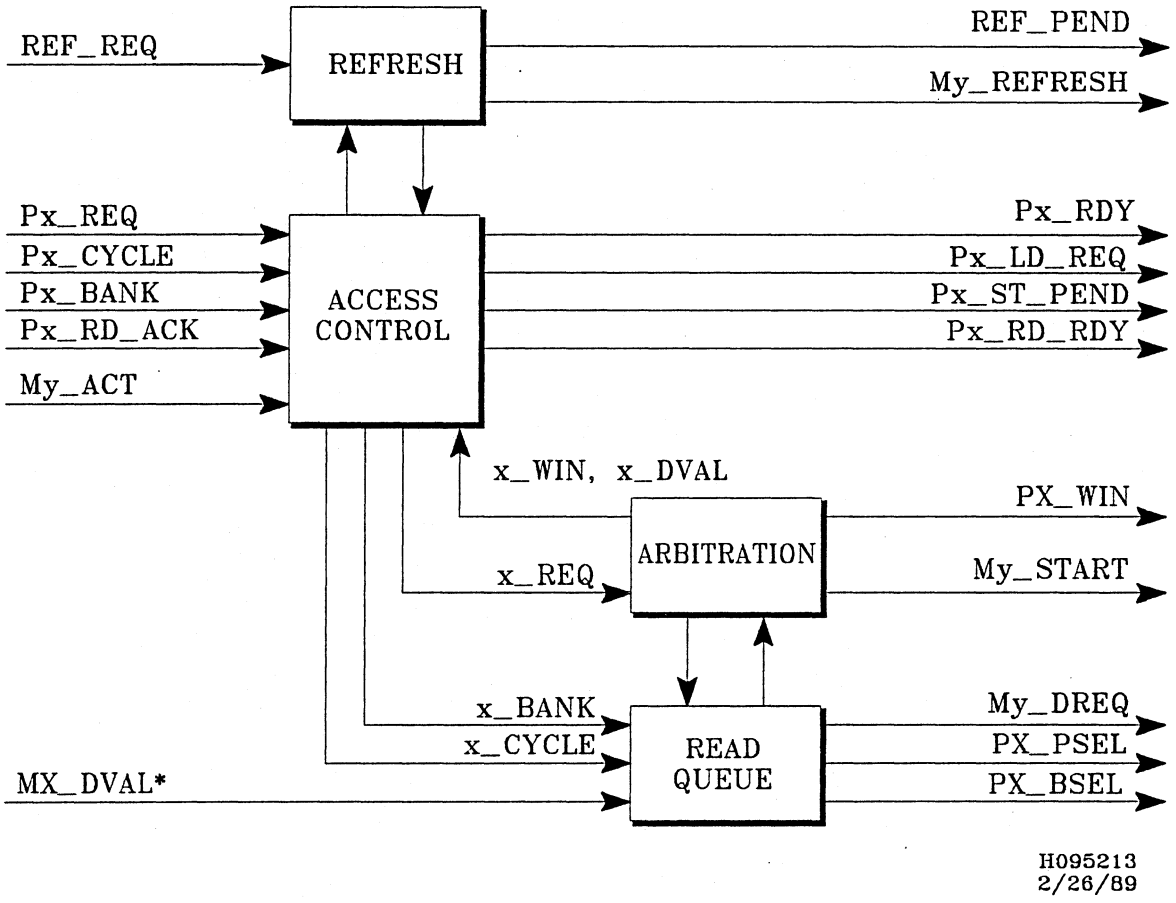
The arbitration controller determines on a clock by clock basis which processor should be granted access to the memory board. It also generates refresh requests to the memory banks upon command from the service processor. Each memory board in the Memory Subsystem has its own independent arbitration controller so requests to one board have no effect on requests to the other boards. A request is considered valid if the desired memory bank is not currently busy or being refreshed and, in the case of a read request, if there is room in the crossbar to hold the returned data. Arbitration is required if more than one valid memory request exists in a given clock cycle. The priority scheme used gives highest priority to the I/O system (port E), but does not grant it access twice in a row if any other processor has a valid request to the board at the same time. The four CPUs (ports A–D) have equal priority. Contentions between CPUs are resolved by granting access to the CPU that won access to memory least recently. The arbitration controller is implemented entirely within a single memory arbitration gate array. The following is a brief description of the arbitration gate array.

The arbitration controller consists of the four main sections:

- Access control logic (with sections for each of the five ports)
- A request arbiter
- A read control queue
- Refresh control logic

Figure 5–7 shows the functional block diagram of the arbitration controller for the C200 Series Memory Subsystem:

Figure 5–7, Arbitration Controller Block Diagram



Memory access requests from the five ports are registered in the access control logic. Two requests per port can be registered—one in the input staging register and one in the overflow register. The input staging register is always loaded first. If this request cannot win access to memory, a second request may be loaded into the overflow register. Further requests are blocked by de-asserting the port ready signal for the appropriate port. Only the requests in the input staging registers can be passed to the request arbiter.

Once a request has been registered, it is tested to see if the requested memory bank is currently busy or scheduled for a forced refresh. In addition, read requests are tested to see if there is room in the crossbar for the data that will be returned from memory. The access control logic maintains a request counter for each port that keeps track of the number of outstanding read requests for that port. The request counter for a given port counts up each time a read request for that port wins access to memory and counts down each time read data is accepted by its corresponding processor. As long as the request count for a port remains less than eight, there is room in the crossbar for additional data.

The access control logic maintains one additional counter per port that is not used in validating memory requests. These data counters are used to keep track of the amount of data actually in the crossbar. The data counter for the port pointed to by port select signal counts up when the multiplexed data valid signal is asserted indicating that data has returned to the crossbar and counts down when the port read acknowledge signal is asserted indicating that data has been accepted by the processor. The corresponding port read ready signal is asserted as long as the data counter for a processor is nonzero.

The request arbiter looks at each of the validated requests (if any) from the five ports and determines which one should be granted access to memory. It generates a three-bit code, the port win signal, to indicate which port has won access to memory and asserts the start signal to the memory bank desired by the winning port. The priority scheme used gives highest priority to port E provided it did not win the previous access. If the E port did win the previous access, it is given lowest priority. The four CPU ports (ports A–D) alternate in priority depending on the order of their previous accesses. The more recently a port won access to memory, the lower its assigned priority.

The read control queue monitors the port win signal from the request arbiter and the bank select and cycle type from the winning port to determine if a read request has won access to memory. If so, the port and bank select codes are written into the read control queue. The outputs of the queue are used to generate the port select, port bank select, and the memory data request signals to control the return of data from the eight memory banks. This ensures that data is returned to the crossbar in the same order that it was requested.

The refresh control logic is responsible for generating refresh requests to the eight memory banks. When the refresh control logic detects a low to high transition on the refresh request signal from the SPU, it sets all eight refresh request registers to one indicating that all eight memory banks need to be refreshed. As long as the refresh request signal remains high, the refresh control logic is in hidden refresh mode. In this mode, an access request to a memory bank will override a refresh request. Only banks that need to be refreshed and are not currently active or being started by the request arbiter will be told to begin a refresh cycle via the memory refresh signal. Once the refresh request signal goes low, however, the refresh control logic enters forced refresh mode. In this mode, any bank which has not yet been refreshed will receive the memory refresh signal as soon as that bank goes inactive. Requests to those memory banks are blocked by the access control logic.

The refresh control logic also maintains eight refresh pending registers. These are set at the same time as the refresh request registers but are cleared when a refresh operation for the associated bank completes instead of when it starts. The arbitration controller asserts the refresh pending signal to the SPU as long as any one of the refresh pending registers remains set.

5.3.5 Address and Data Crossbar

The address and data crossbar is responsible for latching the address and control information plus any write data associated with a memory access request and for passing this information to the memory banks when that request is honored. The crossbar is also responsible for holding any read data returned by the memory banks until it is accepted by the requesting processor.

The address and data crossbar consists of the following:

- Five processor interfaces
- One memory interface
- Eight crossbar gate arrays

Each crossbar gate array consists of the following two basic logic blocks:

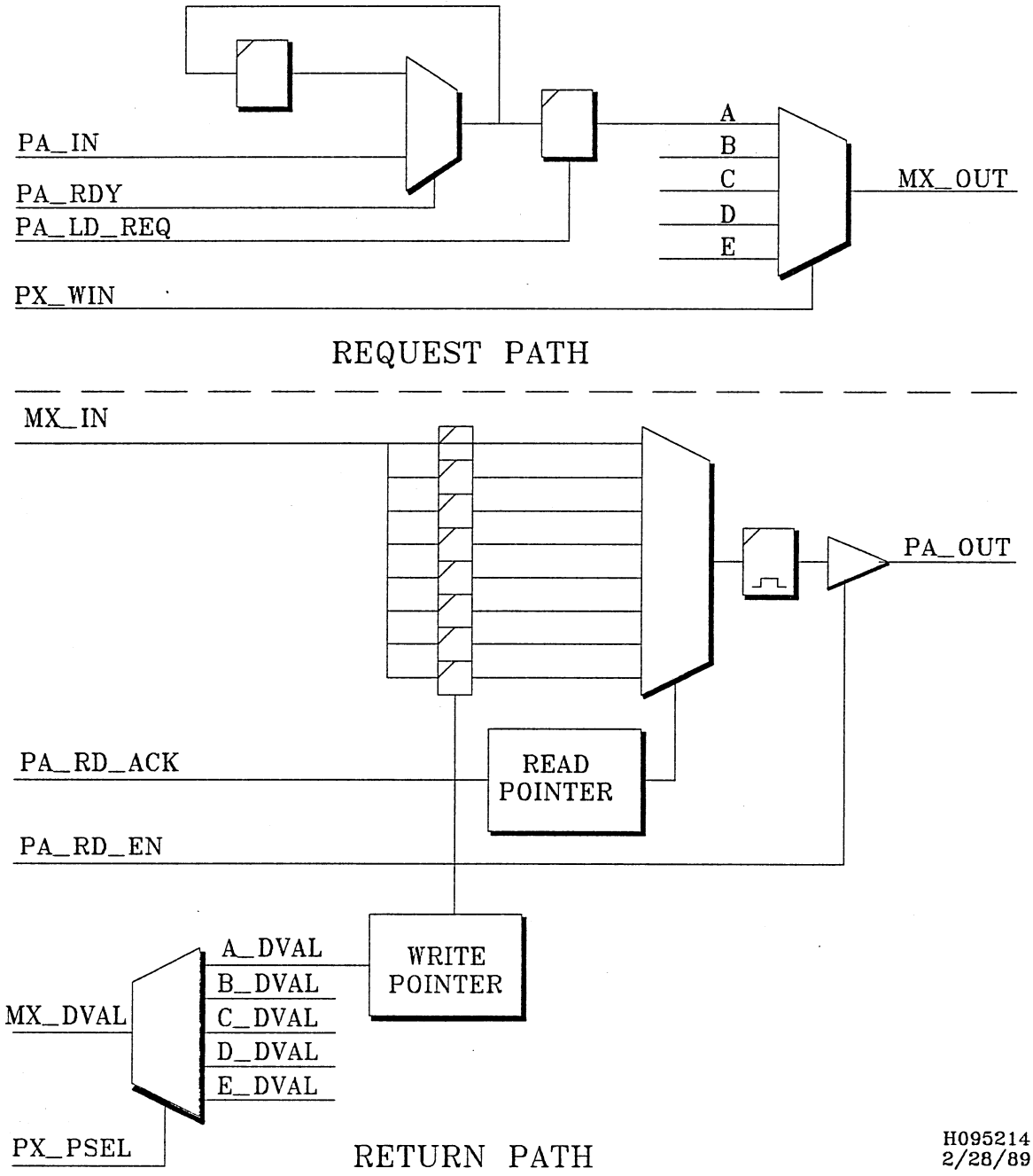
- A write multiplexer
- A read multiplexer

When the arbitration controller grants one of the processors access to memory, the address, data, and control associated with that access is moved from the selected processor interface to the memory interface where it is available to the memory banks. Data returning from memory follows a reverse path from the memory interface to one of the processor interfaces. This path is also directed by the arbitration controller. Each crossbar gate array handles a portion of the address, data, and control.

The following is a brief description of the crossbar gate array. The crossbar gate array is basically a multiport data switch with internal storage. The write mux data path is nine bits wide and the read mux path is five bits wide. Both operate under the control of the arbitration gate array.

Figure 5–8 shows the functional block diagram of the address and data crossbar for the C200 Series Memory Subsystem:

Figure 5–8, Address and Data Crossbar Block Diagram



The input structure of the write mux is identical to that of the arbitration gate array and consists of the following:

- Five input staging registers
- Five overflow registers

Only information in the input staging registers can be passed to the output of the write mux. The port load request signal controls the loading of the input staging registers. When asserted, the port load request signal tells the crossbar to load new information into the corresponding input staging register. This information comes from either the device inputs or the associated overflow register depending on the state of port ready signal. The port ready signal also controls the loading of the overflow register. The output of the write mux is controlled by the port win signal. This three-bit code from the arbitration controller selects the contents of one of the five input staging registers and passes them to the write mux output.

The read multiplexer logic is duplicated for each of the five processor ports. Each copy of the read multiplexer logic consists of the following:

- Eight 5-bit latches
- An 8-to-1 multiplexer
- A write pointer
- A read pointer

All five sets of latches are fed by the common memory input port. The five write pointers are used along with the port select signal from the arbitration controller to select which (if any) of the latches is to receive new data from memory. When data is being returned from memory, the port select signal will select the port to receive the data. The latch pointed to by the write pointer for that port will open when the memory clock signal goes low. When no data is being returned from memory, the port select signal is set to 7 and none of the latches will open. The combination of the port select signal being any value except 7 ((!= 7) and the multiplexed data valid signal active causes the write pointer for the selected port to advance.

The read pointer in each of the five sections of the read mux selects which of the data latches will pass data to its processor interface. The selected data is output to the processor when the corresponding port read enable signal is active. The read pointer advances when the port read acknowledge signal is received from the processor indicating that the data has been accepted.

5.3.6 Read Multiplexer

The read multiplexer selects read data and parity from one of the eight memory banks and returns it to the crossbar where it is held until read by the requesting processor (CPU or I/O).

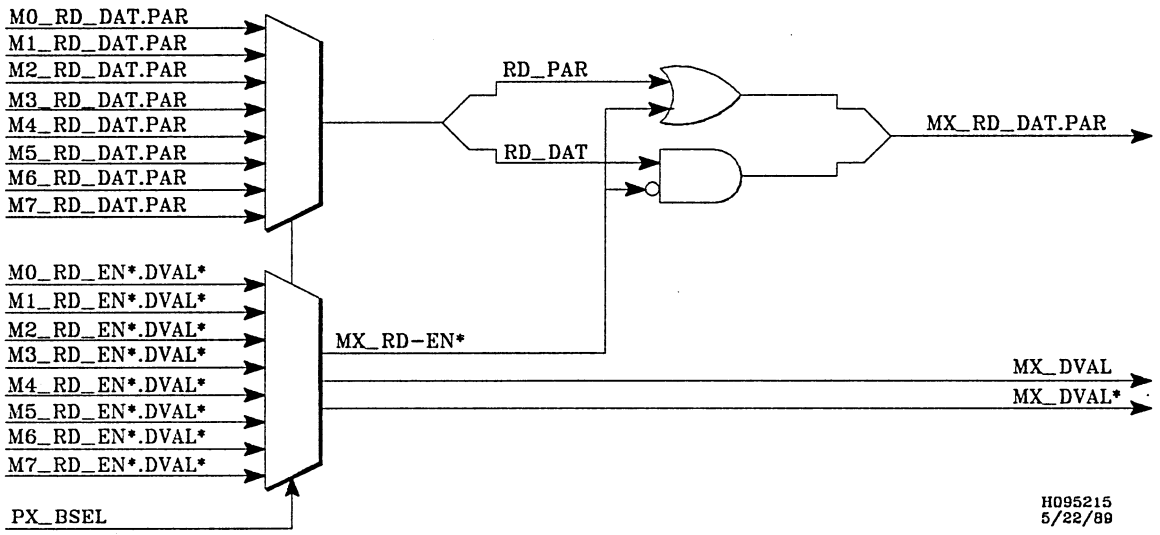
The read mux consists of the following:

- A 36 bit wide 8-to-1 multiplexer
- A bus buffer

The bus buffer is used to prevent unstable data (and thus potentially bad parity) from reaching the crossbar.

Figure 5–9 shows the functional block diagram of the read multiplexer for the C200 Series Memory Subsystem:

Figure 5–9, Read Multiplexer Block Diagram



The arbitration controller determines which memory bank is to return read data via the port select signal. This three-bit code also selects the appropriate multiplexed data valid signal and read bus enable signal. The selected data valid signal is passed on to the arbitration controller and the crossbar to complete the read operation handshake sequence. The selected read bus enable signal is used by the read multiplexer to decide whether or not to pass the selected read data to the crossbar. When the read bus enable signal is false, the output of the read multiplexer is forced to data of all zeroes and parity of all ones to ensure that the crossbar always receives valid parity.

5.3.7 Clock Generator

The clock generator generates and distributes the various clocks needed by the MCM. The clock generator consists of the following:

- A set of gates and registers
- Tuning delay lines
- Buffers

The set of gates and registers generate the basic clock signals, the tuning delay lines de-skew the clock edges, and the buffers distribute the clocks around the board.

5.3.8 Scan Control

The scan control logic serves as the scan data and control interface to the MCM scan rings. The scan control logic consists of the following:

- Registers
- Decode logic

The registers are used to stage the incoming scan control signals from the SPU. The outputs of the registers are either used directly or are decoded to control the operating mode of the MCM.

5.3.9 Win Queue

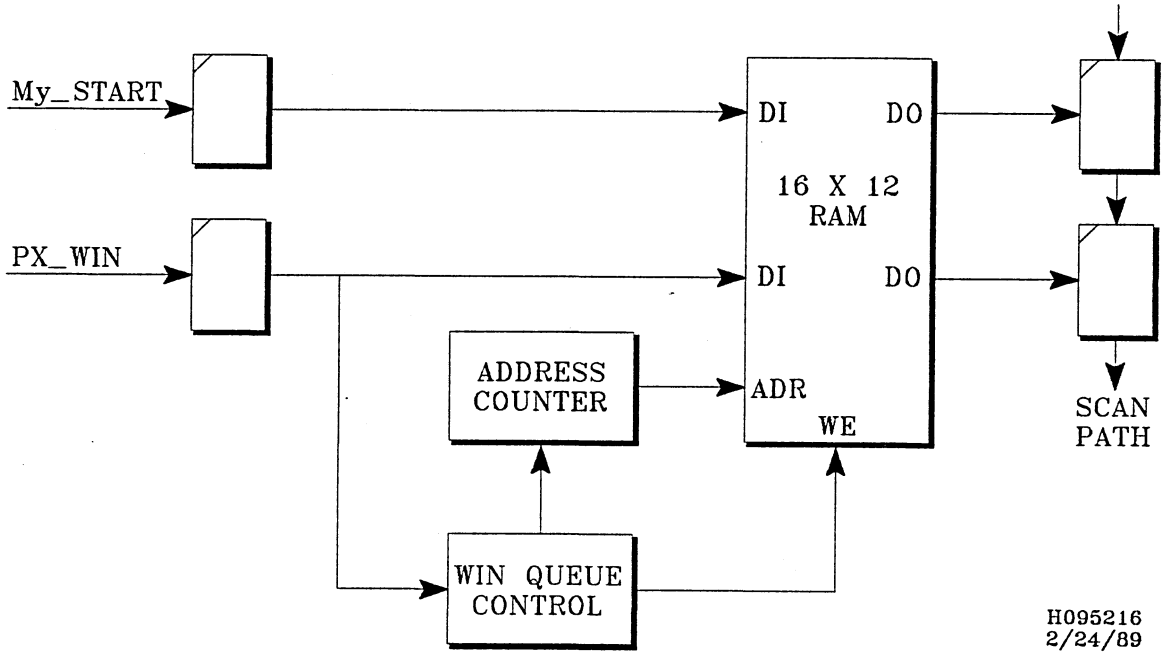
The win queue is a diagnostic aid that keeps a record of the most recent 16 accesses to memory. The win queue consists of the following:

- A 16-word by 12-bit RAM
- An address counter
- Scannable registers
- Associated control logic

When reset, the win queue RAM and the address counter are cleared to zero.

Figure 5–10 shows the functional block diagram of the win queue for the C200 Series Memory Subsystem:

Figure 5–10, Win Queue Block Diagram



The win queue monitors the port win signal and the start signals from the arbitration controller. The port win signal is a three-bit code that indicates which memory port (if any) is currently being granted access to memory. The start signals are eight discrete signals that indicate which one (if any) of the eight memory banks is to be accessed. These signals are registered and then tested for the presence of a memory "win," i.e., the port win signal is not equal to 7. If a win is detected, the win queue writes the port win signal and the start signal to the RAMs and then increments the address counter. When the address counter reaches `0xf`, an increment will cause it to wrap back to `0x0` and begin overwriting the oldest entries in the queue.

It is also possible to write the win queue via scan. If a zero is scanned into the win queue WRITE* bit in the MCM scan ring and a scan HOLD clock is generated, the contents of the port win signal and the start signal (which may also be loaded via scan) will be written into the win queue RAM. In this case, the win queue address will not be incremented. This function is intended to assist testing of the win queue.

The outputs of the win queue RAM are sent to registers that are part of the MCM scan ring. The contents of the RAM location pointed to by the win queue address are transferred to the scan ring at each normal or scan LOAD clock edge. If the win queue is read via scan LOAD, the win queue address will automatically decrement to allow easy access to the entire queue.

5.3.10 Miscellaneous Logic

The miscellaneous logic block consists of the following:

- The COP logic
- Red and green LEDs
- Hard and soft error logic
- The memory bank reset logic

The COP logic consists of the following:

- A non-volatile memory
- A control pulse generator
- ECL-TTL translators

The COP chip is used to hold details (part and serial numbers, revision letters, etc.) about the board. The COP chip is accessed via the four least significant bits of the MCM scan ring.

The red and green LEDs are controlled from bits <5> and <4> of the MCM scan ring. Since they are driven from the same ECL-to-TTL translator as the COP chip, a scan hold clock must be given after loading the scan ring to actually write new values to the LEDs.

The hard error logic ORs the fatal error signals on the MCM into bit <6> of the MCM scan ring. Once the hard error is registered, it is driven off the memory board as the hard error signal and halt signal. If hard errors are disabled via bit <7> of the MCM scan ring, no hard errors are reported. Hard errors are also blocked if the reset or diagnostic mode signals are active.

Bits <8..15> of the MCM scan ring are reset enables for the eight memory banks. If enabled, the reset signal will be asserted to the appropriate memory bank(s) whenever the reset signal is active. This reset control allows the non-memory portions of the MCM to be reset without potentially destroying the contents of memory.

The soft error logic is similar to the hard error logic. It ORs the various non-fatal error signals into bit <6> of the MCM error logger scan ring. Once the soft error is registered, it is driven off the memory board as the soft error signal. If soft errors are blocked via bit <7> of the log ring, by the reset signal, or the log diagnostic mode signal no soft errors are reported. Bits <0..5> of the log ring are unused.

5.3.11 Memory Bank

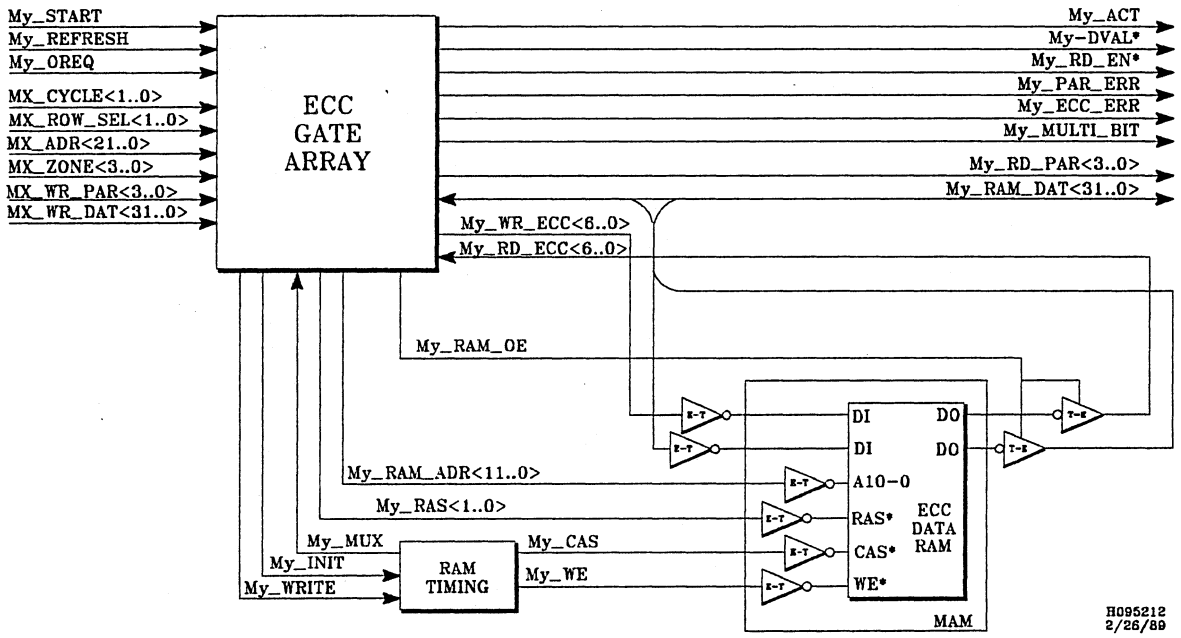
An MCM contains eight independent memory banks. A memory bank consists of the following:

- An Error Checking and Correction (ECC) gate array
- Delay lines
- ECL-to-TTL and TTL-to-ECL level translators
- Dynamic RAMS (DRAMS) for one bank

The DRAMs for two memory banks are located on a Memory Array Module (MAM).

Figure 5–11 shows the functional block diagram of a memory bank for the C200 Series Memory Subsystem:

Figure 5–11, Memory Bank Block Diagram



The ECC gate array is the controlling element of the memory bank; it performs all functions necessary to read from and write to memory. The delay lines are used to guarantee the proper sequencing of address and control signals to the MAM. Read/write data is converted from/to TTL by the ECL-to-TTL level translators to correctly interface to the DRAMs on a MAM. Additional translators are provided on a MAM to convert the address and control signals to TTL levels.

5.3.11.1 ECC Gate Array

The Error Checking and Correction (ECC) gate array is the main controlling entity for the memory bank. A state machine inside the ECC gate array generates the appropriate signals to generate the following four basic types of memory cycles:

- Read
- Write
- Read-modify-write
- Refresh

Additional signals are generated to allow the arbitration controller to keep track of memory bank activities. The ECC gate array also handles data transfer between the crossbar and RAM, including the generation and checking of parity bits for crossbar data and ECC bits for RAM data. Any single-bit errors in the RAM data are corrected before the data is returned to the crossbar or used in a read-modify-write cycle. Latches inside the ECC gate array are used to hold the address, data, and control information needed to perform a memory operation. There is also circuitry to perform row/column address multiplexing for the DRAMs and an address generator for refresh operations.

5.3.11.2 Memory Array Module (MAM)

A Memory Array Module (MAM) is a removable board that contains the following:

- The DRAMs for two banks
- ECL-to-TTL translators

The ECL-to-TTL translators are for the address and control signals needed to access the DRAMS. The rows of DRAMs share addresses and write and read data. There are two copies of the write enable and Column Address Strobe (CAS) signals. Only the Row Address Strobe (RAS) signal is distinct for each row in a bank.

The DRAMs for a bank are organized into from one to four rows of 39 single-bit-wide devices. A process can access one row of data RAMs in a single memory cycle. Thirty two bits of each 39-bit word are used to store data; the remaining seven bits are used to store parity bits for Error Checking and Correction (ECC).

Depending on the size and the alignment of the operand, a single memory access may involve even, odd, or both halves of memory.

5.4 Capacity

Each Memory Control Module (MCM) contains 4 Memory Array Modules (MAMs) and each MAM contains two memory banks. Therefore, each MCM contains a total of eight independent memory banks. A memory bank can consist of from one to four 32-bit wide rows of memory with each row containing as many as 4,194,304 locations. Since the memory devices themselves actually reside on removable MAMs, it is possible to change the capacity of an MCM simply by replacing the MAMs.

The current MCM design supports five sizes of MAMs:

- 4 Mbytes (2 rows of 256 Kbit DRAMs per bank)
- 8 Mbytes (1 row of 1 Mbit DRAMs per bank)
- 16 Mbytes (2 rows of 1 Mbit DRAMs per bank)
- 32 Mbytes (1 row of 4 Mbit DRAMs per bank)
- 64 Mbytes (2 rows of 4 Mbit DRAMs per bank)

The memory chip size to MAM relationships are shown in the following table:

Table 5–3, Memory Chip Size to MAM Population

POPULATION	CHIP SIZE		
	256 Kbit	1 Mbit	4 Mbit
Half	N/A	8 Mbyte	32 Mbyte
Full	4 Mbyte	16 Mbyte	64 Mbyte

The following table shows the total memory system capacity for various combinations of MCMs and MAMs:

Table 5–4, MAM to MCM Configuration

MCM	MAM SIZE				
	4 Mbyte	8 Mbyte	16 Mbyte	32 Mbyte	64 Mbyte
1 MCM	16 Mbyte	32 Mbyte	64 Mbyte	128 Mbyte	256 Mbyte
1 MCM Pair	32 Mbyte	64 Mbyte	128 Mbyte	256 Mbyte	512 Mbyte
2 MCM Pairs	64 Mbyte	128 Mbyte	256 Mbyte	512 Mbyte	1 Gbyte
3 MCM Pairs	96 Mbyte	192 Mbyte	384 Mbyte	768 Mbyte	1.5 Gbyte
4 MCM Pairs	128 Mbyte	256 Mbyte	512 Mbyte	1 Gbyte	2 Gbyte

As shown in the preceding table, the range of memory system capacity is 16 Mbytes for a single MCM with 4-Mbyte MAMs to 2 Gbytes for four pairs of MCMs with 64-Mbyte MAMs. As can be seen, there are several ways to achieve the same memory capacity.

> **NOTE**
>
> Installing fewer memory boards decreases the degree of interleaving causing a reduction in the available memory system bandwidth.

5.5 Memory Interleaving

Interleaving is the process of using low order address bits to access the independent memory banks in odd and even memory. Since each memory board holds eight independent banks, eight memory requests may apply to the memory on one board. If a series of requests is sequential, each request goes to a different bank and the memory processes the requests in parallel.

The degree of interleaving is equal to the number of independent memory banks. A single memory board (with eight banks) has eight degrees of interleaving. Since boards must be paired, one odd and one even, one board pair has a degree of 16 for word accesses, or 8 for longword accesses.

If the system contains more than one board pair, interleaving may extend from banks to boards. For example, if a sequence has 16 sequential longwords, the first eight requests go to the first board and the last eight requests go to the eight banks on the second board pair. For four board pairs, interleaving can cross all four boards for longword interleaving of degree 32.

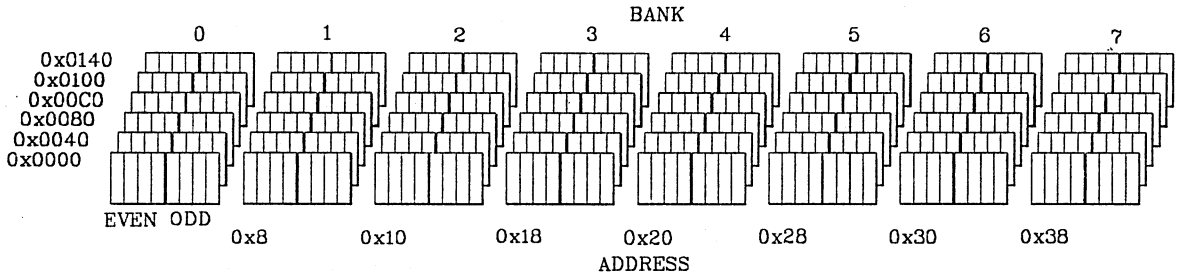
The degrees of interleaving for word and longword accesses are listed in the following table:

Table 5–5, C200 Series Interleaving and Numeric Precision

PRECISION	MCM PAIRS			
	1	2	3	4
32-Bit	16	32	32/16	64
64-Bit	8	16	16/8	32

Figure 5–12 shows eight-way interleaving of one MCM pair in the C200 Series Memory Subsystem:

Figure 5–12, Eight-Way Interleaving of One MCM Pair



8–WAY INTERLEAVING (1 MCM PAIR)

H095144
3/20/89

5.6 Bandwidth

The eight memory banks on each Memory Control Module (MCM) are completely independent. This allows them to be interleaved to support a pipelined access rate of one read cycle or one full word write cycle per clock (not including memory refresh). Assuming a single pair of MCMs with a 40-ns clock cycle, this corresponds to a memory bandwidth of 200 Mbyte/sec (2 words/clock).

All processors (CPU or I/O) attempting to access the same pair of memory boards will be competing for the same 200 Mbyte/sec of bandwidth. The entire bandwidth of a board pair is available to one processor, **only** if no other processors have made requests to that pair. By interleaving memory at the board level and the bank level, a fully configured memory system (four board pairs) will provide a bandwidth of 800 Mbyte/sec.

Although the memory system can consist of up to four board pairs, installing anything less than four pairs will reduce the system bandwidth. Additionally, board level interleaving requires 2 or 4 board pair combinations.

The following table shows the available bandwidth (in bytes per second) possible for various combinations of MCMs.

Table 5–6, C200 Series Memory Subsystem Bandwidth

MCM PAIRS	AVAILABLE BANDWIDTH (bytes per second)
1	200 Mbytes
2	400 Mbytes
3	600 Mbytes
4	800 Mbytes

5.7 Memory Contentions

The C200 Series Memory Subsystem is designed to handle simultaneous memory access requests from any of its five ports without one processor blocking the entire system. To guarantee each processor equal access to memory, The arbitration controller on each MCM reassigns the memory port priorities with each request. Memory contentions will occur if more than one processor attempts to access the same MCM during the same clock, or an attempt is made to access a memory bank without allowing time for comletion of previous accesses to the same bank.

5.8 Memory Addressing

The initial operating system version for the C200 Series processor will be ConvexOS 6.2. This version uses an address space of 30 bits, so two bits of the physical address space will always be zero. Since later versions of the operating system (7.0 and above) will use the entire address space, mapping involving bits <30> and <31> needs to be replaced by lower order memory addressing bits not used for version 6.2 of the operating system. This assumes that machines using ConvexOS version 6.2 have memory boards populated with no larger than 1-Mbit memory RAMs.

The ConvexOS Version 6.2 of the operating system addressing mode affects memory board select and MADDR mapping as described in following paragraphs.

In order to efficently use memory, it is desirable to maximize the time between accessing the same memory bank on transfers involving a large number of consecutive addresses. This reduces the possibility of finding a bank busy when its next transfer is requested.

With more than one memory board pair in the system, all banks in each pair should be accessed before a bank is repeated. Since there are eight banks per board pair and one longword per access, addresses `XXXXXX00` through `XXXXXX3f` will require all eight banks of a board pair. The next address should, therefore, be to another board (if installed). To this end, board selects are done with memory address bits <6> or <7> in place of bits <29> and/or <30> (or <27> for 6.2 addressing mode). Depending on the number of memory board pairs installed, memory will be interleaved by a longword factor of from 8 to 32.

All five ports transmit only 29 of the 32 bits of physical addressing. For reads, the three least significant bits are not used since memory will return the entire word or longword (depending on the board(s) requested) regardless of the byte address. It is necessary, however, to specify the bytes to be affected by a write operation. This is done by an 8-bit ZONE field. Each ZONE bit represents a byte write enable during a memory write operation. A zero in a ZONE bit position causes the corresponding byte to remain unchanged.

Memory boards contain one, two, or four rows of RAMs. One row select bit (least significant bit) is used when memory boards contain two rows of RAMs. If the memory boards contain four rows of RAMs, a second row select bit (most significant bit) is required.

The least significant row select bit is always memory address bit <28>. The most significant row select bit (used only when memory boards have four rows of RAMs) will be memory address bit <31> (or bit <26> for the 6.2 address mode).

The following table lists the memory addressing decode in the C200 Series Memory Subsystem:

Table 5–7, MCM Memory Addressing

MCM PAIRS	REAL MEMORY	DCODE <30>	ROW 0	BOARD 1	BOARD 0	ADDRESS <21..0>	BANK 0 .. 7	MCM E = 0	BYTE 0 .. 3
1	128 Mbyte	-	<28>	-	-	<25..6>	<5..3>	<2>	<1,0>
2	256 Mbyte	-	<28>	-	<6>	<25..7,29>	<5..3>	<2>	<1,0>
3	384 Mbyte	0	<28>	<30>	<6>	<25..7,29>	<5..3>	<2>	<1,0>
		1	<28>	<30>	<29>	<25..6>	<5..3>	<2>	<1,0>
4	512 Mbyte	-	<28>	<7>	<6>	<25..8,30,29>	<5..3>	<2>	<1,0>

Figure 5–13 shows a diagram of the memory addressing decode in the C200 Series Memory Subsystem:

Figure 5–13, MCM Memory Addressing in the Memory Subsystem

```
31  30  29 28   27 26  25..9    8 7 6  5 4 3   2     1  0    Physical Address
    ------ --   -----  -------  -----  -----  ---   ----

    Board  Row  Unused 1 Mbit   MADDR  Bank   MCM   Byte     (with align -> Zone)
    or                Ram                   Odd
    MADDR             Address               or
                                            Even
```

Memory Address	Bus Address
Row sel<0>	ADDR<28>
Ram addr<21..3>	ADDR<25..9>
Ram addr<2..0>	MADDR<2..0> = ADDR<8,7,6> or ADDR<8,7,29> or ADDR<8,30,29>
Bank sel<2..0>	ADDR<5..3>

NOTE: MADDR = Least significant 3 bits of Ram Address

5.9 Memory Loading

The C200 Series Memory Subsystem is organized into 32-bit (4-byte) words to improve memory system performance for word operations. Word-aligned word operands can be written using a simple write cycle instead of a longer read-modify-write cycle.

A single memory access may involve even, odd, or both halves of memory, depending on the size and the alignment of the operand. One row of DRAMs per bank is accessed during any single memory cycle.

The DRAMs for a bank are organized into from one to four rows of 1-bit wide devices. Thirty-two bits (word) are used to store data. The rows of DRAMs share addresses and write and read data. There are two copies of the write enable and Column Address Strobe (CAS) signals. Only the Row Address Strobe (RAS) signal is distinct for each row in a bank.

Figure 5–14 shows an example of an 8-bit load in the C200 Series Memory Subsystem:

Figure 5–14, An 8-Bit Load

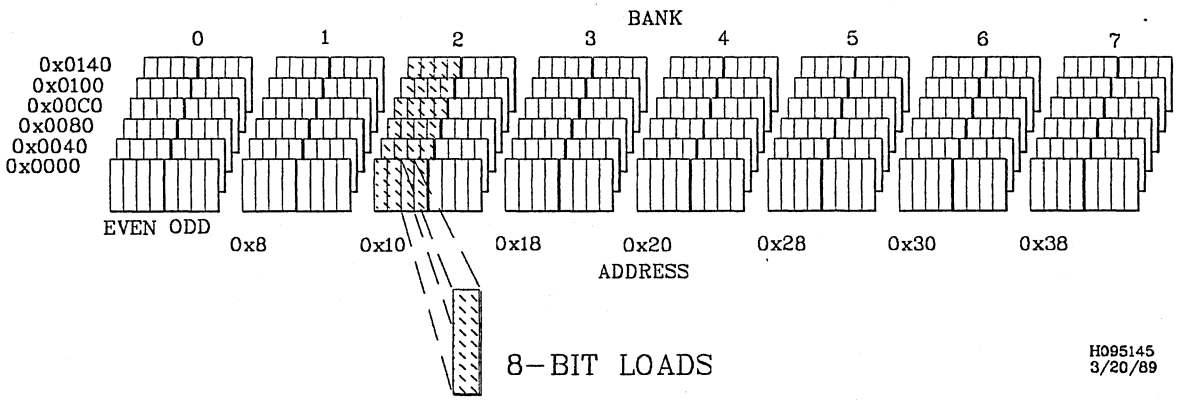


Figure 5–15 shows an example of a 32-bit load in the C200 Series Memory Subsystem:

Figure 5–15, A 32-Bit Load

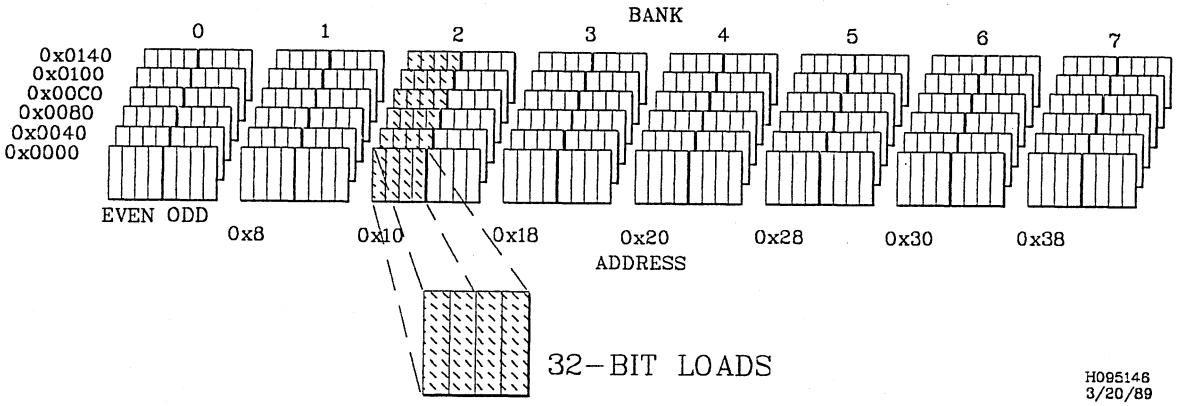
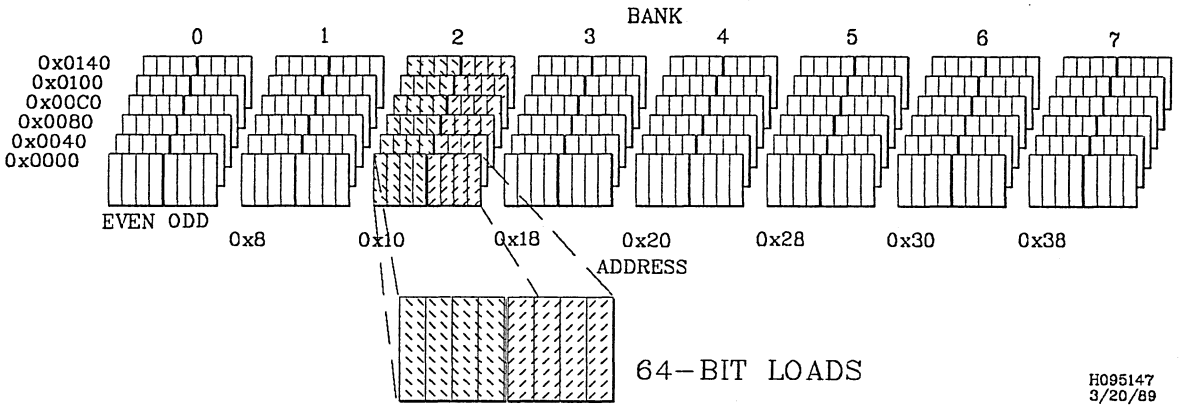


Figure 5–16 shows an example of a 64-bit load in the C200 Series Memory Subsystem:

Figure 5–16, A 64-Bit Load



5.10 MCM Signals

The following sections describe the various MCM signals.

- **Processor Port Signals** — Lists the input and output signals for a processor port (A through E) on an MCM.
- **Control Signals** — Lists the control signals used for clock generation, error reporting, scan operation, and refresh control on an MCM.
- **Arbitration Controller Interface Signals** — Lists the input and output signals for the memory arbitration controller.
- **Address and Data Crossbar Interface Signals** — Lists the input and output signals for the address and data crossbar.
- **Read Multiplexer Interface Signals** — Lists the input and output signals for the read multiplexer.
- **Clock Generator Interface Signals** — Lists the input and output signals for the clock generator.
- **Scan Control Interface Signals** — Lists the input and output signals for scan control.
- **Win Queue Interface Signals** — Lists the input and output signals for the win queue
- **Miscellaneous Logic Interface Signals** — Lists the input and output signals for the miscellaneous logic block.
- **Memory Bank Internal Signals** — Lists the signals internal to a memory bank.
- **Memory Bank Interface Signals** — Lists the input and output signals for a single memory bank.

