CONVEX Theory of Operation (C3800 Series)

Order No. DHW-210

First Edition
January 1991

Internal Use Only

DRAFT DISCLAIMER

This document is a draft. As such, it is not approved for field or customer distribution. It is approved for internal use only as a working copy for this document's CONVEX review team. CONVEX makes no claims as to the accuracy of the nformation contained herein and does not approve its distribution or use outside the review team defined in this document's QA Plan.

CONVEX Computer Corporation
Richardson, Texas USA

Table of Contents

Introduction

Overview .1-1
Description of Major Units .1-3
  Central Processing Unit (CPU) .1-3
    Scalar Processor Board .1-3
    Vector Processor Board .1-3
  Memory Subsystem .1-3
  Crossbar .1-4
  CPU Utilities Board .1-4
    Communication Registers .1-4
    Control Register Address Space .1-4
    ASAP .1-5
    Trap and Interrupt Structure .1-5
    Master Clock Generation and Control .1-5
    SPU Interface .1-5
    Expansion Port Interface .1-5
    Communication Register Polling .1-5
    CPU Deadlock Monitoring .1-5
  Service Processor Unit .1-5
  Diagnostic Hardware .1-6
  I/O Subsystem .1-6
  Power Subsystem .1-6

Architecture Overview

Overview .2-1
Register sets .2-1
Memory management .2-1
  Virtual address space .2-2
  Process structure and control .2-2
  Terminology .2-3
  Segment descriptor registers .2-4
  Page table entries .2-4
  Address Translation .2-4
  Shared and Unshared Memory .2-4
  Reference and Modified Bits .2-5
Multiprocess- ing introduction .2-5
Automatic self-allocating processors .2-5
  CPU states .2-6
  CPU scheduling .2-6
  Forking .2-6
  Parallel processing .2-8
The communication registers .2-9
  Communication index registers .2-9
  Communication register partitions and rings .2-9
  Communication register addressing 2-10
  Communication registers modified bits 2-10
  Hardware communication registers 2-[illegible]
    Trap instruction registers 2[illegible]
    Thread allocation mask and count 2-11
    Fork event communication registers 2-12
    Segment descriptor registers 2-13
    CPU execution timer registers 2-13

CPU deadlock detection . . . 2-13
CPU timers . . . 2-13
Execution Timer . . . 2-14
Thread timer . . . 2-14
CTR and TTR timer updating . . . 2-14

Instruction Processor

Overview . . . 3-1
Instruction Cache . . . 3-1

Scalar Processor

Overview . . . 4-1
Major Functions . . . 4-1
Major Units . . . 4-1
Data Return Queue . . . 4-1
Scalar Data Structure . . . 4-1
Register File / ALU Gate Array . . . 4-2
Data Cache . . . 4-2
Floating-Point and Miscellaneous Integer Function Unit . . . 4-2
Address Generation Logic . . . 4-2
PTE Cache . . . 4-2
Scratch RAM . . . 4-2
Scalar Processor Microsequencer . . . 4-3
Source Hazard Logic . . . 4-3
Scalar Processor Operations . . . 4-3
Address Translation . . . 4-4
Memory Management Terminology . . . 4-4
Basic Steps . . . 4-4
Segment Descriptor Registers . . . 4-5
Page Table Entries . . . 4-5
Address Translation Hardware Operation . . . 4-6
Cache Operations . . . 4-7
Cache Read . . . 4-7
Indirect Cache Read . . . 4-7
Cache Write . . . 4-7
Cache Purge . . . 4-8
Memory Operations . . . 4-8
Memory Read . . . 4-8
Memory Write . . . 4-8
Scalar Instruction Execution . . . 4-8

Vector Processor

Overview . . . 5-1
Major Functions . . . 5-1
Major Units . . . 5-1
Input Staging . . . 5-1
Vector Dispatch Interface . . . 5-1
Function Pipes . . . 5-1
Pipe Controllers . . . 5-2
Vector Register File . . . 5-2
Output Staging . . . 5-3