5.10.1 Processor Port Signals

The following table lists the I/O signals for a processor port (A through E) for an MCM:

> **NOTE**
>
> To simplify matters, these signals are listed in the form "Px_(signal_name)", where "**x**" refers to one of the five processor ports A through E.

Table 5–8, MCM Processor Port Signals

Signal Name	I/O	Function
Px_REQ	I	Port A-E request
Px_CYCLE<1..0>	I	Port A-E cycle type
Px_ROW_SEL<1..0>	I	Port A-E row select
Px_ADR<21..0>	I	Port A-E address
Px_BANK<2..0>	I	Port A-E bank select
Px_ZONE<3..0>	I	Port A-E byte write zones
Px_WR_PAR<3..0>	I	Port A-E write parity
Px_WR_DAT<31..0>	I	Port A-E write data
Px_RDY	O	Port A-E ready
Px_ST_PEND	O	Port A-E store pending
Px_RD_EN	I	Port A-E read enable
Px_RD_PAR<3..0>	O	Port A-E read parity
Px_RD_DAT<31..0>	O	Port A-E read data
Px_RD_RDY	O	Port A-E read ready
Px_RD_ACK	I	Port A-E read acknowledge

5.10.2 Control Signals

The control signals for an MCM are used for clock generation, error reporting, scan operation, and refresh control.

The following table lists the control signals for an MCM:

Table 5–9, MCM Control Signals

Signal Name	I/O	Function
PI-M.100MHZ	I	Master clock
PI-M.PHASE<1..0>	I	Clock phase
SP-M.RUN[1]	I	Board run
SP-M.LOG_RUN[1]	I	Log run
SP-MM.RESET	I	Board reset
BP-BP.HALT	I/O	Board halt
MM-SP.HARDERR	O	Hard error detected
MM-SP.SOFTERR	O	Soft error detected
SP-MM.DMODE	I	Diagnostic mode
SP-MM.SCTL<1..0>	I	Scan control
SP-MM.LOG_DMODE	I	Log diagnostic mode
SP-MM.LOG_SCTL<1..0>	I	Log scan control
SP-M.ODENA	I	Scan output data enable
SP-M.SCNDAT	I/O	Scan data
SP-MM.REF_REQ	I	Refresh request
M-SP.REF_PEND	O	Refresh pending

[1] This signal is active low.

5.10.3 Arbitration Controller Interface Signals

The following list describes each of the input and output signals for the memory arbitration controller.

> **NOTE**
>
> Many of these signals are duplicated for each processor port, or for each memory bank. To simplify matters, these signals are listed in the form "Px_(signal_name)" or "My_(signal_name)", where "**x**" refers to one of the five processor ports A through E and "**y**" refers to one of the eight memory banks 0 through 7. A signal followed by an "*" is active low.

INPUTS:

- **Px_REQ — Request.** From the backplane. Asserted by a processor when it wants access to memory. Requests are only accepted when the arbitration controller is ready as indicated by the corresponding output signal, **Px_RDY.**
- **Px_CYCLE — Cycle type.** Buffered from the backplane. Selects the type of memory cycle to be performed. Must be valid during the same clock period that **Px_REQ** is asserted. The cycle types are:

CYCLE	Operation
00	No-Op
01	Read
10	Write
11	Test-And-Modify

- **Px_BANK — Bank select.** From the backplane. Indicates which memory bank to access. Must be valid during the same clock period that **Px_REQ** is asserted.
- **Px_RD_ACK — Read acknowledge.** From the backplane. Asserted to acknowledge acceptance of read data from the crossbar. This signal is ignored by the arbitration controller if **BD_RUN** is not asserted.
- **My_ACT — Active.** From the memory banks. Asserted by a memory bank when it is processing a memory request.
- **MX_DVAL* — Multiplexed data valid.** From the read multiplexer. Indicates that the currently selected memory bank (refer to output signal **Px_BSEL<2..0>** below) has valid data on the read bus.
- **REF_REQ — Refresh request.** From the backplane. Generated by the SPU to tell the arbitration controller when to refresh memory.
- **CLK — Clock.** From the clock generator. This is the normal 25 MHz system clock.
- **CLK_MX — Memory clock (XBAR version).** From the clock generator. This clock never stops.

- **RESETB — Reset (buffered).** Buffered copy of RESET from the scan control logic block. Used to initialize the arbitration controller.
- **BD_RUN — Board run.** From the scan control logic block. Indicates that the memory board is in the normal run mode, i.e., the **SP-M.RUN*** signal is active and the **BP-BP.HALT** and **SP-MM.DMODE** signals are inactive.
- **GA_SCTLD — Gate array scan control (copy "D").** Buffered copy of **GA_SCTL** from the scan control logic block. These signals control the operating mode of the arbitration controller. The modes are:

SCTL	Mode
00	Normal
01	Hold
10	Load
11	Scan (right)

- **SCAN_IN — Scan in.** From the scan control logic block. This is a buffered copy of the backplane signal **SP-M.SCNDAT.** At this point, it is the Most Significant Bit (MSB) of the right shift scan chain.

OUTPUTS:

- **Px_RDY — Memory ready.** Asserted by the arbitration controller when it is ready to accept a memory request.

> **NOTE**
>
> A buffered version of this signal is also sent to the crossbar.

- **Px_LD_REQ — Load request.** Tells the crossbar to load the current address, data, and control information into its input staging registers.
- **Px_ST_PEND — Store pending.** Asserted as long as a write operation remains latched inside the arbitration controller (and the crossbar).
- **Px_RD_RDY — Read ready.** Asserted whenever read data is available in the crossbar.

- **Px_WIN — Win.** Three bit code indicating which port has just won access to memory. Combinations "101" (5) and "110" (6) are not used. The codes and the winners they define are:

Code	Winner
000	CPU A
001	CPU B
010	CPU C
011	CPU D
100	I/O (Port E)
101	Not Used
110	Not Used
111	- none -

- **Px_PSEL — Port select.** Three bit code indicating which port should receive data from the current read request. The codes are the same as for the output signal **Px_WIN** above.
- **Px_BSEL — Bank select.** Three bit code indicating which bank is currently returning read data.
- **My_START — Start.** Tells a memory bank to initiate a memory access.
- **My_DREQ — Data request.** Tells a memory bank to return read data to the crossbar.
- **My_REFRESH — Refresh.** Tells a memory bank to initiate a refresh cycle.
- **REF_PEND — Refresh pending.** Indicates that a refresh operation has been requested but not all of the memory banks have completed a refresh cycle.
- **SDA_X — Scan data - ARB to XBAR.** Right shift scan data from the arbitration controller to the crossbar.

5.10.4 Address and Data Crossbar Interface Signals

The following is a list of the input and output signals for the address and data crossbar.

> **NOTE**
>
> Many of these signals are duplicated for each processor port. To simplify matters, these signals are listed in the form "Px_(signal_name)", where "**x**" refers to one of the five processor ports A through E.

INPUTS:

- **Px_CYCLE — Cycle type.** Buffered from the backplane. Indicates the type of memory cycle to perform. The cycle types are:

CYCLE	Operation
00	No-Op
01	Read
10	Write
11	Test-And-Modify

- **Px_ROW_SEL — Row select.** From the backplane. Indicates which row of memory devices to access within the selected memory bank.
- **Px_ADR — Address.** From the backplane. This is the actual RAM address used to access memory within the selected row and bank.
- **Px_ZONE — Zone.** From the backplane. This field is only used during write or test-and-modify operations. It indicates which bytes within the 32-bit word are to be updated.
- **Px_WR_PAR — Write parity.** From the backplane. Contains the parity bits for data on the corresponding **Px_WR_DAT** bus.
- **Px_WR_DAT — Write data.** From the backplane. Contains the data bits to be used in a write or test-and-modify operation.
- **Px_RDY — Ready.** Buffered copy of RDY from the arbitration controller. Indicates that the arbitration controller is ready to accept a new memory request. Used in the crossbar to control the input overflow registers.
- **Px_LD_REQ — Load request.** From the arbitration controller. Tells the crossbar to load new information into its input staging registers.

- **PX_WIN — Win.** From the arbitration controller. Three bit code that multiplexes data from the input staging registers of one of the five ports to the MX_ outputs of the crossbar. The win codes and the ports they select are:

> **NOTE**
>
> Port E is selected by default (codes 100 <4>, 101 <5>, 110 <6>, or 111 <7>) if no memory requests are present.

WIN	Port Selected
000	Port A
001	Port B
010	Port C
011	Port D
100	Port E **(Default)**
101	Port E **(Default)**
110	Port E **(Default)**
111	Port E **(Default)**

- **MX_RD_PAR — Multiplexed read parity.** From the read multiplexer. Read parity from the memory bank selected by Arbitration Controller Interface input signal, **PX_BSEL.**
- **MX_RD_DAT — Multiplexed read data.** From the read multiplexer. Read data from the memory bank selected by **PX_BSEL.**
- **MX_DVAL — Multiplexed data valid.** From the read multiplexer. Indicates that the currently selected memory bank has valid data on the read bus.
- **PX_PSEL — Port select.** From the arbitration controller. Tells the crossbar which port should receive data from the current read request.
- **Px_RD_EN — Read enable.** From the backplane. Enables the port read parity and read data drivers.
- **Px_RD_ACK — Read acknowledge.** From the backplane. Indicates acceptance of read data from the crossbar.
- **CLK — Clock. From the clock generator.** This is the normal 25 MHz system clock.
- **CLK_E — Early clock. From the clock generator.** Shifted version of **CLK.** The rising edge of **CLK_E** occurs 6 ns prior to the rising edge of **CLK,** assuming a 40 ns clock cycle.
- **CLK_MX — Memory clock (XBAR version).** From the clock generator. This clock never stops.

- **RESETB — Reset (buffered).** Buffered copy of **RESET** from the scan control logic block. Used to initialize the crossbar.
- **GA_SCTLC, GA_SCTLD — Gate array scan control (copies "C" and "D").** Buffered copies of **GA_SCTL** from the scan control logic block. These signals control the operating mode of the crossbar gate arrays. The modes are:

SCTL	Mode
00	Normal
01	Hold
10	Load
11	Scan (right)

- **SDA_X — Scan data - ARB to XBAR.** From the arbitration controller. Right shift scan data from the arbitration controller to the crossbar.

OUTPUTS:

- **MX_CYCLE, MX_ROW_SEL, MX_ADR, MX_ZONE, MX_WR_PAR, MX_WR_DAT — Multiplexed memory address, data, and control signals.** These are the cycle type, row select, address, zone, write parity, and write data from the input staging registers of the port selected by **PX_WIN.** This information goes to all eight memory banks. Only the bank receiving a **My_START** signal from the arbitration controller (if any) will latch this information and use it to access memory.
- **Px_RD_PAR — Read parity.** Parity for the corresponding **Px_RD_DAT** bus. Parity will only be driven if the corresponding **Px_RD_EN** is active.
- **Px_RD_DAT — Read data.** Read data returned to the processor. Data will only be driven if the corresponding **Px_RD_EN** is active.
- **SDX_E — Scan data - XBAR to ECC.** Right shift scan data from the crossbar to the bank 7 ECC gate array.

5.10.5 Read Multiplexer Interface Signals

The following list describes each of the input and output signals for the read multiplexer.

> **NOTE**
>
> Many of these signals are duplicated for each memory bank. To simplify matters, these signals are listed in the form "My_(signal_name)", where "**y**" refers to one of the eight memory banks 0 through 7. A signal followed by an "*****" is active low.

INPUTS:

- **My_RD_PAR — Read parity.** From the memory banks. Parity for data on the corresponding **My_RAM_DAT** bus.
- **My_RAM_DAT — RAM data.** From the memory banks. Read data from the RAMs or the ECC gate array in the memory bank. This bus is bi-directional inside the memory bank.
- **My_DVAL* — Data valid.** From the memory banks. Indicates that the data on the corresponding **My_RAM_DAT** bus is valid, i.e., either there were no ECC errors or else a correction cycle has been performed.
- **My_RD_EN* — Read bus enable.** From the memory banks. Indicates that the data on the corresponding **My_RAM_DAT** bus is stable. The read multiplexer will not pass any data that is not stable to prevent bad parity from reaching the crossbar. Instead, it will default to data of all zeroes and parity of all ones.
- **Px_BSEL — Bank select.** From the arbitration controller. Selects one of the eight memory banks to return read data. It also selects the corresponding **My_DVAL*** and **My_RD_EN*.**

OUTPUTS:

- **MX_RD_PAR — Multiplexed read parity.** Read parity from selected memory bank.
- **MX_RD_DAT — Multiplexed read data.** Read data from selected memory bank.
- **MX_DVAL, MX_DVAL* — Multiplexed data valid.** Indicates that the data and parity from the currently selected memory bank is valid.

5.10.6 Clock Generator Interface Signals

The following list describes each of the input and output signals for the clock generator.

> **NOTE**
>
> A signal followed by an "*" is active low.

INPUTS:

- **PI-M.100MHZ — 100 MHz.** From the backplane. This is the master clock from which all others are derived.
- **PI-M.PHASE — Phase.** From the backplane. Defines the four phases of the basic 25 MHz system clock.
- **SP-M.RUN* — Run.** From the backplane. Controls the generation of several of the memory board clocks. In particular, it controls **CLK, CLK_E,** and **CLK_WE*.**
- **SP-M.LOG_RUN* — Log run.** From the backplane. Controls the generation of clocks for the log scan ring.
- **HALT — Halt.** From the miscellaneous logic block. Indicates that a fatal system error has occurred. This signal is a buffered version of **BP-BP.HALT** from the backpane. It is used to halt the clocks **CLK, CLK_E,** and **CLK_WE*** in order to preserve the state of the memory subsystem in the event of a fatal system error.

OUTPUTS:

- **CLK — Clock.** This is the normal 25 MHz system clock. Nine copies are generated.
- **CLK_E — Early clock.** Shifted version of CLK used in the crossbar read logic. The rising edge of **CLK_E** occurs 6 ns prior to the rising edge of **CLK** assuming a 40 ns clock cycle. Three coies are generated.
- **CLK_M — Memory clock.** It never stops. Used to clock operations that cannot be stopped without destroying the contents of memory. Four copies are generated.
- **CLK_MD — Memory clock (delayed).** This is the same as **CLK_M** except that it has been delayed by 2 ns. Four copies are generated.
- **CLK_MX — Memory clock (XBAR version).** Same as **CLK_M.** Four copies are generated.
- **LOG_CLK — Log clock.** Clock used exclusively for the error logger scan ring. Four copies are generated.
- **CLK_WE* — Write enable clock.** Clock use to generate write enable strobes for the win queue. One copy is generated.

5.10.7 Scan Control Interface Signals

The following list describes each of the input and output signals for the scan control.

> NOTE
>
> A signal followed by an "*" is active low.

INPUTS:

- **CLK_MX — Memory clock (XBAR version).** From the clock generator. Free running clock used to register incoming scan control signals. The duty cycle was unimportant.
- **SP_MM.RESET — Reset.** From the backplane. Used to initialize the memory subsystem.
- **SP_M.RUN* — Run.** From the backplane. Enables clocks on the MCM.
- **SP_M.LOG_RUN* — Log run.** Enables error logger clocks on the MCM.
- **HALT — Halt.** From the miscellaneous logic block. Indicates that a fatal system error has occurred and that clocks have been stopped.
- **SP_MM.DMODE — Diagnostic mode.** From the backplane. Used to put the MCM in diagnostic mode. This signal must be asserted for all scan operations.
- **DMODEB — Diagnostic mode (buffered).** This is a buffered copy of **DMODE** from the scan control logic block. This is a registered and buffered version of **SP_MM.DMODE.**

- **SP_MM.SCTL — Scan control.** From the backplane. These signals control the operating mode of the scan ring. These signals must both be zero if **DMODE** is not asserted. The modes are:

DMODE	SCTL	Mode
0	00	Normal
1	00	Load
1	01	Scan (left)
1	10	Scan (right)
1	11	Hold

- **SCAN_MSB — Scan MSB.** From the win queue. Most significant bit of the scan ring. This bit is output on a left shift scan read.
- **SCAN_LSB — Scan LSB.** From the miscellaneous logic block. Least significant bit of the scan ring. This bit is output on a right shift scan read.
- **SP_MM.LOG_DMODE — Log diagnostic mode.** From the backplane. Used to put the MCM error logger in diagnostic mode. This signal must be asserted for all log scan operations.
- **SP_MM.LOG_SCTL — Log scan control.** From the backplane. Controls the operating mode of the error logger scan ring. The **LOG_SCTL** must be zero if **LOG_DMODE** is not asserted. The log scan modes are the same as the scan modes described above.
- **LOG_MSB — Log MSB.** From the miscellaneous logic block. Most significant bit of the error logger scan ring. This bit is output on a left shift log scan read.
- **LOG_LSB — Log LSB.** From the miscellaneous logic block. Least significant bit of the error logger scan ring. This bit is output on a right shift log scan read.
- **SP_M.ODENA — Output data enable.** From the backplane. Enables scan data onto the backplane for scan read operations.

OUTPUTS:

- **RESET, RESET* — Reset.** Registered copy of **SP_MM.RESET** from the backplane.
- **BD_RUN — Board run.** Indicates that the MCM is in the normal run mode, i.e., **SP_M.RUN*** is active and **BP-BP.HALT** and **SP_MM.DMODE** are inactive. Three copies are generated.
- **DMODE, DMODE* — Diagnostic mode.** Registered copy of **SP_MM.DMODE** from the backplane.
- **SCTL, SCTL* — Scan control.** Registered copy of **SP_MM.SCTL** from the backplane.

- **GA_SCTL — Gate array scan control.** Controls the operating mode of the gate arrays on the MCM. This is decoded from **DMODE** and **SCTL** as follows:

DMODE	SCTL	GA_SCTL	Mode
0	00	00	Normal
1	00	10	Load
1	01	01	Hold
1	10	11	Scan (right)
1	11	01	Hold

- **LOG_DMODE — Log diagnostic mode.** Registered copy of **SP_MM.LOG_DMODE** from the backplane.
- **LOG_SCTL — Log scan control.** Registered copy of **SP_MM.LOG_SCTL** from the backplane.
- **GA_LOG_SCTL — Gate array log scan control.** Controls the operating mode of the error logger logic in the ECC gate arrays on the MCM. This is decoded from **LOG_DMODE** and **LOG_SCTL** in the same manner as **GA_SCTL.**
- **SP_M.SCNDAT — Scan data.** Always an input and sometimes an output for data in the MCM scan rings. On a scan write operation, the SPU sources this signal. On a scan read operation (**SP_M.ODENA** asserted), the MCM sources this signal. In either case, the data on **SP_M.SCNDAT** is buffered (refer to **SCAN_IN** below) and fed to the inputs of the MCM scan rings.
- **SCAN_IN — Scan in.** Buffered copy of **SP_M.SCNDAT.** Used to source new data or to recirculate old data in the MCM scan rings.

5.10.8 Win Queue Interface Signals

The following list describes each of the input and output signals for the win queue.

> **NOTE**
>
> Some of these signals are duplicated for each memory bank. To simplify matters, these signals are listed in the form "My_(signal_name)", where **"y"** refers to one of the eight memory banks 0 through 7. A signal followed by an **"*"** is active low.

INPUTS:

- **My_START — Start.** From the arbitration controller. Indicates which memory bank (if any) is being told by the arbitration controller to begin a memory access.

- **PX_WINB — Win.** Buffered copy of WIN from the arbitration controller. Three bit code indicating which port (if any) has just won access to memory. Combinations "101" <5> and "110" <6> are not used. The codes and the winners they define are:

Code	Winner
000	CPU A
001	CPU B
010	CPU C
011	CPU D
100	I/O (Port E)
101	Not Used
110	Not Used
111	- none -

- **CLK — Clock.** From the clock generator. This is the normal 25 MHz system clock.
- **CLK_WE* — Write enable clock.** From the clock generator. Used to generate write enable strobes for the win queue.
- **RESETB — Reset (buffered).** Buffered copy of **RESET** from the scan control logic block. Used to initialize the win queue RAM and address counter to zero.
- **DMODEB, DMODE* — Diagnostic mode (buffered).** Buffered copy of **DMODE** from the scan control logic block. Used to put the win queue in diagnostic mode. This signal must be asserted for all scan operations.
- **SCTL, SCTLB, SCTL* — Scan control.** From the scan control logic block. Controls the operating mode of the scan ring. **SCTL** must be zero if **DMODE** is not asserted. **SCTLB** is a buffered copy of **SCTL.** The scan modes are:

DMODE	SCTL	Mode
0	00	Normal
1	00	Load
1	01	Scan (left)
1	10	Scan (right)
1	11	Hold

- **SDM_WQ — Scan data - miscellaneous to win queue.** From the miscellaneous logic block. Left shift scan data from the miscellaneous logic block to the win queue.
- **SDE_WQ — Scan data - ECC to win queue.** From memory bank 0. Right shift scan data from the ECC gate array in memory bank 0 to the win queue.

OUTPUTS:

- **SCAN_MSB — Scan MSB.** Most significant bit of the left shift scan path.
- **SDWQ_M — Scan data - win queue to miscellaneous.** Right shift scan data from the win queue to the miscellaneous logic block.

5.10.9 Miscellaneous Logic Interface Signals

The following list describes each of the input and output signals for the miscellaneous logic block.

> **NOTE**
>
> Many of these signals are duplicated for each memory bank. To simplify matters, these signals are listed in the form "My_(signal_name)", where **"y"** refers to one of the eight memory banks 0 through 7. A signal followed by an **"*"** is active low.

INPUTS:

- **My_PAR_ERR — Parity error.** From the memory banks. Indicates that memory bank **"y"** has detected bad parity on the write bus.
- **My_ECC_ERR — ECC error.** From the memory banks. Indicates that memory bank **"y"** has detected bad ECC on a read from memory.
- **My_MULT_BIT — Multi-bit ECC error.** From the memory banks. Indicates that memory bank "y" has detected a fatal ECC error (more than one bad bit) on a read from memory.
- **CLK — Clock.** From the clock generator. This is the normal 25 MHz system clock.
- **RESET, RESET* — Reset.** From the scan control logic block. Used to generate **My_RESET** to the memory banks and to block the assertion of **BP-BP.HALT, MM-SP.HARDERR,** and **MM-SP.SOFTERR.**
- **BP-BP.HALT — Halt.** Wire-ORed on the backplane. Halts the system clocks to preserve state in the event of a fatal system error. This signal is both an input and an output of the MCM.
- **DMODE, DMODE* — Diagnostic mode.** From the scan control logic block. This signal must be asserted for all scan operations.

- **SCTL, SCTL* — Scan control.** From the scan control logic block. Controls the operating mode of the scan ring. **SCTL** must be zero if **DMODE** is not asserted. The scan modes are:

DMODE	SCTL	Mode
0	00	Normal
1	00	Load
1	01	Scan (left)
1	10	Scan (right)
1	11	Hold

- **SCAN_IN — Scan in.** From the scan control logic block. This is a buffered copy of the backplane signal **SP_M.SCNDAT.** At this point, it is the input to the LSB of the left shift scan ring.
- **SDWQ_M — Scan data - win queue to miscellaneous.** From the win queue. Right shift scan data from the win queue to the miscellaneous logic block.
- **LOG_CLK — Log clock.** From the clock generator. A 25 MHz clock for the error logger scan ring.
- **LOG_DMODE. — Log diagnostic mode.** From the scan control logic block. This signal must be asserted for all log scan operations.
- **LOG_SCTL — Log scan control.** From the scan control logic block. Controls the operating mode of the error logger scan ring. The **LOG_SCTL** signal must be zero if **LOG_DMODE** is not asserted. The log scan modes are the same as the scan modes described above.
- **SCAN_INB — Scan in (buffered).** Buffered copy of **SCAN_IN** from the scan control logic block. At this point, it is the input to the LSB of the left shift error logger scan ring.
- **LDE_M — Log data - ECC to miscellaneous.** From memory bank 0. Right shift log scan data from the ECC gate array in memory bank 0 to the miscellaneous logic block.

OUTPUTS:

- **My_RESET — Bank reset.** Used to reset the memory banks. Asserted when **RESET** is active if enabled via the scan ring.
- **MM_SP.HARDERR — Hard error.** Indicates that a fatal error has been detected by the MCM.
- **MM_SP.SOFTERR — Soft error.** Indicates that a nonfatal error (i.e., single bit ECC error) has been detected by the MCM.
- **HALT — Halt.** Buffered copy of **BP_BP.HALT.** Indicates that a fatal system error has occurred and that the MCM state should be preserved.

- **SDM_WQ — Scan data - miscellaneous to win queue.** Left shift scan data from the miscellaneous logic block to the win queue.
- **SCAN_LSB — Scan LSB.** Least significant bit of the scan ring.
- **LOG_MSB — Log MSB.** Most significant bit of the error logger scan ring.
- **LOG_LSB — Log LSB.** Least significant bit of the error logger scan ring.

5.10.10 Memory Bank Internal Signals

The following list describes the significant signals internal to a memory bank.

> **NOTE**
>
> Since each memory bank has its own copy of these signals, the signals are listed in the form "My_(signal_name)", where **"y"** refers to the memory bank (0-7) to which the signal belongs.

- **My_RAS — Row address strobe.** From the ECC gate array. Four bit signal used to strobe the row address or the refresh address into the DRAMs on the MAM. Only two of the four bits are used.
- **My_INIT — Init. From the ECC gate array.** Used to generate **My_MUX** to control row/column address multiplexing and to generate **My_CAS** once the column address is stable.
- **My_MUX — Mux.** Delayed 15 ns from **My_INIT.** Used to perform row/column address multiplexing for the DRAMs on the MAM. See **My_RAM_ADR** below.
- **My_CASA, My_CASB — Column address strobe (copies "A" and "B").** Delayed 30 ns from **My_INIT.** Used to strobe the column address into the DRAMs on the MAM.
- **My_WRITE — Write.** From the ECC gate array. Used to generate **My_WE** to write data into memory.
- **My_WEA, My_WEB — Write enable (copies "A" and "B").** Delayed 20 ns from **My_WRITE.** Used to strobe write data into the DRAMs on the MAM.
- **My_RAM_OE — RAM output enable.** From the ECC gate array. Used to enable read data and read ECC from memory onto the **My_RAM_DAT** bus.
- **My_WR_ECC — Write ECC.** From the ECC gate array. ECC check bits generated for the data to be written into memory.
- **My_RD_ECC — Read ECC.** From the MAM. Contains the ECC check bits associated with the data bits read from memory. Used to determine if there is an error in the read data.

- **My_RAM_ADR — Ram address.** From the ECC gate array. Contains the address to be used to access memory. The upper 11 bits of **My_RAM_ADR** always contain the 11 odd numbered bits of the 22 bit address. The lower 11 bits are controlled by **My_MUX, My_REFRESH,** and the RAS precharge count programmed in the ECC gate array as follows:

Tpre	REFRESH	MUX	RAM_ADR
≠0	0	0	MX_ADR "odd" bits
≠0	0	1	MX_ADR "even" bits
≠0	1	x	Refresh address
0	x	x	MX_ADR "even" bits

5.10.11 Memory Bank Interface Signals

The following list describes each of the input and output signals for a single memory bank.

> **NOTE**
>
> Some of these signals are common to all eight memory banks. Others are duplicated with each memory bank having its own version of the signal. To simplify matters, the duplicated signals are listed in the form "My_(signal_name)", where **"y"** refers to the memory bank (0-7) to which the signal belongs. A signal followed by an **"*"** is active low.

INPUTS:

- **My_START — Start.** From the arbitration controller. Tells the memory bank to initiate a memory access. The memory bank ignores this signal if **BD_RUN** is not active.
- **My_REFRESH — Refresh.** From the arbitration controller. Tells the memory bank to initiate a refresh cycle.
- **My_DREQ — Data request.** From the arbitration controller. Tells the memory bank to return read data to the crossbar. This signal will remain active until the memory bank responds with **My_DVAL*.**
- **My_RESET — Bank reset.** From the miscellaneous logic block. Asserted when **RESET** is active if enabled via the scan ring.

- **MX_CYCLE — Cycle type.** From the crossbar. Goes to all eight memory banks. Defines the type of memory operation to perform. Only the bank receiving **My_START** (if any) will use this to begin a memory access. The cycle types are defined as follows:

CYCLE	Operation
00	No-Op
01	Read
10	Write
11	Test-And-Modify

- **MX_ROW_SEL — Row select.** From the crossbar. Goes to all eight memory banks. Defines which row of memory devices to access. Only the bank receiving **My_START** (if any) will use this to begin a memory access.
- **MX_ADR — Address.** From the crossbar. Goes to all eight memory banks. Defines which address inside the selected row of memory devices to access. Only the bank receiving **My_START** (if any) will use this to begin a memory access.
- **MX_ZONE — Zone.** From the crossbar. Goes to all eight memory banks. Defines which byte(s) within the 32 bit word are to be updated in a write or test-and-modify operation. Only the bank receiving **My_START** (if any) will use this to begin a memory access.
- **MX_WR_PAR — Write parity.** From the crossbar. Goes to all eight memory banks. Parity for data on the **MX_WR_DAT** bus. Only the bank receiving **My_START** (if any) will test for bad parity. Parity is checked for all cycle types except no-op.
- **MX_WR_DAT — Write data.** From the crossbar. Goes to all eight memory banks. Contains the write data to be used in a write or test-and-modify operation. Only the bank receiving **My_START** (if any) will use this to begin a memory access.
- **CLK — Clock.** From the clock generator. This is the normal 25 MHz system clock. Only the scan ring in the ECC gate arrays is controlled by this clock.
- **CLK_M — Memory clock.** From the clock generator. Same as **CLK** except that it never stops. It controls most of the memory bank operations. This allows the memory bank to finish memory accesses and to respond to refresh requests even when the normal clock has been halted.
- **CLK_MD — Delayed memory clock.** From the clock generator. This is a delayed (2 ns) version of **CLK_M.** It is used by the ECC gate array to avoid exceeding its limit on simultaneously switching outputs.
- **BD_RUN — Board run.** From the scan control logic block. Indicates that the memory board is in the normal run mode, i.e., **SP-M.RUN*** is active and **BP-BP.HALT** and **SP-MM.DMODE** are inactive.

- **GA_SCTLA, GA_SCTLB — Gate array scan control (copies "A" and "B").** Buffered copies of **GA_SCTL** from the scan control logic block. Controls the operating mode of the ECC gate arrays. The modes are:

SCTL	Mode
00	Normal
01	Hold
10	Load
11	Scan (right)

- **SDX_E — Scan data - XBAR to ECC.** From XBAR. Right shift scan data from gate array #7 (inside the crossbar), to the ECC gate array inside memory bank 7.
- **LOG_CLK — Log clock.** From the clock generator. 25 MHz clock for the error logger scan ring.
- **GA_LOG_SCTLA, GA_LOG_SCTLB — Gate array log scan control (copies "A" and "B").** Buffered copies of **GA_LOG_SCTL** from the scan control logic block. Controls the operating mode of the ECC gate arrays. The modes are the same as for **GA_SCTL** above.
- **SCAN_IN — Scan in.** From the scan control logic block. Buffered copy of backplane signal **SP-M.SCNDAT.** At this point, it is the MSB of the right shift log scan chain.

OUTPUTS:

- **My_ACT — Active.** Indicates that the memory bank is processing a memory request. This is the result of either a valid start command or a refresh command.
- **My_DVAL* — Data valid.** Indicates that the data on the **My_RAM_DAT** bus is valid, i.e., there were no ECC errors or else a correction cycle has been performed.
- **My_RD_EN* — Read bus enable.** Indicates that the data on the **My_RAM_DAT** bus is stable. This is to prevent the read multiplexer from passing unstable data (and thus likely bad parity) to the crossbar.
- **My_PAR_ERR — Parity error.** Indicates that the memory bank has detected bad parity on the write bus from the crossbar.
- **My_ECC_ERR — ECC error.** Indicates that the memory bank has detected bad ECC on the read bus from memory.
- **My_MULT_BIT — Multi-bit ECC error.** Indicates that the memory bank has detected a fatal ecc error (more than one bad bit) on the read bus from memory.
- **My_RD_PAR — Read parity.** Parity generated for data returning to the read multiplexer.
- **My_RAM_DAT — RAM data.** Read data from either the RAMs or the ECC gate array in the memory bank. This bus is bi-directional inside the memory bank.
- **SDE_WQ — Scan data - ECC to win queue.** Right shift scan data from the ECC gate array in memory bank 0 to the win queue.
- **LDE_M — Log data - ECC to miscellaneous.** Right shift log scan data from the ECC gate array in memory bank 0 to the miscellaneous logic block.

Chapter 6
Input/Output Subsystem

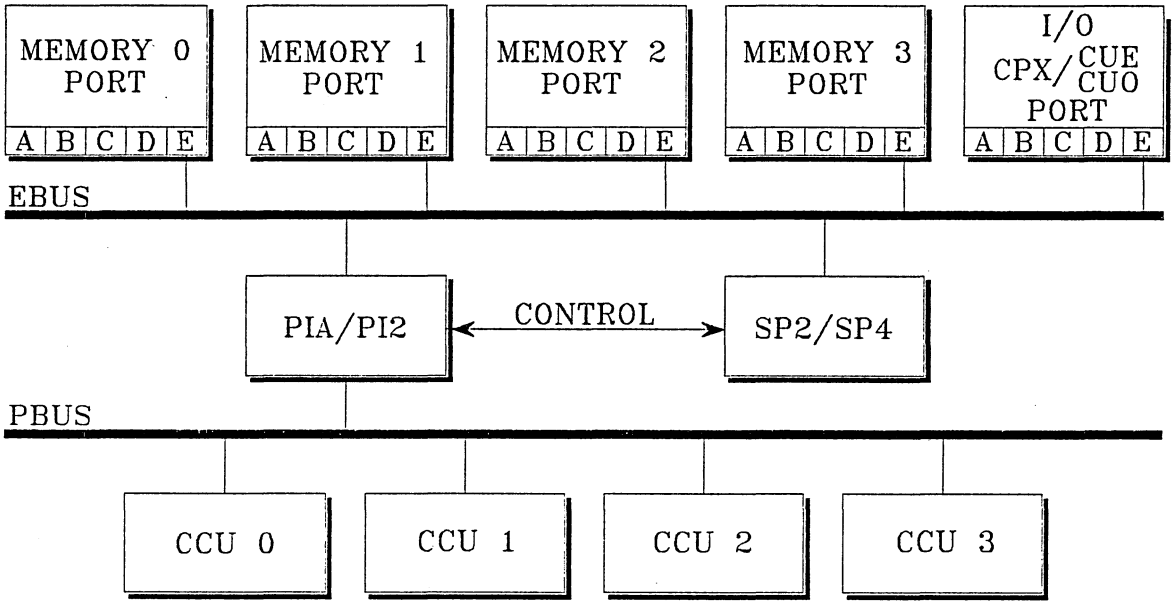
6.1 Overview

The Input/Output Subsystem for the CONVEX C200 Series systems uses the same I/O processors as the CONVEX C100 Series systems. These processors interface to the 80 MByte/sec block transfer CONVEX PBUS. Each I/O processor is a potential master of the PBUS. The bus slave is the interface to the host system's physical memory. The CONVEX C200 Series systems use a single PBUS for all I/O processors. The PBUS is located in the CPU and memory cabinet.

The C200 Series Memory Subsystem dedicates one memory port (port E) for I/O. Both the PBUS interfaces and the C200 Series Service Processor (SP2/SP4) use this port. The **only** exception is for C232i, where an additional I/O subsystem is connected to port D. Unlike the C100 Series systems, the SP2/SP4 in the C200 Series systems does not reside on the PBUS. The SP2/SP4 interfaces directly to the Memory Subsystem through port E.

Figure 6–1 shows the I/O subsystem functional block diagram:

Figure 6–1, I/O Subsystem Functional Block Diagram



NOTE: C232i Second I/O Subsystem connects to Memory Port D.

H004119
9/20/90

The current CONVEX system architecture requires three types of processors:

- One or more CPUs which execute user jobs and operating system code.
- One or more CCUs (sometimes referred to as I/O Processors or IOPs) which manage data transfers between main memory and peripherals.
- A service processor (SP, SPU, SP2, SP4) which runs diagnostics and boots the CPUs and CCUs.

The three processor types communicate exclusively through the use of interrupts and data structures in main memory.

6.2 PIA—PBUS Interface Adapter

The PIA provides the memory control and data path to system memory for both Channel Control Units (CCU) and the Service Processor (SP2/SP4). For CCU requests, header transfers and write data must be accepted and read return transfers must be driven on the PBUS. For SP2/SP4 requests, only memory control functions are performed on the PIA. Also, the PIA detects and reports errors found while processing these requests.

The PIA contains the system clock generator. The oscillators, clock switching (for margin testing) and drivers provide master clock and phase signals for all boards in the system.

The PIA is also responsible for system interrupt arbitration. Interrupts are polled round robin until an interrupt request is detected. Upon completion of the interrupt sequence, the next device is polled.

Also, the PIA provides TTL to/from ECL translation for signals between the PBUS and EBUS.

The PIA and the SP2/SP4 use the same address and data lines for the E port of the memory bus (the EBUS). The PIA provides arbitration and return data ordering for these two potential EBUS masters. EBUS arbitration is control over when a given master is allowed to drive the memory system address, control, and data signals. Return data ordering ensures that memory read data are returned to the requesting EBUS master in the same order that the requests were issued.

When multiple PBUS device requests are present, each device is serviced in a round robin fashion. Once a transfer is initiated, the PIA will allow the transfer to go to completion before moving on to the next PBUS device. Between each PBUS request, the service processor is issued at least a one cycle window to master the EBUS. If the service processor has a request during its window, it will be allowed to complete a single transfer before the next PBUS device can be granted master of the bus.

6.2.1 PBUS Interface Capacity

From one to four Channel Control Units (CCU) can be connected to the PIA on a single PBUS interface. The PIA is designed to emulate a memory interface to the CCUs upward compatible with that of the CONVEX C100 Series systems. Each CCU is capable of being the bus master on any given bus cycle. All transfers on the PBUS are either to or from the PIA as CCU to CCU transfers are not supported.

6.2.2 PBUS Interface

The PBUS interface is a synchronous, 10 MHz multi-master, block transfer bus protocol used as an I/O interface to main memory.

The PBUS interface must arbitrate the bus, check for error conditions, interpret/forward memory requests and drive memory return data.

6.2.2.1 PBUS Arbitration

PBUS arbitration determines which CCU is given access to the PBUS for the duration of a transfer. The arbitration is a fixed algorithm. All arbitration is done in state 0 and applies until state 0 is reentered. The algorithm is designed to sequence the bus to CCU0, CCU1, CCU2, CCU3 and back to CCU0. When state 0 is entered at the end of a transfer, the next requesting device in the sequence (starting one after the device just finished) will immediately be granted the bus. If no requests are pending, the currently finishing device is remembered until a request or requests become active.

6.2.2.2 PBUS Arbiter

The PBUS arbiter is that portion of the PIA responsible for the following functions:

- PBUS arbitration
- Receive and interpret CCU sourced PBUS transfers
- Sequence return data from memory to the PBUS
- Monitor/detect PBUS errors

6.2.3 EBUS Description

The PIA connects to the C200 Series Memory Subsystem through Port E (EBUS) of the Memory Subsystem. This interface provides full memory capabilities for all I/O processors as well as the Service Processor (SP2/SP4). The data path portion of the memory interface is shared between the SP2/SP4 and the PIA. All the memory control signals are sourced by the PIA alone.

The C200 Series Memory Subsystem is organized as a 5-port memory array. Each port is capable of transferring 64 bits (longword) during each 40ns bus cycle. Each longword transfer references two banks of memory, an even bank and an odd bank. In addition to the memory space, there exists an I/O space for miscellaneous timer registers and CPU referenced and modified bits. I/O space consists of even banks only but otherwise behaves identically with memory space. I/O space resides on the CPU utilities board (CPX).

The C200 Series operating mode of operation has from one to four memory board pairs, where each pair consists of an odd and an even memory board. Each board consists of eight banks and, depending on the number of board pair present, may be from 8 to 32 way interleaved.

6.2.3.1 Memory Data Path

The memory data path consists of those lines used by a requester to initiate a specific memory function and those signals used by the memory as a result of a memory request.

6.2.3.2 Memory Control Path

The memory control path consists of those signals used by the arbiter to request and acknowledge an information transfer. In the case of a memory request, these signals indicate that the memory will accept a new transfer. For read return data, these signals indicate that the data is available and/or received.

6.2.4 EBUS Interface

The PIA interface to the EBUS contains two logical blocks. The first block is the EBUS arbiter. The arbiter is responsible for selecting the current EBUS master, generating the required memory request handshakes and driving PBUS sourced EBUS data paths. The arbiter must also drive default data on the write data and parity buses to assure proper parity during cycles when the bus is idle.

The second block is the Return Queue. The Return Queue is responsible for selecting the memory board to drive the read data and parity buses, generating the required memory read request handshakes and forwarding PBUS requested read data to the PBUS interface. The Return Queue also checks for proper parity from memory on the read data bus and sources the memory base pointer value upon a PBUS request.

6.2.4.1 EBUS Arbitration

EBUS arbitration determines whether the SP2/SP4 or the PBUS will be given access to the EBUS for the duration of a transfer. The arbitration scheme is a fixed algorithm. All arbitration is done in state 0 and applies until state 0 is reentered. When state 0 is entered at the end of a transfer, the SP2/SP4 is given a one cycle window where, if a SP2/SP4 request is pending, the SP2/SP4 wins the bus. If the SP2/SP4 does not have a request, the PBUS can be granted the bus. After the first cycle, any simultaneous SP2/SP4 and PBUS request will result in the SP2/SP4 winning the bus. If only one of the two are requesting, then the requesting device will win the bus.

This algorithm is based on the fact that SP2/SP4 transfers are infrequent while the possibility of a long PBUS transfer or multiple PBUS requests could lock out the SP2/SP4 for a significant period of time. The algorithm prevents the PBUS from hogging the EBUS interface for longer than one full PBUS request.

6.2.4.2 Physical Address Mapping

EBUS memory requests supply a 32-bit physical addressss. Several system factors control the most efficient way to map the address to the real memory in the system. All memory requesters (all ports) must use the same algorithm to assure consistency in memory addressing.

The algorithm used is dependent on the following system characteristics:

- Machine type
- Revision level of the operating system
- Number of memory cards installed
- Memory ram configuration

The EBUS arbiter is responsible for the entire address mapping function for PBUS sourced transfers. The SP2/SP4 provides its own address mapping function.

The initial operating system port for the C200 Series processor will be UNIX 6.2. This version uses an address space of 30 bits. Therefore, two bits of the physical address space will always be zero. Since operating system revision 7.0 and beyond will use the entire address space, mapping involving bits 30 and 31 need to be replaced by lower order bits not used in memory addressing for 6.2.

The assumption adopted requires machines using UNIX 6.2 to have memory boards limited to 1 Mbit memory RAMs. The 6.2 addressing mode affects memory board select and MADDR mapping as described in following paragraphs.

In order to utilize memory in the most efficient way, it is desirable to maximize the time between accessing the same memory bank on large transfers involving consecutive addresses. This reduces the possibility of finding a bank busy when its next transfer is requested.

With more than one memory pair in the system, all banks in each pair should be accessed before a bank is repeated. Since there are eight banks per board pair and one longword per access, addresses XXXXXX00 through XXXXXX3f will require all eight banks of a board pair. The next address should therefore be to another board (if installed). To this end, memory address bits 6 and/or 7 are used in place of bits 29 and/or 30 (or 27 for 6.2 addressing mode) for board selects. Depending on the number of memory board pairs installed, memory will be from 8 to 32 way longword interleaved.

The EBUS only transmits 29 of the 32 bits of physical addressing. For reads, the three least significant bits are not used since memory will return the entire word or longword (depending on the board(s) requested) regardless of the byte address. It is necessary, however, to specify the bytes to be affected by a write operation. This is done by an 8-bit ZONE field. Each bit represents a byte write enable during a memory write operation. A zero in a ZONE bit position causes the corresponding byte to remain unchanged.

Memory boards can consist of either two or four rows of RAMs. Regardless of the organization, the least significant row select is always memory address bit 28. If there are four rows of memory, memory address bit 31 (or bit 26 for the 6.2 address mode) is used for the most significant row select.

6.2.4.3 EBUS Arbiter

The EBUS arbiter is that portion of the PIA responsible for the following functions:

- EBUS arbitration.
- Memory request handshake for both SP2/SP4 and PBUS sourced transfers.
- Memory request address and write data path for PBUS sourced transfers.
- Default EBUS write bus driver.

6.2.4.4 EARB/PBUS Arbiter Interface

The PBUS interface sends header and data transfers from the PBUS to the EBUS arbiter. The PBUS interface is responsible for checking the validity of a header as well as parity of the header or data transfer. The EBUS and PBUS communicate using several specialized handshake signals.

6.2.4.5 EARB/Return Queue Interface

The Return Queue is responsible for memory return data and handshake signals and the forwarding of return data to the PBUS interface. The EBUS arbiter and the Return Queue communicate through the Write/Control Queue and a few control signals.

The eight bits of the Write/Control Queue used by the Return Queue contain per transfer information about memory board select, non-present memory access and last transfer.

6.2.4.6 EARB and SP2/SP4 Interface

The SP2/SP4 shares the data portion of memory port E with the PIA. When the SP2/SP4 needs a memory transfer, it must compete with active PBUS transfers through the EBUS arbiter.

6.2.4.7 EARB Internals

The EBUS arbiter is composed of eight functional blocks. The following paragraphs contain a brief descriptions of each block. The functional blocks of the EARB are:

- Write/Control Queue
- Arbiter Input Byte Counter
- Arbiter Input Address Counter
- EBUS Address Counter
- EBUS Write/Parity Logic
- EBUS Address and Control Logic
- PCM Logic
- Internal Control

The **Write/Control Queue** is a 16 register by 72 bit memory file. During read sequences, only the upper eight bits are used. These bits contain the information about each memory request that the Return Queue will need to sequence the return data from memory. During PBUS write sequences, the queue contains the data and parity to be written to memory. The queue logic contains three counters. The four-bit head pointer points to the next location to be used for storing data. The four-bit tail pointer points to the first location containing valid data. The third counter tracks the fill level of the queue. When the queue is empty, the head and tail pointers will be equal and the fill level will be zero. The queue is never allowed to exceed fifteen elements.

The **Arbiter Input Byte Counter** is a 13-bit counter used to track the number of longwords remaining in a PBUS transfer. Since PBUS transfers are dispatched in longword increments, the counter loads only the upper 13-bits of the 16-bit header byte count. On the first read transfer dispatched to the EBUS or the first write transfer data written to the queue, the counter control will align the count with the actual number of full or partial longword requests required. The counter is loaded with the byte count from the PBUS header.

The **Arbiter Input Address Counter** monitors the physical address of transfers entering the Write/Control Queue. The counter is loaded with the longword start address from the PBUS header and is incremented for each piece of write data received or each read transfer dispatched to the EBUS. The output of this counter is used to check the current PBUS address against the memory present in the Memory Subsystem.

The **EBUS Address Counter** monitors the address of the requests on the EBUS. When a transfer is dispatched to the EBUS, the current value of the counter is sent to the output register as the memory address. For reads, this counter will always be equal to the Arbiter Input Address Counter. For writes, the two counters may vary by as much as fifteen longwords. The EBUS Address Counter is loaded with the longword start address from the PBUS header and is incremented each time a new request is dispatched to the EBUS.

The **EBUS Write/Parity logic** consists of registers and output drivers for the EBUS interface. Information dispatched to the EBUS will remain in the output registers until accepted by the Memory Subsystem. This logic also contains control to drive good parity (zero data/one's for parity) during idle bus cycles and control to disable the output drivers when the SP2/SP4 is granted the bus.

The **EBUS Address and Control Logic** consists of registers and output drivers for the EBUS interface. Address information dispatched to the EBUS as well as memory cycle type remain in the registers until accepted by the Memory Subsystem. The logic also contains control to manipulate the byte masking (zone) field for partial longword transfers at the beginning and/or end of a PBUS write transfer.

The **PCM Logic** consists of a 1024 by 2-bit ram used to hold information concerning the configuration of physical memory. The most significant bit set indicates that a block of memory is not present. The least significant bit provides odd parity for the RAM.

The **Internal Control** for the EARB monitors the state of the EBUS Arbiters interfaces and provides control to the other blocks as required. The control functions include state machine maintenance, counter control, EBUS arbitration, error log maintenance and EBUS memory status.

6.2.4.8 Return Queue

The Return Queue logic is responsible for accepting the read data that has been requested from the memory unit by the EBUS Arbiter. It passes this data on to the PBUS Interface logic. To maximize memory throughput, a 16 deep queue provides temporary storage for the returning read data and parity from PBUS initiated transactions. The width of this queue is 74 bits. Read transfers initiated by the SP2/SP4 are not put into the queue. Having no state machines for control, the logic of the Return Queue is fairly passive. Control information for the Return Queue is received from the EBUS Arbiter logic and from the PBUS Interface logic.

Read data and parity from the memory system are registered on the rising edge of early clock by the Return Queue logic in a scannable register. Parity checkers are used to check that the data on the memory read data bus has the correct odd parity. If a parity error is detected, the input register will be locked with the errant data in it. It will remain locked until the parity error bit is cleared. Visibility is provided in the scan ring as to which byte or bytes had improper parity. The outputs from the read data register proceed to the data inputs of the queue.

The queue is implemented with ECL register file chips. Queue output is on a 74-bit return queue bus. Associated with the queue are three counters. Each of these counters is four bits wide. Two of the counters are configured as circular pointers and provide the write and read addresses for the queue. The write address counter is referred to as the head pointer. The location pointed to by the head pointer will be written on every cycle. If the data written is actually return data for a PBUS transaction, the head pointer will advance to the next location. The location currently pointed to by the head pointer will be referred to as the top of the queue.

The read address counter is known as the tail pointer. If the tail pointer is pointing at a returned element, the Return Queue will tell the PBUS Interface logic that valid data is available from the output of the queue. After the PBUS Interface logic signals that it has taken the data, the tail pointer is allowed to advance to the next location. The location pointed to by the tail pointer will be called the bottom of the queue.

The third counter in the Return Queue is used to keep track of the number of elements in the queue. Unlike the head and tail pointers, which are only allowed to count up, this counter can count down as well as count up. If return data is written into the top of the queue without data being taken out from the bottom of the queue, then the counter will advance. Conversely, cycles during which data is taken out of the bottom of the queue, without return data being written into the top of the queue, will cause the counter to decrement. If return data is written into the queue at the same time that data is taken out of the queue, the counter will remain the same.

The memory boards assert their read ready signals when requested data is available to the EBUS. The read enable signals determine which memory boards are allowed to drive data on the EBUS. When there are read requests outstanding, the Return Queue receives information from the EBUS Arbiter logic to tell it which boards to enable. The proper return sequencing is maintained because the EBUS Arbiter logic puts the control information for the Return Queue into a control queue in the order that the requests were made. If the read ready signals correspond to the memory boards that are enabled, then the Return Queue will accept the data, providing that the queue is not already full. The Return Queue indicates to the memory that it has accepted the data by asserting the proper read acknowledge signals. The enables are registered on the rising edge of early clock and are held in the register until the data has returned.

The PIA is required by the EBUS definition to always have a pair of boards enabled on the read data bus. This is true whether or not there are read requests outstanding, since parity is continually checked on the bus. During times when no read requests are outstanding, the Return Queue will drive a pair of default board read enables. The default enables are decoded from two bits in the PIA scan ring, which are scanned in during system initialization.

In addition to reading data from the memory boards, the PIA is also capable of reading the time of century counter on the CPU Utility Unit (CPX). The CPX is located on the even half of the memory read data bus. Whenever the Return Queue has the read enable active for the CPX, it will also activate the read enable for the default odd memory board. The odd default memory board is enabled to ensure good parity on the odd portion of the bus.

The Memory Base Pointer (MBP) resides in the Return Queue logic. This ten bit value points to the location of the base of physical memory and it is scan loaded during system initialization. Two scannable parity bits are also associated with the MBP and need to be initialized. The outputs from the MBP can be disabled and are wire-ored with the register file outputs of the queue. A memory base pointer read signal from the PBUS Interface logic controls whether the queue outputs or the MBP outputs are enabled. The position of the 10 bits of the MBP in the return queue bus is bits <31..22>. The return queue bus bits <69..68> are where the parity bits are wire-ored. The bits of the parity byte correspond to bytes four and five of the data long word. When the MBP is enabled, the bits <63..32> and <21..0> of the return queue bus will be 0's. Ones will be forced on bits <71..70> and bits <67..64> of the parity byte to provide proper parity for the data bytes which are 0's. MBP reads are handled solely by the PBUS Interface logic without any intervention from the EBUS Arbiter logic or from the Return Queue control logic.

6.2.5 Interrupt Interface

Interrupts in the CONVEX C200 Series system are used for communications between the various processing elements of the system. Under normal operation, processors within the system will use interrupts to signal completion of a task or to request action from each other. Interrupts are transmitted via dedicated interrupt lines on both the EBUS and PBUS backplanes and are arbitrated by logic on the PIA.

There are eight devices capable of requesting interrupts. The four of these devices residing on the PBUS backplane are the four Channel Control Units (CCUs). The remaining four devices reside on the EBUS backplane. They are CPU A, CPU B, the Service Processor (SP2/SP4) and the CPU Utility Unit (CPX).

6.2.5.1 Interrupt Arbiter

The Interrupt arbiter is that portion of the PIA responsible for the following functions:

- Interrupt bus arbitration.
- TTL/ECL interrupt vector translation
- Interrupt state sequencing

6.2.5.2 Interrupt Arbitration

The Interrupt arbiter is responsible for granting the interrupt bus to a device with a pending interrupt request. Interrupt bus arbitration consists of sequentially polling each device capable of requesting the bus. If the currently selected device has a pending interrupt request, it will be granted the bus for the duration of a single interrupt sequence. After the completion of the interrupt sequence, the next sequential device will be polled.

6.2.5.3 Interrupt Level Translation

The Interrupt arbiter services devices that reside on both the PBUS (TTL levels) and the EBUS (ECL levels). Interrupt data and handshake signals must be translated from the logic level of the source to the opposite logic level. This logic resides on the PIA as part of the Interrupt Arbiter.

6.2.5.4 Interrupt State machine

The Interrupt State Machine is responsible for tracking the progression of an interrupt request.

6.2.6 PIA Data Flow

The PIA provides the memory control and data path to memory for both Channel Control Units (CCUs) and the Service Processor (SP2/SP4). To service a CCU request, header transfers and write data must be accepted and read return transfers must be driven on the PBUS. For SP2/SP4 requests, only memory control functions are performed on the PIA. The PIA is also responsible for detecting and reporting errors found while processing these requests. The following sections describe the data path associated with the various transfer types.

6.2.6.1 PBUS Header Transfers

When a PBUS device receives a grant from the PIA, the first data transfer expected is a header transfer. The header transfer contains information concerning the address, type and length of the PBUS request. The header is expected to be valid on the PBUS one cycle after a CCU receives a grant. The valid data signal will not be asserted during a header transfer.

The PBUS header comes on to the PIA via the backplane connector and propagates to the PBUS Input Deskewing Latch. The latch provides deskewing between the PIA and the CCU and inverts the incoming data and parity. At this point the PBUS is s checked for proper (even) parity. The header is then registered by the PBUS Input Register.

On the following PBUS cycle, the header data (but not the parity) is inverted. This returns the header to its high asserted PBUS value while converting the parity to its odd sense. The header is now on the data bus. If no parity error exists, the header information is decoded by the PBUS interface to determine what action the current transfer type requires. For transfers requiring the EBUS interface, the PBUS interface will signal the EBUS interface that a valid header is present on the data bus. The PBUS state machine will maintain the the value of the PBUS Input Register until the EBUS interface signals that the header has been accepted.

When the EBUS interface is idle and the PBUS signals that the data bus has valid data, the EBUS interface will accept the header transfer. During this cycle, the header information propagates into the EBUS interface where the address field is loaded into the EBUS Address Counter and the Arbiter Input Address Counter while the byte count field is loaded into the Arbiter Input Byte Counter. The EBUS interface also uses the transfer type to determine what further action the transfer requires.

6.2.6.2 PBUS Write Transfers

When a PBUS write request is decoded from a header transfer, the PBUS interface is set-up to monitor and control data transfers from the PBUS to the EBUS. A valid data transfer on the PBUS requires the valid data signal and the memory buffer available signal to be asserted. For write transfers, the valid data signal indicates that the PBUS requester is driving valid write data on the PBUS and the memory buffer available signal indicates that the PIA can accept write data on the current PBUS cycle. The second cycle after the header transfer is the first cycle which can have a write transfer. Thereafter, each PBUS cycle can contain new write data. When the last data transfer (based on the header byte count) is accepted by the PIA the PBUS interface wilremove the current PBUS grant and return to its idle state.

A PBUS write transfer comes on to the PIA via the backplane connector and propagates to the PBUS Input Deskewing Latch. The latch provides deskewing between the PIA and the CCU and inverts the incoming data and parity. At this point the PBUS is checked for proper (even) parity. The write data is then registered in the PBUS Input Register.

On the following PBUS cycle, the write data (but not the parity) is inverted. This returns the write data to its high asserted PBUS value while converting the parity to its odd sense. The write data is now on the data bus. If the data is free of parity errors and the target memory address exists, the PBUS interface signals the EBUS interface that new write data is available.

Once a write header is received by the EBUS interface, the EBUS state machine transitions to the start write state. The the upper ten bits of the address loaded in the Arbiter Input Address Counter are applied to the PCM RAM to check the next address to be written against the system configured memory. When the PBUS interface signals that new write data is available, the write data and parity propagate to the Write/Control Queue. At the next EBUS rising clock edge following the new write data signal, the Arbiter Input Address Counter, the Arbiter Input Byte Counter, the Write/Control Queue head pointer and the Write/Control Queue element counter are updated to reflect a new queue entry.

Data may continue to enter the queue at a maximum rate of two longwords for each five EBUS cycles. The queue is capable of holding up to 15 longwords destined to be written to the EBUS. Once the thirteenth element is loaded in the queue, the EBUS interface will notify the PBUS interface that the queue is temporarily full. If data transfers are being received every PBUS cycle, there will already be one transfer on the bus and one in the PBUS input register. These two data transfers will fill the queue and therefore no more transfers can be accepted. Hence, on the following PBUS cycle, the PBUS interface will deassert the memory buffer available signal thus effectively blocking further valid PBUS transfers. Since data need not be on the PBUS every cycle it, is possible for the write control queue to be one or two elements short of full when the PBUS interface stops further valid PBUS transfers. Once the queue drops below thirteen elements, the write wait signal will be de-asserted causing the memory buffer available signal to be reasserted and valid PBUS write transfers to resume. When the EBUS interface detects that the next data transfer will be the last, it signals the PBUS interface using the last data transfer signal.

Due to the highly interleaved design of the Memory Subsystem, it is desirable to perform transfers in consecutive memory bursts as opposed to transfers with many idle cycles interspersed. To implement the burst mode for the EBUS, the EBUS interface will not begin to dispatch transfers unless there are either eight transfers or the final transfer (for the current request) in the queue. Once a burst of transfers begin, the EBUS interface will continue dispatching transfers until the queue is empty. Each transfer will drive a longword on the EBUS although the the first and/or last transfer may write only a partial longword. The control for how much will be written is contained in the zone logic. The initial byte count and start address are used to calculate the starting and ending byte positions within a longword.

Data to be written to the EBUS comes from the element of the Write/Control Queue that the queue tail pointer addresses. This element is loaded into the EBUS Write Data Register coincident with the assertion of the write signal. At the same time, the current value of the EBUS Address Counter is loaded into the EBUS Address Register. The EBUS interface determines which memory board(s) need to accessed based on the mode and configuration of memory. All this information is registered. At the end of the cycle, the EBUS Address Counter and the Write/Control Queue tail pointer are incremented. On the following EBUS cycle, the appropriate memory ready signals are checked to see if memory can accept the registered transfer. Regardless of how many bytes are actually to be written, the memory for the entire longword transfer must be ready. If not, the registers are held until the required memory(s) are ready. The ready signals enable the memory requests to the EBUS and the transfer is active immediately. At the end of the EBUS cycle, new data and address information is registered for the next EBUS cycle. For non-error conditions, the write transfer completes after the last piece of PBUS data is queued and, subsequently, the Write/Control Queue empties. At this point, the EBUS state machine returns to its idle state.

6.2.6.3 PBUS Read Transfers

When a PBUS read request is decoded from a header transfer, the EBUS interface is set up to request all the memory longwords needed to satisfy the requested byte count. Once a read header is received by the EBUS interface, the EBUS state machine transitions to the start read state. On the following EBUS cycle, the upper ten bits of the EBUS Address Counter are applied to the PCM RAM to check the next address to be read against the system configured memory.

If no address error is found, the current Arbiter Input Address Counter value is loaded into the EBUS Address Register. The EBUS interface determines which memory board(s) need to be accessed based on the mode and configuration of memory. All this information is registered and both address counters are incremented. The write data and parity registers are loaded with zeros and ones respectively to insure good parity. At the same time, the write/control queue head element is loaded with control information. Bit 71 indicates that the current queue element is the last of the current request. Bit 70 indicates that the current queue element is a request to non-existent memory. Bits 69 through 67 indicate the memory boards required for this transfer. Bit 66 indicates that the queued request was requested by a PBUS device. Bits 65 and 64 will always be asserted for PBUS reads indicating a full longword was requested. At the end of the cycle, the Write/Control Queue head pointer will be incremented.

On the following EBUS cycle, the appropriate memory ready signals are checked to see if memory can accept the registered transfer. If not, the registers are held until the required memory(s) are ready. The ready sigignals enable the memory requests to the EBUS and the transfer is active immediately. At the end of the EBUS cycle the next read request may be registered.

Due to the highly interleaved design of the Memory Subsystem, it is desirable to burst read transfers to memory. To do this, the PIA will dispatch read transfers to the EBUS until the Write/Control Queue is full (count is 15). At that time, read requests will be suspended until the queue count drops below eight. This insures that memory requests will be in bursts of at least eight.

Once the read requests are sent to memory, the EBUS interface must wait for read return transfers from memory. Initially, the queued control information at the tail of the queue advances to the Return Queue Input Register. From here, the memory board select information is used to enable the appropriate memory board for driving the E port read data bus. The control information is held until the entire longword is on the EBUS as signaled by the memory read ready signals. On that cycle, the read return data is registered in the Read Data/Parity Register and the next Write/Control Queue tail element is loaded in the input register. In addition, each time the RQ Input register is loaded, the queue tail pointer advances.

Once the read data is captured, its parity is checked. Then data and parity along with the last and non-existent memory control bits are written to the head of the return queue. This process will continue as long as requests are in the RQ Input Register and the return queue is not full. If the return queue becomes full, the EBUS interface will ignore the memory ready signals until at least eight queue locations are available. When the RQ input register is empty, the default memory board selects are used to enable memory (for parity reasons) and no read ready signals are expected. Once the last transfer is forwarded to the return queue, the EBUS state machine returns to its idle state.

After receiving a read header from the PBUS, the PBUS interface waits in the transfer state for read return data. As data becomes available, the Return Queue tail element is loaded into the PBUS Output Register and the tail pointer is incremented. The PBUS interface will then attempt to send the return data back to the requesting CCU. A valid return data transfer requires both the valid data signal and the CCU buffer available signal to be asserted. The data valid signal indicates that the PIA has valid return data on the PBUS. The CCU buffer available signal indicates that the CCU can accept return data on the current PBUS cycle. When a valid return data transfer is sent, the next Return Queue tail element is loaded in to the PBUS Output Register. Output data propagates from the output register through the PBUS Output Latch where the data and parity are inverted and translated to PBUS levels. Finally, the data is inverted through the PBUS Output Buffers and the parity is buffered (non-inverted) creating high true PBUS data and even PBUS parity.

6.2.6.4 PBUS TAM Transfers

PBUS test and modify transfers behave much like PBUS read transfers. The only differences are the EBUS write data and the EBUS transfer type. Since TAM transfers are limited to a single byte only one longword request is issued on the EBUS. If the request is a Test and Set, the write data will be all ones. For a Test and Clear, the write data is all zeroes. Test and Modify instructions are handled differently than reads within the Memory Subsystem.

6.2.6.5 PBUS Memory Base Pointer Read Transfer

The Memory Base Pointer Register is addressed as I/O space even though it resides on the PIA. The CCU devices on the PBUS can access the register by requesting an I/O read as the transfer type in a PBUS header transfer. When the PBUS interface detects an I/O read with an address of (hex) 6FFFFFF8, a Memory Base Pointer Read access is initiated.

On the PBUS cycle after the header transfer is received, the MBP read transfer is decoded. On the next PBUS cycle, the PBUS interface signals the Return Queue logic to disable the output of the Return Queue and enable the output of the Memory Base Pointer Register on the return queue bus. At the end of the PBUS cycle, the data is loaded into the PBUS Output Register. On the following PBUS cycle, the MPB propagates through the PBUS Output Latch and the PBUS Output Buffer and onto the PBUS. The valid data signal is asserted and the transfer is sent regardless of the state of the CCU buffer available signal. The CCU buffer available signal is assumed to be asserted for the first (in this case only) transfer of a CCU request.

6.2.6.6 SP2/SP4 Write Transfers

Service processor write requests are initiated via discrete SP2/SP4 to PIA signals. The SP2/SP4 is capable of requesting one or both words of a single longword that makes up the entire SP2/SP4 request. At the time of the request, the PIA cannot determine whether the request is for a read or a write. The PIA will eventually know by monitoring the EBUS cycle type during the actual memory request on the EBUS. Therefore, all SP2/SP4 requests are treated as reads to the EBUS arbiter control. Note that although all PIA boards in a system must monitor the progress on an SP2/SP4 request (to track arbitration), only the PIA in slot PIY asserts the return handshakes to the SP2/SP4.

An SP2/SP4 request is initiated by the SP2/SP4 setting one or both of its memory request signals to the PIA. The board selects and TAM indication are assumed valid on the same EBUS cycle. When the PIA sees a new SP2/SP4 request, EBUS the request processor registers the request information. On the next cycle, the request processor notifies the EBUS arbiter that the SP2/SP4 has a memory request pending. On the next EBUS cycle (or at the end of the current EBUS burst), the arbiter will grant the EBUS to the SP2/SP4. The PIA will monitor the appropriate memory request signals until they indicate that the memory can accept the request. The SP2/SP4 will drive the EBUS data parity and control lines as long as it is granted the bus. Once memory accepts the transfer, the grant signal will be deasserted and request cycle acknowledge signal will be asserted. The PIA will detect that the EBUS request was a write transfer and thus will return the SP2/SP4 control to its idle values.

6.2.6.7 SP2/SP4 Read/TAM Transfers

Service processor read requests are initiated via discrete SP2/SP4 to PIA signals. The SP2/SP4 is capable of requesting only one word per SP2/SP4 request. At the time of the request, the PIA cannot determine whether the request is for a read or a write. The PIA will eventually know by monitoring the EBUS cycle type during the actual memory request on the EBUS. Therefore, all SP2/SP4 requests are treated as reads to the EBUS arbiter control.

No action is taken on the SP2/SP4 request until the PIA arbitrates to the SP2/SP4. Once the SP2/SP4 is granted the bus, the SP2/SP4 will drive all the address and parity buses required for the transfer. At the same time, the PIA calculates the required board selects based on the mode and configuration of memory. By the end of the EBUS cycle, the EBUS Output Registers are loaded and the Write/Control Queue is written with control information denoting last transfer (since only one transfer per SP2/SP4 request), no non-existent memory (this is pre-checked by the SP2/SP4), the SP2/SP4 board selects, not a PBUS request and which word of the longword the SP2/SP4 requested. On the next cycle, the appropriate memory ready signals checked to see if memory can accept the transfer. The board selects are held until the memory is ready. On the cycle the memory becomes ready, the transfer to memory takes place and the SP2/SP4 is signaled to stop driving the bus.

The information written to the Write/Control Queue advances to the RQ Input Register. This information is used to enable the memory board that will return the requested data. The RQ Input Register is held until the memory signals that the requested data is on the EBUS. On the same cycle, the memory is informed that the data is being accepted and the SP2/SP4 is informed that the data is available. At this point, the transfer is complete.

6.2.7 Error Handling

The PIA is responsible for detecting and reporting error conditions on both the PBUS and the EBUS as well as internally. Error conditions fall into two general categories. The first is a hard error condition. These errors are determined to be significantly catastrophic that no meaningful recovery can be attempted. These errors result in the blocking of further system clocks. The second class of error conditions are soft errors. These errors cause pertinent information to be saved in spepecial error logging registers and some form of hardware clean-up to re-set the PIA to a usable state. The clean-up consists of aborting the current transfer and returning the PBUS and EBUS interfaces to their idle states.

6.2.7.1 Hard Error Conditions

There are three conditions on the PIA that can cause a hard error. Each are listed below with a corresponding scan bit that identifies the given error.

- PCM parity error (pia:ea_pcm_pe)
- Interrupt arbiter error (pia:ia_harderr)
- Memory data parity error (pia:re_byte_perr)

A PCM parity error occurs whenever an accessed PCM location contains bad parity. The error can occur even when the EBUS interface is idle. This implies that all location of the PCM RAM must contain valid (even) parity.

An Interrupt Arbiter error indicates one of two possible error conditions. First, if a device is granted the interrupt bus to send an interrupt and the device removes his request before the interrupt cycle completes, a hard error indication will be asserted. The second case will detect a device acknowledging an interrupt vector when the vector is not valid.

A memory data parity error indicates that the data read from the EBUS Read Data BUS contains a parity error. The memory is required to drive correct (even) parity at all times. A common cause of bad parity is the PIA enabling a non-present memory board to drive the EBUS. Other causes include improper clock calibration or hardware failure.

6.2.7.2 Soft Error Conditions

There are six conditions on the PIA that can cause a soft error. Each are listed below with a corresponding scan bit that identifies the given error.

- EBUS reference of non-existent memory (lpia:ea_softerr)
- PBUS parity error on a write data transfer (lpia:wpar_berr)
- PBUS reference of non-existent memory (lpia:npm_berr)
- PBUS illegal transfer type on a header transfer (lpia:ill_pb_hdr_berr)
- PBUS parity error on a header transfer (lpia:hdr_par_berr)
- CCU hard error (lpia:cc_hrderr[3:0])

An EBUS reference to non-existent memory occurs when a read or write request references a 4M byte region that is mapped as non-existent in the PIA's PCM RAM. A check of the PCM is made before the request is loaded into the EBUS Output Registers (read transfer) or before the request is queued into the Write/Control Queue (write transfers). In either case, the EBUS portion of the PIA log ring is locked with the upper 10 bits of the failing address, the data and parity at the PCM address and the transfer type currently being processed. This error is only detected for memory (not I/O) requests from the PBUS. The SP2/SP4 is responsible for checking its own addresses against its own copy of the PCM.

When an EBUS reference to non-existent memory is detected during a PBUS read transfer, the request is queued as the last request (bit 71 of the queue) and as a request to non-existent memory (bit 70 of the queue). When the error element reaches the Return Queue Input Register (ie. all previous read transfers have returned from memory), bits 70 and 71 are immediately forwarded to the Return Queue. Memory read ready need not be checked since EBUS read request was not dispatched. In turn, the error transfer is presented to the PBUS interface which processes it as a PBUS reference to non-existent memory.

When an EBUS reference to non-existent memory is detected during a PBUS write transfer no transfer is queued into the Write/Control Queue. The EBUS interface signals the PBUS interface that a request to non-present memory has occurred when the data for the error address is in the PBUS Input Register. This results in a PBUS transfer to non-existent memory error in the PBUS interface. Meanwhile, the EBUS interface allows the write data received ahead of the error transfer to go out to the memory.

When the PBUS interface detects a PBUS parity error on a write data transfer, the bus error signal is asserted on the following PBUS cycle. This error will cause the offending CCU to abort the transfer. In addition, the data transfer is prevented from going to the EBUS interface by deasserting the new write data signal. The error will also cause the PBUS portion of the PIA log ring to lock containing the byte(s) containing a parity error, which CCU caused the error and the transfer type.

When the PBUS interface detects a PBUS reference to non-existent memory during a write transfer, the bus error signal is asserted on the following PBUS cycle. This will cause the PBUS requester to abort the current request. The PBUS interface will signal the EBUS interface that an error was found. The data associated with the error location is not written into the Write/Control Queue but all previously received write data is allowed to be written to memory. When the EBUS interface goes idle, the abort signal is removed and the PBUS interface returns to its idle state. Error information can be found in the PIA log ring.

When the EBUS interface detects a PBUS reference to non-existent memory during a read transfer, the EBUS set bit 70 of the Write/Control Queue to indicate the error condition and bit 71 to indicate that this is the last transfer. When the error element is loaded into the Return Queue Input Register, the error and last transfer indication are immediately forwarded to the Return Queue. At this time, the EBUS interface will enter its error state. When the error element is pointed to by the Return Queue Tail Pointer, the PBUS interface sets the error condition in its log ring and sources the bus error signal. This informs the requesting CCU that the an error has been detected. The PBUS interface will also signal the EBUS interface that an abort condition was found. The abort will cause both interfaces to return to their idle states. Error information can be found in the PIA log ring.

When the PBUS interface detects an illegal header, it will assert the the bus error signal. This informs the requesting CCU that an error was found. The PBUS interface will then return to its idle state. Error information can be found in the PIA log ring.

When the PBUS interface detects a parity error on a header transfer, it will assert the bus error signal. This informs the requesting CCU that an error was found. The PBUS interface will then return to its idle state. Error information can be found in the PIA log ring.

When a CCU detects an internal hard error condition, it drives its CCU hard error signal on the backplane. The PIA will register and hold this indication while sourcing a soft error condition to the SP2/SP4. The PIA can continue with requests from/to other CCUs.

6.2.8 Clock Generation Logic

The clock generation on the PIA is significantly more complex than any other board in the system. It is the privilege of the PIA to provide the 100 MHz master clocks and phase bits for the entire system. Another piece of complexity that the PIA clock generation has to deal with is interfacing one bus protocol running at 10 MHz to another bus protocol running at 25 MHz.

The heartbeat of the system clock generator as with most clock generators is provided by an oscillator. Actually there are three different values oscillators on the PIA. The different values are used for margining the system. The frequencies of the oscillators on the PIA are 90 MHz, 100 MHz, and 110 MHz. Outputs from the oscillators are buffered to provide the raw clocks for the system. The output of the 100 MHz oscillator is also divided in half by a flip-flop to provide a 50 MHz clock for low speed operation. Another source for the raw clock generation is from an external frequency generator. This signal is brought onto the PIA via one of the PIA's bnc connectors.

The SP2/SP4 is responsible for determining what frequency the clock generator is running at. The clock frequency signals coming to the PIA from the SP2/SP4 are decoded on the PIA into enables for the appropriate clock frequency. The clock command signal coming from the SP2/SP4 is actually the signal that initiates switching the clock generator over to a new frequency. After the clock command has been asserted, the clock generator will stop the system clocks at the point where all of the phase signals as registered on the boards are high and the master clock is low. The system clocks will be held in this state for forty cycles of the new clock frequency. This delay is provided to allow any metastable state in the clock generator caused by the switch to resolve itself. After the delay, the master clock resumes at the new selected frequency.

There are five clock synchronizers in the system clock generator, one for each of the possible sources of the master clock. It is these synchronizers which are responsible for counting the forty cycles during which metastables are resolved. The synchronizers control the gates which do the actual multiplexing of the different frequencies. The outputs of the multiplexers are further buffered by gates to become the master clocks which are driven out onto the backplane.

The master clocks are divided into four distinct groups so that clock skew can be minimized across the system. One of the groups is the group for processor B. This group contains the ASP board, the SFU board, the IPP board, the DCU board, the VPC board, and the VPD board for processor B. The second group of the master clocks contains the processor A board-set. It contains a similar compliment of boards as processor B. The third master clock group contains the memory 0 board pair, the memory 1 board pair, the SP2/SP4 board, and the CPX board. The fourth master clock group contains the memory 2 board pair, the memory 3 board pair, and the PIA. Group four is also the source of the monitor signal which is connected to the other bnc connector on the PIA. This signal makes it possible to monitor the operating frequency of the system clock generator. These signals were manually routed to give them the proper length of wire to equalize delays.

The heart of the system clock generator is a state machine that operates nominally at 100 MHz. The signal paths involved in this state machine are minimal. There are five signals which correspond to the encoded state bits of the clock generator state machine. These five signals come out of a pair of 100 MHz clocked registers and are inputs to a pair of 4 ns pals. The outputs from the pals then feed back into the inputs of the 100 MHz state registers. It is from these state registers that the phase bits for the system are derived.

The five state bits of the state machine are five of the phases produced by the clock generator. A sixth phase is produced to create the TTLGATE clock. TTLGATE is used by the SP2/SP4 and PIA to coordinate timing of control signals which cross the 25 MHz/10 MHz boundaries. If one were to assume the nominal operating frequency of 100 MHz for the master clock then the phases would have the following frequencies:

- **Phase 0** corresponds to state bit 0 and can be used to create a **50 MHz clock**.
- The frequency of **phase 1 is 25 MHz**. This is the phase used to generate the normal system clock. The EXCLUSIVE OR of phase 0 and phase 1 is used to produce the early clock for registering read data returning from memory.
- **Phase 2** is used for generating the **20 MHz clocks** required by the PBUS interface.
- **Phase 3** provides the means to generate the **10 MHz clock** for the PBUS interface.
- **Phase 4** actually corresponds the the phase for generating TTLGATE and not to state bit 4. The frequency of phase 4 varies, but is such that the TTLGATE clock will precede the rising edge of the 10 MHz clock by at least 20 ns.
- State bit 4 corresponds to **phase 5** and has a nominal operating frequency of **5 MHz**.

The phase bits for the system are driven out of eight registers. The registers serve as buffers, creating more copies of the phases that come from the state machine of the clock generator. As with the master clocks, the phase registers are also separated into the four distinct board groups. These registers are clocked by 100 MHz clocks that are derived off of the same group of buffers that drive the group's master clocks. Phase bits going to processor B are clocked with a clock from the processor B master clock group and so on. It is from the phase bits and the master clock that the boards derive their actual system clocks. In general, each board receives one master clock and two phase bits. Two exceptions to this are the PIA and the SP2/SP4. The PIA needs six phase bits to generate all of the various clocks it uses. Four phase bits are used by the SP2/SP4 for its clock generation.

Due to test requirements, the phase bits and 100 MHz clock which the PIA uses to generate its internal clocks are brought out to the backplane and then back into the board via backplane connector pins. After the phase and 100 MHz signals come back onto the board they are first taken into delay lines before being used to generate the internal clocks. The delay lines are used to reduce the clock skew between the PIA and other boards in the system.

An example of the typical method by which the PIA generates its internal clocks is the 25 MHz system clock. To generate this clock, the clock generation logic takes the phase 1 bit into a gate to buffer it. One of the outputs of this buffer is taken into a five input OR gate along with the registered outputs of the board run signal and the board halt signal. The run bit is the method by which the SP2/SP4 controls the clocks of every board in the system. If the run bit is asserted, then the clocks are allowed to run. The halt signal also has the ability to stop the internal clocks of the boards in the system. Every board drives as well as receives halt. If a hard error condition is detected, then the board asserts halt to stop the clocks in the system. The output of the OR gate is taken into the D input of a register. This register is clocked with the PIA's 100 MHz master clock. The output of this register is the raw 25 MHz clock. It is further buffered by gates to create more versions of the clock. There is also a delay line in the path between the output of the register and the input of the gate for reducing clock skew.

A set of guidelines was set up for designing the individual clock generators on each of the boards so that clock skew could be minimized. The 100 MHz master clock signal should not have more than three loads. There should be no gate delays into the register's clock pins that the master clock drives. In regard to the phase bits, all gating should be at the center of the board, close to the clock generator registers. In order to assure sufficient setup time, there should be at most 4.1 ns of delay (including trace) from the first gate input on a phase line to the D input of the register. This allows for at least 2 levels of logic prior to the D input of the register. At a minimum there should be at least one gate level into the D input of the register to assure sufficient hold time. The halt and run bits should have no more than one gate level into the D input of the clock generator registers. Maximum allowed delay (including trace) is 1.9 ns. The gate clock drivers themselves should have their separate data inputs driven from a tunable delay line. The common enable inputs on the gate drivers should be avoided, as they have more skew. Devices used in clock generation that have complementary outputs should have both outputs terminated, even if only one output is used. This is done to minimize transients in the power supply and should be done wherever high frequency signals are run through complementary output devices.

It is also the responsibility of the PIA to provide the clocks required by the PBUS. There are potentially up to four Channel Control Units (CCUs) that can be installed in a C200 Series system (more in a C232i system). Each CCU receives its own copy of the clock signals. The signals required by a CCU to generate its internal clocks are a free running 20 MHz clock, a free running 10 MHz phase and a 10 MHz phase which may be halted. The SP2/SP4 sends the PIA four phase control signals. These signals are the equivalent of the run bits by which the SP2/SP4 controls the rest of the system clocks. The circuitry generating the 10 Mhz phase signals will cause them to latch in their present state if their respective 10 Mhz phase control signals are not asserted. This is unlike other clocks in the system which stop in the high state when their run bits are not asserted. The clock generation circuitry for the CCU clock signals is constructed on the PIA with ECL components. Before the signals go out to the backplane they are translated into TTL levels.

A great deal of complexity was added to the clock generation circuitry for the PBUS in order to be able to properly handle single stepping the I/O Subsystem. The inherent difficulty in interfacing the two bus protocols was brought on by the ratio of the two clock frequencies. There are not always the same number of 25 MHz edges during a 10 MHz cycle. During one 10 MHz cycle there may be either two or three 25 MHz edges depending upon what part of the twenty state sequence, the clock generator's state machine is in. Due to this relationship, there are only specificic windows where it is legitimate to start the clocks if the system is to stay synchronized. Since the SP2/SP4 is in control of when the run bits are issued, it must be kept informed as to what state the stopped 10 MHz clocks on the PIA are in relation to the free running clock state machine.

The PIA sends a clock synchronize signal to the SP2/SP4. When this signal is asserted, the stoppable 10 MHz clocks are aligned with the free running clocks. On the PIA there are a pair of counters for keeping track of the state of the clocks. One of the counters keeps track of the state of the free running clocks. The other counter keeps track of the state of the stoppable 10 MHz clocks. The counters are clocked with a free running 25 MHz clock. The counter which tracks the free running clocks is allowed to increment once every 40 ns. Since the clock generator state machine sequence repeats once every 200 ns, the counter will increment 5 times per sequence. The count increments from 0 through 4. A clear counter signal is generated which clears the counter after it reaches 4. The clear counter signal occurs at the same point in time during every clock state machine sequence. This assures that the count values at any point in the sequence will be consistent from one sequence to the next.

The counter tracking the stoppable 10 MHz clocks is not allowed to count if the 10 MHz clocks are not running or if the PIA is being scanned. The clear counter signal will also clear this counter, if the counter is actually being allowed to increment. As with the other counter, it counts from 0 through 4. When the value of the free running clock counter is equal to the value in the 10 MHz stoppable clock counter, it means that the stoppable clocks are at the same point in the clock sequence that the free running clocks are. The outputs of both of the counters are taken into a pal which acts as a comparator to generate the clock synchronize signal.

Three other signals are sent from the PIA to the SP2/SP4 with clock state information. These signals are also derived off of the state counters. The scan OK signal informs the SP2 that the PIA clocks are at a point where it is legitimate to scan the 10 MHz ring. Another signal sent to the SP2/SP4 is the CCU scan OK signal which is asserted when the clocks are in a position that the CCUs can be scanned. The third signal that the PIA sends to the SP2/SP4 with clock state information is the two clock high signal. When this signal is asserted, it indicates that two system clocks are required to give the CCU a single clock. If this bit is not asserted, then three system clocks are required to give the CCU a single clock. The software uses this signal to calculate the number of clocks to issue when wanting to single clock or count clocks relative to CCU 10 MHz operation.

The clock generation logic on the PIA also contains the scan interface logic for the PIA and CCUs. There are three scan rings resident on the PIA. One of the scan rings is the normal 25 MHz scan ring. This ring contains scannable system clock registers as well as early clock registers. During scan operations of this ring, the logic which generates the early clocks, switches over to the same phase that generates the system clocks. This allows for one contiguous ring with the same clock phase. The 25 MHz ring is not fully bidirectional. Some of the devices in the EBUS ARBITER logic are only unidirectional. These devices are located at the top of the ring. The bottom of this ring has the signals for controlling the COP chip and LEDs. Another ring on the PIA is the 10 MHz ring. The third ring is the log ring for the PIA. This ring keeps track of soft error conditions that arise on the PIA and CCUs.

Since there is only one Scan Data (SCNDAT) line for the I/O Subsystem , the PIA is responsible for multiplexing the proper scan data from the three PIA scan rings onto the line. A pal is used to multiplex the data. The data selected is determined by which run bit is asserted, whether or not the Output Data Enable (ODENA) line is asserted, which Dmode (DMODE or LOG_DMODE) line is asserted, and the direction of the scan operation. During scan write operations, the write data is sent to the PIA from the SP2/SP4 over the SCNDAT line.

There is also only one scan data (CCUSCNDAT) line for the PBUS from the SP2/SP4. Since there are potentially four CCU's resident on the PBUS, their scan rings must be multiplexed onto this signal at the appropriate times. The PIA has the honor of doing the multiplexing and level translation of the CCU scan data. Each CCU has two scan data lines. One is for the most significant data out, and the other is for the least significant data out. The PIA handles this multiplexing with a pal. It uses similar information as does the PIA scan ring multiplexer pal for determining which data to source. The run bits in the case of the CCUs are the 10 Mhz phase control signals. The PIA also uses the 10 Mhz phase control signals to determine which ODENA signal to assert for scan read operations.

6.3 PBUS—Peripheral Bus

The hardware which implements the interrupt and data transfer mechanisms for the CCUs is collectively known as the PBUS.

The data transfer portion of the PBUS is a multi-master, 80 MBs, synchronous, block transfer bus. CCUs are masters of the PBUS and 'main memory' is the slave. All data transfers are between a CCU and main memory. Direct CCU to CCU transfers are not possible. The data bus is 64 bits wide with byte parity. There are individual request and grant signals for each CCU and data handshake lines which are bused to all devices on the bus. Transfer sizes from 1 byte to 64 Kbytes are supported.

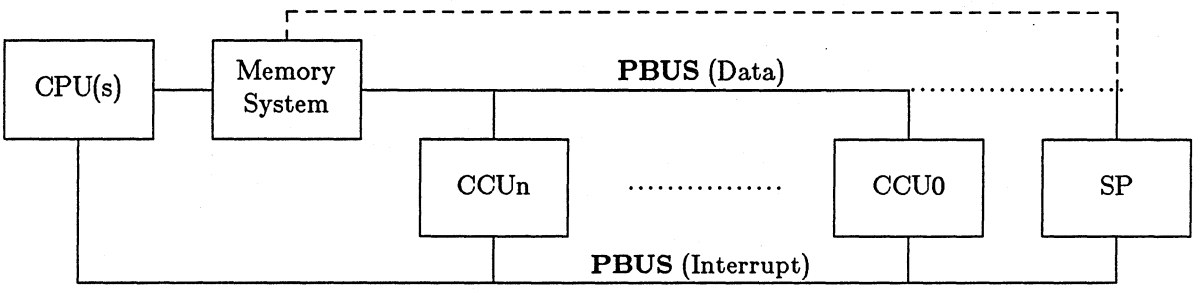
After arbitrating for the bus a CCU executes a single operation and then relinquishes control. At the beginning of the operation the CCU uses the data bus to specify the starting byte address, byte transfer count, and transaction type. Currently, eight transaction types are defined: four data transfer operations, two semaphore operations, a 'scrub' operation to correct single-bit ecc errors, and a nop. It is not necessary for all implementations of the PBUS to have all eight transaction types (none currently do).

The interrupt portion of the PBUS provides 256 system wide interrupts. Interrupts 0 through 7 are reserved for CPUs, and interrupts 8 through 15 are reserved for the SP. The remaining interrupts are available for CCUs. The particular interrupt(s) assigned to a CCU are dependent on diagnostic and operating system conventions and not forced by the hardware in any way. Any processor can interrupt any other processor, including itself. It is also possible for several interrupt receivers to be programmed for the same interrupt, providing a broadcast interrupt mechanism.

The actual hardware connection of the PBUS to the memory system varies from system to system, as does the maximum number of CCUs and CPUs. The connection of the SP2/SP4 to the memory system also varies; on older systems the SP2 is attached to the PBUS, while on newer systems the SP2/SP4 is attached to one of the memory system ports.

Figure 6–2 presents a highly simplified block diagram of a CONVEX C100/C200 Series PBUS structure showing the interconnection of the various parts:

Figure 6–2, CONVEX C100/C200 Series PBUS Structure



6.3.1 PBUS Data Transfer Operations

This section describes the operation of the data transfer part of the PBUS. The sequence of events necessary to acquire bus mastership, transfer data, and relinquish bus mastership is described.

6.3.1.1 Bus Arbitration

The PBUS is a multi-master bus with a centcentralized arbiter. Existing implementations use a round robin algorithm to sequence through requesters, but this specification does not prohibit the use of other algorithms.

6.3.1.2 Bus Acquisition

Each CCU on the PBUS has a unique pair of request/grant signals to the arbiter. The PBUS is acquired by a requesting CCU when it receives its grant from the arbiter. A CCU may assert its request at any time, provided that its grant is not asserted. The arbiter may assert grant as soon as the tick following the assertion of request. A reasonable arbiter design will typically require one tick more than this, and the maximum time to receive a grant is dependent on other traffic on the bus.

6.3.1.3 Bus Release

The header which the CCU sends to the arbiter at the beginning of a PBUS operation contains the byte count for the transfer (among other things). Both the CCU and the arbiter keep track of the actual byte count transferred as the operation progresses. In a normal termination, the CCU and the arbiter simultaneously remove request and grant at the beginning of the tick after the last requested piece of data has moved on the data bus.

However, both the CCU and the arbiter have the option to terminate the transfer earlier by deasserting request or grant, respectively. By definition, no data transfer takes place during a tick in which either request or grant is deasserted, regardless of the state of the data handshake signals.

If a CCU terminates a transfer early, it must tristate its bus drivers at the same time it deasserts request. If the arbiter terminates a transfer early, it tristates its bus drivers at the same time it deasserts grant. The arbiter then waits for the CCU to recognize the loss of grant and remove its request before issuing any new grant. In order to prevent tristate overlap with the new master, the CCU which was 'kicked off' the bus must tristate its drivers and the same time it deasserts its request. CCUs must be designed so that request is removed during the tick following the loss of grant.

6.3.1.4 Bus Lock

The Bus Lock mechanism may be used by a CCU to guarantee that a transfer will not be terminated early due to any "bandwidth sharing" or "latency" mechanisms built into the arbiter. It does *not* guarantee that any other restrictions which a particular arbiter design places on transfers are magically circumvented. The assertion of Bus Lock during a transfer affects the arbiter only until that transfer completes. The arbiter is then free to grant the use of the bus to another requester. Note that Bus Lock does not guarantee that the memory system will be capable of supporting continuous transfers at the maximum bus bandwidth and that the data stream can still be modulated by the data handshake signals. Bus Lock is accomplished by asserting the bus lock signal during the header tick and holding it asserted until the transfer is complete.

6.3.1.5 Arbitration Considerations

The PBUS is an important system resource and must be managed effectively. This requires tradeoffs between bus bandwidth utilization and latency. The trend in PBUS designs is to set the bandwidth/latency ratio in the CCU by choosing an appropriate transfer block size. With this strategy, the most effective arbiter design allows all PBUS transfers to run to completion. It is very important that the arbiter maximize bandwidth under heavy load and not cause thrashing by terminating transfers early.

6.3.2 PBUS Transactions

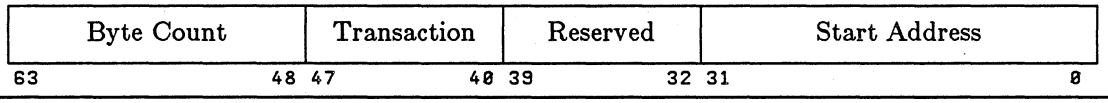
Once a CCU has obtained mastership of the PBUS, it performs a single bus operation known as a transaction. A transaction consists of a single longword of Header information optionally followed by one or more bytes of data. The data within a transaction is a sequential string of bytes associated with incrementing addresses in memory. In effect, a single transaction performs a block move of data between main memory and a CCU. Non-sequential memory references are handled with a transaction per reference.

6.3.2.1 Header Longword

The Header longword contains address, byte count and transaction type information about the current transfer. The Header longword is driven on the bus during the tick immediately following the assertion of grant by the arbiter.

The format of the Header is shown in Figure 6–3:

Figure 6–3, Header Longword



Start Address is the starting physical byte address for the transfer. In the original C100 Series implementation of the PBUS, bit 31 of the address is reserved, giving a PBUS address space of 2 GBytes. Addresses from 0 to 1 GByte are physical memory, and addresses from 1 to 2 GBytes are I/O space. The enhanced C200 Series version of the PBUS adds address bit 31, expanding the address space to 4 GBytes. Memory and I/O space are differentiated by the Transaction field, thus enlarging both spaces to 4 GBytes each.

Byte Count is positive number of bytes to be transferred. The field is 16 bits, giving a byte count range of 1 byte to 64 KBytes (a count of 0 is 64KB).

The eight-bit Transaction field specifies the PBUS operation to be done. Eight of the 256 possible operations are currently defined.

Table 6–1 shows the transactions for the two current implementations of the PBUS:

Table 6–1, PBUS Transaction Types

Transaction	C100	C200
0xf8	Write	Memory Write
0xf9	Read	Memory Read
0xfa	TAS	TAS
0xfb	Illegal	TAC
0xfc	Scrub	Illegal
0xfd	Illegal	I/O Write
0xfe	Illegal	I/O Read
0xff	NOP	NOP

6.3.2.2 Transaction Types

As previously mentioned, Memory Space and I/O Space are part of a single address space on C100 Series system and separate address spaces on C200 Series system. Thus the above table shows a single set of Read/Write transactions for a C100 Series system and two sets (Memory and I/O) for a C200 Series system. Accesses to uninstalled memory or unimplemented I/O space are detected by the arbiter and result in a bus error. The enlargement of the Memory Space on C200 Series system is due primarily to the desire of customers to take advantage of the decreasing cost and increasing density of dynamic memory to run *very* large programs.

The Test-And-Set (TAS) transaction is an indivisible read/modify/write operation on a semaphore byte in main memory. This transaction is used for locking queues and data structures used for interprocessor communication. In this transaction the current contents of the addressed byte is returned to the requester and the byte is written to a hex "FF". This operation is defined for a single byte only: any other count in the byte count field should result in a bus error from the arbiter. Unfortunately, the C100 Series implementation does not check the byte count. Although it is possible, therefore, to do a TAS of more than one byte, CCUs are not capable of generating such a transaction. It is highly recommended that new implementations enforce a byte count of one.

The Test-And-Clear (TAC) operation is identical to the TAS operation except that the byte written to main memory is hex "00". This operation is useful for implementing certain interprocessor communication protocols in a multiple CPU environment. This operation is new to the C200 Series PBUS implementation.

A memory Scrub transaction is used to try to correct previously detected single-bit memory errors. It is implemented in the Memory Subsystem as a read/modify/write with no new data. The data in memory is simply read, corrected and written back. If the error was a true soft error (due to an alpha particle, etc.) the bad data has been replaced with good data and the chance of a double-bit error occurring is minimized. If, on the other hand, an actual failure has occurred in a RAM cell it is impossible to fix the failing data word and subsequent accesses will continue to elicit single-bit errors. The SP2/SP4 is responsible for scrub operations. Since the path to main memory for the C100 Series Service Processor (SP) is the PBUS, Scrub is a legal transaction in the C100 Series PBUS implementation. The SP2/SP4 has a different path to main memory on C200 Series and therefore Scrub is not implemented on the C200 Series PBUS.

The No Operation (NOP) transaction is included both for completeness and testing purposes. The address and byte count fields are ignored but parity is checked on the entire header. The arbiter also ignores the data handshake lines and removes grant as soon as the NOP command is decoded.

6.3.2.3 Alignment and Partials

In a PBUS transfer, a sequential string of bytes beginning at the Stardress for a total number of bytes equal to the Byte Count are moved between a CCU and memory or I/O space. There are no 'alignment' restrictions on the values of the Address or Count. This leads to a situation in which the transfer can begin and end on any byte in a longword. The simplest transfers consist of a complete longword or multiple complete longwords. More complicated transfers, such as a partial longword or a block of longwords preceded and followed by partial longwords are equally possible.

Note that partial longword reads are really not terribly interesting in memory space. They are, however, quite useful in I/O space where registers are frequently less than 64 bits. Partial longword writes, however, are a different story. They are absolutely essential in both memory and I/O space so that pointers, registers, status indicators and a host of other things need not be padded to 64 bits and aligned on longword boundaries. The C100 Series/C200 Series CPU architecture is byte addressable with no special alignment restrictions and the PBUS mirrors this.

There is a cost for arbitrary alignment and byte count, however. The most hardware intensive transaction is a leading partial longword followed by at least some portion of another longword. While it may not sound very complicated, it took more hardware than would fit in the original C100 Series PBUS arbiter and was therefore not implemented. The C100 Series arbiter gets around the problem by accepting the partial longword and then removing grant. This forces the remainder of the transfer into another PBUS transaction, which is then longword aligned. *This is the only situation in which Bus Lock does not guarantee that a transfer request will complete in one transaction.* The C200 Series PBUS implementation has no such restriction.

6.3.2.4 Data Transfer

In a PBUS transaction, the sequence starts when a CCU asserts its request to the PBUS arbiter. Some number of ticks later the arbiter asserts grant. In the tick immediately following the assertion of grant the CCU drives it's header onto the data bus. Now the bus is set up for a data transfer. All ticks following the header until either request or grant is deasserted can be used to transfer data. The state of the data handshake lines in a given tick determines whether data actually moves across the bus.

The handshaking of data transfers within a transaction is accomplished with three signals:

- Data valid
- Memory buffer available
- Channel buffer available

The assertion of data valid signal during a tick indicates that the data sender is driving good data onto the data bus during that tick. The assertion of one of the buffer available signals, memory buffer available for writes, channel buffer available for reads, during a tick indicates that the receiver of data can accept data during that tick. A data transfer occurs when both data valid and buffer available signals are asserted during the same tick.

This handshake allows for latency and pipeline bubbles in either the data sender or receiver. The Memory Subsystem, for instance, will always have a multiple tick startup latency on memory read operations, and may have pipeline bubbles due to memory refresh and CPU accesses.

6.3.2.5 Errors

The PBUS arbiter monitors transfers for a number of error conditions. When an error occurs the particulars are stored in an error log for examination by the SP2/SP4. In general, the arbiter informs the current bus master of errors as soon as it detects them.

The arbiter checks the header longword and all data received from a CCU for parity. If bad parity is detected, the bus error signal is asserted to the CCU the tick after the bad data was on the PBUS. The arbiter only checks write data when the CCU asserts the data valid signal. The state of the data bus is considered indeterminate if the data valid signal is deasserted and therefore parity is not checked.

Note that it is possible for the last write data in a transaction to have a parity error, resulting in a bus error the tick after both request and grant have been deasserted. CCUs must check for bus errors during this tick.

The arbiter checks several things in the header longword besides parity. The header is checked for valid address, byte count, and transaction combinations. Obvious errors such as illegal transactions or access to unimplemented I/O space are generally detected during the tick following the header and the transaction is then terminated with a bus error before any data is transferred.

More subtle addressing errors, such as a transfer which starts in installed memory and runs over into uninstalled memory, are generally detected and reported at the time the CCU oversteps the boundary. For writes, the bus error is asserted during the tick following the one in which the first out of range data is written. For reads, the bus error is asserted during the tick in which the first out of range data would have been returned.

6.3.3 PBUS Interrupts

The interrupt portion of the PBUS provides 256 system wide interrupts. CPUs use interrupts to initiate I/O requests, and CCUs use them to signal I/O completion. Interrupts are also used in booting CCUs and to signal some error conditions to the SP. The interrupt bus is logically part of the PBUS, but it does not share any signals or circuitry with the data transfer portion of the PBUS.

The interrupt bus consists of a central arbiter, an interrupt transmitter/receiver for each processor, and the necessary backplane connections. A processor which has an interrupt to transmit asserts its request to the arbiter. The arbiter services requests in strictly round robin order. Since the interrupt cycle is fixed length, the maximum latency to send an interrupt is equal to the cycle length (including arbitration) times the number of installed processors.

Once the arbiter has recognized a request and issued a grant, the requester drives the interrupt vector for the target processor onto the interrupt bus vector lines. All interrupt receivers then examine the vector. The target processor's receiver responds with an interrupt acknowledge if the particular interrupt has been enabled. The transmitter responds to acknowledge by deasserting its request and notifying its processor of successful interrupt transmission. The receiver is responsible for interrupting its processor in some appropriate fashion.

If the interrupt receiver is disabled or no receiver on the interrupt bus has been programmed to respond to the current vector, the cycle terminates with no acknowledge. The arbiter then proceeds to the next requester. The interrupt transmitter has the option to immediately terminate its request and report failure to its processor, or to keep trying to send the interrupt until its processor gives up based on a software or microcode timeout.

6.3.3.1 Interrupt Vector Assignment

By convention, interrupt vectors 0 through 7 are reserved for CPUs, vectors 8 through 15 are reserved for the SP2/SP4, and the remaining vectors are assigned to CCUs. The actual number of interrupt receivers per processor is implementation dependent, but four is a very common number. The assignment of a particular set of interrupt vectors to a CCU is dependent on diagnostic and operating system conventions. It is generally derived from a formula based on the system type and the particular backplane slot number which the CCU occupies.

6.3.3.2 Interrupt Signal Timing

All transitions (assertion, deassertion) of interrupt bus signals are timed from the rising edge of the system clock. As in the discussion of the data bus, events on the interrupt bus are described in terms of events which occur on successive system clocks. In the following discussion, the period of the system clock is referred to as a tick. The interrupt arbiter is responsible for all deskewing of interrupt bus signals.

6.3.3.3 Interrupt Bus Cycle

While the time from the assertion of a request to the assertion of grant is variable depending on other pending interrupts and the requester's position in the round robin, the number of ticks dks during which grant is asserted for a particular interrupt cycle is fixed at five. One tick is required between interrupt cycles to prevent tristate overlap between transmitters, so the minimum interrupt cycle time is six ticks. The arbiter may introduce additional time in between interrupt cycles depending on how the round robin algorithm is actually implemented.

The interrupt transmitter uses it's grant to enable the interrupt vector onto the bus. The vector must be stable on the bus by the time the arbiter asserts the interrupt vector valid signal at the beginning of the tick after grant. The vector must not change until after grant is deasserted. The interrupt vector valid signal indicates that a valid vector is on the bus and should be examined by all interrupt receivers. The interrupt vector valid signal lasts exactly three ticks.

An interrupt receiver may assert the interrupt target acknowledge signal as soon as it decodes its own vector with the interrupt vector valid signal asserted. The arbiter allows two ticks for the decode to occur. The arbiter does not actually check the state of the ack signal until the third tick of the interrupt vector valid signal. If ack is asserted during this tick, the arbiter generates the interrupt source acknowledge signal to the transmitter during the following tick, which is also the last tick of grant. This indicates to the transmitter that the interrupt has been accepted. The transmitter then removes its request at the beginning of the tick after the interrupt source acknowledge signal. While this may seem like a strange way to implement the acknowledge, it actually simplifies the transmitters and receivers since all timing and deskewing is handled by the arbiter.

If the interrupt is not accepted by any receiver, no ack occurs and the arbiter simply removes grant at the end of five ticks without asserting the interrupt source acknowledge signal. The transmitter may then abort its request or leave it asserted until a software timeout occurs and its processor aborts the request.

6.3.4 PBUS Signal Line Characteristics

While a number of physical implementations of the PBUS are possible, it is intended that all boards plug into a single physical backplane. Should it be necessary to split the PBUS across two backplanes, great care must be taken to control the impedance continuity of the interconnect. Any such setup must undergo careful timing analysis to insure that the setup and hold times presented in this specification are met.

The backplane on which the PBUS resides provides power distribution and signal line line impedance control. It is intended that PBUS interfaces be implemented in the FTTL logic family. The nominal PBUS signal line impedance of 75 ohms was chosen to match the output characteristics of FTTL drivers.

All point to point control lines should be terminated at the receiver in a 220 ohm resistor to +5 and a 330 ohm resistor to ground. Termination of bused lines more difficult. A fully populated PBUS may have quite a stubs and be anywhere from a few inches to over a foot long. At a minimum all bused signals should be terminated with a 220/330 terminator on the card which interfaces to the Memory Subsystem. This ensures that the bus lines stay out of the transition region when the bus is tristated. This keeps the bus receivers on all cards from turning into oscillators and injecting lots of noise into the system.

PBUS drivers and receivers for bused signals should be placed as close as possible to the connector on CCUs to minimize stub lengths. This helps keep down reflections and wiring capacitance.

6.4 CCU—Channel Control Unit

The CONVEX VME I/O Processor (VIOP) Subsystem provides a powerful, high-speed connection between the PBUS and the VMEbus peripheral controllers. It consists of a VIOP, a VME Bus Control Unit (VBCU) and a VMEbus chassis for housing the peripheral controllers, backplanes, power supplies and a circulation fan. The VME IOP Subsystem replaces the Multibus IOP Subsystem as the primary means of connecting peripheral devices to CONVEX computers. The VIOP communicates with disks, tapes, printers, communications, DMA devices, plotters, etc. A VME IOP supports two VMEbuses, with a sustained throughput of 8 MBytes or greater per bus.

Using a high performance processor and static no wait state memory, the VME IOP requires less latency time to process I/O operations and complete transfers than the Multibus IOP. The VIOP connects to the VBCU, residing on the VMEbus, through three 60-pin cables. The VIOP cable interface protocol allows 32-bit data transfers and 22-bit address transfers. In addition, the handshake allows pipelining of data transfers within a cable reducing transmission times for data burst on longer cables.

The VIOP provides a high performance channel control unit that is capable of performing certain functions and relieving the CPU from medial taskings that would normally slow processing. CONVEX version of VME uses the split VMEbus which speeds up the peripheral controllers transfer to the VIOP. The split bus contains two VBCUs, each with four controller slots. This reduces the latency time that a controller must wait to access the bus. The arbiter on the VBCU provides arbitration to each controller on an equal basis. Information transfers are handled by the arbiter in a round-robin fashion, under the premise of first in first out. The cabling scheme allows for the increase of data transfer as well as address transfer increases. This means that the processing of this data is accomplished quicker than on a Multibus Subsystem. The unique concept of the VME Subsystem allows communication between devices without the interruption of internal activities on other devices interfaced with the VMEbus.

6.4.1 VME Subsystem Data Path

In a CONVEX C200 Series supercomputer, the VIOP connects to a Peripheral Interface Adapter (PIA). The PIA is the interface between the VIOP and main memory. The C200 Series standard PIA configuration contains a single PIA and up to four VIOPs.

VME peripheral controllers contained in a CONVEX expansion chassis are connected to a CONVEX C200 Series computer's main memory through a Channel Control Unit (CCU) and the VBCU. The VIOP, which is a VME Input/Output Processor, controls all data transfers between main memory and the VBCU. The VBCU controls all data transfers on the VMEbus.

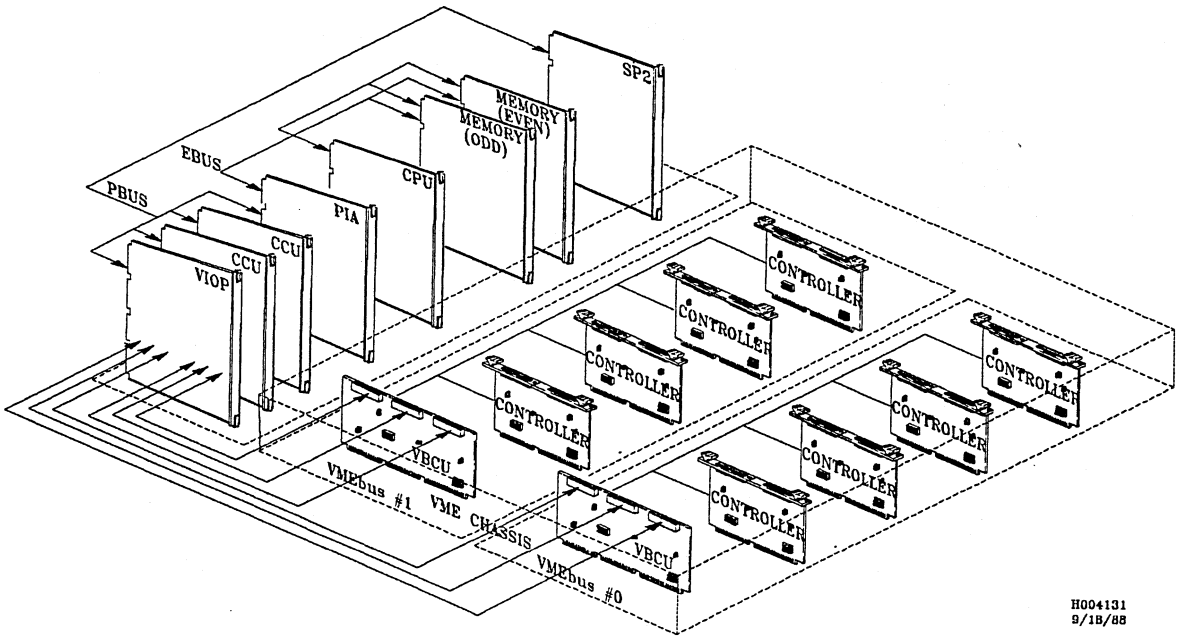
The intelligence for the subsystem is contained in the VIOP. Depending on the type of CONVEX computer, from one to five VIOPs can be installed in a C200 Series computer card cage.

One version of the CONVEX VME Subsystem contains dual VMEbuses and each VMEbus can contain a VBCU. Up to four VME controllers can be installed in each of the dual VMEbuses. A single VIOP can be used for the dual bus, because the VIOP has two VBCU communication ports. The VIOP ports alternate between the two VBCUs as needed.

The VIOP's transfer bandwidth to and from main memory is many times greater than the VME controller's bandwidth. Therefore, there is a minimum of controller waiting time to gain access to its VIOP port. The theoretical transfer speed of this configuration is approximately 12-Mbytes/VIOP or 6-Mbytes/VBCU. When only a single port is used, the theoretical transfer speed is 10 Mbytes for both the VIOP and VBCU.

Figure 6–4 illustrates the relationships between the major components in the CONVEX VME Subsystem and a C200 Series computer:

Figure 6–4, C200 Series VME Subsystem Block Diagram



6.4.2 VIOP—VME I/O Processor

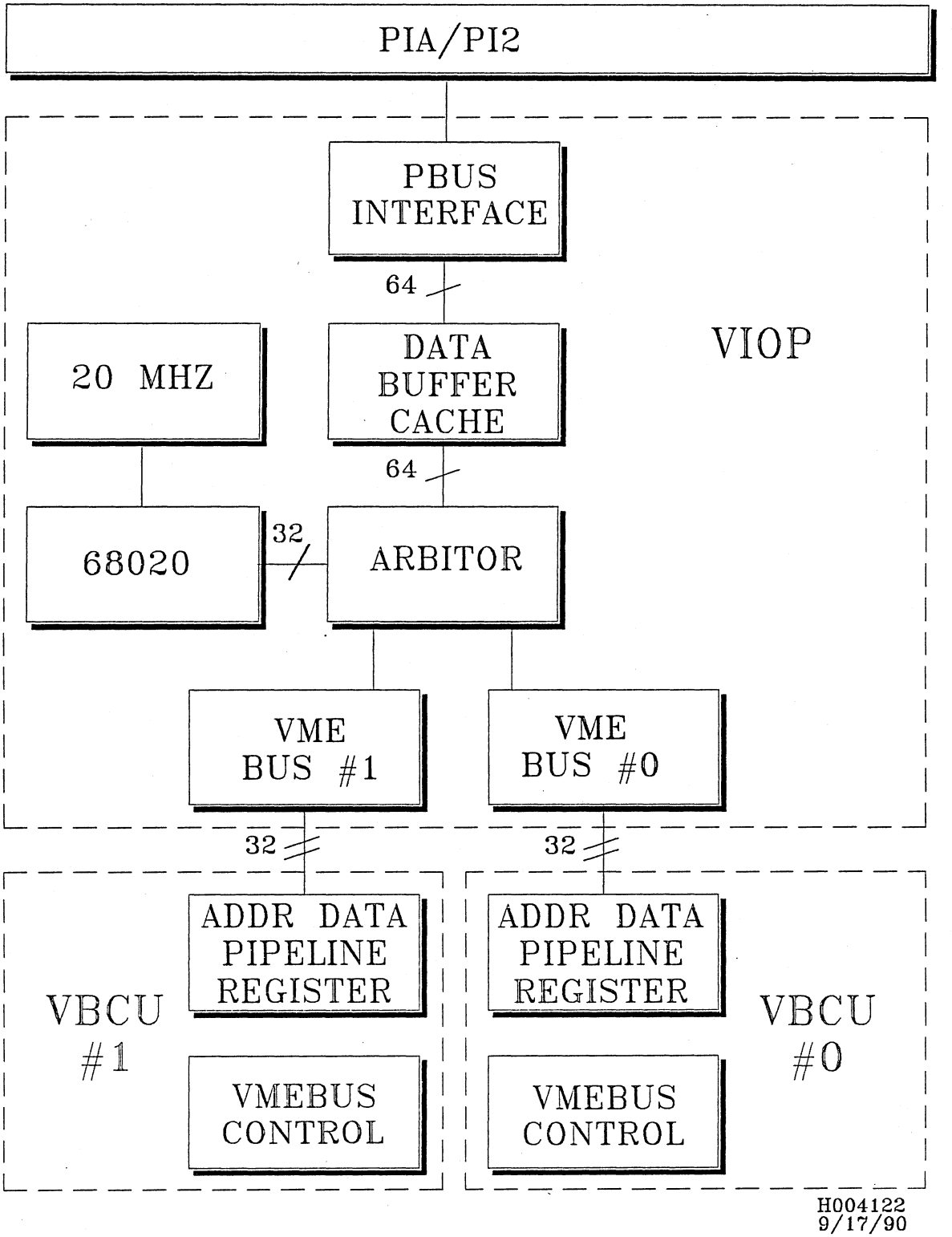
The CONVEX VIOP provides the high-speed connection between the PBUS and the VBCU. The VIOP supports two VMEbus card cages with up to four controller slots each, and an aggregate bandwidth of 8 Mbytes per second. This VIOP acts as a channel controller within the CONVEX I/O Subsystem. The VIOP's VBCU interface architecture allows 8-, 16-, and 32-bit data transfers coupled with 22-bit addressing.

A VMEbus I/O Processor consists of a single CONVEX daughter card (an VIOP), connecting cables, and a VMEbus Control Unit (VBCU) card within each VMEbus cage. A VIOP can be inserted into any I/O backplane slot (CCU0 through CCU3).

Each VIOP has two VMEbus Adapter buses (0 and 1) to which the VMEbus controllers are attached.

Figure 6–5 shows a functional block diagram of a C200 Series VMEbus I/O Processor:

Figure 6–5, VIOP Functional Diagram



Data is pipelined between the VIOP and VBCU in both directions. Data movement is controlled by an asynchronous protocol. The pipelining and asynchronous protocol compensate for propagation delays, introduced by long cable lengths. These features enable the maximum possible transfer bandwidth for a given peripheral device configuration.

Features of the VME I/O Processor are as follows:

- The processor and its memory operate asynchronously from the rest of the VIOP
- Equal interrupt priorities from the VME chassis are serviced in a round-robin manner
- Communication from the processor to the SP2/SP4 is by a 16-bit scan ring register
- The processor is able to determine the type of system it is in by using an additional register
- A VME short address space replaces the Multibus I/O address space and is expanded to 64 Kbytes
- The SP2/SP4 can margin the processor's clock via scan
- An increase in cache windows allows 4 Mbyte of memory accessible through the cache
- Window mapping address in the PMAP registers increased to 20 bits, allowing 32-bit PBUS addresses
- Page protection is on a per controller basis
- PMAP registers containing a TAC bit to test and clear PBUS cycles and a I/O bit that will start an I/O read write PBUS cycle
- Has two VBCU interfaces that use asynchronous protocol and pipelining
- Contains a 128-Kbyte high-speed local memory (cache) for temporary storage of data between main memory and VMEbus controllers
- Has 32-bit data paths between the processor, the cache, and the VME chassis
- The VIOP maintains an encoded slot number of the current VMEbus master. This enables cache window access rights to be enforced on a specific VME chassis and slot number
- Each VME slot number is contained in a VME diagnostic register that is used during loopback testing to identify the device under test
- The processor is able to write bad data parity on any combination of bytes of its 4-byte wide data path during the diagnostic mode of operation
- The processor contains a parity/error address register to identify the address with bad parity, or the address that caused a bus error

6.4.3 VBCU—VME Bus Control Unit

The VME Bus Control Unit provides the interface between the VIOP and the VMEbus. The peripheral controllers attached to the VMEbus, cable to the VBCU for interfacing to the VIOP. An arbitration circuit on the VBCU ensures that there is time for each VME controller on its bus. Data transfers between the controller and the VBCU are handled in a round-robin fashion, but on a first come, first serve basis for controllers with the same priority levels.

Special features of the VBCU are:

- Supports 8-, 16-, and 32-bit data fields on a 32-bit data path and a 22-bit address path, with parity generation and checking on each
- Address mapping between the 4 Megabyte cable address range and the 16 Megabyte VME standard address space will be performed separately in both directions on 4 Megabyte boundaries
- Address mapping between the 4 Megabyte cable address range and the 4 Gigabyte VME extended address space will be performed separately in both directions on 4 Megabyte boundaries
- The VBCU generates VME address modifiers 2D hex (short), 3D hex (standard), or 0D hex (extended) on VIOP 68020 cycles to the VME chassis, and responds to address modifiers 39 and 3D hex (standard), or to 09 and 0D hex (extended) on VMEbus controller cycles to the cache
- VMEbus timeout is controlled by the VBCU and is set at 768 microseconds
- The VBCU control register handles the power supply margining and shutdown
- Air flow sensor and bus arbiter timeout error bits are available in the VBCU error log
- The VME interrupt acknowledge daisy-chain is eliminated through the use of a non-standard backplane
- Seven unique bus request levels are available through the non-standard backplane
- The VIOP provides an encoded slot number of the current VMEbus master, allowing cache window access rights to be enforced based on VME chassis and slot number
- VME chassis reset is automatically asserted on reset to the VBCU, and can be set separately through the VBCU control register
- Provides a 16-MHz VME system clock for the VMEbus
- The VBCU is a triple height (9u), single width VME board
- All VIOP cabling is attached through a three 60-pin foreplane connector
- Contains a low-airflow sensor for the VME chassis

6.4.4 VMEbus Arbitration

A non-standard backplane allows up to 7 bus requests and grant levels. Each peripheral controller contains a separate bus request output and bus grant input in the slot adjacent to the system controller. Bus requests are serviced in a round-robin scheme. This allows placing controllers in any slot, without reguard to the integrity of the VMEbus grant daisy-chain and the resulting slot-dependent priority. This also allows the current VMEbus master to be uniquely identified by slot, allowing the cache window access rights being assigned on a chassis-and-slot basis, and allows identification of an errant controller.

6.4.5 MIOP—Multibus I/O Processor

CONVEX Multibus I/O Processor (MIOP) provides the connection between the PBUS and Multibus-based peripheral controllers. A Multibus I/O Processor Subsystem consists of a single CONVEX daughter card (an MIOP), connecting cables, and a Multibus Control Unit (MBCU) card within each Multibus cage. An MIOP supports up to two Multibus card cages with up to eight controller slots each, and an aggregate data bandwidth of 8 Mbytes per second. A MIOP can be inserted into any I/O backplane slot (CCU0 through CCU3).

Each MIOP has two Multibus Adapter buses (0 and 1) to which the Multibus controllers are attached. Each controller then connects to a local controller bus that ultimately connects to the devices.

A CONVEX MIOP supports devices such as disks, tape drives, printers, and terminals using Multibus (IEEE P796) controllers. These devices have maximum transfer rates of 1 to 2 Mbytes per second. The maximum throughput available to any one device attached to an MIOP is 4 Mbytes per second.

The following list describes the MIOP features and capabilities.

- The Multibus (IEEE P796) is the host bus for all peripheral controllers.
- The MIOP hosts I/O device drivers, offloading the CPU.
- CONVEX supports multiple MIOP's per system.
- The MIOP processor is a 68000 that runs on the 10-MHz system clock.
- The card contains 512 Kbytes RAM and provides byte parity.
- The SP2 initiates diagnostics.
- Bootstrapping is done with a combination of local EPROM and scan rings.
- Aggregate bandwidth from the Multibuses to the PBUS is 8 Mbytes per second.
- Each MIOP supports two Multibus cages.
- A Multibus cage may be placed up to 25 feet from its associated MIOP.
- A map is provided allowing a 20-bit Multibus address to be mapped into the 32-bit CONVEX physical address space. This 32-bit space is composed of 1 Gbyte of physical memory space and 1 Gbyte of physical I/O space.
- Local Memory may be placed within a Multibus cage, but one Mbyte of Multibus memory address space is mapped into the MIOP Cache. The 68000 uses a register on the MBCU to determine which 1 Mbyte out of 16 is mapped into the MIOP Cache.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 7
Utility Subsystem

7.1 Overview

The C200 Series Utility Subsystem may consist of a CPX or CUE/CUO pair, a System Control Monitor (SCM or ESM), and the Service Processor (SP2/SP4). The combination is model dependent. The utility functions that occupy I/O space are accessed by the CPUs and the PIA/PI2 via the even memory bus. All data is passed on the upper 16 bits of the 32-bit data bus, along with parity for the entire data bus. The utility functions that occupy I/O space include the following:

- The Referenced and Modified (R&M) bits monitor the activity on up to four memory buses.
- The Physical Configuration Map (PCM) indicates the physical blocks of memory present in the system and monitors all memory accesses to verify address validity on up to four memory buses.
- The Interval Timer (ITC) generates an interrupt to a programmable vector location.
- The Time of Century Counter (TOC) is used for time stamps and process timing.

Other utility functions that are accessible only by the CPUs include the following:

- The Communication Registers provides a communication link between the processors and storage for Page Table Entries.
- The CPU Interrupt Arbiter monitors the system interrupt bus and processor status to determine when a CPU interrupt is generated and which processor should process the interrupt.

The System Control Monitor (SCM/ESM) prevents situations which could cause damage to the C200 Series system. The SCM/ESM verifies that the system is configured correctly, and that the correct boards and power supplies are installed in the machine.

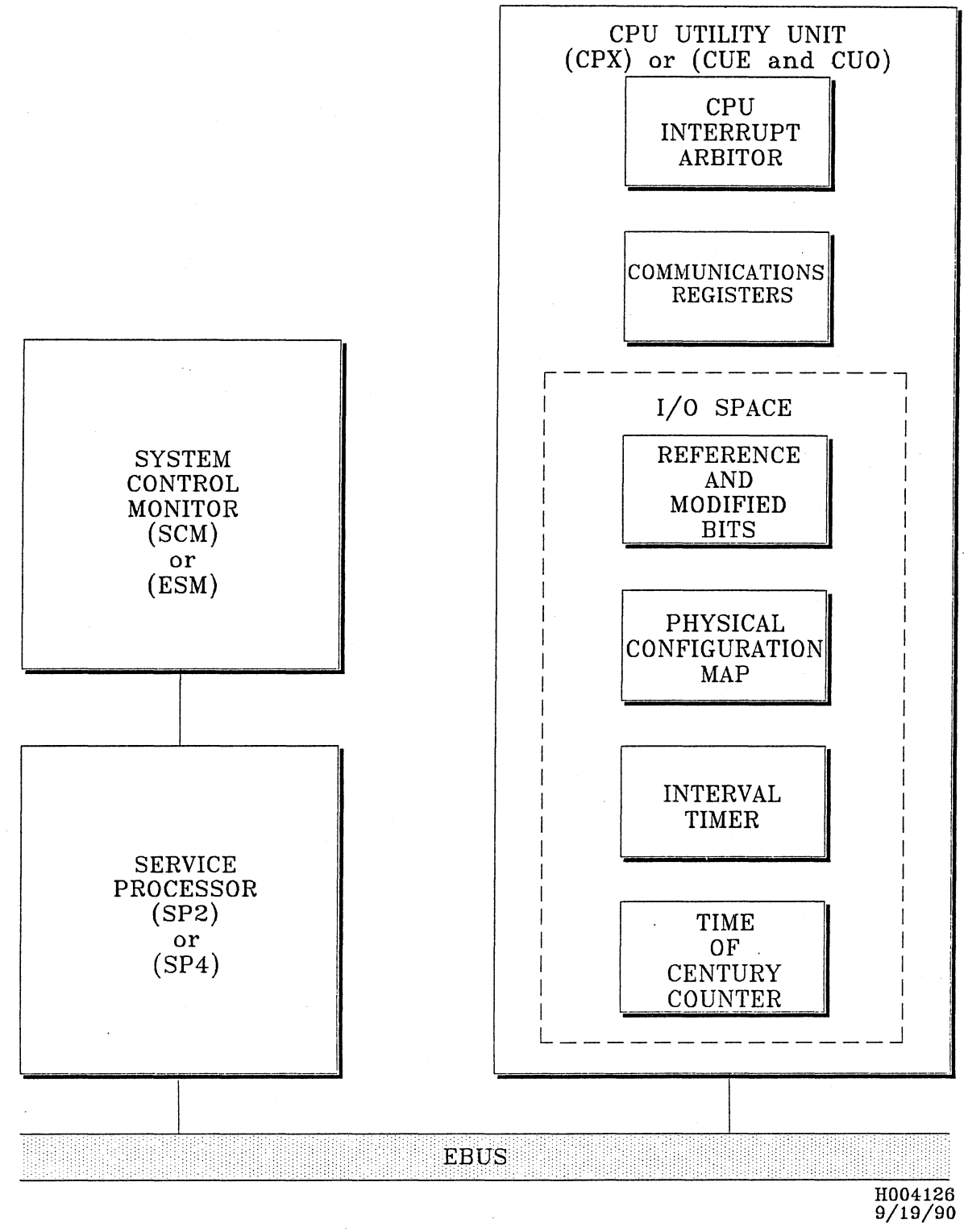
Then, when directed by the user, the SCM/ESM powers up the system in an orderly sequence. When the system is up, the SCM/ESM proceeds to monitor status of the power supplies, current sharing, voltage levels, and the environment of the system.

If the SCM/ESM receives hardware interrupts during the normal monitoring of the system, it powers down the system in an orderly sequence and generates system status hex codes to the front panel.

The SP2/SP4 maintains an 8-bit bidirectional data bus for communication with the SCM/ESM. The SP2/SP4 serves as the user's interface with the power environment of the machine. The SP2/SP4 may request an operation from the SCM/ESM at any time while the system is powered up. The SP2/SP4 relays a request to the SCM/ESM, and the SCM/ESM returns information to the user. During normal operation, the SCM/ESM accepts commands from the SP2/SP4 as interrupts.

Figure 7–1 shows a block diagram of the C200 Series Utility Subsystem:

Figure 7–1, Utility Subsystem



7.2 CPU Utility Unit

The CPU utility unit can be most easily understood by dividing its functions into two sections:

1. Functions that occupy I/O space and may be accessed by the CPUs and the PIA/PI2.
2. Functions that are only accessible by the CPUs.

The functions in the I/O space that may be accessed by the CPUs and the PIA/PI2 include the following:

- **Referenced and Modified Bits**—The Referenced and Modified (R&M) bits are used by the operating system to optimize the use of accelerated blocks of memory by monitoring the activity on up to four memory buses.
- **Physical Configuration Map**—The Physical Configuration Map (PCM) indicates physical blocks of memory present in the system and monitors all memory accesses to verify address validity on up to four memory buses.
- **Interval Timer**—The Interval Timer (ITC) is a 16-bit programmable timer which generates an interrupt to a programmable vector location.
- **Time of Century Counter**—The Time of Century Counter (TOC) is a 64-bit 1 Mhz counter for such uses as time stamps and process timing.

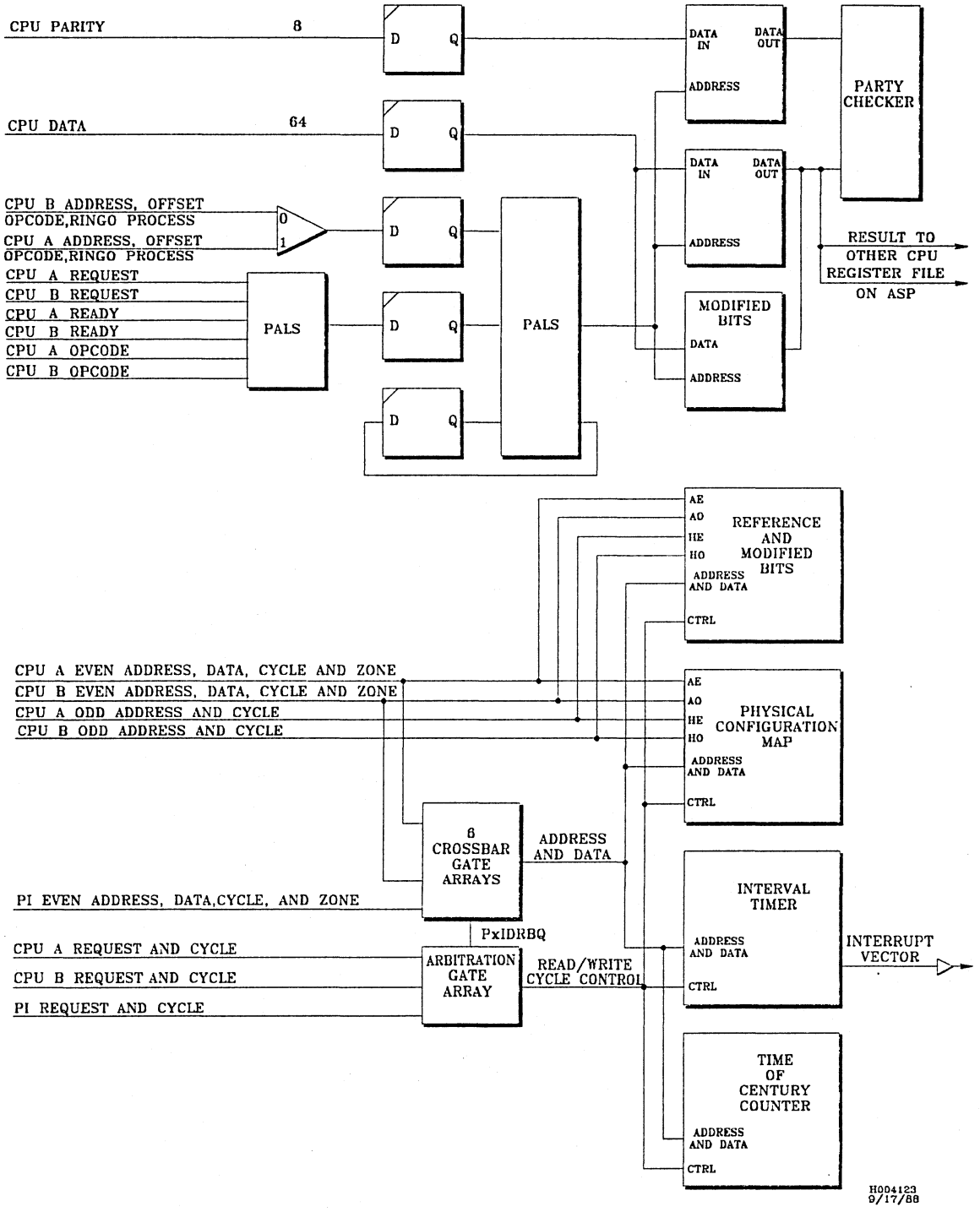
The functions that are accessible only by the CPUs include the following:

- **Communication Registers**—The Communication Registers are 1K by 72 bits, including 8 parity bits and an associated lock-bit that provide a communication link between the processors and storage for Page Table Entries.
- **CPU Interrupt Arbiter**—The CPU Interrupt Arbiter monitors the system interrupt bus and processor status to determine when a CPU interrupt is generated and which processor should handle the interrupt.

> **NOTE**
>
> The illustration that follows describes the CPX version of the Utility Subsystem. For an illustration of the CUE/CUO version of the Utility Subsystem, see the system functional block diagrams in the *C200 Series Major Functional Areas* subsection of Chapter 1. The system functional block diagrams in Chapter 1 also show which version of the Utility Subsystem (CPX or CUE/CUO) goes with each C200 Series model.

Figure 7–2 shows the CPX Functional Block Diagram:

Figure 7–2, CPX Functional Block Diagram



7.2.1 I/O Access Ports

The I/O space is selected when bit <9> of the level 2 Page Table Entry (PTE) of the addressed page is set. This provides 2 Gbytes of separate I/O space, in addition to the 2 Gbytes of physical memory address space. Address translation for I/O space references is performed in the same manner as for physical memory references including all the protection checks. Due to board I/O limitations, the CPU Utility Subsystem is only accessible from the even memory bus, and only the 16 most significant bits of the corresponding data bus are used for data transfers.

Read/write access from CPU and PIA/PI2 ports is controlled by an arbitration controller and crossbar, similar to that of the Memory Subsystem. The Utility Subsystem uses 6 crossbar gate arrays, while the memory boards use 8. The arbitration controller selects between direct requests from any of the 5 ports. The crossbar passes data between the selected port and the R&M bits, the Interval Timer, the Time of Century Counter, or the Physical Configuration Map. Normal CPU to memory handshakes are performed for the transfer of address and control information, write data, and read data. The gate arrays can queue data and addresses for up to 8 read and 2 write requests per processor port. As long as the request queue is not full, the arbitration gate array should be asserting a Utility Subsystem ready line to each of the processors, indicating that it is capable of accepting requests.

7.2.2 Arbitration and Crossbar Gate Arrays

I/O read or write requests may be made by CPUs or the Ebus. They are handled by the arbitration and crossbar gate arrays so that the counters process only one request at a time, and the source of the request is unknown to the counters.

The arbitration gate array functions as the request coordinator for the Utility Subsystem functions located in I/O space. It has separate handshakes for the processors and the PIA/PI2, just as the crossbar gate arrays have separate read and write ports for each of the processors.

The arbitration gate array will accept requests from any processor while its ready line to that processor is asserted. When the arbitration gate array receives a request matched with its own ready signal, it asserts the load request signal corresponding to that port to the crossbars that latched the data on the corresponding port.

The arbitration gate array then generates a start signal to the indicated block, if the block's active signal is not asserted. If the request was a write, the only indication that the request has completed will be the removal of the active signal. If the request was a read, the assertion of a block's data valid signal will indicate that the read data is asserted on the proper data bus, and will remain asserted until the data request signal for that block is asserted by the arbitration gate array.

7.2.3 Referenced and Modified Bits

The purpose of the Referenced and Modified bits is to record all CPU reads (references) and writes (references and modifies) to physical memory. There are even and odd memory address buses for each CPU, and any combination may have simultaneous requests. A portion of the R&M memory banks is dedicated to each of these address buses, and is able to record intermittent or successive memory accesses to any 4 Kbyte page made on that memory address bus.

The R&M has two banks, designated 0 and 1, of static RAMs dedicated to each memory address bus. The RAMs require ECL to TTL translators on all addresses, controls, and data to be written, as well as TTL to ECL translators on read data. The combined translation and access times alone are almost one clock cycle. To solve the timing problem, banks 0 and 1 on a particular memory address bus monitor accesses on alternating clocks, as determined by the select signal. The select signal is the inverse of its previous state, clocked on half clock, so that on the rising edge of one clock it will be true and on the rising edge of the next clock it will be false. One bank accepts requests when the select signal is true, the other bank when the select signal is false. The two sets of RAMs monitoring each memory bus are 128K locations deep which allows the CPX to monitor an address space of 512 Mbytes. The RAMs contain a referenced bit, and a modified bit and parity on the modified bit only.

The default mode for the R&M block is to monitor memory accesses. The memory request indicator and memory address are both latched in another section of the board and are shared with the PCM. Cycle bits are also latched, but are used only by the R&M block. The address and cycle information are set up to the mux-latch combinations within the block, and latched at about half way through the clock cycle following the memory request by the corresponding signal for that block. The data latched in the muxes is held for about a clock cycle and a half, to be sure that stable data is available to the RAMs for the time required to write to the RAMs within the block. Before being written to the RAMs, the data is translated from ECL to TTL levels. Under normal conditions, new data should be set up on each bank of RAMs, available to be written, every other clock cycle. The banks appropriate signal, is used in a pal to determine if a write pulse should be generated for a bank of RAMs on the current clock. Thus if no memory request or if an I/O read request was made, the data that has already been set up will not be written into the RAM. If a valid request is indicated, the cycle bits will indicate whether the referenced bit for a memory read, or both the referenced and modified bits for a write are to be written. Upon completion of this cycle, the block is ready to process another potential memory request or an I/O request.

When an I/O request is made by one of the CPUs or on the Ebus, the gate arrays send a start pulse to the control logic of the R&M block along with information on the type of request, zone, address, and data. On receiving the start signal, the state machine asserts the active signal which indicates that the block is currently processing an I/O request.

The I/O state machine consists primarily of signals which indicate the current clock cycle of an I/O request. While it is possible to process many memory requests or memory requests and an I/O request at the same time, it is not possible to process more than one I/O request at a time, and the ACTIVE signal is used to inhibit further I/O dispatches to this block.

The first signal of the I/O state machine, from latching the start signal, is the request which is latched simultaneously with the request inhibit. On the next rising edge of the system clock, request inhibit is latched to generate a request which is true during the second and third clocks of an I/O request. On the clock edge during the start signal, the R&M block may accept more memory accesses, and begin the I/O request, by generating a request inhibit to be sent to the CPUs to halt memory requests for two clock cycles. A request is also asserted at this time for two cycles. Two cycles allows both banks within all memory monitoring RAMs to begin the I/O instruction, and at the end of the two cycles the bank which accepted the I/O request first will have finished and be able to process any incoming memory monitoring accesses. On the next rising edge, a request is asserted, for a total of two clocks.

During this time, the memory banks process the I/O request, rather than the usual memory request. Once the I/O access has begun on both banks, the request inhibit is removed to allow the bank that has finished the I/O request to resume normal activity. If the I/O request is a write, the instruction cycle is only four clocks long.

If the request is a read, the instruction will take five or more clocks, although in either case memory requests are only disabled for two cycles. During an I/O read request, read data is latched on alternating clocks, from the first bank at the end of cycle three and from the second bank at the end of cycle four. A PAL and register are used to capture the data at the correct time, and to preserve the data from the first bank until data from the second bank is ready, and they may be 'or'ed together. The signal which indicates to the arbitration gate array that valid data is on the read data bus is asserted on the fifth clock cycle, and will remain asserted until another signal is asserted by the arbitration gate array when it is ready to handle data on the read bus.

During I/O accesses, memory requests are never held up for more than two clock cycles. Since any request of the RAMs actually uses them for only two cycles, the bank which accepted the I/O request first will be ready to process an I/O request the clock after the second bank accepts the I/O request. Holding the crossbar data at the inputs to the RAMs to provide stable address, data and cycle information as well as the read data state machine on the outputs of the RAMs allows maximum efficiency given the current timing constraints.

Since physical disks and tapes allow a system to have access to much more information than actual physical memory, the operating system uses the Referenced and Modified (R&M) bits to maintain only the most used 4 Kbyte pages in physical memory. The R&M bits monitor memory accesses on the CPU even and odd ports only. For each 4 Kbyte page in memory space, there is one Referenced bit and one Modified bit located in bits 16 and 24, respectively, of the most significant half word of each corresponding I/O address.

The R&M logic monitors all CPU accesses to memory by updating the address' Referenced bit, and updating it's Modified bit for any write within the 4kbyte space. This information allows the operating system to maintain the most often used memory, while replacing any memory that is not necessary for the current operations. Currently, the R&M are capable of monitoring 512 Mbytes of physical memory.

The implementation of the R&M bits is more complicated. Monitoring all memory accesses means constant activity over four memory buses for the C200 Series system. The implementation is also complicated by the speed of the machine and the size of its virtual memory space, in that no static RAMs are currently available that are fast enough to be updated during one machine cycle and of a size to allow a full R&M functional block to fit with the other necessary functions on one board.

There are a pair of CMOS memory banks large enough to hold a Referenced, Modified, and a Modified parity bit for each 4 Kbyte page in memory space plus translators to and from ECL for the addresses and data for each of the four memory buses. Each memory bank requires two clock cycles per memory access, and therefore a pair of memory banks which accept requests on alternate clocks is necessary to monitor memory requests on consecutive clocks. With each of the eight banks monitoring different memory requests, within a short time none of them will contain the same data. Therefore, when reading a particular location from I/O space, the data from the corresponding location in each bank is **OR**ed to be sure that any reference or modification is not missed.

All locations of the R&M bits may be read or written using any normal scalar or vector memory reference instruction except for the *test-and-set* and *test-and-clear* instructions. Parity checking and address error detection are also included in this functional block.

7.2.4 Physical Configuration Map

The main functionality of the Physical Configuration Map (PCM) consists of four RAMs, and the logic to support their monitoring of all memory requests and process any PCM I/O requests. Each RAM is dedicated to monitoring a particular memory bus. During I/O requests, data from the four RAMs is **OR**ed together for a read request, or all four RAMs are written simultaneously for a write request.

The PCM contains a bank of memory which includes one bit plus parity for every possible memory address, divided into 4 Mbyte segments. The PCM is initialized when the machine is powered up to correctly show all 4 Mbyte blocks of physical memory in the system. The initialization consists of setting all bits, via bit 31 of the PCM I/O space addresses, in the PCM RAMs which correspond to memory present in the system. After initialization, the block may be read or written from I/O space, but its primary function is to monitor all memory accesses and to determine if the accesses are to valid addresses.

> **NOTE**
>
> Accesses to 2 Mbyte blocks are legal, and will access the previous 4 Mbyte block, to allow for C100 Series compatibility.

The PCM receives its I/O addresses from the crossbar gate arrays and its memory monitoring addresses from the latched versions of the upper ten bits of the actual memory address. The crossbar address and the corresponding data, cycle and zone information are latched by the PCM block only during active I/O requests to that block.

The address information is multiplexed at the PCM, with a separate set of two-to-one muxes for each of the four memory buses multiplexing the memory bus information with I/O information, and sending that to one of the RAMs.

During normal operation, the PCM monitors memory requests. A request signal is asserted when a valid access is made. On the next clock, the PCM uses the latched version of this signal and the corresponding latched address to access the data in the PCM RAM at that location. The data read from the pal is qualified with the latched request line and checked to verify that the location contains resident physical memory.

If the RAM data is zero, the memory is not present, and the PCM asserts a memory error, which will generate a system hard error on the following clock. The pal also checks the parity at the RAM, and if the data and its corresponding parity are not correct, that same pal will drive a soft error, which will also drive the memory error. The latter is driven high because in a parity error situation, there is no real way to determine if the parity or the data itself is bad.

Like the referenced and modified bits, the PCM must be able to process a possible memory access every machine cycle for four memory buses. A memory bank and various registers and muxes exist for each of the four memory buses. Memory monitoring for the PCM is a one clock process that starts when a memory signal is received from one of the processors. The address to be accessed is latched in, and is used to address the PCM memories. If the address is valid, the location corresponding to the address will contain a one. When a one is confirmed, the bank prepares to accept a new address. If the memory location contains a zero, an address error is generated which produces a halt signal, and the error log and log ring are stopped to preserve the erroneous state.

All I/O read and write requests are made to the PCM through the gate arrays. Only I/O write requests modify the data in the PCM RAMs, and since all RAMs are modified simultaneously, they should all contain the same data. A parity bit is written with the data, but its primary use is during address monitoring. Data is written high to indicate memory present, and like the Referenced and Modified bits, data from each of the blocks is ored together for I/O reads. This functional block also has internal parity and address error detection.

In the event of an I/O access to the PCM, the arbitration gate array will generate a start signal to initiate the PCM state machine. This signal is sent to the PCM and latched near the gate array itself, just to return the signal to the gate array as an active signal as quickly as possible. In addition, the start signal is latched at the error log to help determine any error source, and to become a request inhibit which combines with certain Referenced and Modified signals to generate a DCU inhibit request. This signal will prevent either DCU from making memory requests during the clock cycles that it is asserted. Thus the PCM does not need to monitor memory accesses during the next cycle (since the request signal and address are latched, and the PCM is always a clock behind the actual requests being made), and will be free to execute the I/O request.

7.2.5 Interval Timer and Time of Century Counter

The two primary tasks of the Interval Timer and Time of Century Counter are to count when indicated and to process I/O requests.

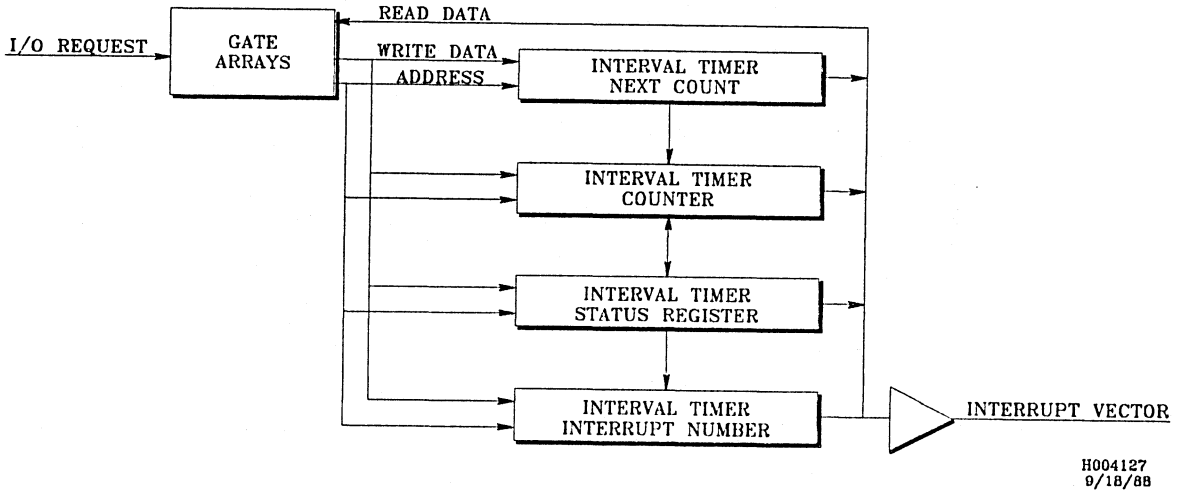
The Interval Timer and Time of Century Counter are not actually counters but a pair of byte wide register files which utilize an Arithmetic Logic Unit and address counters to increment or decrement counter data as needed. Register file R0 is a non-readable counter. When writing any data into the counters, it is first latched in a series of input registers, which hold data, address, cycle type and zone bits while the I/O access is active. The data immediately passes through an 8-bit wide 4-to-1 multiplexer, which during I/O accesses should select either the Most Significant Byte (MSB) or Least Significant Byte (LSB) of I/O data. During writes, the data will be either written into the register files or the status registers. I/O writes will write identical data into the same locations of the two register files. I/O reads will store the requested data in the registers following register file R1, which will eventually drive the data back to the crossbars. Accesses to the register files do not necessarily correspond to I/O addresses.

7.2.5.1 Interval Timer

The Interval Timer (ITC) provides the system with the ability to program when, and to which functional unit to send an interrupt. The functional group consists of the Next Interval Timer Count register (NITC), the Interval Timer Counter (ITC), the Interval Timer Status Register (ITSR), and the Interval Timer Interrupt Number (ITIN).

Figure 7–3 shows the Interval Timer functional diagram:

Figure 7–3, Interval Timer Functional Diagram



The items in the Interval Timer functional group are directly accessible in I/O space. The actual timer is a 16-bit down counter which counts once per 10 microseconds. The timer is parallel loaded from the Next Interval Timer Count register on the next count after counting down to zero, or may be loaded directly from the I/O address. Also upon counting to zero, it loads the status register as described in the following text.

The status register contains the following information:

- FULL—Readable only at bit 16, this bit is set on the next count after the counter reaches zero, and indicates that an interrupt is to be sent. Cleared when read.
- OVF—Readable only at bit 17, this bit is set if the counter reaches zero and the FULL bit is already set. Cleared when read.
- ON—Bit 18 may be read or written, but must be high for the timer to decrement.
- RESERVED—Bits 19–31 are reserved for future use.

Upon the next count after the ITC reaches zero, an interrupt request is sent over the System Interrupt Bus (SIB). The arbiter of the bus will then generate a bus grant, and in response the Utility Subsystem transmits the contents of the ITIN register as the interrupt vector number for the duration of the bus grant signal. The FULL and OVF bits, once set, will remain set until the ITSR is read. It is recommended that the counter be turned off when loading the NITC register.

The Interval Timer is a 16-bit down counter, which decrements every 10 microseconds. On the next count pulse after the counter has reached zero, an interrupt will be requested on the system interrupt bus at the address loaded by the user into the Interval Timer Interrupt Number register. Also at this time, the counter will be loaded with the Interval Timer Next Count. The full and overflow bits will be set as discussed in the previous section on the Interval Timer.

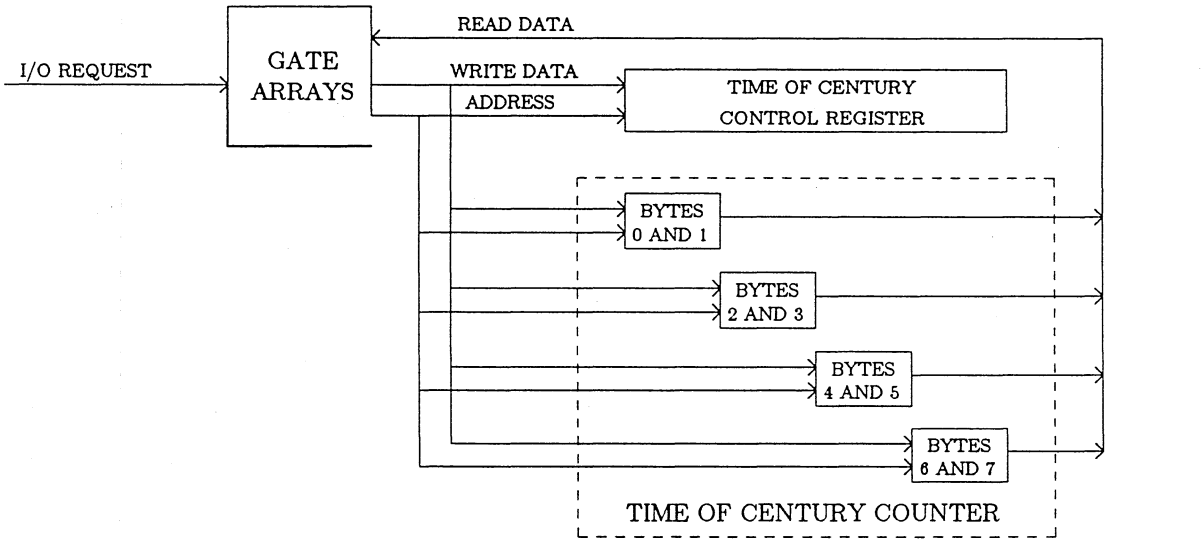
> **NOTE**
>
> Note that I/O writes to the Interrupt Number write a location in the register file and a separate register; only the register file data is accessible by I/O read, the register data is used to assert the interrupt address on the system interrupt bus during active interrupts.

7.2.5.2 Time of Century Counter

The Time of Century Counter (TOC) is a 64-bit up counter that is incremented once every microsecond. The TOC has a simple one-bit status register accessible in I/O space which is set to allow the TOC to count. Typically, the TOC will only be loaded when the machine is powered up, since it may count for over 500,000 years without being reset. When writing to the counter, the TOC may be written to directly in I/O space, but only two bytes can be written at a time.

Figure 7–4 shows the Time of Century Counter functional block diagram:

Figure 7–4, Time of Century Counter Functional Diagram



In order to load a correct 64-bit value, it is therefore necessary to write to the TOC at four separate locations with four separate instructions. To avoid an incorrect load by possibly counting during this time, it is advisable to clear the control register and disable counting until the load is complete.

During a read of the TOC, the 64-bit counter can only be read two bytes at a time, but it is not desirable to turn off the counter each time it is read. To avoid disabling the TOC yet provide an accurate value during a read, there are two copies of the TOC. One copy is locked while being read and the other copy which cannot be read is continually updated and used to correct the readable copy when the reads are complete.

Any read of the TOC should begin with the least significant halfword, which will increment an 8-bit lock counter. A nonzero value in the lock counter will discontinue updating of the readable copy of the TOC. The lock counter is incremented when the most significant halfword of the holding register is read, and decremented when the most significant is read. The size allows it to be simultaneously read by several processors. Once the lock register has been decremented to zero, the holding register resumes continuous updating. No attempt has been made to prevent the overlapping of writes to the TOC.

The Time of Century counter should be initialized shortly after the machine is powered up, and usually will not be written to again. It is advised that the counter status register be cleared to disable counting when the counter is being written. When reading that counter, the least significant two bytes should be read first, as this increments a lock counter which inhibits counting of the readable copy of the counter. Reading the most significant two bytes will decrement the lock counter. To allow multiple processes to access the Time of Century counter, the lock counter has a maximum count of 255, and the readable copy of the counter will not be updated while the lock counter is nonzero. There is a writable but not readable copy of the Time of Century counter which is always updated when a count pulse is received, and is used to update the readable copy with a current count when the lock counter is zero.

7.2.6 Communication Registers

The Communication Registers provide a communication link between the processors, as well as storage for Page Table Entries (PTEs) in a single processor or multiple processor configuration. The Communication Registers are a series of 1K x 64-bit words with corresponding 8 parity bits, a lock bit plus parity, and a modified bit per block of 64 register locations. The registers are equally divided among eight potentially running processes, and within those divisions are designated to hardware or rings 0 through 4.

All locations are accessible to either processor and may be manipulated by the Communication Register instruction set. These instructions allow access to either half or the complete 64-bit long word location, manipulation of the interrupt arbiter, read/write access to the modified bits, and read/write access to the lock bits.

The Communication Registers are accessed by a 16-bit address bus. The addresses can have ring access restrictions, as well as process number and function designations depending on the virtual address. Accesses through addresses 3C00–3FFF are direct accesses to the registers and are for privileged users only.

To access the Communication Registers, a processer asserts its request line, the opcode for the desired action, and the correct Communication Index Register (CIR) code and address. When the Communication Registers are able to process the request, a ready line is asserted and on the following clock edge, the information is latched and the request begins processing. A write request requires one clock cycle, and while on that same cycle, the Registers accept another request. A read request requires two clock cycles, and requests are not accepted during the first cycle. Read data is accompanied by a data valid signal to the processor, and lock or word swapping information is required.

When a request line is asserted, the input multiplexers on the address, CIR bits, opcode and ring bits will be set to select the data from processor A. This information is latched on the immediately following clock, as well as the cycle request signal which indicates the first clock cycle of a valid request. If the request is a write, specifically a *snd_l* instruction, it will actively use the registers for only one clock cycle. Once this request is latched in, the address is sent through a series of pals, which translate the virtual address into a physical address with the proper enables and ring qualifications, and then access the lock bit to determine if the data may be written into that register.

If the lock bit is set and the instruction used requires that the bit be cleared, the register has been locked by some process and the data will not be updated. All write instructions do not require that the lock bit be polled. In this example, the lock bit has not been set and the new data is written. When a location is modified, the Modified bit which corresponds to the 64-location block in which that location belongs is also set. This allows a user to rapidly determine if a series of registers needs to be saved off during a context store. There are sixteen modified bits, each corresponding to a 64-register section or the Communication Registers. Modified bits may be written to directly with a *wr_cmod* instruction and read with a *rd_cmod* instruction.

If the second instruction cycle is for other than a CPUA read, the request line for the appropriate CPU is asserted while processing the A request, but as the previous request is being completed, the new CPU request is noted and the state machines set up so that the next ready will be the ready signal for the new CPU. The assertion of the new CPU ready signal also switches the selects on the address, etc., multiplexers to that information from the new processor, which is latched on the following clock. This request is a read request, any of which require two clock cycles, so that on the next clock cycle no request can be accepted. On that cycle, the cycle request is asserted, and the data in the RAMs at the correct address is accessed. On that cycle and the following cycle, the data will be asserted on the backplane bus. On the following cycle, all necessary handshakes are asserted along with the data. Parity on the data read from the RAMs is only checked on the second clock. This is the only time parity is checked on the Communication Register data. Data with bad parity could be written into the registers and not be detected until this time. Also at this time, the ready line is reasserted, since the read request will be complete at the end of the current cycle and a new request may be latched in.

If the third request is a different CPU read, it is completed in the same manner as a request from the current CPU. Once the input data and request have been met with a ready signal and latched in, the Communication Registers themselves are blind to which CPU made the request. The various read handshakes are handled by the same state machines which determine which processor receives the next ready signal.

7.2.7 CPU Interrupt Arbiter

The CPU Interrupt Arbiter is a specialized part of the Communication Registers which monitors the system interrupt bus and accepts only interrupt vectors less than eight. There are 256 total system interrupts, with 248 of these used by the I/O processors.

When receiving an interrupt vector between 0 (highest priority) and 7, the arbiter determines the priority of the interrupt, and decodes it into a one-of-eight vector. This vector is then compared with any pending interrupt. The next pending interrupt vector is determined to be the interrupt with the highest priority. The arbiter uses interrupt masks and idle state information from each processor to determine whether the requested interrupt is allowed, and which processor is to receive the interrupt. In the case of a valid interrupt to a processor, the arbiter sends an interrupt request to that processor and continues to interrupt until an acknowledge signal is received.

The interrupt arbiter uses the same CPU arbitration and latches as the Communication Registers, but with its own specific instructions. The *eni* and *dsi* instructions enable and disable the CPU interrupt arbiter's propagation of selected interrupts from the system interrupt bus to the CPUs. An *eni* instruction enables the arbiter to send detected CPU interrupts, but the interrupts are disabled either by a *dsi* instruction or after sending an interrupt to a CPU, so that a maximum of one interrupt is received by a CPU after an *eni* instruction.

The *wr_imask*, *wr_imode*, and *wr_tcpu* instructions effect only the CPU interrupt arbiter. The *wr_imask* instruction allows the user to write the masking level for interrupts, which allow only particular interrupt levels to be sent to the CPUs. The *wr_imode* instruction installs an eight-bit field which determines which of the CPU interrupts will be global or local. The *wr_tcpu* instruction indicates the CPU to be targeted by any interrupts, and may select any CPU or, all, but is qualified by the presence of the target CPU in the system.

If the interrupt arbiter receives an interrupt within the proper range, and no interrupt was generated on the last clock, it decodes the interrupt from 0 to 7 into [00000001] to [10000000], so that only one bit is set per vector. This facilitates any masking or prioritizing which must be done to the requested level of interrupt. This vector must be compared to any existing interrupt, as well as the masking and mode bits to determine if either processor is to receive the interrupt.

The masking and mode vectors are similar to the decoded interrupt vector, with one bit each corresponding to a level of interrupt in an 8-bit vector although any number of bits may be set. These two vectors are combined with the interrupt enable vector from each CPU and any interrupt from previous clocks to determine if that interrupt is valid, and which processor may be its potential recipient. At this point, an interrupt vector may have several bits set, i.e. be requesting several levels of interrupts. The vector is sent through a pal, which picks up only the interrupt of highest priority, and that interrupt is then compared to any pending interrupt so that the next interrupt sent to a CPU will be of the higher priority.

If interrupts are not enabled at this time, the next interrupt vector will be the combined pending vector and any from the system interrupt bus within the 0 to 7 range. If interrupts are enabled via the *eni* instruction, the next interrupt vector will be the ORed version of the highest priority interrupt and any other pending interrupt. The vector sent to the CPU will reflect only the highest priority interrupt, but this **OR**ing retains any lower level interrupts for later processing.

Once the interrupt vector is ready, it is sent through an encoder which will translate the 8-bit vector into a 3-bit vector of only the highest priority interrupt. This interrupt is latched, along with the selected destination interrupt request, to be sent to the processor on the following clock. An interrupt request signal will be asserted until met with an interrupt acknowledgement from the correct IPP. The target CPU is selected from a combination of the interrupt levels enabled by the CPU, the target CPU as indicated by the most recent *wr_tcpu* instruction, and the idle status of the CPUs. If only one CPU is installed in the system, then eventually the interrupt must go to that CPU. If an interrupt is targeted for a specific CPU, it must go to that CPU. If all CPUs are installed and have equal vectors enabled. Any CPU may be the target, so the interrupt will go to the next idle CPU.

7.2.8 Miscellaneous Logic and Error Detection

A substantial portion of the board has been dedicated to the detection and preservation of erroneous states, both hard and soft errors. The hard error logger detects any errors which are considered fatal to the correct operation of the system, and on detection of such an error both the hard and soft log rings will latch and hold, to preserve as much information about the problem as possible.

The Utility Subsystem contains a soft error log, which latches and holds if a soft or hard error is detected. It contains information on R&M and PCM addresses being accessed, soft error information from the IPP, DCU and VCU, and other internal information which may be helpful in pin-pointing the source of an error. The data will be held from the time an error is detected until the log is read via scan.

7.2.9 Board Level Control

Clocks used to scan in system level controls are free running, but those which effect only on-board functions other than scan are haltable by latched versions of halt or run.

7.3 System Control Monitor—(SCM or ESM)

The major function of the System Control Monitor (SCM/ESM) is to prevent a number of situations which could cause damage to the C200 Series system. This is accomplished by first verifying that the system is configured correctly, and that the correct boards and power supplies are installed.

Then, when directed by the user, the SCM/ESM powers up the system in an orderly sequence. Once the system is up, the SCM/ESM proceeds to monitor the running status of the power supplies, current sharing, voltage levels, and the environment of the system. If hardware interrupts are received during normal monitoring of the system, the SCM/ESM powers down the system in an orderly sequence and generates system status hex codes.

The SCM board is mounted along the side of the system between the assembly wall and the air intake plenum (C201, C202, C210, C220) in order to provide a central location for the many wire/cable harnesses that attach to the SCM. The ESM board is mounted in the second card cage (C230, C240, C232i).

The SCM/ESM interfaces to the following devices:

- SP2/SP4 via backplane connection
- PS1 and PS3 through PS8 control connectors
- CPU front panel, including the key lock switch and reset button
- CPU AC controller
- One additional peripheral AC controller
- Two in-system air-flow sensors
- Six or eight Fan air-flow sensors (model dependent)
- Intake and exhaust air temperature sensors
- Individual C200 Series daughter connections
- Backplane voltage levels

7.3.1 SP2/SP4 and SCM/ESM Communications

The SP2/SP4 maintains an 8-bit bidirectional data bus for communication with the intelligent control on the SCM/ESM. Three handshake lines control the data transfer between the SP2/SP4 and the SCM/ESM. Communication over this bus is commanded only by the SP2/SP4. Thus, the SCM/ESM may not initiate data transfers to the SP2/SP4. The data transfers are multiplexed commands followed by data. The read/write designation of the transfer is encoded in the command field.

The SP2/SP4 may request an operation from the SCM/ESM at any time while the system is powered up. The SP2/SP4 serves as the C200 Series user's interface with the power environment of the machine.

If the user should request that power supplies be margined to plus or minus five percent of nominal, or request the voltage levels of the supplies, the SP2/SP4 relays the request to the SCM/ESM, and the SCM/ESM returns information to the user. All requests for the environmental status of the machine, or the execution of diagnostic commands which exercise the links between the SCM/ESM and SP2/SP4 are also performed in this manner.

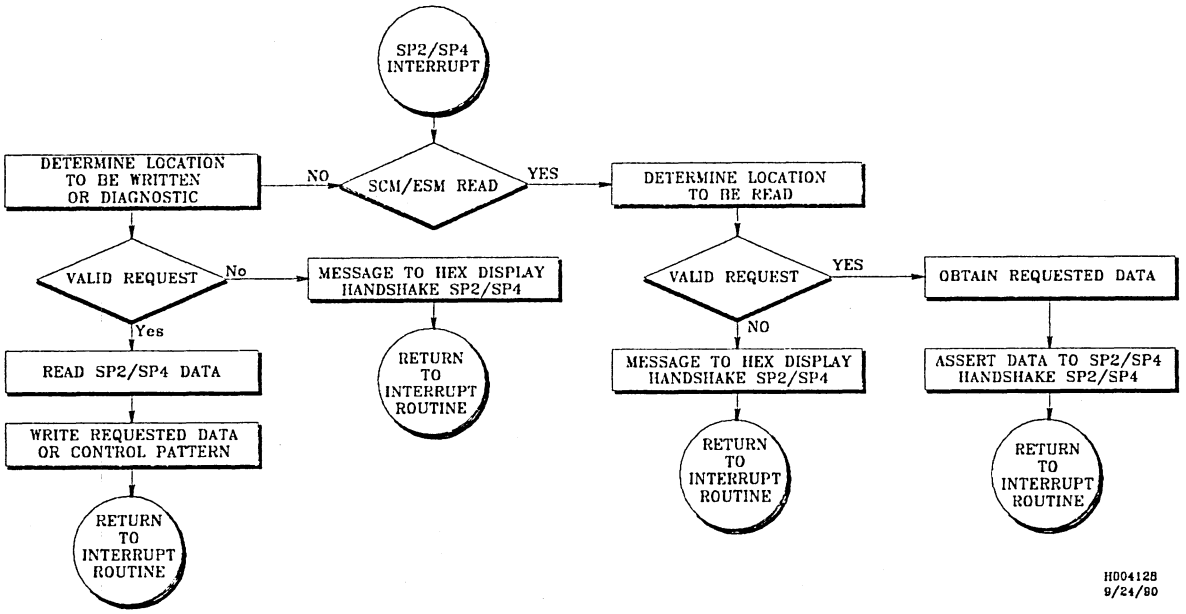
During normal operation, the SCM/ESM accepts commands from the SP2/SP4 as interrupts. Once an SP2/SP4 interrupt has been received, the interrupt handler determines the command code.

SP2/SP4 commands consist of the following types:

- Reads of many of the levels regularly monitored by the SCM/ESM
- Writes to various writable locations on the SCM/ESM
- Diagnostic tests to verify the SCM/ESM-SP2/SP4 interface

Figure 7–5 shows SP2/SP4 command sequencing:

Figure 7–5, SP2/SP4 Command Sequencing



Then the commands are sorted by code, and the proper response is made. If the command is a read of a recognizable location, the SCM/ESM proceeds to either read the indicated register or set up the A/D converter to receive the requested data. When the data has been retrieved, the SCM/ESM asserts the data on the bidirectional bus and then indicates to the SP2/SP4 that the data is stable on the bus.

If the SCM/ESM detects that the request is a read of an unrecognizable location, the SCM/ESM will complete the handshake sequence with invalid data on the bus, and send a error code to the hex displays that indicates that an illegal command was sent by the SP2/SP4.

Similarly, if the command is a write or a diagnostic code, as determined by the most significant bit of the command code, the SCM/ESM will sort through the list of known codes. When the code is recognized, the SCM/ESM will read the data asserted on the data bus and handshake the receipt. If the code is a simple write, the SCM/ESM will write the data to the specified location.

If the command is a margin, the SCM/ESM must first determine if the supply(s) to be margined are already margined, and if so to return the supply(s) back to normal if they are margined in the opposite direction. Then the supply can be margined as requested. but particular care is taken not to margin a supply both +5% and -5% at the same time. Again, if the command code is not recognized, the handshake sequence is completed but an error code to the hex displays indicates that an illegal command has been received.

Each diagnostics code requires only one action, but often a second code is required to remove any undesired state. In particular, when writes to local and remote are used, they must be followed by the *e1* command to clear a flag set by those instructions that disable the SCM/ESM ability to read the LOCAL and REMOTE options on the front panel key switch. If the flag is not cleared, the SP2/SP4 may continue to incorrectly report the status of the key switch. Other diagnostics codes should be polled first to record the current status before altering the status so that the machine can be returned to the current status at a later time.

7.3.2 SCM/ESM Reads and Writes

Most SCM/ESM reads and writes are for monitoring, state reporting to the front panel, or power supply control. There are several exceptions that should be noted, such as a write to the deadman timer. The purpose of the timer is to help verify the operation of the SCM/ESM, and in the event of a failure, to protect the machine from potential damage.

The deadman timer is clocked by the 60 Hz clock, and if it should be allowed to count to its full count and roll over, the carry will turn off the power supplies and send a **00** to the front panel. This timing gives the SCM/ESM 267 milliseconds to write to the panel before powering down the machine. The current hardware should clear the timer approximately every 40 milliseconds.

A read of location 1FB is used to initialize the state machines that control the SP2/SP4-SCM/ESM command interface. A read of 1FC clears the counters used at the fan assembly in the exhaust air path to allow the SCM/ESM to determine how fast the fans are turning.

The external RAM of the SCM/ESM resides outside the normal address space. To access it, the least significant bit of the A I/O port of the microprocessor must be set, which disables accesses to all devices except the program eprom, and overlays the RAM addresses with what would otherwise be device addresses.

7.3.3 Interrupts

The SCM/ESM receives internal timer interrupts and external hardware generated interrupts. The timer interrupts are enabled at various times in every program that the SCM/ESM uses. Timer interrupts are used to reset the deadman timer which informs the user of SCM/ESM fatal internal errors and whose maximum count will turn off all power to the machine. Timer interrupts are also used to time various wait cycles, for example the two second wait on powering up the main CPU power controller, or waiting 100 milliseconds between reseting the PIA/PI2 and SP2/SP4. Timer interrupts are approximately 40.96 milliseconds apart, but can be longer if temporarily disabled.

Hardware interrupts are enabled only during normal monitoring and while a hardware interrupt is not currently being processed. The microprocessor has one enable bit for interrupts. So to enable timer interrupts while disabling hardware interrupts requires writing a one to a particular location. This is necessary while not in normal power up mode to avoid false interrupts.

The sources of hardware interrupts are:

1. The command line from the SP2/SP4 used to begin the SP2/SP4-SCM/ESM command-request sequence.
2. The power interrupt acknowledge signal generated by the SP2/SP4 in response to either a power up or power down interrupt from the SCM/ESM.
3. The combinatorial signal used to interrupt the SCM/ESM immediately when the power supplies are in normal powered up mode and AC power is no longer within tolerance.
4. The combinatorial signal of the outputs of the -10% DC compariters which monitor the backplane voltage levels.
5. The signal from the reset button on the machine front panel, which initiates a reset sequence to the SP2/SP4 when pressed (key lock switch in LOCAL or SECURE modes).

Each of these conditions should arise only when the machine is completely powered up and in the normal monitoring mode, thus they are all disabled with hardware at all other times.

The third interrupt indicates that a major problem may exist in a power supply. The fourth interrupt indicates that a monitored backplane voltage level is below ten percent of nominal. For either failure, the SCM/ESM will enter its interrupt routine, interrupting the SP2/SP4 with a power down interrupt and waiting in a tight loop for the power interrupt acknowledge signal generated by the SP2/SP4. When the signal is received or the wait loop times out, the error condition is retested and if it still exists the system is powered down. If the condition is no longer present, the SP2/SP4 is reinterrupted with a power up interrupt, and again the SCM/ESM waits in a tight loop for the interrupt acknowledge signal. When the signal is received, the system resumes normal operation.

The final interrupt is generated by the user, and results in the standard reset signals being sent to the PIA/PI2 and the SP2/SP4. The reset to the SP2/SP4 includes a power up interrupt and wait loops for the acknowledge signal.

7.3.3.1 Interrupt Handling

During times when the interrupts are enabled, the processor will take the interrupt, and initially read the vector to determine the source of the interrupt. Once the source has been determined, the processor can act either to reset the SP2/SP4, acknowledge SP2/SP4 commands, or in the event of a power failure, power down the machine. The hardware interrupts are valid only after the machine has been powered up and stabilized. During all other times, when an interrupt is recognized, the processor writes a 1 to the most significant bit at location 1E3, which disables the external source of interrupts, but allows the processor to continue to recognize the internal timer interrupts. It is vital that the timer interrupts are processed, since they are used to clear the deadman timer and to time front panel signals.

7.3.4 Pre-power Up Sequence

The SCM/ESM begins operation as soon as the C200 Series system is plugged in and the main power breaker is closed. First, the SCM/ESM clears the on board deadman timer. The purpose of the timer is to turn off the machine in the event of an SCM/ESM failure. The timer is a 4-bit counter clocked with the 60 Hz line clock. If the deadman timer rolls over, its carry out will turn off all power supplies. The SCM/ESM also initializes all writable registers and clears the various state machines, to insure correct operation later.

Next, the SCM/ESM must begin to determine the machine status. It starts by checking which boards are installed in the system (by instructing the internal Analog-to-Digital converter to convert the voltage levels presented by the board id resistor networks). The SCM/ESM requires that boards be in their correct slots. When all slots have been checked, the load current requirements of the installed boards are used to determine if there are sufficient power supplies. Any errors will be encoded and flashed to the user via the front panel hex display. Should more than one error be found, the SCM/ESM will display them in the order of their detection (the first for about six seconds, and up to four additional errors for about two seconds each). In addition, the SCM/ESM will flash the front panel attention light when the CPU power is off, when an error is indicated by SP2/SP4, or when supplies are margined.

After determining which boards are installed, the SCM/ESM polls the power supply exist lines to determine which power supplies are present. The front panel power supply exist lights should be turned on at this point to indicate which supplies measured present.

Once board ID and power supply information is available, the SCM/ESM determines if sufficient power is available for the machine configuration, and if not, will indicate that on the hex display. If the SCM/ESM determines that the board configuration is illegitimate, or that available power is insufficient, it will not allow the system to be powered up.

Once machine state has been determined, the SCM/ESM will monitor the front panel key switch position to determine if the user wants to power up the machine. The SCM/ESM will continue to monitor the machine state, to detect any changes since initial power up. The keylock switch status will be returned to the front panel, to be shown in the front panel lights. When the key switch is positioned to power up the machine, the SCM/ESM once again verifies the existing power supplies and board configuration. If it is a legitimate configuration, the SCM/ESM will pass into the power up phase of its code. A reset from the front panel big red switch will have no effect at this time. The number of -2.0 and -4.5 V supplies actually required by a system is dependent on the type and number of boards in the system. When there are no known problems, the SCM/ESM sends an FF to the hex displays (although the front panel attention light will flash when the system is not powered up).

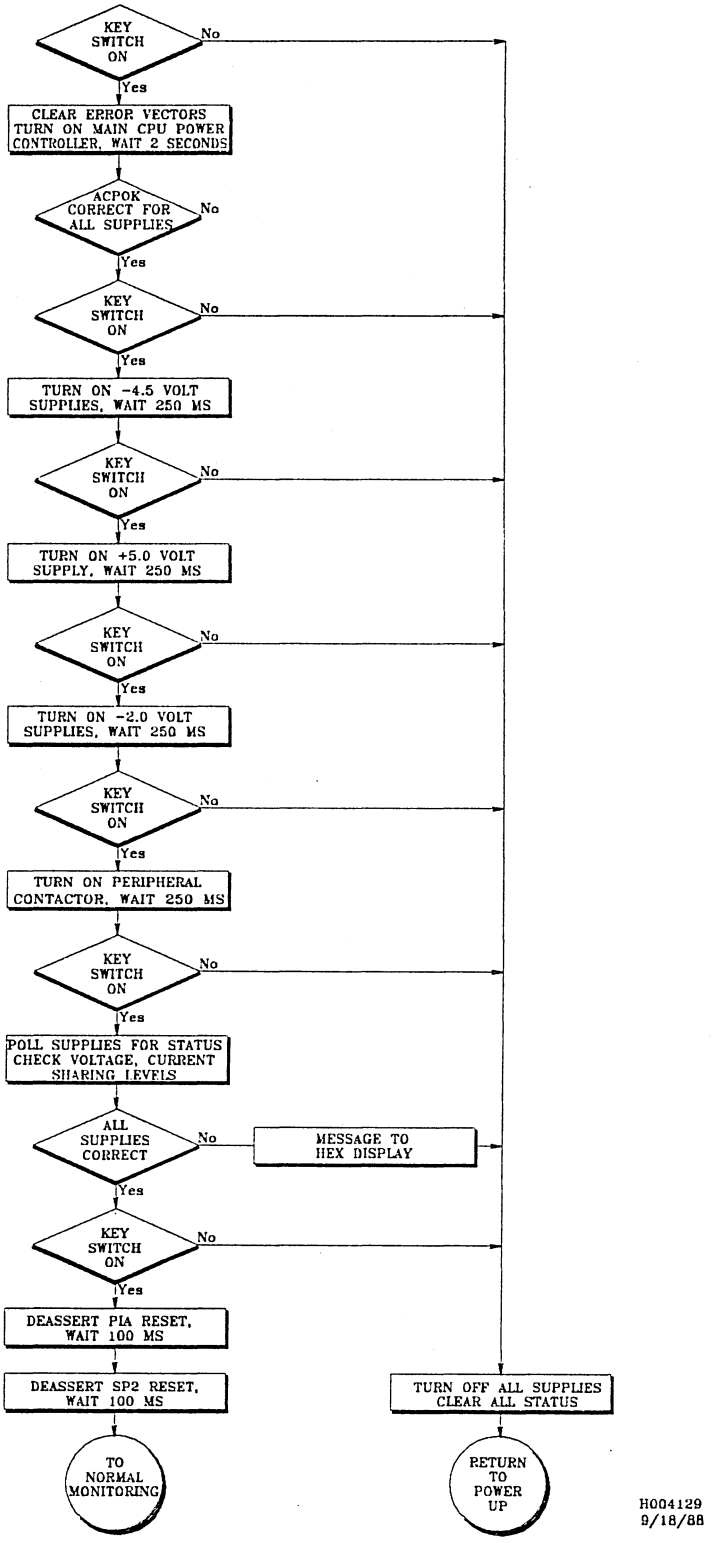
7.3.5 Power-Up Sequence

After once again verifying the power supply and board configuration, the SCM/ESM will begin powering up the rest of the system. After closing the contacter on the power controller and waiting about two seconds for settling time, the SCM/ESM checks all AC OK lines for correct AC power to the supplies. If AC power is correct, the SCM/ESM will begin to enable the power supplies by voltage level, turning on the -4.5 V, then the +5 V and finally the -2 V and peripheral supplies with at least 250 milliseconds between enables to minimize surging. Note that the -5, +/-12 volt supply is on when the main CPU contactor is closed. After turning on each supply, the SCM/ESM monitors the front panel keylock switch to be certain that the key remains in the on position. If the key should return to the off position during the power up sequence, all supplies will immediately be turned off.

When all supplies have been turned on, the SCM/ESM will monitor voltage levels, current sharing levels, and AC power OK to verify that they are all within specified ranges. At this time, the SCM/ESM will recode the front panel power supply lights to indicate that the supplies are operating correctly. When all supplies are on, the SCM/ESM will wait 100 millisecond to remove the reset to the SP2/SP4, and then go into its normal power and environment monitoring mode. Note that the SCM/ESM does not require that any boards be in the system to power up.

Figure 7–6 shows the SCM/ESM Power-UP flow diagram:

Figure 7–6, SCM/ESM Power-Up Flow Diagram



7.3.6 Normal Operation

During normal operation, the SCM/ESM regularly polls backplane voltage levels and current sharing levels. There are also a series of comparitors which monitor each of the backplane voltage levels, and interrupt the microprocessor if any should fall 10% below the nominal value. The AC levels for supplies are also monitored during normal operation only as interrupts, so that if any supply find its AC power out of tolerance, the SCM/ESM will get that information as soon as possible, and power down the machine if the condition persists.

Whether a power down is due to failures or the front panel key-lock switch returning to the off position, the SCM/ESM interrupts the SP2/SP4 to inform it of the pending power down, and then powers down all supplies at once. Should the power down be due to an error condition, the SCM/ESM will not attempt to power the machine back up until the key-lock switch has been returned to the off position.

During normal operation, the SCM/ESM will simply continue to monitor power supply levels, current sharing levels, temperature and fan activity. It will now respond to all requests from the SP2/SP4, as discussed in the following text, and has enabled all hardware interrupts. During all states, it must continue to regularly reset the deadman timer to avoid powering down the system.

Then the various power supply levels and signals are read, and verified to comply with expected levels. If a supply is not within the expected ranges, the SCM/ESM has the ability to determine if the machine may operate safely and correctly without that supply, and may indicate to the user to power down and contact the field service person but that the machine may be turned on again without problems. It has been determined that an unmargined machine should operate with its DC voltage levels within 3% of nominal to be without warnings, and when beyond 7% of nominal will be turned off by the SCM/ESM. A margined supply is allowed to extend these limits to 7% for warnings and 10 % for power down, but the correct operation of the machine is not expected at those levels. Current sharing levels are relative, and should be within about +/- 10% of each other to avoid powering down a machine. The actual supply levels seen by the A/D have been calibrated to allow the level and the variation to fall within the 0 V to +5 V range of the converter.

The next value polled will be the current sharing levels from the various supplies. These are read through the D input of the A/D and at the analog mux select values 0 through 5. There are levels for each of the four -4.5 V supplies and the two -2.0 V supplies. All current sharing levels for each voltage will be measured and simply compared to see that they are within ten percent of each other. If a current sharing level is above the ~10% margin, it will be reported as the failing supply; if more than one level is above that margin, the lowest current sharing level will be reported as erroneous, since the lowest level is used to determine the acceptable current sharing levels. Again, if any are out of tolerance, the SCM/ESM will enter the failure mode.

7.3.7 Environment Monitoring

While all power supplies are on, the SCM/ESM monitors temperature and airflow sensors to verify that the system is running within range to allow correct operation of all parts of the system. To assist the SCM/ESM in determining this, thermistors have been placed in the path of intake air flow, exhaust air flow and one on the SCM/ESM itself. These functions provide warning and shutdown mechanism for proper operating temperature control.

When the SCM/ESM senses that a thermistor is operating within the warning range, a warning message is sent to the front panel hex display. If the thermistor is operating within the shutdown range, the SCM/ESM initiates a power down interrupt sequence with the SP2/SP4, then checks if the shutdown condition persists. If it does, the machine is powered down, and a error message is sent to the hex display. If the thermistor has moved out of the shutdown range, a power up interrupt is sent to the SP2/SP4.

The airflow sensors are open collector output devices which drive their outputs high for normal airflow. Airflow sensors have been placed on the CPX/CUE-CUO, PIA/PI2, and each fan assembly in the system (to help insure proper rotation). If the SCM/ESM detects a single fan airflow failure, it sends a warning error code to the front panel hex display. For multiple fan airflow failures, or CPX/CUE-CUO or PIA/PI2 airflow failures, the SCM/ESM initiates a power down sequence to the SP2/SP4 and powers down the machine.

The SCM/ESM then moves on the check the environmental state of the machine. There are six/eight (model dependent) fan monitors, as well as two air flow monitors on boards in the system and thermistors on both intake and exhaust air as well as the SCM/ESM. The airflow and fan monitors are simple ttl open collector signals and if any of these signals is low, indicate insufficient air flow. The airflow monitors in the system are on the CPX/CUE-CUO and PIA/PI2, and the SCM/ESM will only evaluate those signals if it has determined that those boards are in the system since the pull-up on the sensor is also on the board. If either of these airflow sensors indicates failure with the board(s) present, the machine will be powered down to avoid potential damage from excess heat in the system.

While a fan failure is not considered fatal to the machine, when a thermistor or air flow senser indicates a failing level, the machine will interrupt the SP2/SP4 during the standard power down sequence. It will recheck the failure, and power down if it remains. Warning levels will result in the appropriate code sent to the hex display, and also inform the SP2/SP4 of an environmental problem.

7.3.8 Power Failure Sequencing

During normal operation of C200 Series system, the SCM/ESM expects all supply levels to be within three percent of their nominal values. If they fall outside this range, but within seven percent, the SCM/ESM sends an appropriate error code to the front panel hex display. If an unmargined supply falls outside the seven percent range, the SCM/ESM powers down the machine. If a supply is margined, a warning is not sent to the SP2/SP4 until the monitored voltage level is seven percent or more beyond the supply's nominal voltage. Power down occurs at ten percent from nominal. At power down, the SCM/ESM interrupts the SP2/SP4 to indicate a power down situation. After the interrupt is acknowledged, the SCM/ESM again monitors the power supplies to detect any changes in the situation. If the situation has cleared up, the SCM/ESM issues a second power up interrupt to the SP2/SP4, otherwise the SCM/ESM turns off the relays which supply power to the C200 Series system. This is the standard failure power down sequence, and is executed by the SCM/ESM when any failure is encountered while the system is powered up.

After the supplies have been turned off, the SCM/ESM continues to indicate the cause of the fatality on the front panel hex display. The SCM/ESM will not attempt to power the machine on again until the front panel keylock switch has been returned to the off position and back on again.

If the SP2/SP4 should detect its own power failing, the SCM/ESM will respond by powering down the machine, and sending the appropriate error code to the front panel hex display.

7.3.9 SCM/ESM Hardware

The SCM/ESM is powered up when the main breaker on the power controller is in the **ON** position. The power controller generates 24 volts AC, which the SCM/ESM rectifies and uses with voltage regulators to provide local +5 and +/-8 volts DC. In addition, the output of the power controller is also used to provide a 60 Hertz clock.

The main element of the SCM/ESM is an 8-bit CMOS microprocessor. This is accompanied by an 8K eprom, which contains various programs for a system before power is applied, power supply sequencing routines, power level and environmental monitoring.

The SCM/ESM monitors both digital and analog status information, the latter of which is made possible through the use of an 8-bit microprocessor compatible Analog-to-Digital converter. Such information as backplane voltage levels are adjusted to the 0 to 5 volt range and converted, then the microprocessor can evaluate the result to determine if it is within the acceptable range or some action must be taken to correct a possibly hazardous situation. The A/D converter is used on thermistor inputs, board ID voltage levels, backplane DC levels, and on current sharing levels.

7.3.9.1 Power Supplies

In the power supply configuration, power supplies one and four are -2 volt supplies, five through eight are -4.5 volt supplies, power supply two is a +5 volt supply, and supply three is actually two supplies: a -5 and a +/-12 volt supply. All but supply three are semi-intelligent supplies, providing the SCM/ESM with supply existence signals, AC power **OK** lines, and +/-5% margining capabilities.

In addition, supplies one and four are current sharing supplies, which allows them to share their loading. The current sharing signal generated by the supply is proportional to the loading of the supply. Supplies five through eight are also capable of current sharing.

The SCM/ESM can determine a supply's status before that supply is powered, thereby protecting the supply and system from a possibly hazardous situation. Unfortunately, this information is not available from the supply combination designated as supply three, and the only means the SCM/ESM has to check those supplies is to verify that the backplane voltage levels driven by those supplies is correct after system power up.

As described in previous text, the SCM/ESM receives signals from each supply to determine the validity of the supply-board configuration before allowing the system to be powered up. If the configuration is valid, the SCM/ESM will begin sequencing up the power supplies when the front panel key-lock switch is moved into the LOCAL, SECURE, or REMOTE positions.

The SCM/ESM begins the power up sequence by energizing the relay to the main CPU power, and waiting approximately 2 seconds before doing anything to allow power surges to settle. Then the SCM/ESM checks various lines to be sure all supplies have acceptable AC levels. If so, the SCM/ESM will continue to energize supplies, starting with the -4.5 volt supplies, then +5, -2, and finally the peripheral power supplies, waiting about 250 milliseconds between each to allow surges to settle.

Supplies providing the same voltage level are powered up at the same time. The -5, +/-12 volt supplies are powered up with the main CPU power in the power controller, since the SCM/ESM has no other control over these supplies.

When all supplies are up and have been allowed to settle, the SCM/ESM will poll the backplane voltage levels being provided by each supply to determine that they are operating correctly. If not, the SCM/ESM will power the machine down again, with information to the front panel hex displays regarding the faulty operation.

The AC levels for the supplies, and current sharing levels are also polled to verify correct operation. If the SCM/ESM finds that all levels are within tolerances, it will remove the reset lines on the PIA/PI2 and SP2/SP4, with about 100 milliseconds between their deassertions, and begin its normal monitoring routines.

7.3.9.2 Front Panel Indicators

The front panel has a pair of hexadecimal LEDs which will display an **FF** during normal operation. If any problems are encountered, the display will show the errors in the order which they are detected. The first error code is displayed for about six seconds, and up to four subsequent codes for about 2 seconds each.

The POWER, RUN, and ATTENTION LEDs on the C200 Series system front panel are the same as those on the C100 Series system front panel.

The functions of the three LEDs are as follows:

- The **POWER** LED indicates that switched AC power is turned on. Actual power indicators for each power phase are located on the AC Power Controller.
- The **RUN** LED displays the state of the RUNLED* signal generated by the SP2/SP4.
- The **ATTENTION** LED displays the assertion ERROR signal during power up or any powered off status. The signal is flashed at a frequency controlled by SCM/ESM firmware.

The CPU front panel contains a four-position key lock switch that provides customer security in establishing CPU operating mode.

The four key lock positions have the following functions:

- **OFF**—All switched AC in the system is off.
- **LOCAL DIAG**—All switched AC is on, the front panel reset switch is enabled, and the local diagnostic enable signal is sent to the SP2/SP4.
- **SECURE**—All switched AC in the system is enabled and the front panel reset switch is disabled.
- **REMOTE DIAG**—All switched AC is on, the front panel reset switch is enabled, and a remote diagnostic enable signal is sent to the SP2/SP4.

7.4 SP2/SP4—Service Processor Unit 2/4

The CONVEX C200 Series Service Processor Unit (SP2/SP4) performs a unique set of functions for C200 Series system. In its role as the diagnostic processor, the SP2/SP4 executes CONVEX SPU OS and controls the remainder of the system via scan rings and the EBUS memory interface.

The SP2/SP4 also controls the system console, a remote diagnostic port, and the system boot device(s). All of the system clocks are generated and controlled on this board.

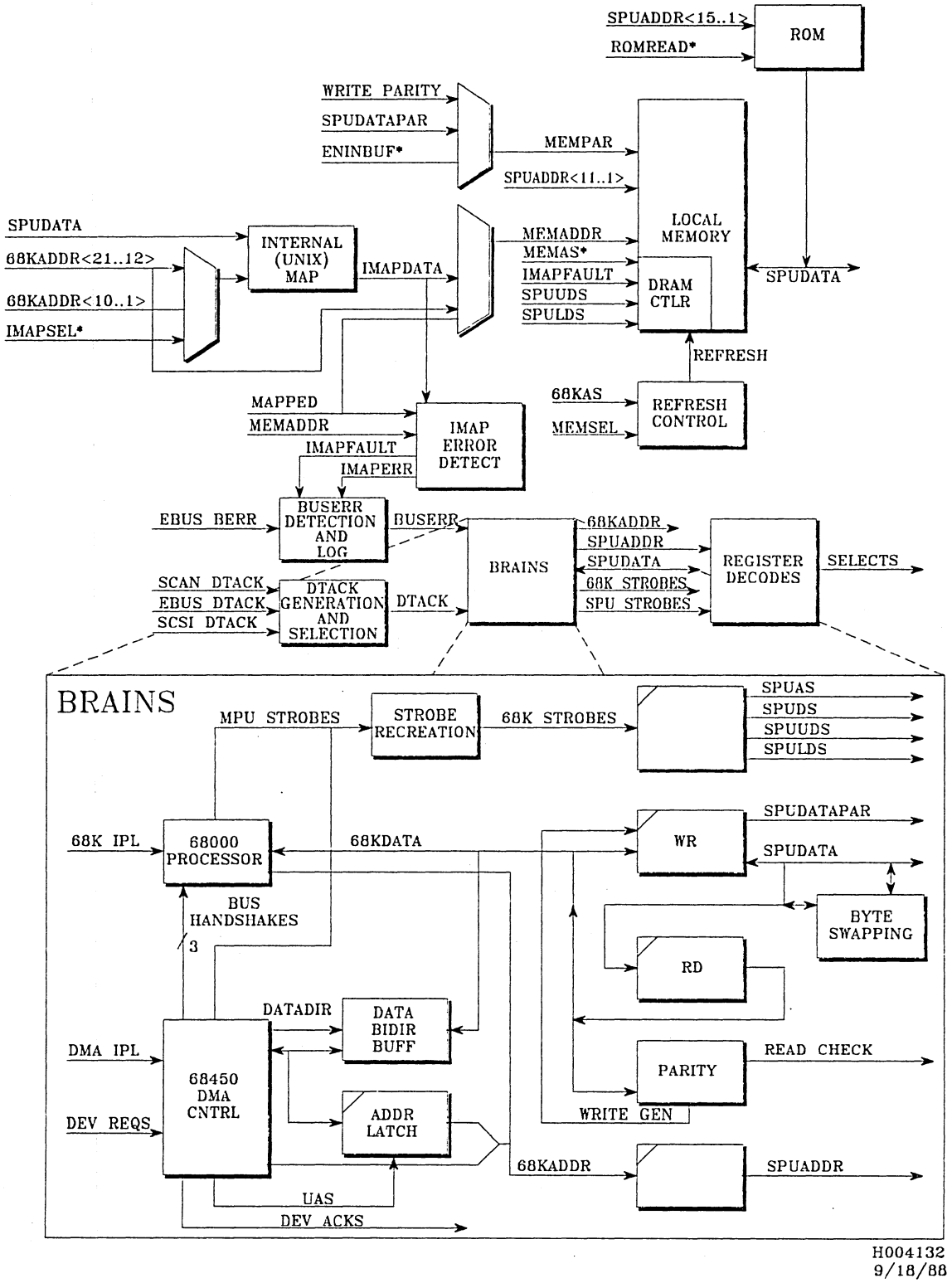
In the remainder of this subsection, the overall structure of the SP2/SP4 is presented. Also, the various address spaces related to the SP2/SP4 and the mapping of addresses between them are described. Later subsections provide detailed discussions of the parts of the SP2/SP4.

7.4.1 SP2/SP4 Structure

The SP2/SP4 system is composed of the SP2/SP4 board, the system console, a remote diagnostic port, and boot device(s).

The SP2/SP4 system is illustrated in Figure 7–7:

Figure 7–7, SP2/SP4 System Block Diagram



H004132
9/18/88

The SP2/SP4 card is composed of four main subsystems:

- The SP2/SP4 processor
- The EBUS interface
- Clock control
- The diagnostic interface

Each of these subsystems is described in detail, in the following subsections.

7.4.2 SP2/SP4 Addressing

The SP2/SP4 must communicate with main memory in order to perform its function. In order to communicate with main memory, part of the memory address space has been made accessible to the SP2/SP4. This accessibility is accomplished by defining SP2/SP4 address space and by providing the means of translating or mapping memory addresses between two address spaces.

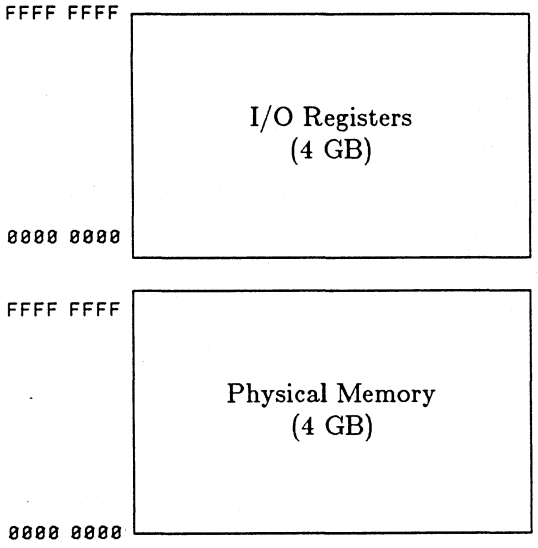
The two address spaces are:

- The CONVEX physical address space
- The SP2/SP4 processor address space

7.4.3 CONVEX Physical Address Space

The physical address space consists of 8 Gbytes, four each of memory space and I/O space, as shown in Figure 7–8:

Figure 7–8, CONVEX Physical Address Space



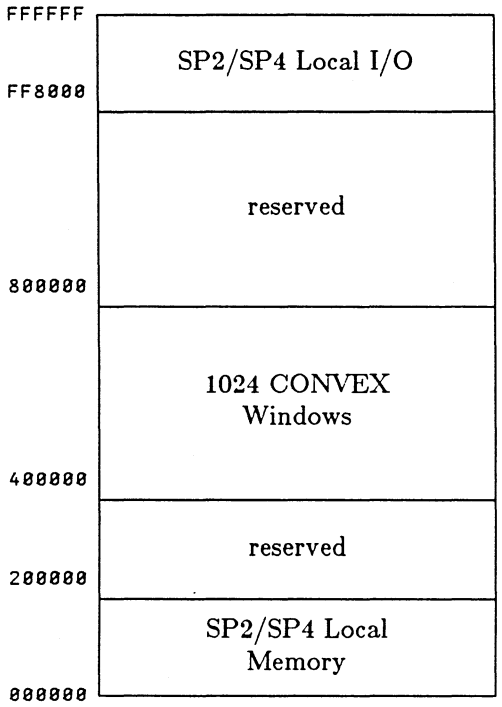
Thus, a CONVEX physical address is 32 bits in length, and enabling accesses to I/O space by setting the proper bit in the address mapping process differentiates between memory space and I/O space.

7.4.4 SP2/SP4 Processor Address Space

The processor of the SP2/SP4, a 68000, is capable of accessing a 16 Megabyte address space. Therefore, a 24-bit address is used for the SP2/SP4 local address space.

This space is divided as shown in Figure 7–9:

Figure 7–9, SP2/SP4 Processor Address Space



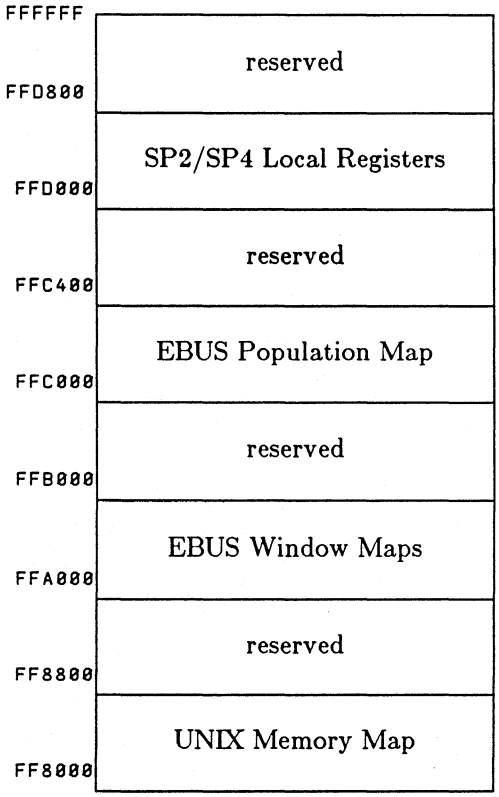
SP2/SP4 Local Memory is populated with 2 MB of dynamic RAM. The area between 400000 and 7FFFFF, called the CONVEX Windows, is used by the SP2/SP4 processor to reference into the CONVEX physical address space via the EBUS. The mapping performed on these accesses is explained in the text that describes CONVEX Windows.

7.4.5 SP2/SP4 Processor I/O Address Space

The top 32 KB of SP2/SP4 processor address space is used for all I/O references. I/O includes devices on the SP2/SP4 card, as well as registers for system control and monitoring.

Figure 7–10 details the I/O portion of the SP2/SP4 processor address space.

Figure 7–10, SP2/SP4 Processor I/O Address Space



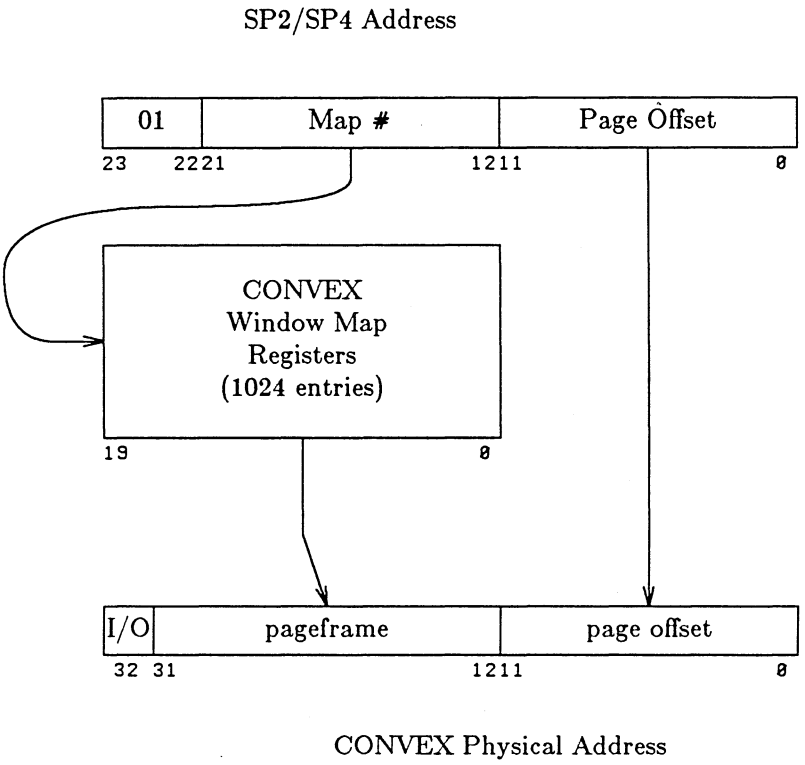
7.4.6 CONVEX Windows

The CONVEX Windows provide a mechanism for the SP2/SP4 processor to access locations anywhere within the 4 Gigabytes of the CONVEX physical memory address space, as well as the 4 Gigabytes of CONVEX I/O space. The processor first programs a set of map registers located in the SP2/SP4 I/O address space. These registers contain bits to control access through the maps and contain the mapping information used to translate an SP2/SP4 address into a CONVEX physical addresses. The EBUS interface allows the SP2/SP4 to perform either byte, halfword, or longword transfers on the EBUS.

Whenever the SP2/SP4 processor references an address between 400000 and 7FFFFF, the SP2/SP4 Window hardware detects this reference and translates it to a corresponding CONVEX physical address.

Figure 7–11 illustrates the address mapping process.

Figure 7–11, CONVEX Window Mapping



7.4.7 SP2/SP4 EBUS Interface

The EBUS interface is the mechanism by which the SP2/SP4 talks to the main memory cards. It is a 32-bit path by which the SP2/SP4 can access either the even or odd EBUS port. The 68000 interface will make requests to the EBUS arbiter and the EBUS interface will handle all arbitration handshaking. The interface will also perform the required address translation for generating the physical MAM addresses, although the Physical Configuration Register (PCR) must be initialized before any MAM accesses can occur.

The EBUS interface will translate 68000 bus cycles into EBUS cycles. Due to the manner in which the 68000 executes bus cycles, read operations will not complete until the MAM returns read data to be accepted by the microprocessor. Write operations will complete when the interface has completed the EBUS request cycle handshaking.

EBUS transfers are "window" transfers, where up to 1024 pages of SP2/SP4 address space may be mapped onto 1024 pages of CONVEX physical address space. The 68000 may then access CONVEX pages as if they were pages on the SP2/SP4. The 68000 may perform read, write, Test-and-Set, and Test-and-Clear operations on memory locations through the EBUS port.

The EBUS commands are shown in Table 7–1:

Table 7–1, EBUS Commands

Command Code	Command Name
00	NO OP
01	Read
10	Write
11	Test-and-Modify

Differentiation between TAC and TAS cycles will be handled by the data supplied to the memory boards. If the required cycle type is a TAC instruction, data of all **1**s should be supplied, and data of all **0**s will indicate a TAC cycle. Thus either byte or half-word data sizes can be accommodated in the Test-and-Modify operations.

7.4.8 EBUS Windows

The EBUS windows provide a mechanism for the programmer to deal with structures in main memory as if they were part of the 68000 address space. The windows allow 1024 selected pages of EBUS address space to be treated as part of 68000 address space. These pages occupy the 68000 address space from 400000 to 7FFFFF. Associated with each page is a register containing both the address of the page in EBUS address space to which it is mapped, and six bits of control information.

During its internal initialization, the EBUS interface clears all the bits to 0 in all 1024 map registers. Before a window can be used, its corresponding map register must have the valid bit set, and the map bits must be initialized to address the desired page in main memory. The 68000 may then read, write, and do TAS or TAC operations in that page.

7.4.9 SP2/SP4 Operation

The SP2/SP4 performs several different tasks in support of the C200 Series system. First, it boots its own operating system (CONVEX SPU OS), allowing it to function as an independent computer in its role of supporting the C200 Series processor. Second, it initializes the C200 Series processor to the proper state so that it can be booted, boots the C200 Series processor, keeps track of errors in the C200 Series processor, and controls the isolation and diagnosis of problems in the C200 Series processor.

7.4.9.1 Error Logging

The SP2/SP4 also takes care of all error logging. There is a specialized area of SP2/SP4 that listens for difficulties elsewhere in the system and then sends the error information to the error log and prints it to the console.

During operation of the C200 Series processor, the SP2/SP4 checks error monitors for error conditions that might need to be put into the error log.

These error conditions include:

- Memory soft (single-bit) errors
- Memory hard (double-bit) errors
- Microstore parity error
- A PTE cache parity error
- Any bus parity error

When single-bit memory errors are detected, the SP2/SP4 enters the address and the data pattern that caused the ECC syndrome into the error log. If a particular RAM chip starts having multiple failures, it becomes immediately apparent.

When a hard error occurs, the function experiencing the error asserts its hard error line and the C200 Series processor is halted. The current SP2/SP4 software includes an error identification daemon that checks to see which error line has been pulled and then tries to determine which board pulled the hard error.

For example, if CPU A were to pull a hard error and shut the system down, all the SP2/SP4 knows at that point is that CPU A pulled hard error. The SP2/SP4 then scans all the CPU A boards to determine which function was the cause.

Once the responsible board has been identified, the SP2/SP4 tries to determine the cause of the hard error—what part of that board was upset and caused the hard error. All error information is then reported to the log.

7.4.9.2 Diagnostics

When the SP2/SP4 is running diagnostics, it has total control over the C200 Series processor. Under a diagnostic, the SP2/SP4 is actually used like a logic analyzer. Since the SP2/SP4 has complete control over the clocks and direct access to the scan ring registers, minute examination of the operation of the C200 Series processor can be accomplished.

The SP2/SP4 does things like scan (load) particular bit patterns into the scan ring registers, gate in a certain number of clocks, and then read the scan ring registers and test to see if everything is as expected. This is why the SP2/SP4 controls the clocks to the boards—it can let one clock go through and read everything again—or after a specific number of clocks. The SP2/SP4 has several different modes in which it allows the clocks to run.

7.4.10 SP2/SP4 Diagnostic Interface

The SP2/SP4 diagnostic interface provides the SP2/SP4 with direct access to areas of the C200 Series processor. The SP2/SP4 directs normal C200 Series processor boot procedures as well as conducts C200 Series processor diagnostics through this interface.

Diagnostic activities are implemented through a set of diagnostic registers on the SP2/SP4. These registers are detailed in the following text.

7.4.11 Test Result Register

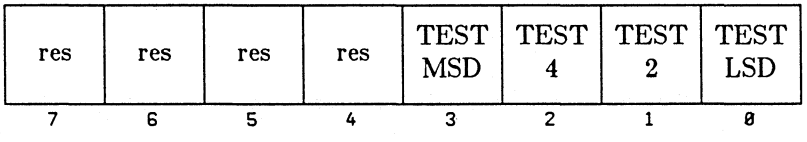
The Test Result Register (TRR) is an 8-bit register, however, only four bits <3..0> are currently used. SP2/SP4 software, through the TRR, controls the status of the four red LEDs on the SP2/SP4 foreplane.

During SP2/SP4 self test, test progress is reported both to the console and to the TRR. If the console should fail during self test, test progress can still be monitored through the TRR. The status of each of the four bits in the register is reported through an LED on the front edge of the SP2/SP4 board. Writing a 1 to a bit in the register causes the corresponding LED to be illuminated. Conversely, writing a 0 to a bit in the register causes the corresponding LED to be extinguished.

All TRR bits are set to 1 by system reset or power up, illuminating all four LEDs. As the self test progresses, the pattern of illuminated LEDs changes to correspond to the number of the test currently executing. When a test fails, the current test number is latched in the register, causing the LEDs to continue to indicate the test number that failed.

The TRR is located at address FFD02B, and its format is shown in Figure 7–12:

Figure 7–12, Test Result Register



7.4.12 Control Panel Register

The Control Panel Register (CPR) is an 8-bit register, however, only four bits <3..0> are currently used. Two bits of this register control the status of the front panel **RUN** and **ATTENTION** LED indicators. The other two bits indicate the position of the front panel keyswitch.

SP2/SP4 software sets or clears bits <3,2> of the CPR, controlling the status of the front panel **RUN** and **ATTENTION** LED indicators. Writing a 1 to either of these bits in the register illuminates the corresponding indicator. These two bits are cleared at power up and by a system reset. The LED indicators are driven by logic on the System Control Module (SCM/ESM) based on the state of the bits in this register.

Bits <1,0> of the register indicate the **1 LOCAL MAINTENANCE** and **1 REMOTE MAINTENANCE** positions of the front panel keyswitch. A 1 is written to the appropriate bit in the register by the SCM/ESM when the keyswitch is in either of these positions. If both of these bits are 0, the front panel keyswitch is in the **1 SECURE EXECUTION** position. Logic on the SCM/ESM senses the position of the keyswitch and reports to the SP2/SP4, which sets the bits in the EMR accordingly.

The CPR is located at address FFD029, and its format is shown in Figure 7–13:

Figure 7–13, Control Panel Register

reserved	Run	Error	Local	Remote
7 4	3	2	1	0

7.4.13 System Reset Register

The System Reset Register (SRR) is an 8-bit register, however, only six bits <6..1> are currently used. These bits control the reset lines to all cards (other than the SP2/SP4) in the processor card cage.

CONVEX SPU OS controls the bits in this register. When a 0 is written into one bit in the SRR, the reset line to the corresponding subsystem is asserted by logic on the SP2/SP4. Note that the PIA/PI2 and CPX/CUE-CUO boards are reset via the I/O bit of the register.

At power up, the System Control Module (SCM/ESM) holds all subsystems in reset—the SP2/SP4 directly, and the remainder of the system through the SRR on the SP2/SP4 (all SRR bits cleared to 0). When the SCM/ESM decides that all is well and applies power to the power supply buses, it takes the SP2/SP4 out of reset. The status of the bits in the SRR (which are still 0s at this point) is then controlled by SP2/SP4 software, which sets them to 1 at appropriate points in the C200 Series processor boot sequence.

When the system is reset via the front panel **SYSTEM RESET** switch, a reset interrupt is sent to the SCM/ESM. The SCM/ESM interrupt handler then resets the system and begins the power-up reset sequence described in the previous text.

The SRR is located at address FFD405 and is formatted as shown in Figure 7–14:

Figure 7–14, System Reset Register

res	IO	PROC D	PROC C	PROC B	PROC A	MEM	res
7	6	5	4	3	2	1	0

7.4.14 Environmental Monitor Register

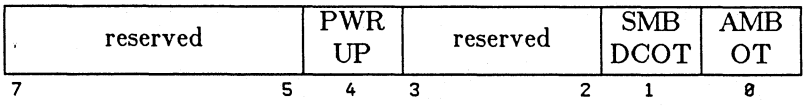
The Environmental Monitor Register (EMR) is an 8-bit register, however, only three bits <4,1,0> are currently used. Two bits in the register indicate that an out of tolerance condition has been detected by the SCM/ESM. The other bit indicates a change in power status.

A 0 in bit <1,0> indicates that the SCM/ESM has detected an out of tolerance condition of the ambient temperature or DC voltages.

Bit <4> indicates power status. When a power interrupt is received from the SCM/ESM, this register indicates whether power is going up or down. A 1 in this bit indicates that power is coming up. The SP2/SP4 may then begin to process the boot code. A 0 in this bit indicates that the system is going down.

The EMR is located at address FFD027 and is formatted as shown in Figure 7–15:

Figure 7–15, Environmental Monitor Register



7.4.15 Error Source Register

The Error Source Register (ESR) is an 8-bit register, however, only six bits <5..0> are currently used. Each bit displays the status of the hard error lines—one each from I/O, memory, and up to four processors (A, B, C, or D).

When an internal hard error is detected by a processor, memory, or channel controller board, an error condition is asserted on the corresponding hard error line. The assertion of any of these error signals causes the Hard Error Interrupt to occur.

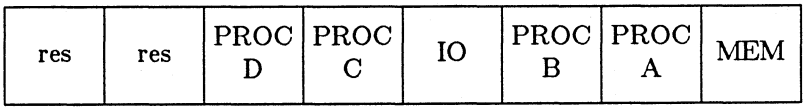
The condition of the hard error lines is monitored by the ESR on the SP2/SP4. The bits in this register are not latched. Once a hard error has been detected, the ESR can be read (by CONVEX SPU OS, diagnostics, error daemon, etc.) to determine which function asserted the error.

When a CPU board asserts its hard error line, it also stops all clocks in the processor so that the state of the processor when the error occurred is preserved for examination by diagnostics.

Detection of an internal error by a channel controller does not stop the system clock. The channel controller's internal processor does, however, save information about the error for diagnostics.

The ESR is located at address FFD411 and is formatted as shown in Figure 7–16:

Figure 7–16, Error Source Register



7.4.16 Soft Error Log

The Soft Error Log (SEL) is an 8-bit register, however, only six bits <5..0> are currently used. Each bit displays the status of the soft error lines—one each from I/O, memory, and up to four processors (A, B, C, or D).

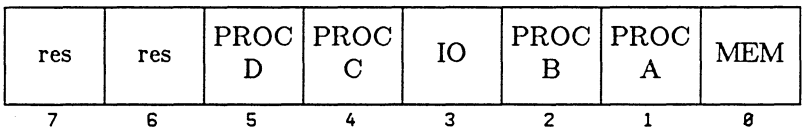
When an internal soft error is detected by a processor, memory, or channel controller board, an error condition is asserted on the corresponding soft error line. The assertion of any of these error signals causes the Soft Error Interrupt to occur.

The condition of the soft error lines is monitored by the SEL on the SP2/SP4. Once a soft error has been detected, the SEL can be read (by CONVEX SPU OS, diagnostics, error daemon, etc.) to determine which function asserted the error.

If a processor soft error is detected, the resulting SP2/SP4 action is dependent on SP2/SP4 software and which function pulled the error. For example, when a memory soft error is detected, the SP2/SP4 can scan the error log and then modify memory to correct the error.

The location of the SEL is at address FFD415 and is formatted as shown in Figure 7–17:

Figure 7–17, Soft Error Logs



7.4.17 SCM/ESM Interface

The SP2/SP4 includes a bi-directional interface to the System Control Module (SCM/ESM). This interface contains an 11-bit wide bus consisting of 8 data bits and 3 handshake bits, and is address/data multiplexed. The SP2/SP4 is always master of the bus.

The purpose of the interface is to provide flexible parallel data communication between the SP2/SP4 and SCM/ESM as opposed to discreet communication – dedicated lines such as power interrupt and ambient interrupt. The interface allows SP2/SP4 software (CONVEX SPU OS, diagnostics) to issue commands to the SCM/ESM and verify the status of those commands.

On the SCM/ESM side of the interface are 128 simulated registers. The SP2/SP4 can write data into these registers (issue commands) or read the contents of the registers (obtain the results of the command). For example, the SP2/SP4 can direct the SCM/ESM to measure some system parameter (fan performance, backplane voltage, etc.) and than read the results of the measurement through the interface.

The SCM/ESM interface controller on the SP2/SP4 is a state machine that operates the interface through sequencing the handshaking lines so that bi-directional communication between the SCM/ESM and the 68000 cpu on the SP2/SP4 runs smoothly.

7.4.18 SP2/SP4 Scan Interface

Scan rings and their interface on the SP2/SP4 provide a direct connection to the internals of the C200 Series. This allows the SP2/SP4 to discharge the following main functions:

- Loading initialization data into the processor before booting ConvexOS
- Examining the state of the system at the instant of death after a crash
- Executing C200 Series diagnostics

A scan ring is considered to be an interconnection of shiftable registers—a string of parallel/serial shift registers—connected via their serial pins. These registers are arranged in a circular configuration with the output of the last register tied back to the input of the first register, thus creating a ring.

Each C200 Series board has at least one scan ring (except the SP2/SP4 which has none), and some boards have two or three. The concept of multiple scan rings per function/board is new for the C200 Series system; the C100 Series system has one scan ring on every board (except the SP2/SP4). A much more complex system, the C200 Series system needs multiple scan rings to adequately represent its structure.

In addition, the C200 Series system employs numerous large gate arrays. Many of these contain a number of complete functions. It is therefore impossible to probe directly many of the elements of functions contained in these gate arrays with a logic analyzer or scope. Scan rings threaded through large gate arrays provide some visibility into the gate array internals, allowing diagnostics to be applied and evaluated.

Some rings in the C200 Series are extremely long—the C200 Series memory boards have scan rings over 3000 bits long (due to crossbar switches, ECCs, etc.). The C100 Series scan rings are 500 bits long or less.

Each scan ring has an interface on the SP2/SP4, through which it is controlled by SP2/SP4 software.

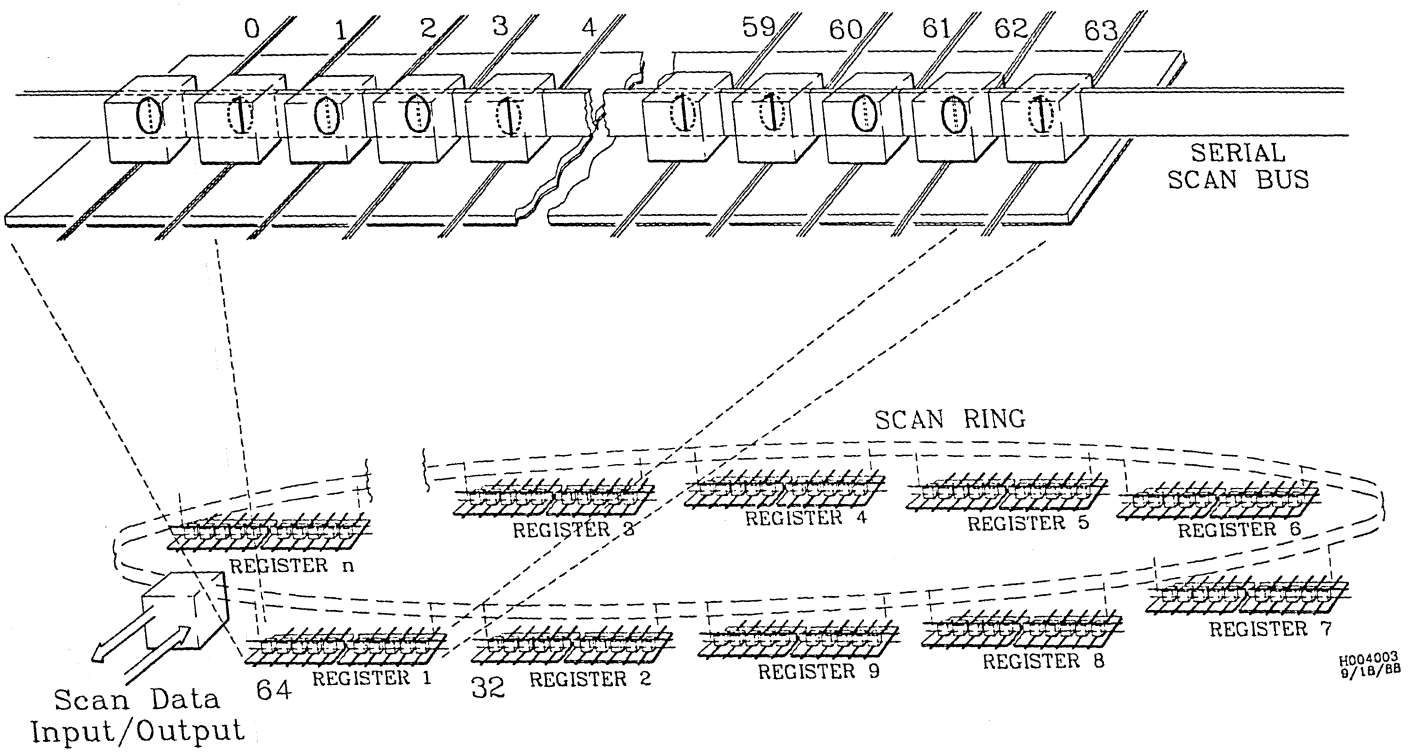
7.4.18.1 Scan Rings

There are registers (and register-like structures) in the system that normally operate in a parallel fashion but include a serial input and output connections. These registers also have the ability to shift data bits right or left on command (in the C200 Series, portions of some rings are unidirectional—they shift in the *right* direction only).

A number of these registers are connected together with the serial output of one register connected to the serial input of the next. Finally, the serial output of the last register in the line is connected back around to the serial input of the first register. The serial configuration of these registers is then circular, creating a scan ring.

Figure 7–18 shows the scan rings:

Figure 7–18, Scan Rings



Each C200 Series board has at least one scan ring; some boards have two or three:

- The PIA/PI2 has three rings
- The Vector Data board, CPX/CUE-CUO, and memory boards each have two rings
- The SP2/SP4 has no scan rings (none)
- All other boards have one ring each

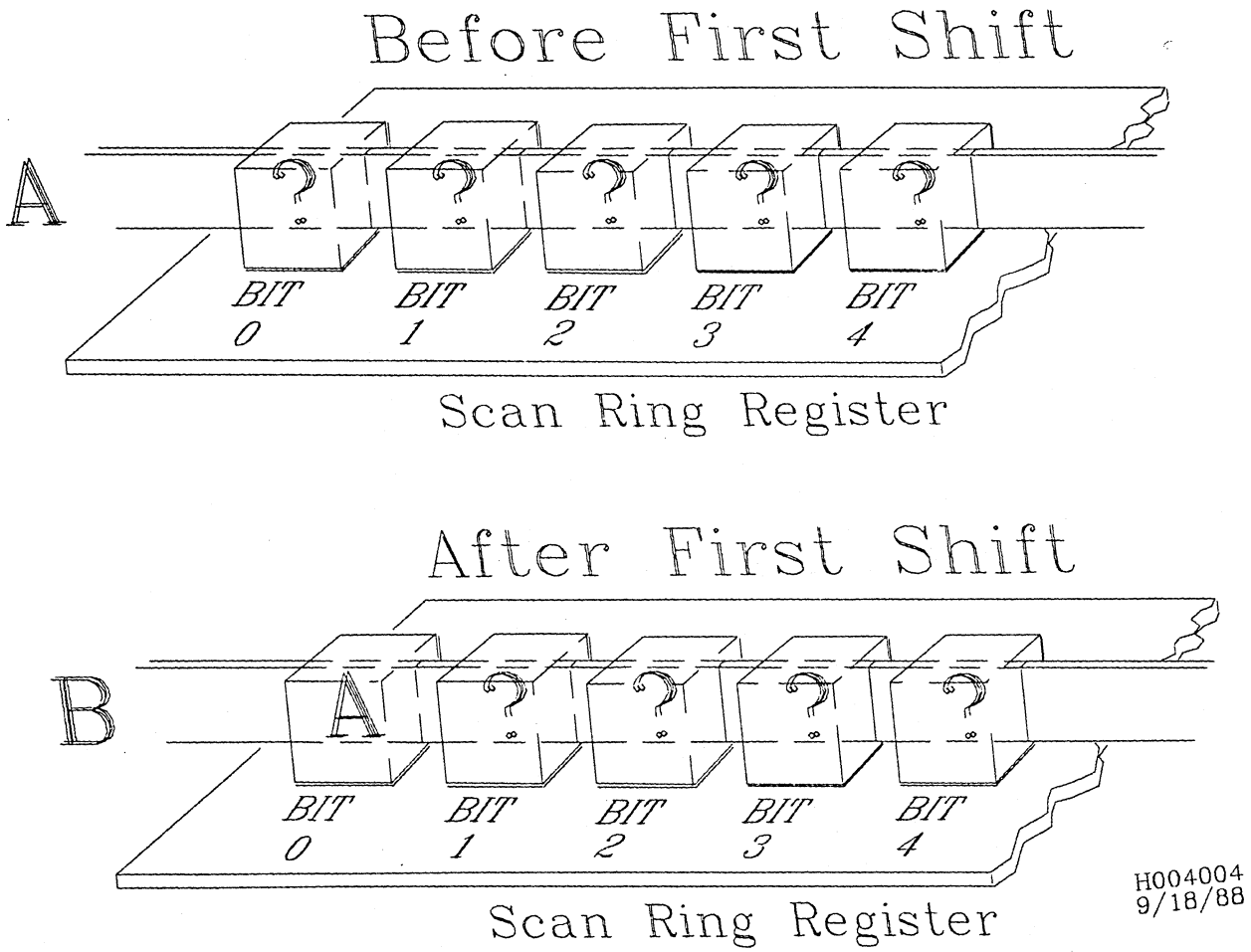
No scan rings cross board boundaries.

The SP2/SP4 controls the scan rings through its scan interface which contains registers, scan control logic, and clock control logic. The other boards in the system are connected by lines to, and controlled by, the scan interface registers during scan operations.

Loading Scan Rings

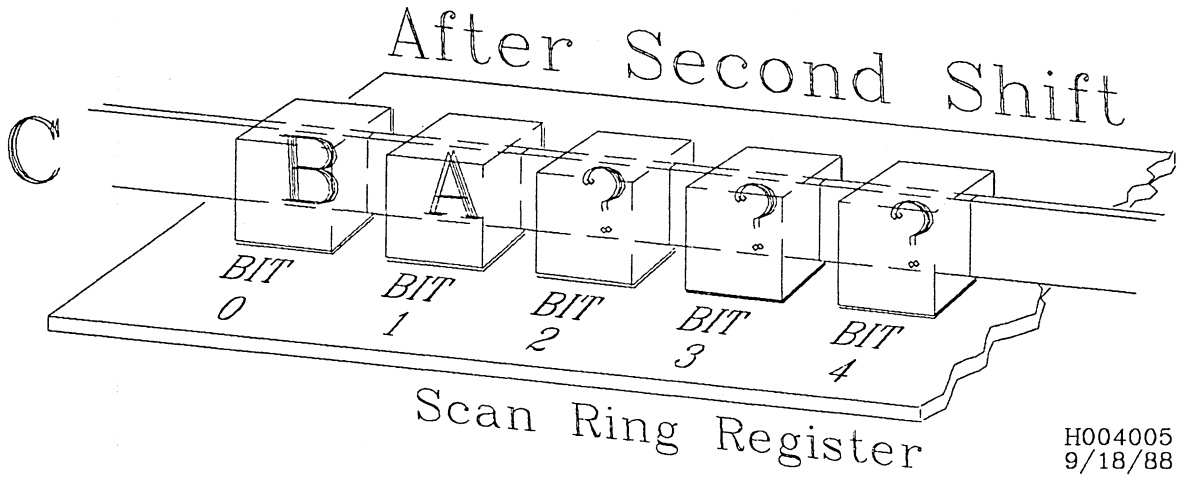
Scan data (initialization or diagnostic) is loaded into a ring sequentially, one bit at a time. The first bit is applied to the serial input of the first register and then the registers are shifted. This moves the first bit into the first bit position of the first register, as shown in Figure 7–19:

Figure 7–19, Loading Scan Ring Registers



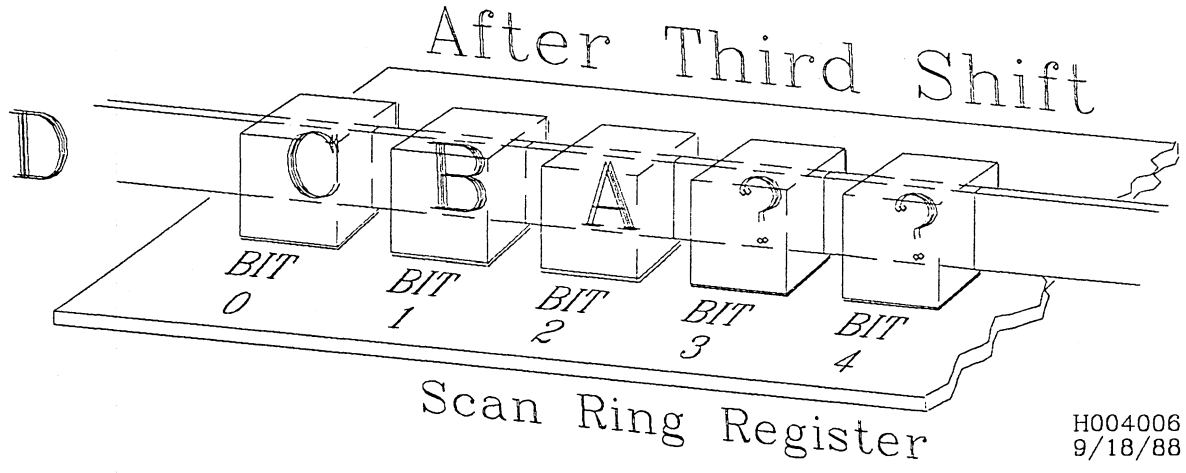
The next bit is then applied to the input of the first register and the registers shifted again (the same direction as before). This moves the first bit into the second position in the register and moves the second bit into the first register location, as shown in Figure 7–20:

Figure 7–20, After Second Shift



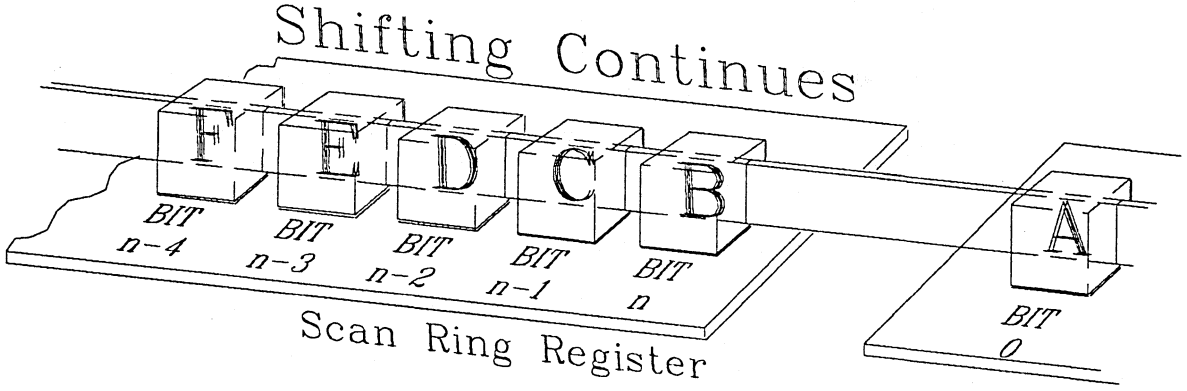
The next shift moves everything over once again—with the third bit being moved into the first register location, as shown in Figure 7–21:

Figure 7–21, After Third Shift



When the first bit emerges from the serial output of the first register, it goes into the serial input next register in the ring on the next shift, as shown in Figure 7–22:

Figure 7–22, Shifting Continues



H004007
9/18/88

This process is repeated until the entire scan ring is loaded with bits, each in their proper position.

In a similar fashion, one bit can be placed in any location in the scan ring by applying it to the input and issuing the required number of shifts to move it to the desired location in the ring.

If a particular scan register is 64-bits wide, it takes 65 shifts to move a bit through the register. A scan ring may include hundreds of registers, and therefore, thousands of bits.

Reading Scan Rings

Scan rings are read by shifting the registers in the ring until the desired bit or bits come around to be read.

Reading a scan ring is non-destructive so long as rings with unidirectional portions are not shifted in the *left* direction. The data remains in the ring after any number of right shifts during a read operation. Data can be restored to its original ring position by shifting the ring the appropriate number of times to move the data full circle—back to its original location in the ring.

7.4.18.2 Initialization and Diagnostics

The main functions of the C200 Series scan rings and the SP2/SP4 scan interface are to allow initialization of the system before booting, and to facilitate error analysis and diagnostics execution after errors are detected.

Initialization

The C200 Series processor must be in a particular state before it can be successfully booted (certain data must be in specific counters and registers before the system can begin operation).

Initialization includes setting system memory to a known condition and loading control stores, memory and I/O configuration information, and the program counter.

Initialization is performed by the SP2/SP4 under the direction of SP2/SP4 software through the SP2/SP4 scan interface. Data to be loaded into the system is applied to a scan ring and the ring shifted appropriately until the data is in the correct position.

This process is repeated for each scan ring requiring initialization data. Once all initialization data has been loaded, the system is ready for booting.

Error Analysis

When a hard error occurs in the system, the SP2/SP4 stops all clocks. This preserves the state of the system at the instant of death. Every internal state variable of the system (program counter, address register, scalar registers, vector registers, controllers, and so on) is retained for analysis.

Scan allows the SP2/SP4 software to examine these areas to see what was where at the moment of death. When a program dies during execution, this data helps determine whether the system actually crashed, what it was doing when it died, and where and why it died.

Diagnostics

Scan is also used to run special diagnostics on the system after errors have been detected. Diagnostic software running on the SP2/SP4 can load test data into the scan rings in a system function, issue a certain number of clocks, and check for known results in particular locations.

This process checks the entire intervening path—gates, translators, registers, etc.—resulting in basic data path and address path verification. This sort of test is generally done between two different rings on a single board, but, in many cases, can also be done within a single scan ring.

Another scan diagnostics activity loads a program, but stops before it begins executing. Then the entire system is single-stepped. After each step, data is scanned out and compared with known correct data. Any variation from expected values indicates a problem with the hardware associated with the register containing incorrect data. This is analogous to a built-in logic analyzer.

7.4.18.3 Scan Control

Scan operations are controlled through the SP2/SP4 scan interface. This interface consists of a number of registers and a micro-system to manage them. The SP2/SP4 scan interface is connected to the rest of the system through six lines. The lines and their functions are as follows:

- SCTL <0>—Controls scan mode (with bit 1, specify diagnostic operation)
- SCTL <1>—Controls scan mode (with bit 0, specify diagnostic operation)
- DMODE—Sets boards to diagnostic mode
- RUN—Issues clocks to scan rings
- ODENA—Enables scan output data onto the backplane for scan read/write operations
- SCNDAT—Scan data

These lines control all aspects of scanning rings. DMODE and SCTL control the operation of the main diagnostic scan chain. Mode changes are allowed only while clocks are stopped high. Selection is controlled by DMODE: whenever it is set to diagnostic mode, the main diagnostic scan chain is selected.

SCTL <1..0>

These two scan control lines from the SP2/SP4 control the operating mode of the scan ring registers during scan operations. The signals must both be zero if DMODE is not asserted. These lines are routed to boards in the system.

The two bits of this signal encode all four possible scan register states. In the diagnostic mode (DMODE line to that functional unit is active), these lines are used to set the scan registers to the proper scan function:

- Shift left
- Shift right
- Hold
- Parallel load

The registers in the scan ring then perform the operation specified by the SCTL lines when the next clock is received.

The SCTL lines control the operation of the scan ring registers as long as the ring is in the diagnostic mode. At all other times, the SCTL lines are ignored and the devices comprising the scan ring operate normally (count, compare, and so on).

DMODE

There is a separate DMODE line for each functional unit, and all boards in a particular functional unit are connect to the single DMODE line for the functional unit.

The SP2/SP4 asserts the DMODE line to a functional unit to put it into the diagnostic mode. Once a functional unit has its DMODE line activated, the boards in that functional unit understand that they are now in the diagnostic mode and do the following:

- Change data paths from normal to diagnostic configuration
- Change control paths from normal to diagnostic configuration
- Decode the scan control lines (SCTL) to determine the correct scan operation
- Perform the specified scan operation when the next clock is received (only the board which receives the clocks)

RUN

Each board in the system (except the SP2/SP4 and SCM/ESM) has a separate RUN line from the SP2/SP4. The status of the RUN line determines whether clocks from the PIA/PI2 are allowed onto the board—when a board's RUN line is asserted, clocks are gated onto the board.

When a board is in the diagnostic mode, clocks cause the registers in the scan ring to operate as specified by the SCTL lines. The SP2/SP4 can put the entire system, a complete functional unit, several boards, or a single board (scan ring) into the diagnostic mode, and then operate the affected scan ring(s) one at a time. The SP2/SP4 does this by asserting the DMODE lines to functional units and RUN lines to individual boards.

ODENA

Output Data Enable is used to multiplex the proper source for scan data and generate the proper ODENA to the desired functional unit.

To reduce I/O pin count at the SP2/SP4/System interface, ODENA, which causes a board to drive its scan data onto the scan bus, is issued on a functional unit basis, and only under scan read operations. If a functional unit board detects that the ODENA and DMODE lines are asserted, it should be prepared to drive scan data upon issuance of the RUN bit.

Associated with the ODENA generation, is the selection of the desired functional unit to be scanned. ODENA controls the multiplexer which selects the appropriate scan bus out of the functional units.

ODENA may be asserted to drive the output of the selected scan chain onto SCNDAT. SCNDAT is also the source of input scan data. Thus, when ODENA is active, any data shifted off the board is also shifted back into the chain. New data may be shifted into the scan chain only when ODENA is inactive.

SCNDAT

The scan chain uses SCNDAT to move data on and off the board. SCNDAT is always an input and sometimes an output for data in the MCM scan rings. On a scan write operation, the SP2/SP4 is the source of SCNDAT. On a scan read operation (ODENA asserted), the MCM is the source of SCNDAT. In either case, the data on SCNDAT is buffered and fed to the inputs of the MCM scan rings.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix A
Essential System Files

A.1 Overview

This appendix describes essential ConvexOS system files. These files require periodic maintenance and must be present for the ConvexOS operating system to function properly.

A.2 List of Files Sorted by Name

Most of the system files reside on the CPU disks. However, the */mnt/errlog* file resides on the SPU disk.

System files for optional CONVEX products (e.g., CONVEX Networking Utilities) are described in the documentation for those products.

For each file listed in this appendix, the following information is provided:

- File format
- File description and maintenance instructions
- Related programs
- Examples
- Caveats and bugs (if any)

The *termcap*(3X) manual (man) page contains a detailed explanation of termcap-format files. Use this manual page to assist in reading the descriptions of */etc/batchcap*, */etc/disktab*, */etc/gettytab*, */etc/printcap*, */usr/spool/queue/queuecap*, */etc/remote*, */etc/stripecap*, and */etc/tapecap*.

The files included in this appendix are listed in alphabetical order by filename (not by full pathname). They are as follows:

- /etc/disktab
- /etc/fstab
- /etc/gettytab
- /etc/group
- /etc/hosts
- /etc/nurc
- /etc/passwd
- /etc/printcap
- /etc/pwrestrict
- /etc/rc.local
- /etc/remote
- /etc/securetty (not until 8.0)
- /etc/stripecap
- /etc/tapecap (not until 8.0)
- /etc/ttys
- /etc/ttytype (not until 8.0)

The ConvexOS system files shown in the previous list, are described in detail on the following pages:

Name

/etc/disktab—disk description table

Format

termcap style

Description

Each *termcap*-style entry in */etc/disktab* describes the partition layout for one type of disk.

Capabilities

Refer to *termcap*(5) for a description of the file layout.

Name	Type	Description
ns	num	number of sectors per track
nt	num	number of tracks per cylinder
nc	num	total number of cylinders on one disk
rm	num	disk's speed (revolutions per minute)
p[*a-h*]	num	partition sizes (sectors)
b[*a-h*]	num	partition logical block sizes (bytes)
f[*a-h*]	num	partition fragment sizes (bytes)

Example

```
dkd-001|DKD-001|eagle|Eagle|Fujitsu Eagle (45 sectors):\
    :ty=winchester:ns#45:nt#20:nc#842:rm#3900\
    :pa#37800:ba#8192:fa#1024:\
    :pb#151200:pc#756000:\
    :pd#37800:bd#8192:fd#1024:\
    :pe#227700:be#4096:fe#512:\
    :pf#75600:bf#4096:ff#1024:\
    :pg#341100:bg#4096:fg#512:\
    :ph#225900:bh#4096:fh#1024:
```

This example specifies data for a disk with several names: *dkd-001*, *DKD-001*, *eagle*, *Eagle*, and *Fujitsu Eagle (45 sectors)*. The final name is rarely typed by hand and is included only as a long descriptor of the entry. This disk is a *winchester* type (the other type is *removable*). It has 45 sectors per track and 20 tracks per cylinder for each of 842 cylinders. The disk spins at 3900 rpm. The *a* partition is 37,800 sectors (19,353,600 bytes) in length. Its major blocks are 8,192 bytes long, and its minor blocks are 1,024 bytes long. The example also specifies 7 other partitions. The block sizes are not specified for either the *b* partition or the *c* partition.

Programs

The *newfs*(8) and *newst*(8) utilities determine physical characteristics for given disks by looking in */etc/disktab*.

Caveats

Choose the layout of file systems carefully to avoid problems of mounting file systems that share cylinders. In particular, note that the *c* partition is made up of the entire disk, and the *g* partition is made up of partitions *d*, *e*, and *f*.

Only change */etc/disktab* when adding a new style of disk to the system.

Name */etc/fstab*—static information about file systems

Format ASCII: %s %s %s %s %d %d \n

Description Each single-line entry in the ASCII */etc/fstab* file describes six fields:

1. The block (not raw) physical device described by the entry. For NFS partitions, this field is
   `machine`:*/full_path_name*

2. The logical name upon which the device is normally mounted

3. The type of partition entry for this device: "4.2" (local disk), "nfs" (network file system), "swap" (swap partition), and "ignore" (comment line).

4. The type of access for this device. Some access options are valid for both 4.2 and *nfs* file systems, while others are only valid for *nfs* file systems. The options valid on all file systems are listed below; the default options are indicated:
   - rw (read/write; default)
   - ro (read only)
   - suid (set uid execution allowed; default)
   - nosuid (set uid not allowed)
   - quota (usage limits enforced)
   - noquota (usage limists not enforced; default)
   - hide (ignore this entry during a *mount -a* command)

   The options available only on *nfs* systems are listed below. (See the *CONVEX Network File System System Manager's Guide* and *fstab*(5) for more details.)
   - bg (if first attempt fails, retry in the background)
   - fg (retry in foreground)
   - retry = *n* (set number of failure retries to *n*)
   - rsize = *n* (set read buffer size to *n* bytes)
   - wsize = *n* (set write buffer size to *n* bytes)
   - timeo = *n* (set *nfs* timeout to *n* tenths of a second)
   - retrans = *n* (set number of *nfs* retransmissions to *n*)
   - port = *n* (set server IP port number to *n*)
   - soft (return error if server doesn't respond)
   - hard (retry request until server responds)

5. An obsolete entry sometimes used to specify the dump frequency (in days)

6. The pass number for parallel dumps (and for running *fsck*(8))

The ASCII file contains an entry for each logical file system. Blanks or tabs separate the fields on each line. A "#" character at the beginning of a line indicates a comment. An entry should be added to this file each time disk partition is added to the system. No program currently writes to */etc/fstab*.

Example

```
/dev/da0d /mnt 4.2 rw 1 3
```

In this example, *mount -a* puts the */mnt* file system on partition */dev/da0d* for read and write access. No program currently examines the fourth field. The *preen*(8) disk-checking program ignores the final "3", although *fsck*(8) checks this file system concurrently with all other file systems with "3" as their pass number.

Programs

df(1), *fsck*(8), *preen*(8), *swapon*(8), *mount*(8), *umount*(8)

Caveats

Attempts to mount overlapping file systems result in quick chaos and probable loss of data on both partitions.

Access records from */etc/fstab* by using *getmntent*(3). The older *getfsent*(3X) may also be used, but it is discouraged.

Order records in */etc/fstab* so that *mount* and *umount* can process them sequentially. For example, if */master* and */master/slave* are partitions, the */master* entry must be included in *fstab* before the entry for */master/slave*.

Disk partitions specified as "swap", "ignore", and "nfs" do not have meaningful dump frequencies or pass numbers and are not checked by *fsck*(8). They are necessary, however.

Information on striped disk partitions can be included in */etc/fstab*. For more information on the use of striped disk partitions, see the chapter on device configuration in the *CONVEX System Manger's Guide* and *stripecap*(5).

See Also

/usr/include/fstab.h, *stripecap*(5), *fstab*(5)

Name

/etc/gettytab—terminal configuration file

Format

termcap style

Description

gettytab is a *termcap*-style file that describes terminal lines. The initial terminal login process *getty*(8) reads the *gettytab* file each time it starts, allowing dynamic reconfiguration of terminal characteristics. Each entry in the database describes one class of terminals.

Capabilities

Refer to *termcap*(5) for a description of the file layout.

The **Default** column lists defaults obtained if no entry is specified and the special default table has no entry.

Name	Type	Default	Description
ap	bool	false	terminal uses any parity
bd	num	0	backspace delay
bk	str	0377	alternate end of line character (input break)
cb	bool	false	use CRT backspace mode
cd	num	0	carriage-return delay
ce	bool	false	use CRT erase algorithm
ck	bool	false	use CRT kill algorithm
cl	str	NULL	screen clear sequence
co	bool	false	console—add \n after login prompt
ds	str	^Y	delayed suspend character
ec	bool	false	leave echo off
ep	bool	false	terminal uses even parity
er	str	^?	erase character
et	str	^D	end-of-file (EOF) character
ev	str	NULL	initial environment
f0	num	unused	tty mode flags to write messages
f1	num	unused	tty mode flags to read login name
f2	num	unused	tty mode flags to leave terminal as
fd	num	0	form-feed (vertical motion) delay
fl	str	^O	output flush character
hc	bool	false	do *not* hang up line on last close
he	str	NULL	host name editing string
hn	str	host name	host name
ht	bool	false	terminal has real tabs
ig	bool	false	ignore garbage characters in login name
im	str	NULL	initial (banner) message
in	str	^C	interrupt character
is	num	unused	input speed
kl	str	^U	kill character
lc	bool	false	terminal has lowercase
lm	str	login:	login prompt
ln	str	^V	"literal next" character

lo	str	*/bin/login*	program to execute when name obtained
nd	num	0	newline (line-feed) delay
nl	bool	false	terminal has (or might have) a newline character
nx	str	default	next table (for auto-speed selection)
op	bool	false	terminal uses odd parity
os	num	unused	output speed
pc	str	\0	pad character
pe	bool	false	use printer (hard copy) erase algorithm
pf	num	0	delay between first prompt and following flush (seconds)
ps	bool	false	line connected to a MICOM port selector
qu	str	^\	quit character
rp	str	^R	line retype character
rw	bool	false	do *not* use raw for input, use cbreak
sp	num	unused	line speed (input and output)
su	str	^Z	suspend character
tc	str	none	table continuation
to	num	0	time-out (seconds)
tt	str	NULL	terminal type (for environment)
ub	bool	false	do unbuffered output (of prompts, etc.)
uc	bool	false	terminal is known uppercase only
we	str	^W	word erase character
xc	bool	false	do *not* echo control characters as ^X
xf	str	^S	XOFF (stop output) character
xn	str	^Q	XON (start output) character

Example

```
default:\
     :ap:fd#1000:im=\r\n\r\nCONVEX UNIX, RELEASE V6.0 (%h)\r\n\r\
\r\n\r:sp#1200:2|std.9600|9600-baud:\
     :sp#9600:
```

where

ap terminal uses any parity (default entry)

fd form-feed delay is a full second

im initial banner message substitutes the name of the system for "%h"

sp default speed is 1200 baud

As this example shows, only a few of the available options are typically used.

The "2" entry in *sp* changes nothing but the terminal speed. Additional entries ("3", "4", etc.) also change the baud rate.

Caveats

If no line speed is specified, speed is not altered from the line speed in use when *getty* is entered. Specifying an input or output speed overrides line speed for the stated direction only. CONVEX does not support split baud rates.

Terminal modes are derived from the Boolean flags specified (*f0, f1, f2*). These modes can be used for message output, login name input, and to ensure that no one changes *tty* modes. If the derivation should prove inadequate, any (or all) of these three may be overridden with the *f0, f1,* or *f2* numeric specification. These specifications identify (usually in octal, with a leading "0") the exact values of the flags. Local (new *tty*) flags are set in the top 16 bits of this 32-bit value.

When it receives a null character, *getty* restarts, using the table pointed to by *nx*. (Null characters are presumed to show a line break.) If no *nx* entry exists, *getty* reuses its original table.

The *cl* screen clear string may be preceded with a (decimal) number indicating the number of milliseconds of delay required to clear the screen (*termcap* style). This delay can be simulated repeatedly using the pad character (*pc*).

To print the host name, include the character sequence *%h* in the initial message (*im*) and the login message (*lm*). (*%%* prints a single "%" character.) The host name is normally obtained from the system, but the *hn* table entry can be used to set it. In either case, the host name can be edited with *he*.

The *he* string is a sequence of characters—each character that is neither "@" nor "#" is copied into the final host name. A "@" in the *he* string causes one character from the real host name to be copied to the final host name. A "#" in the *he* string causes the next character of the real host name to be skipped. Surplus "@" and "#" characters are ignored.

When *getty* executes the login process given in the *lo* string (usually */bin/login*), it sets the environment to include the terminal type, as shown by the *tt* string (if it exists). The *ev* string can be used to enter additional data into the environment. It is a list of comma-separated strings, with each string having the form *name=value*.

If a nonzero time-out is specified with *to*, *getty* exits within the indicated number of seconds, either having received a login name and passed control to *login*, or having received an alarm signal and exited. This may be useful for hanging up dial-in lines.

Output from *getty* is even parity unless *op* is specified. Specify *op* with *ap* to allow any parity on input, but generate odd parity output. This only applies while *getty* is being run; terminal driver limitations prevent a more complete implementation. *getty* does not check parity of input characters in *RAW* mode.

Bugs

Some administrators change the default special characters, so it is wise to always specify (at least) the erase, kill, and interrupt characters in the *default* table. The characters "#" or "^H" in a login name are treated as erase characters; "@" is treated as a kill character.

The delay implementation is not flexible, and some of the delay algorithms are not used. The terminal driver should support useful delay settings.

Because *login*(1) resets the environment, there is no point setting it in *gettytab*.

Programs

getty(8)

See Also

termcap(5)

Name

/etc/group—group file

Format

%s:%s:%d:[^\n]\n

Description

This ASCII file contains the following information for each group:

- Group name
- Encrypted password (OBSOLETE)
- Numerical group ID
- A comma-separated list of users allowed in the group

Each line describes one group, and fields are separated by colons.

The file maps group names (the first field) to numerical group identification numbers (third field) and vice versa. Programs like *ls*| use this file.

Example

```
daemon:*:1:smith,jones
```

Group *daemon* (with "*" in the obsolete password field) is also known as group ID "1". Users *smith* and *jones* are in this group in addition to the group specified for them in */etc/passwd.*

Programs

Many programs use */etc/group.*

Caveats

Users become members of groups in two different ways:

1. By having the group number listed in the */etc/passwd* file
2. By having their user name listed in the */etc/group* file

See Also

/etc/passwd

bill(1), *setgroups*(2), *initgroups*(3X), *crypt*(3), *passwd*(1), *passwd*(5), *getgrent*(3), *getgrgid*(3), *getgrname*(3), *setgrent*(3), *endgrent*(3), *getgroups*(2)

Name

/etc/hosts—host name database

Format

%s [^\n]\n

Description

The *hosts* file contains information on the known hosts on the DARPA Internet. For each host, a single line containing the following information should be present

- Official host name
- Internet address (in 4-octet format)
- Aliases by which the host is also known

Blanks or tabs separate the items. A "#" shows the beginning of a comment; characters after "#" up to the end of the line are not interpreted by the routines that search the file. This file is normally created from the official host database maintained at the Network Information Control Center (NIC). Local changes may need to be made to add unofficial aliases or unknown hosts (for example, a local host name).

Network addresses are specified in the conventional "." notation using the *inet_addr*() routine from the Internet address manipulation library, *inet*(3N). Host names may contain any printable character except a field delimiter, newline, or comment character.

Example

The following example shows the local portion of the */etc/hosts* database. The lines following it in */etc/hosts* come from NIC distribution tables or from the CONVEX distribution.

```
# Berkeley Host Database
127.1  localhost
#  Convex 10Mb/s local Ethernet
100.0.1.0x82  convexs-ex    convexs sw
100.0.0.1 convexe-ex
100.0.0.6 convex1-ex    convex1
```

Lines beginning with "#" are comments. The *127.1 localhost* line defines a well-known host address (for this host) that is used for testing, loopbacks, and some daemons.

The local hosts (100.*) are members of the local Ethernet. The "100" prefix was arbitrarily chosen (choices should be in harmony with the existing host numbers, of course).

When an unknown name is requested, the *arp*(4) name-resolution protocol broadcasts packets requesting identification.

Programs

ftp(1), *rcp*(1), *rlogin*(1), *sendmail*(8)

Bugs

A name server should be used instead of a static file. A binary indexed file format should be available for fast access.

See Also

gethostent(3N), *arp*(4)

CONVEX Network File System System Manager's Guide

Name */etc/nurc—nu* defaults database

Format %s:%s

Description

The */etc/nurc* file contains default constants for the *nu*(8) utility. The *nu* utility adds new users to the system; *nu* uses the constants defined in */etc/nurc* if no other values are provided and if an alternate default file is not specified in place of */etc/nurc*.

Lines in the *nurc* file have the form
`tag: value`
where *tag* is the parameter to modify, and *value* is the default constant for the parameter. The tags and their default values are listed below.

Name	Default	Description
uid	(see below)	user ID
gid	EMPTY	group ID
group	staff	group name (used only if gid field not given)
directory	/mnt	path under which home directory will be placed
protection	0755	home directory protection
shell	/bin/csh	login shell
password	(see below)	new user's initial password
username	username	user's full name (for the *finger* command)
office	office	user's office (for the *finger* command)
extension	extension	user's work extension
homephone	homephone	user's home phone number
minwks	1	minimum # of weeks for password aging
maxwks	52	maximum # of weeks for password aging
diskquota	6000,8000,1200,1500	*blksoft, blkhard, inodesoft, inodehard* quotas
skeleton	/usr/skel	directory containing files to copy into home directory
homedir	*<directory>/<login>*	home directory to create for this user
typed	OFF	should this user have password typing restrictions?
aged	OFF	should this user have password aging restrictions?
quota	OFF	should home directory file system have quotas initialized?
nouidfile	OFF	ignore the contents of */etc/uidcount*?

For more information on these parameters, see *nu*(8).

Example

An *nurc* file might look like this:

```
directory: /mnt
protection: 0700
shell: /bin/csh
group: swtst
username: M. Atwood
typed
quota
diskquota: 3000, 4096, 1000, 1500
```

Here, a new user's directory is (by default) created in the */mnt* directory, and that directory is readable, writable, and executable by the user only (*protection: 0700*). The new user's default group is *swtst*. Typing restrictions are activated (although password aging restrictions are not), and the user's home directory file system has quotas initialized.

See Also

chfn(1), *finger*(1), *nurc*(5), *passwd*(5), *nu*(8)

Name

/etc/passwd—password file

Format

ASCII: %s:%s:%s:%s:%s:%s:%s\n

Description

Each single-line entry in the ASCII */etc/passwd* file maps a sign-on name to seven fields:

1. Name
2. Encrypted password
3. Numerical userd-identification number
4. Group identification number
5. Full user name, office, extension, and home phone number
6. User's home directory
7. User's default command interpreter

The ASCII file contains an entry for each user on the system. Colons separate the fields on each line. If the password field is empty, no password is required. A null command interpreter field defaults to the */bin/sh* interpreter.

Generally, the *nu*(1) program appends new users to this file. Several programs modify the user information when necessary

- *chsh*(1) changes the user's shell
- *chfn*(1) (change finger name) changes the user's identification field
- *passwd*(1) changes the user's password

Example

```
smith:2sadfNQDlxsMg:11:19:Rob &,22A,228,3823133:/mnt/smith:/bin/csh
```

In this example, Rob Smith has the sign-on name *smith* (first field). The second field is his encrypted password. His user identification number is 11; his group ID is 19. His full name is Rob Smith (the "**&**" causes the user name to be substituted appropriately); his office is 22A with office phone 228 and home phone of 382-3133. His home directory is */mnt/smith*, and he uses the standard C shell command interpreter.

Note that the fifth field (user identification) has four comma-separated subfields.

Programs

ls(1), *finger*(1), *passwd*(1), *nu(1)*, *chfn*(1), *chsh*(1), *vipw*(8), *login*(1)

Caveats

All system users are generally permitted read privileges to */etc/passwd* because the passwords themselves are encrypted. System managers usually grant write permission only to superuser. Because several programs

use the */etc/passwd* file to map user IDs to user names, read permission is important to the system's general well-being.

While the password file can be edited with a text editor, it is important to ensure that only one person at a time edits the file. The editing programs *chfn*(1), *chsh*(1), *passwd*(1), and *vipw*(8) are preferred because they perform the required file locking. Be aware that writing out an ill-formatted *passwd* file may cause severe problems with programs that use the ill-formatted lines and may introduce security holes.

See Also

/etc/group

bill(1), *crypt*(3), *group*(5), *getpwent*(3), *getpwuid*(3), *getpwnam*(3), *setpwent*(3), *endpwent*(3)

Name

/etc/printcap—printer capability database

Format

termcap style

Description

printcap is a *termcap*-style file that describes a system's set of line printers. Because the spooling system reads the *printcap* file every time a file is to be printed, printers can be added and deleted dynamically. Each entry in the file describes one printer.

Refer to the "4.2BSD Line Printer Spooler Manual" in the *CONVEX UNIX Tutorial Papers* for instructions on setting up an entry for a given printer.

Capabilities

Refer to *termcap*(5) for a description of the file layout.

Name	Type	Default	Description
af	str	NULL	name of accounting file
br	num	none	if *lp* is a tty, set the baud rate (*ioctl* call)
cf	str	NULL	cifplot data filter
df	str	NULL	TEX data filter (DVI format)
fc	num	0	if *lp* is a tty, clear flag bits (*sgtty.h*)
ff	str	\f	string to send for a form feed
fo	bool	false	print a form feed when device is opened
fs	num	0	like *fc* but set flag bits
gf	str	NULL	graph data filter (*plot*(3X) format)
ic	bool	false	driver supports (nonstandard) *ioctl* to indent printout
if	str	NULL	name of text filter that does accounting
lf	str	/dev/console	error logging filename
lo	str	lock	name of lock file
lp	str	fl/dev/lp	device name to open for output
mx	num	1000	maximum file size (in BUFSIZ blocks), zero = unlimited
nd	str	NULL	next directory for list of queues (unimplemented)
nf	str	NULL	*ditroff* data filter (device independent *troff*)
of	str	NULL	name of output filtering program
pl	num	66	page length (in lines)
pu	bool	false	propogate user information to filters (*lpd-acct*(5))
pw	num	132	page width (in characters)
px	num	0	page width in pixels (horizontal)
py	num	0	page length in pixels (vertical)
rf	str	NULL	filter for printing FORTRAN style text files
rm	str	NULL	machine name for remote printer
rp	str	lp	remote printer name argument

rs	bool	false	restrict remote users to those with local accounts
rw	bool	false	open the printer device for reading and writing
sb	bool	false	short banner (one line only)
sc	bool	false	suppress multiple copies
sd	str	/usr/spool/lpd	spool directory
sf	bool	false	suppress form feeds
sh	bool	false	suppress printing of burst page header
st	str	status	status filename
tf	str	NULL	*troff* data filter (cat phototypesetter)
tr	str	NULL	trailer string to print when queue empties
vf	str	NULL	raster image filter
xc	num	0	if *lp* is a *tty*, clear local mode bits (*tty*(4))
xs	num	0	like "xc" but set bits

The *CONVEX UNIX Programmer's Manual* describes these capabilities.

Example

```
fast|Fast Imagen serial printer:\
    :af=/usr/adm/imagen:\:br#19200:\
    :df=/usr/local/lib/idvi:fc#0177777:ff=\00:fo=0:\
    :fs#040:gf=/usr/local/lib/igraph:if=/usr/local/lib/ipf:\
    :lf=/usr/spool/nipd/ilog:lo=lock:lp=/dev/imagen2:\
    :mx=#10000:nf=/usr/local/lib/idimp:pl#60:pw#80:\
    :px#2016:py#2624:rf=/usr/local/lib/ifort:rw:\
    :sb=CONVEX Computer Corporation:sc:sd=/usr/spool/nipd2:\
    :sh:st=/usr/spool/nipd2/pstatus:vf=/usr/local/lib/imp:
```

where

fast Printer name stands for "fast Imagen printer."

af Accounting information goes to the */usr/adm/imagen* file.

br Baud rate is 19200.

df Text data filter */usr/local/lib/idvi* preprocesses the data before it is printed when the *lpr* command is used with the TEX option.

fc All *sgtty.h* style bits are cleared on start-up.

ff Form-feed strings are nulls.

fo No form feed is sent when the device is opened.

fs Flag bits 040 (octal) are set on start-up.

gf Graphics filter */usr/local/lib/igraph* preprocesses the data before it is printed when the *plot*(3X) command is used.

if Accounting data is processed by filter */usr/local/lib/ipf.*

lf Errors are logged to the */usr/spool/nipd/ilog* file.

lo Lock file is *lock* (the default).

lp Printer name is */dev/imagen2.*

mx	Maximum file size is 10,000 blocks.
nf	*ditroff* output goes through the */usr/local/lib/idimp* filter.
pl	Page length is 60 lines.
pw	Page width (line length) is 80 characters.
px	Page width (horizontal dimension) is 2,016 pixels.
py	Page height (vertical dimension) is 2,624 pixels.
rf	FORTRAN listings go through the */usr/local/lib/ifort* filter.
rw	Line is opened in read/write mode.
sb	Short banner is "CONVEX Computer Corporation."
sc	Multiple copies are suppressed.
sd	Spool directory is */usr/spool/nipd2*.
sh	Burst page headers are suppressed.
st	Status messages appear in */usr/spool/nipd2/pstatus*.
vf	Raster images go through the */usr/local/lib/imp* filter.

Programs

lpc(8), *lpd*(8), *pac*(8), *lpr*(1), *lpq*(1), *lprm*(1)

Caveats

Be careful to specify "#" or "=" appropriately, depending on whether a parameter has a numerical or string value. Although the above example does work, it contains some counterintuitive assignment characters.

See Also

lpd-acct(5), *termcap*(5), *lpd*(8)

"4.2BSD Line Printer Spooler Manual" in the *CONVEX UNIX Tutorial Papers*

Name

/etc/pwrestrict—password restrictions file

Format

ASCII: %s:%s:%s:%s:%s\n

Description

Each single-line entry in */etc/pwrestrict* maps a sign-on name to five fields:

1. Name
2. Encrypted password
3. Password aging information
4. Password type flag (Y or N)
5. Numerical user identification number

pwrestrict contains an entry for each user on the system. Colons separate the fields on each line. If the aging information field is empty, no password aging is enforced by the system.

Generally, the *nu*(l) program appends new users to this file. *passwd*(1) modifies the user's password and the password aging information.

Example

```
smith:ENEaEinahyIAo:1,2,Oct 13 1986:Y:469
```

In this example, the sign-on name is *smith* (first field). The second field is the encrypted password. The password age field denotes that the password was last changed on Oct 13 1986; the user cannot change the password for one week (from the given date) and will be forced to change the password after Oct 27 1986 (2 weeks after the last-changed date). The fourth field is a flag (Y or N); if the flag is "N", the password may be composed of any sequence of characters. f the flag is "Y", the user's password must follow the guidelines for passwords described in *passwd*(1). The user identification for *smith* is 469 (the last field).

Programs

passwd(1), *pwage*(1), *nu(8)*, *vipw*(8), *login*(1)

Caveats

All system users are generally permitted read privileges to */etc/pwrestrict* because the passwords themselves are encrypted. System managers usually grant write permission only to superuser. Because several programs use the */etc/pwrestrict* file to map user IDs to user names, read permission is important to the system's general well-being.

Although the *pwrestrict* file can be edited with a text editor, only one person should edit the file at a time. The editing programs *passwd*(1), *vipw*(8), and *nu*(8) are preferred because they perform the required file locking. Writing an ill-formatted *pwrestrict* file will probably cause severe problems with programs that use the ill-formatted lines. If inconsistencies are found between the */etc/passwd* file and the */etc/pwrestrict*

file, information from the password file is used; the restriction information is ignored.

See Also

/etc/passwd, */etc/group*

crypt(3), *group*(5), *getpwrestent*(3), *getpwrestuid*(3), *getpwrestnam*(3), *setpwrestent*(3), *endpwrestent*(3)

Name

/etc/rc.local—system-specific start-up information

Format

Shell script

Description

The system start-up routine processes */etc/rc.local* after it has mounted all the file systems but before it has started the daemons. The *rc.local* file contains the shell commands used to

- Set the host name
- Set up networks
- Check quotas
- Save core dumps
- Start local daemons
- Remove temporary files after crashes

Example

```
/bin/hostname convex
/etc/ifconfig ex0 convex-ex up arp -trailers    >/dev/console
echo -n 'check quotas: '                        >/dev/console
        /usr/etc/quotacheck -a
echo 'done.'                                    >/dev/console
/usr/etc/quotaon -a
echo -n 'local daemons:'                        >/dev/console
if [ -f /etc/routed ]; then
        /etc/routed & echo -n ' routed'         >/dev/console
fi
if [ -f /etc/rwhod ]; then
        /etc/rwhod & echo -n ' rwhod'           >/dev/console
fi
if [ -f /usr/local/bin/stat ]; then
        /usr/local/bin/stat&
        echo -n ' statistics'                   >/dev/console
fi
if [ -f /etc/restart ]; then
        /etc/restart s 1 v &
        echo -n ' batch'                        >/dev/console
fi
echo '.'                                        >/dev/console
rm -f /tmp/Emacs* /tmp/queue?/*
rm -f/usr/spool/notes/.locks/*
/usr/convex/spu -r /mnt/os/vmunix > /vmunix
/usr/convex/spu -r /mnt/errlog > /errlog.back
```

See Also

Descriptions of the daemons, editor temporary files, and administrative support programs.

Name

/etc/remote—remote host description file

Format

termcap style

Description

The systems known by *tip*(1C), and their attributes, are stored in the *termcap*-style file */etc/remote*. Each line in the file describes a single remote host.

The first entry is the name of the host system; vertical bars separate multiple names for a single system.

The *tip* program and the *cu* interface to *tip* use entries named "tip*" and "cu*" for defaults as follows. When *tip* is invoked with only a phone number, it looks for an entry of the form "tip300", where "300" is the baud rate with which the connection is to be made. When the *cu* interface is used, it looks for entries of the form "cu300".

Capabilities

Capabilities are either strings (str), numbers (num), or Boolean flags (bool). A string capability is specified by *capability=value*, for example, "dv=/dev/harris". A numeric capability is specified by *capability#value*, for example, "xa#99". A Boolean capability is specified (as true) by listing the capability.

Refer to *termcap*(5) for a description of the file layout.

Name	Type	Default	Description
at	str		Auto call unit type.
br	num	300	The decimal baud rate to be used.
cm	str		Initial connection message to be sent to the remote host.
cu	str	(dv)	Call unit if making a phone call.
di	str		Disconnect message sent to the host when requested by user.
du	bool		This host is on a dial-up line.
dv	str		UNIX device(s) to open to establish a connection.
el	str	NULL	Characters marking an end-of-line.
fs	str	BUFSIZ	Frame size for transfers.
hd	bool		The host uses half-duplex communication (do local echo).
ie	str	NULL	Input end-of-file marks.
oe	str	NULL	Output end-of-file string (sent at EOF after transfers).
pa	str	even	The type of parity to use when sending data to the host.
pn	str		Telephone number(s) for this host.
tc	(str)		Continue this capability at *string* in the named description.

Example

```
UNIX-1200:\
        :dv=/dev/cau0:el=^D^U^C^S^Q^O@:du:at=ventel:ie=#$%:oe=^D:\
        :br#1200:
arpavax|ax:\
        :pn=7654321%:tc=UNIX-1200:
```

The *UNIX-1200* device (*dv*) uses */dev/cua0*, which connects to a ventel modem (*at*) on a dial-up (*du*) line. Any of the entire line (*el*) list characters (^D, ^U, ^C, ^S, ^Q, ^O) cause the entire line and the character to be sent to the remote host. The connect program sends an output end-of-file (*oe*) string ^D after transferring a file. The characters "#", "$", and "%" signify end-of-input-file when preceded by "\n" (they're the shell prompts). The *arpavax* name specifies a phone number (*pn*) to dial and then the standard *UNIX-1200* characteristics.

Programs

tip(1C)

Caveats

If *dv* is used to refer to a terminal line, *tip* tries to open it for exclusive use.

The "~" escapes are only recognized by *tip* after an *el* character or after a carriage-return.

Parity may be set to "even", "odd", "none", "zero" (always set bit 8 to zero), or "one" (always set bit 8 to 1).

If the phone number (*pn*) field contains an "@" sign, *tip* searches the */etc/phones* file for a list of telephone numbers.

Bugs

Some files may not work using the *ie* facility for specifying input EOF.

See Also

phones(5)

Name

/etc/securetty—list of teletypes on which root can log in

Format

%s\n

Description

The */etc/securetty* file contains a list of teletypes on which root can log in. If the file does not exist, then root can log in anywhere. If the file does exist, then root can only log in on a listed teletype.

The teletype names are listed one per line.

Example

```
/dev/console
tty00
```

In this example, the root user can log in only on the console and tty00.

Programs

login(3N)

Caveats

Deleting */etc/securetty* enables root to log in anywhere.

Name

/etc/stripecap—striped disk partition description database

Format

termcap style

Description

The */etc/stripecap* file is a database that describes striped disk partitions. It is used by *putst*(8) and *newst*(8).

Striped disk partitions are logical disk partitions that span several physical disk partitions. Striped disk partitions exploit performance improvements made possible by the parallel operation of the several disk arms. UNIX file systems can be mounted on striped disk partitions just as with conventional disk partitions.

Striped partitions are described in */etc/stripecap* by a *termcap*-style descriptor. This descriptor contains information both on the physical disk partitions that constitute the striped partition and on the layout of the logical sections that make up the stripes.

The *newst*(8) and *putst*(8) utilities assume that each stripe partition has one name of the form *stX*, where *X* is a numerical digit corresponding to the minor device number of the stripe disk pseudo device */dev/rstX*.

Capabilities

Refer to *termcap*(5) for a description of the file layout.

Name	Type	Description
np	num	number of partitions constituting the stripe file system.
M?	num	major device number of partition ?.
m?	num	minor device number of partition ?.
D?	num	number of devices (partitions) in section ?.
B?	num	"stripe blocksize" (interleave factor) of section ?.
S?	num	number of blocks in section ? contributed by each partition.

Example

The following command sequence updates */etc/stripecap*:

```
/etc/newst /dev/rst6 da4a dkd-001 da4h dkd-001 da4g dkd-001
```

The output is a *stripecap* entry similar to

```
st6:\                                   Name of this stripe device.
    :np#3:\                             Number of partitions included (three).
    :M0#5:m0#38:\                       Device numbers of three partitions
    :M1#5:m1#39:\                           sorted from largest size to
    :M2#5:m2#32:\                               smallest size.
    :D0#3:B0#37800:S0#128:\             First section uses all three partitions.
    :D1#2:B1#188100:S1#128:\            Second section uses first two partitions.
    :D2#1:B2#115200:S2#128:\            Third section uses first partition.
```

Entries may be continued on multiple lines if "\" is specified as the last character of a line. Empty fields (adjacent colons) may be included for readability (for example, between the last field on a line and the first field on the next).

All fields have names that are two-character codes. For most of the field names, the second character is uniquely chosen from the set [0-9a-zA-Z] in an ascending sequence. For example, the minor device numbers of the first five disk partitions are *m0*, *m1*, *m2*, *m3*, and *m4*.

The name of this stripe device is *st6*, which means that */dev/st6* is the name on which to mount the file system. It includes three partitions on device number pairs 5,38 and 5,39 and 5,32. The order of these is important: 5,38 must be the largest partition; 5,32 must be the smallest. The stripe system interleaves all three partitions until the third one's space is exhausted. It then uses the first two partitions, and finally only the first partition (assuming unequal sizes of partitions, which is not necessarily the case).

In this example, the first section uses parts of all three partitions (for the first 37,800 sectors). The *S?* parameter directs the striping system to take the first 128 sectors from 5,38; the next 128 sectors from 5,39; then the next 128 sectors from 5,32. The fourth set of 128 sectors comes from 5,38 (again), and the cycle continues until 37,800 sectors have been used from each partition.

The second section repeats the sequence by using 128-sector chunks from only the first and second partitions. The third section uses only the remaining space on the first partition.

Programs

newst(8), *putst*(8), *st*(4)

Name

/etc/tapecap—tape device capability database

Format

termcap style

Description

tapecap is a *termcap*-style file that describes the characteristics of a system's tape devices. Each entry in the file describes one tape device.

Capabilities

Refer to *termcap*(5) for a description of the file layout.

Name	Type	Default	Description
an	str	system dependent	allocation name
bl	bool	false	block mode tape drive
de	num	1600	density of tape unit in bpi
re	bool	false	rewind upon close
sp	num	tape unit dependent	tape unit speed in ips
to	num	60	time-out value in minutes for tape deallocation

Example

```
/dev/rmt8|unit0:\
    :an=/usr/spool/tape/alloc/unit0:de#1600:re:to#60:
/dev/rmt9|unit1:\
    :an=/usr/spool/tape/alloc/unit1:de#1600:re:to#60:
/dev/rmt12:\
    :an=/usr/spool/tape/alloc/unit0:de#1600:to#60:
/dev/rmt13:\
    :an=/usr/spool/tape/alloc/unit1:de#1600:to#60:
/dev/rmt16:\
    :an=/usr/spool/tape/alloc/unit0:de#6250:re:to#60:
/dev/rmt17:\
    :an=/usr/spool/tape/alloc/unit1:de#6250:re:to#60:
/dev/rmt20:\
    :an=/usr/spool/tape/alloc/unit0:de#6250:to#60:
/dev/rmt21:\
    :an=/usr/spool/tape/alloc/unit1:de#6250:to#60:
```

In this example, the system has two tape units: *unit0* and *unit1*. */dev/rmt8*, */dev/rmt12*, */dev/rmt16*, and */dev/rmt20* all refer to different characteristics of the same physical tape unit. The allocation file (*af*) that stores information for these tape devices is */usr/spool/tape/alloc/unit0*. Devices (*de*) */dev/rmt8* and */dev/rmt12* are 1600-bpi tape devices, whereas */dev/rmt16* and */dev/rmt20* are 6250-bpi tape devices. Devices */dev/rmt8* and */dev/rmt16* are tape devices that rewind (*re*) on close. Devices */dev/rmt12* and */dev/rmt20* do not rewind on close. All devices time out (*to*) and are automatically deallocated by *tpd*(8) when they have been idle for 60 minutes.

Programs

tpalloc(1), *tpc*(8), *tpd*(8), *tpdealloc*(1), *tpmnt*(1), *tpq*(1), *tprm*(1), *tpumnt*(1)

Caveats

Be careful to specify "#" or "=" appropriately, depending on whether a parameter has a numerical or string value. Although the above example does work, it contains some counterintuitive assignment characters.

See Also

termcap(5)

Name

/etc/ttys—terminal initialization data

Format

%s [^\n]\n

Description

The *init*(8) program reads the *ttys* file to learn which terminal special files to create getty processes for. (These processes enable users to login.) There is one line in the *ttys* file per terminal special file.

The first character of a line in the *ttys* file is either "0" or "1". If the first character is "0", *init* ignores that line (i.e., no one can log in on that line). If the first character is "1", *init* creates a getty process for that line.

The *init* program uses the second character on each line as an argument to *getty*(8) (which performs such tasks as baud-rate recognition, reading the login name, and calling *login*). See */etc/gettytab* and the */etc/gettytab* entry in this appendix for the meaning of each of these characters. They specify the hard-wired baud rate of the terminal (or the sequence of baud rates to use if more than one rate is possible). The remainder of the line is the terminal's entry in the device directory, */dev*.

Example

```
14console
12tty00
02tty02
02ptypf
```

In this example, *init* sets up *login* processes for the console and tty00. The tty02 teletype and ttypf pseudo-teletype are currently disabled—they appear for the benefit of *getlogin*(3).

Programs

init(8)

Caveats

getlogin(3) and the file */etc/utmp* rely on the order of */etc/ttys*. Changing the order (even inserting new entries in the middle of the file) invalidates some file entries (for example, */etc/utmp*), causing *getlogin* to return erroneous results. Adding entries to the end of the file does not cause these problems.

If the UNIX system is running multi-user mode when */etc/ttys* is changed (for example, turn a teletype on or off or change a baud rate), the "kill -1 1" signal must be used to inform *init*.

CAUTION

Do not accidentally type "kill 1 1" because this signal reboots the system in single-user mode.

See Also

/etc/utmp

gettytab(5), *getty*(8), *login*(1), *on*(1), *off*(1), *getlogin*(3), *init*(8)

Name

/etc/ttytype—database of terminal types by port

Format

%s %s\n

Description

The */etc/ttytype* file is a database that contains a database definition for each *tty* port on the system. There is one line per port. Each line contains the terminal type (as a name listed in *termcap*(5)), a space, and the name of the *tty* (with */dev/*). The *tset* and *login* programs read this information in order to initialize the *TERM* environment variable at login time.

Example

```
vt100n tty00
network ptypc
```

In this example, *tty00* is a vt100 (with no initialization), and *ptypc* (like all pseudo teletypes) is a "network-style" teletype.

Programs

tset(1), *login*(1)

Caveats

This file may not be useful if a MICOM (or other) port selector that connects random terminals to random teletype lines is used. Likewise, defining terminal types by port for dial-up lines may be difficult. In these cases, the user may specify the *TERM* environment variable by executing the command

% **set term** = *type*

where *type* is the terminal type as specified in */etc/ttytype* (that is, vt100n).

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix B
C200 Vector Instruction Operation

B.1 Vector Instruction Overview

The vector processor is dispatched to process all instructions which operate on user visible registers contained in the vector processor. The user visible registers include:

- Vector registers V0 thru V7
- Vector Merge (VM) register
- Vector Length (VL) register

Although the Vector Stride (VS) register is used for vector load and store operations, it is not physically located on either of the vector processor boards. The vector stride register is located in the scalar processor section of logic which generates memory addresses. The vector processor is not dispatched when the vector stride register is accessed.

All instructions dispatched to the vector processor are staged by the vector dispatch logic (VDL). The VDL holds the instruction until all the hardware resources required by the instruction are available.

The checks include:

- Micro controllers are not busy
- The source and destination registers do not have hazards
- A scalar for the instruction has been received (scalar vector instructions)
- The output staging logic is not busy (reductions and stores)

When all checks pass, the instruction is dispatched to one of the three micro controllers (ALU, multiply, or load/store).

There are many different vector instructions which are executed in the vector processor. These instructions can be divided into logical groups (similar execution characteristics). The vector instruction groups and vector processor manipulation of the user visible registers are described in the following text.

B.1.1 Load Vector Register

Instructions in group:

ld.(b|h|w|l|s|d) *<effa>, Vk*
ld.(b|h|w|l|s|d).(t|f) *<effa>, Vk*

These instructions load the vector registers with data from memory. The load/store micro controller (UL) is dispatched to execute the instruction once all hazards are clear. The load/store micro controller starts the load back door controller (LBD) when it is idle.

The LBD controls the scalar processor interface handshakes, the input staging gate arrays, and the write to the vector register file gate arrays.

The Input Staging Gate Arrays (IS GAs) receive the transfers from memory and stage it for the the Vector Register File Gate Arrays (VRF GAs). The even elements of a vector go to the even VRF GAs and the odd elements go to the odd VRF GAs.

The load instructions which operate under mask pass a vector merge register bit (VM bit) per vector element to the scalar processor. The scalar processor requests data from memory for the vector elements which have a corresponding VM bit set. All VM bits are saved in the memory read return queue on the scalar processor. All VM bits are read out of the memory read return queue and returned to the vector processor. If the VM bit from the queue is set, then the corresponding read data from memory is returned with the VM bit. The vector processor only writes data which has the VM bit set to the vector register file.

The load instructions which do not operate under mask do not send a VM bit to the scalar processor. However, the scalar processor forces the VM bit which is returned with each data element to be set. This allows masked and non-masked loads to be treated the same by the load back door controller. The load back door controller forces a zero VM bit at the end of an odd vector length load to inhibit writing an extra vector register file element.

B.1.2 Store Vector Register

Instructions in group:

st.(b|h|w|l|s|d) *Vk, <effa>*
st.(b|h|w|l|s|d).(t|f) *Vk, <effa>*

The load/store micro controller is started to execute the operation when all required resources are available. These include access to vector register Vk, an ALU or multiply reduction instruction is not executing, and the load/store micro controller (UL) is not busy. The UL controller maintains a VL counter which indicates when enough vector elements have been stored.

The UL controller transfers VL vector elements and VL VM bits. The VM bits inform the scalar processor whether the corresponding vector element should be masked (not stored). For those operations which are not under mask, the VM bits are forced to one so that all elements are stored to memory.

The UL controller starts the output staging controller (OS) and VM bit output staging controller (VMO) each UL cycle to transfer two vector elements/VM bits. If the vector length is odd the the last OS and VMO start will transfer only one element/VM bit.

The vector elements are read from the vector register file gate arrays (VRF GAs) under control of the UL controller. The OS controller handles the scalar handshakes for the transfer and controls the output staging gate arrays (OS GAs). The VMO controller controls the VM bit transfer handshake signals.

B.1.3 Load Vector Register/Vector Index

Instructions in group:

ldvi.(b|h|w|l|s|d)	*Vj, Vk*
ldvi.(b|h|w|l|s|d).(t|f)	*Vj, Vk*

The UL micro controller is started to execute the operation. As viewed from the vector processor, the instruction is the combination of a vector load and a vector store executing concurrently.

The instruction must transfer a vector register to the scalar processor to be used in the address generation of a vector to be loaded. VM bits are transferred to the scalar processor with the vector of indexes.

The UL controller controls the vector read from the VRF GAs and the VM bit read from the vector merge gate arrays (VM GAs). The OS controller handles the transfer of the vector to the scalar processor, the VMO controller transfers the VM bits, and the LBD controller handles the transfer of the vector to be loaded and writes the vector into the VRF GAs. If the instruction is not executing under mask then the VM bits are forced to one.

B.1.4 Store using Vector Index

Instructions in group:

stvi.(b|h|w|l|s|d)	*Vk, Vj*
stvi.(b|h|w|l|s|d).(t|f)	*Vk, Vj*
stvi.(b|h|w|l|s|d)	*Sk, Vj*
stvi.(b|h|w|l|s|d).(t|f)	*Sk, Vj*

The UL micro controller is started after all required resources are available. The UL controller starts the OS controller to control the transfer of data to the scalar processor and the VMO controller to control the transfer of VM bits. The VM bits are forces to one for non under mask operations.

For operand sizes less than or equal to 32 bits, the instruction is executed similar to a vector store. The vector register data (byte, halfword, or word) is transferred on the least significant 32 bits of the bus while the corresponding vector index is transferred on the most significant 32 bits. The UL controller generates the control signals which inform the VRF GAs to merge the vector data and vector indexes on the bus.

For long word vector stores two vector indexes are transferred on one UL controller cycle and two 64 bits vector data elements are transferred on the next cycle. This is one of the few instructions which is not able to proceed at a two element (even and odd) per cycle rate.

B.1.5 Store Scalar Extended Under Mask

Instructions in group:

ste.(b|h|w|l|s|d).(t|f) Sk,<effa>

This instruction only requires VM bits to be transferred to the scalar processor. The vector dispatch logic (VDL) dispatches the UL controller when it is not busy. The UL controller starts the VMO controller to transfer the VM bits to the scalar processor.

B.1.6 Move Scalar to Vector Element

Instructions in group:

mov Si,Sj,Vk

The UL micro controller is started to execute the instruction. The UL controller starts the load back door controller (LBD) to request two items be transferred to the vector processor. The first is the index of the vector register element where the data is to be stored. The second is the data which is to be stored in the vector register element.

When the index is received by the input staging gate arrays (IS GAs) it is replicated across the even and odd data busses which go to the VRF GAs. The replication allows each VRF GA to have a copy of the index so that each VRF GA knows which element is to be written. The least significant bit of the index is used as an even or odd element write select. The data which is to be written is presented to both the even and odd VRF input busses and the vector register element is written.

B.1.7 Move Vector Element/Scalar

Instructions in group:

mov Vi,Sj,Sk

The load/store controller starts the LBD controller to control the transfer of the vector register element index. Once the index is received it is replicated across the even and odd data busses which go to the VRF GAs. The replication allows each VRF GA to have a copy of the index so that each VRF GA knows which element is to be read. Once the index is loaded into the VRF GAs the corresponding data elements (even and odd) are read. The OS controller is started to transfer the data element to the scalar processor. The OS controller uses the least significant bit of the index to select whether to transfer the even or odd vector register element.

B.1.8 Load VL

Instructions in group:

mov	*Ak, VL*
ld.w	*#N, VL*
ld.l	*<effa>, VLS*
mov.w	*Sk, VL*

The load/store micro controller enables the vector length register logic to request a new VL value. The VL logic stores the vector length value into the VL register. Both the scalar processor and the vector processor have a copy of the VL register. All VL load instructions must load the VL registers on both processors, but stores only access the copy on the scalar processor.

B.1.9 Load VM

Instructions in group:

ld.x	*<effa>, VM*
mov	*Sj,Sk, VM*
mov	*Sk, VMU*
mov	*Sk, VML*

These instructions load the 128 bit vector merge register. The *ld.x* instruction loads all 128 bits of the VM register and the *mov* instructions load only 64 bits. The 64 bits to load is determined by either the instruction dispatched or by a scalar transferred from the scalar processor. The load/store micro controller controls execution of the instruction. The UL controller starts the LBD controller to request the transfers from the scalar processor. The data to be loaded is received by the vector merge gate arrays and is loaded under control of the UL micro controller.

B.1.10 Store VM

Instructions in group:

st.x	*VM, <effa>*
mov	*VMU,Sk*
mov	*VML,Sk*
mov	*Sj, VM,Sk*
plc.(t|f)	*VM,Sk*

These instructions read the 128 bit vector merge register and transfer the data to the scalar processor. The *st.x* and *plc* instructions read both upper and lower halves of the VM register and transfer both to the scalar processor. The *mov* instructions transfer only one 64 bit piece of the VM register. The 64 bits to transfer is determined by either the instruction dispatched or by a scalar transferred from the scalar processor. If the 64 bits is determined by a transfer from the scalar processor then the UL micro controller starts the LBD controller to request the transfer. The output staging controller (OS) controls the transfer of the VM register to the scalar processor.

B.1.11 ALU Pipe Vector Result Operations

Instructions in group:

Diatic Instructions		Monatic Instructions	
add.(b|h|w|l|s|d)	*Vi, Vj, Vk*	*neg.(b|h|w|l|s|d)*	*Vj, Vk*
add.(b|h|w|l|s|d).(t|f)	*Vi, Vj, Vk*	*neg.(b|h|w|l|s|d).(t|f)*	*Vj, Vk*
sub.(b|h|w|l|s|d)	*Vi, Vj, Vk*	*shf*	*Sj, Vk*
sub.(b|h|w|l|s|d).(t|f)	*Vi, Vj, Vk*	*shf.(t|f)*	*Sj, Vk*
add.(b|h|w|l|s|d)	*Vi, Sj, Vk*	*not*	*Vj, Vk*
add.(b|h|w|l|s|d).(t|f)	*Vi, Sj, Vk*	*not.(t|f)*	*Vj, Vk*
sub.(b|h|w|l|s|d)	*Vi, Sj, Vk*	*plc.t*	*Vj, Vk*
sub.(b|h|w|l|s|d).(t|f)	*Vi, Sj, Vk*	*plc.t.(t|f)*	*Vj, Vk*
sub.(s|d)	*Si, Vj, Vk*	*tzc*	*Vj, Vk*
sub.(s|d).(t|f)	*Si, Vj, Vk*	*tzc.(t|f)*	*Vj, Vk*
shf	*Vi, Vj, Vk*	*lzc*	*Vj, Vk*
shf.(t|f)	*Vi, Vj, Vk*	*lzc.(t|f)*	*Vj, Vk*
shf	*Vi, Sj, Vk*	*cvt*	*Vj, Vk*
shf.(t|f)	*Vi, Sj, Vk*	*cvt.(t|f)*	*Vj, Vk*
and	*Vi, Vj, Vk*	*frint.(s|d)*	*Vj, Vk*
and.(t|f)	*Vi, Vj, Vk*	*frint.(s|d).(t|f)*	*Vj, Vk*
or	*Vi, Vj, Vk*		
or.(t|f)	*Vi, Vj, Vk*		
xor	*Vi, Vj, Vk*		
xor.(t|f)	*Vi, Vj, Vk*		
and	*Vi, Sj, Vk*		
and.(t|f)	*Vi, Sj, Vk*		
or	*Vi, Sj, Vk*		
or.(t|f)	*Vi, Sj, Vk*		
xor	*Vi, Sj, Vk*		
xor.(t|f)	*Vi, Sj, Vk*		

These instructions are executed on the ALU function pipe. The vector dispatch logic dispatches the ALU micro controller when the necessary resources are available. One of the resources which is required by some of the instructions is a scalar data value. The VDL informs the LBD controller to request a scalar to be used by the instruction it is holding. The LBD controller requests the data to be transferred and when it has received it the LBD controller informs the VDL logic. The scalar is held in the input staging gate arrays until the instruction can be started on the ALU micro controller. During the first cycle that the micro controller is executing the instruction the scalar value is transferred to the VRF GAs.

The ALU micro controller (UA) controls the reading of the operands for the instruction being executed. If the instruction is a scalar/vector instruction then the scalar value is selected as one of the operands. The UA controller starts the AFU controller which controls the SALU gate arrays. The operation which is to be performed by the SALU gate arrays is read from a table in the VDL logic. Once results are available from the SALU gate arrays the UA micro controller starts the ALU back door write controller (ABD). The ABD controls writing the results to the VRF GAs.

If the instruction is being performed under mask then the VM bits corresponding to the vector register elements to be read are presented to the VRF GAs. The VM bits are used to select either the vector register element or an identity element for the operation being performed. The VRF

GAs have builtin logic to produce all required identity constants (all 0's, all 1's, minimum and maximum). The identity element is used to inhibit any status bits which could be generated from masked elements. The VM bits are also staged by the ABD logic and used to inhibit writing masked vector elements.

B.1.12 ALU Pipe Compare Operations

Instructions in group:

le.(b|h|w|l|s|d)	*Vj, Vk*
le.(b|h|w|l|s|d).(t|f)	*Vj, Vk*
lt.(b|h|w|l|s|d)	*Vj, Vk*
lt.(b|h|w|l|s|d).(t|f)	*Vj, Vk*
eq.(b|h|w|l|s|d)	*Vj, Vk*
eq.(b|h|w|l|s|d).(t|f)	*Vj, Vk*
le.(b|h|w|l|s|d)	*Sj, Vk*
le.(b|h|w|l|s|d).(t|f)	*Sj, Vk*
lt.(b|h|w|l|s|d)	*Sj, Vk*
lt.(b|h|w|l|s|d).(t|f)	*Sj, Vk*
eq.(b|h|w|l|s|d)	*Sj, Vk*
eq.(b|h|w|l|s|d).(t|f)	*Sj, Vk*

The compare instructions are executed on the ALU micro controller. The scalar/vector instructions obtain a scalar in the same manner as the ALU vector result operations described above. The UA controls reading the operands from the vector register file gate arrays. The UA controller starts the AFU logic to control the SALU gate arrays and starts the ABD controller to control writing the result of the compares in the vector merge gate arrays (VM GAs).

For operation under mask VM bits are applied to the VRF GAs to select identity constants and are staged by the ABD logic to mask writes to the vector merge register. Those instructions which are not under mask force the VM bits to be ones.

B.1.13 ALU Pipe Reduction Operations

Instructions in group:

sum.(b|h|w|l|s|d)	*Vk*
sum.(b|h|w|l|s|d).(t|f)	*Vk*
max.(b|h|w|l|s|d)	*Vk*
max.(b|h|w|l|s|d).(t|f)	*Vk*
min.(b|h|w|l|s|d)	*Vk*
min.(b|h|w|l|s|d).(t|f)	*Vk*
all	*Vk*
all.(t|f)	*Vk*
any	*Vk*
any.(t|f)	*Vk*
parity	*Vk*
parity.(t|f)	*Vk*

These instructions operate on a vector register and produce a single result. The less frequently used instructions reduce the vector to a single result entirely on the vector processor. The more frequently used instructions (sum) reduce part of the vector register on the vector processor and complete the operation on the scalar processor.

The VDL logic dispatches the UA micro controller to execute the instructions. The UA controls operand selection and starts the AFU controller to control the SALU gate arrays. The instructions are completed by first reducing all even elements on the even ALU function pipe and all odd elements on the odd ALU function pipe. Next, the odd partial results are transferred to the even pipe using the AFU Odd to Even Stage Register (shown in figure 1-2). The even and odd partial results are reduced to a few partial results and either transferred to the scalar processor (sum instructions) or reduced to one result on the even AFU function pipe. The ABD write controller is not used since the vector registers are not written by the instructions.

The reduction instructions are executed under mask by using the VM bit to select between the vector register file element and an identity constant. The identity constant inhibits status bits from being generated by masked elements and will not contribute to the result of the reduction.

B.1.14 Multiply Pipe Multiply/Divide/Square Root Operations

Instructions in group:

mul.(b|h|w|l|s|d)	*Vi, Vj, Vk*
mul.(b|h|w|l|s|d).(t|f)	*Vi, Vj, Vk*
div.(b|h|w|l|s|d)	*Vi, Vj, Vk*
div.(b|h|w|l|s|d).(t|f)	*Vi, Vj, Vk*
mul.(b|h|w|l|s|d)	*Vi, Sj, Vk*
mul.(b|h|w|l|s|d).(t|f)	*Vi, Sj, Vk*
div.(b|h|w|l|s|d)	*Vi, Sj, Vk*
div.(b|h|w|l|s|d).(t|f)	*Vi, Sj, Vk*
div.(s|d)	*Si, Vj, Vk*
div.(s|d).(t|f)	*Si, Vj, Vk*
sqrt(s|d)	*Vj, Vk*
sqrt(s|d).(t|f)	*Vj, Vk*

These instruction are executed on the multiply function pipe. The vector dispatch logic dispatches the multiply micro controller (UM) when the necessary resources are available. One of the resources which is required by some of the instructions is a scalar data value. The VDL informs the LBD controller to request a scalar to be used by the instruction it is holding. The LBD controller requests the data to be transferred and when it has received it the LBD controller informs the VDL logic. The scalar is held in the input staging gate arrays until the instruction can be started on the UM micro controller. During the first cycle that the micro controller is executing the instruction the scalar value is transferred to the VRF GAs.

The UM controls the reading of the operands for the instruction being executed. If the instruction is a scalar/vector instruction then the scalar value is selected as one of the operands. The UM controller starts the MFU controller for multiply operations and the DFU controller for divide and square root operations. The MFU logic controls the SMUL gate arrays and the DFU logic controls the DIVX gate arrays. The operation which is to be performed is read from a table in the VDL logic. Once results are available from the gate arrays the UM micro controller starts the multiply back door write controller (MBD). The MBD controls writing the results to the VRF GAs.

If the instruction is being performed under mask then the VM bits corresponding to the vector register elements to be read are presented to the VRF GAs. The VM bits are used to select either the vector register element or an identity element for the operation being performed. The VRF GAs have builtin logic to produce the identity constant (the value 1). The identity element is used to inhibit any status bits which could be generated from masked elements. The VM bits are also staged by the MBD logic and used to inhibit writing masked vector elements.

B.1.15 Multiply Pipe Vector Edit Operations

Instructions in group:

cprs.(t|f)	*Vj, Vk*
xpnd.(t|f)	*Vj, Vk*
merg.t	*Vi, Vj, Vk*
merg.(t|f)	*Vi, Sj, Vk*
mask.t	*Vi, Vj, Vk*
mask.(t|f)	*Vi, Vj, Vk*
mask.(t|f)	*Vi, Sj, Vk*

These instructions manipulate vector register elements based on the vector merge register value. The UM micro controller is started to execute the instructions. The UM controller enables the vector edit logic (VE) to control the vector edit operations and starts the MBD logic to control writing the vector register file gate arrays.

The mask instruction uses a data path internal to the VRF GAs to stage data from the vector register read data path to the vector register write data path. The other instructions use the internal data path as well as the external data paths which couple the even VRF GA and odd VRF GA. The external data paths use the vector edit even and odd stage registers to couple the even and odd VRF GAs.

B.1.16 Multiply Pipe Reduction Operations

Instructions in group:

prod.(b|h|w|l|s|d)	*Vk*
prod.(b|h|w|l|s|d).(t|f)	*Vk*

These instructions operate on a vector register and produce a single result. The VDL logic dispatches the UM micro controller to execute the instructions. The UM controls operand selection and starts the MFU controller to control the SMUL gate arrays. The instructions are completed by first reducing all even elements on the even multiply function pipe and all odd elements on the odd multiply function pipe. Next, the odd partial results are transferred to the even pipe using the MFU Odd to Even Stage Register (shown in figure 1-2). The even and odd partial results are reduced to a few partial results and transferred to the scalar processor.

The reduction instructions are executed under mask by using the VM bit to select between the vector register file element and an identity constant. The identity constant inhibits status bits from being generated by masked elements and will not contribute to the result of the reduction.

B.2 Memory Fault Operation

The C200 vector processor handles all memory faults identically, independent of the vector instructions executing. There are two routines, state save and state restore, which are executed when a fault occurs. All state is saved and restored whether or not the state is meaningful when the memory fault occurs.

Fault processing is initiated when two signals are both asserted. The signals indicate that a memory fault has occurred and that the memory system has returned all outstanding read requests. When both signals are asserted the vector processor clock logic micro interrupts the load/store, add, and multiply micro controllers to start executing at location zero.

The load/store micro controller generates control store addresses for all three micro controllers. This allows the micro fields which normally control address generation for the add and multiply micro controllers to be used for special state save/restore functions. All vector processor state is held until it can be saved by transferring it to memory.

The steps which are required to the save state are as follows:

1. The data contained in the output staging gate array's registers is transferred to memory. This allows the output staging data paths to be used to stage other state.
2. Next the registers implemented in non gate array devices with scan capability are saved. This state includes all control registers on the VPC and the PSW register on the VPD. The state is saved in eight 64-bit transfers to memory. Each transfer contains one bit from each eight bit control register. The registers are shifted to the right one bit after each transfer is made.
3. After all control state is saved the vector processor control which was scan saved is reset to an idle state.
4. Next the vector register file gate array state is saved. The load/store micro controller is used to select state from the VRF gate arrays to be staged through the output staging gate arrays and transferred to memory. Once the state is saved all internal data path register are allowed to clock as normal.
5. The SALU, SMUL, DIVX gate array state and TTL stage registers are staged through the VRF gate arrays and output staging gate arrays to memory.
6. The last pieces of state to be transferred are from the input staging gate arrays.

Once all state is saved, the load/store micro controller waits until a signal is asserted indicating that the instruction processor is ready to dispatch valid instructions. The load/store micro controller then allows all interface signals to operate as normal, and the three micro controllers go idle.

Processing resumes after the operating system has corrected the memory fault condition. The operating system executes a RTNC instruction which restores the state of the processor from memory. The load/store micro controller on the vector processor is dispatched to the restore micro code. The micro controller then waits until the rest of the vector processor is idle so that all instructions which are executing complete normally.

The load/store micro controller forces the add and multiply micro controllers active for the remainder of the restore context instruction. The load/store micro controller generates the control store addresses as was the case with state save.

The state of the vector processor is restored as follows:

1. The output staging gate array state is transferred from memory through the input staging and vector register file gate arrays to the output staging gate arrays.
2. The state for the SALU, SMUL, DIVX and TTL staging registers is restored through the input staging and vector register file gate arrays.
3. The vector register file gate array state is restored through the input staging gate arrays.
4. The all input staging gate array state is restored except the first stage register.
5. Next the register implemented in non gate array devices is restored. The state is transferred from memory to the vector merge gate arrays. The data is shifted to the right into the register.

Once all state is restored the load/store micro controller waits until a signal is asserted indicating that the scalar processor has completed its state restore sequence. The load/store micro controller then allows all interface signals to operate as normal and the three micro controllers return to the operation they were performing when the state was saved.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix C
Reporting Problems

C.1 Overview

This appendix introduces the CONVEX Technical Assistance Center (TAC) and the *contact* utility. The *contact* utility is an online system for reporting problems to the TAC. To learn *contact* by using it, enter **contact** at the system prompt and then answer the questions as they appear on the screen. To find out more about using *contact*, read through this appendix. It describes prerequisites and tips for using *contact* and the step-by-step process *contact* takes you through.

C.2 Technical Assistance Center

The CONVEX Technical Assistance Center (TAC) is staffed by technical specialists who can address the diverse questions and problems that arise in a supercomputing environment. If you have a hardware, software, or documentation problem, contact the TAC. This group stands ready to solve such problems.

C.3 The *contact* Utility

The TAC recommends using the *contact* utility to report a hardware, software, or documentation problem. The *contact* utility is an interactive utility that helps the TAC track reports and route them to the the CONVEX personnel most qualified to fix them.

After invoking *contact*, it prompts for information about the problem. When you finish your report, *contact* electronically mails it to the TAC. You are notified within 48 hours that the TAC has received your report.

C.4 Prerequisites

To use *contact* requires

- a UNIX-to-UNIX Communication Protocol (UUCP) connection to the TAC
- the full path name of the program or utility in question
- the version number of the program or utility in question

C.4.1 UUCP Connection

Before using *contact*, check with your system administrator to be sure there is a UUCP connection to the TAC. A UUCP connection allows files to be copied from one UNIX system to another. The *uucp* (UNIX-to-UNIX copy) command relies on either a dial-up or hard-wired UUCP communication line.

C.4.2 Finding the Program Path Name

To determine the full path name of the program or utility in question, use the *which* command. The following screen illustrates using the *which* command to find the full path name of the loader (*ld*) utility:

```
>which ld
/bin/ld
>
```

In this example, the full path name of the loader is */bin/ld*.

For more information on the *which* command, refer to the *which*(1) man page. You can also use the *info* online information system. Enter **info which** at the system prompt. If you use the C shell (*csh*), you can also use the *whence* command to find the program path name. The *whence* command works like *which*, only faster.

C.4.3 Finding the Program Version Number

To determine the version number of the program or utility in question, use the *vers* command. The following screen illustrates using the *vers* command (enter **vers**, then the path name of the program or utility) to find the version number of the loader (*ld*) utility.

```
>vers /bin/ld
/bin/ld: 7.0
>
```

In this example, the loader utility version number is 7.0.

For more information on the *vers* command, refer to the *vers*(1) man page. You can also use the *info* online information system. To do so, enter **info vers** at the system prompt.

C.5 Tips on Using the *contact* Utility

The *contact* utility is interactive and easy to use. This section lists tips to help use it efficiently. In particular, this section tells how to

- use a *.contact* file
- abort a contact session
- resubmit an aborted report
- suspend a contact session
- move from one prompt to another
- use tilde-escape sequences in the *contact* utility

C.5.1 Using a *.contact* File

When invoked, *contact* prompts for information regarding the problem. The first prompt is for your name, title, phone number, and company name. You can, however, create a *.contact* file to skip this first prompt. Follow these steps:

1. Create a *.contact* file in your home directory.
2. Enter your name, job title, phone number, and company name, each on a new line.

When you invoke *contact*, it automatically includes the *.contact* file as input for the first prompt and proceeds to the next prompt.

C.5.2 Aborting the Report

To abort a contact report, either enter the interrupt key (usually CTRL-C) or choose the abort option when prompted by the *contact* utility. Using CTRL-C to abort does not save the contents of the report. Using the abort option saves the contents of the report in a file named *dead.report* in your home directory.

C.5.3 Submitting the *dead.report* File

When aborting a contact session, the *contact* utility saves the report in a file named *dead.report* in your home directory. Using the *contact* command with the *-r* option automatically merges the contents of the *dead.report* file into the new contact session. Enter

contact -r

and *contact* finds the *dead.report* file in your home directory and merges it into the contact report. You can then edit the report. When you end the editing session, *contact* returns to the final prompt, which asks you to review, edit, submit, or abort the report.

C.5.4 Suspending a Report

Sometimes it is necessary to stop in the middle of a contact report and return to the shell (for instance, to suspend the contact session to find the program path name or version number). To suspend the contact session, press CTRL-Z. To return to the contact session, enter **fg**. Using CTRL-Z and the *fg* (foreground) command lets you switch back and forth between the *contact* utility and the shell. You cannot, however, use CTRL-Z and *fg* to switch back and forth if you are using a Bourne shell (*sh*).

C.5.5 Ending a Response

The *contact* utility prompts for information pertinent to your hardware, software, or documentation question. Some prompts require one-line responses; to move to the next prompt, press RETURN. Other prompts require more than a one-line response; to move to the next prompt, press CTRL-D.

C.5.6 Tilde-Escape Sequences

The *contact* utility treats input beginning with a tilde (~) as a special sequence. The character following the tilde is considered a request for a special function. The following tilde sequences are recognized by *contact*:

~e	Start the text editor (defined in your EDITOR environment variable).
~h	Display a list of available tilde-escape sequences.
~p	Print the contact report to the terminal screen.
~r *filename*	Read the contents of *filename* as a response to the current prompt. Some prompts require only a one-line response. This tilde-escape sequence only works for prompts that allow more a than one-line response.
~~	Insert a single tilde as the first character in the line.

C.6 Using the *contact* Utility

The *contact* utility prompts for the following information:

- your name, title, phone number, and corporate name
- the name and version of the product involved
- a one-line summary of the problem
- a detailed description of the problem
- the priority of the problem
- instructions on how to reproduce the problem
- comments about the problem
- comments about the documentation supporting the problem
- files to include in the contact report

The following is a step-by-step discussion of these prompts:

1a. To invoke the *contact* utility, enter **contact** at the system prompt. The system responds with a welcome message and a series of questions regarding your hardware, software, or documentation question. The following screen illustrates the *contact* command and the system response:

```
>contact
Welcome to contact version 0.11 ()

Enter your name, title, phone number, and corporate name (^D to terminate)
>
```

1b. If there is a *.contact* file in your home directory, *contact* skips the first prompt. The following screen illustrates the *contact* command and the system response when a *.contact* file is in your home directory:

```
>contact
Welcome to contact version 0.11 ()

Enter the name of the product involved
>
```

2. The *contact* utility prompts for the version number of the product. If you do not know the version number, use CTRL-Z to suspend the session. Use the *which* (or *whence* if using *csh*) and *vers* commands to find the version number of the product. Use the *fg* command to return to the session and enter the version number in the form X.X or X.X.X.X.
3. The *contact* utility prompts for a one-line summary of the problem. This summary is the subject header in any further correspondence regarding the problem. Make this summary as descriptive as possible in one line.
4. The *contact* utility prompts for a detailed description of the problem. Make this description as complete as possible. Include source code and a stack backtrace whenever possible. (Refer to the *adb*(1) or *csd*(1) man page for information on obtaining a stack backtrace.) The more information provided, the quicker the TAC can isolate and solve the problem.
5. The *contact* utility prompts for the priority of the problem. The following screen illustrates this prompt and the priority levels from which to choose; you must enter a priority number.

```
Enter a problem priority, based on the following:
1) Critical    - work cannot proceed until the problem is resolved.
2) Serious     - work can proceed around the problem, with difficulty.
3) Necessary   - problem has to be fixed.
4) Annoying    - problem is bothersome.
5) Enhancement - requested enhancement.
6) Informative - for informational purposes only.
>
```

6. The *contact* utility prompts for an explanation of how to reproduce the problem. Include the command syntax and options you used and anything else you did to make your program run.
7. The *contact* utility prompts for any other pertinent comments. Include any relevant information.
8. The *contact* utility prompts for suggestions regarding the documentation supporting the product. Indicate if the documentation could be revised to address the question.
9. The *contact* utility asks for the names of files necessary to reproduce the problem. The following screen illustrates the *contact* prompt and sample user response:

```
Are there any files that should be included in this report (yes | no)?
>yes
Please enter the names of the files, one to a line (^D to terminate)
>test.f
>~/subroutines/sub.f
>
```

NOTE

Tilde-escape sequences are not recognized in responses to this prompt. Instead, *contact* treats a tilde in this section to mean your home directory. This convention is based on use of the tilde for expanding file names in *csh*.

If the files specified are small text files, they are automatically included in the contact report. If the files are too big to be included in this report, *contact* gives further instructions on how to submit these files.

To specify a directory, combine the directory files into a single file using the *tar* command (refer to the *tar*(1) man page for further information) or enter each file name in the directory on a single line in the contact report.

10. The *contact* utility prompts you to review, edit, submit, or abort the contact report. The following screen illustrates this prompt:

```
Please select one of the following options:
1) Review the problem report.
2) Edit the problem report.
3) Submit the problem report.
4) Abort the problem report.
>
```

Choose the number of the option you want to select. These options let you do the following:

Review — Review the text of your contact report. You are then prompted again to select an option.

Edit — Edit the text of the contact report. If you choose to edit the report, *contact* puts you in your default text editor.

Submit — Send the report to the CONVEX TAC. You are notified within 48 hours that the TAC has received the report. This option exits the *contact* utility and returns you to the shell environment.

Abort — Save the text of your report in a file named *dead.report* in your home directory. This option exits the *contact* utility and returns you to the shell environment.

Index

Numeric

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

CONVEX Theory of Operation
(C200 Series)
Document No. 081-005030-000, Second Edition

Reader's Forum

You are invited to submit comments about the clarity and service of this manual. Constructive critical comments are most welcome, and will help us continue in our efforts to generate quality customer documentation. Please list the document page number with your questions and comments.

_____

_____

_____

_____

_____

_____

_____

_____

From:

Name_____ Title_____

Company_____ Date_____

Address and Phone No._____

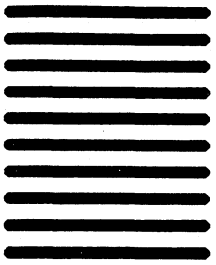
FOR ADDITIONAL INFORMATION OR DOCUMENTATION:

Location	Phone Number
From all locations in continental U.S.	1(800)952-0379
From locations in Alaska, Hawaii, & Canada	1(214)952-4379
From all other locations	Contact nearest CONVEX office

Direct mail orders to:

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

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 1046 RICHARDSON, TEXAS

POSTAGE WILL BE PAID BY ADDRESSEE

CONVEX Computer Corp.
Customer Service
P.O. Box 833851
Richardson, TX 75083-3851

(Fold Here)

(Tape or Staple)