Crossbar

Overview . . . 6-1
Crossbar operations . . . 6-1

Memory transfers 6-1
Processor-to-crossbar handshaking 6-3
Memory read or write request 6-3
Data return . . . . . .
No operation . . . . . .
Test and modify 6-5
CPU utilities board transfers 6-5
Data transfers and status request 6-5
CU board and memory transfer differences 6-5
Crossbar-to-utilities board handshaking 6-7
Status request and return 6-7
Status and control transfers 6-7
Request pending and stores pending 6-7
Processor request overflow 6-7
Crossbar functional units 6-7
Crossbar boards 6-9
Gate array functions 6-9
Arbitration 6-11
Overflow queue 6-12
Physical configuration map 6-12
Parity checking 6-12
Control crossbar 6-12

Memory Subsystem

Overview 7-1
Memory Boards 7-1
Major Functional Units 7-2
Input Staging 7-2
Bank Control . . . . . .
Read EDC 7-2
Write EDC 7-2
Memory Card 7-2
Interface 7-2
Error Detection and Correction 7-2

CPU Utilities ~~Board~~ Hardware

Overview 8-1
The Communication Registers 8-3
Control Register Address Space 8-4
ASAP Accelerator 8-7
Interrupts and Microtraps 8-7
Interrupts 8-8
Microtraps 8-8
Process Deadlock Detection 8-9
Hardware 8-9

Clock and Diagnostic Hardware

Overview 9-1
Scan engine 9-1
Scan memory 9-1
SPU interface . . . . . .
Clock generator . . . . . .

~~Service Processor Unit~~ Workstation Interface Board

Overview 10-1

PRELIMINARY

move to Chap 1

SPU Workstation 10-1
SPU Operating Mode and Modem Key Switches 10-2
Workstation Interface Board 10-4
DIO Bus Interface 10-4
Address Maps 10-6
Select Code Switches 10-8
Card ID / Reset Register 10-10
Card Size Register 10-11
Card Status and Control 10-13
Parallel Interface 10-14
Data Transfer Sequence of Events 10-16
Error Conditions 10-17
Interrupt Register Logic 10-19
BPC Interfaces 10-21
Cable Interlock 10-21
Serial Interface 10-21
Reset Control 10-21
Key Switch Interface 10-21
Printer Interface 10-22
Utility and Diagnostic Hardware 10-23
Miscellaneous Control Register 10-23
Parity Error Force Register 10-24
Loopback Data Register 10-25
CU Address Register 10-25
~~Peripherals~~ 10-26

Input / Output Subsystem

Overview 11-1
Interface adapter 11-2
Input/output channel interface 11-3
Read data queue 11-4
Write data queue 11-4
Memory interface 11-5
Arbitration 11-6
Read queue arbiter 11-6
Write queue arbiter 11-7
Port arbiter 11-7
Interrupt handling 11-7
CPU-to-CPU interrupts 11-7
CPU-to-CCU interrupts 11-8
CCU-to-CPU interrupts 11-8
PBUS 11-8
PBUS transfers 11-9
PBUS data path 11-9
PBUS control path 11-10
PBUS request 11-11
PBUS grant 11-11
Bus lock 11-11
Header 11-11
Data valid 11-12
Memory buffer available 11-12
Channel buffer available 11-12
Bus error 11-12
Memory error 11-12
Expansion port 11-12
XP and PBUS differences 11-12
XP transfer types 11-12
XP header transfers 11-13
Expansion port interface signals 11-13
XP data bus 11-13

PRELIMINARY

XP bus data handshake signals 11-13
IA bus error 11-14
XP interrupt bus 11-14

Power Subsystem

Overview 12-1
Input AC Power 12-3
Bay Power Unit (BPU) 12-3
Input Power Harmonizer 12-3
Bay Power Supply (BPS) 12-3
Bay Power Controller (BPC) 12-3
Power Pallets 12-3
Channel Control Unit Power Pallet (CCUPP) 12-3
Logic Board Power Pallet (LBDPP) 12-3
Crossbar Power Pallet (XbarPP) 12-4
Power Pallet Controller 12-4
Power Enable Keylock 12-4

PRELIMINARY

12-13-90

Introduction

1.1 Overview

The CONVEX C3800 Series computer can have up to eight processors and up to four gigabytes of physical memory. Interprocessor communication and memory transfers are controlled by a central crossbar (see Figure 1-1).

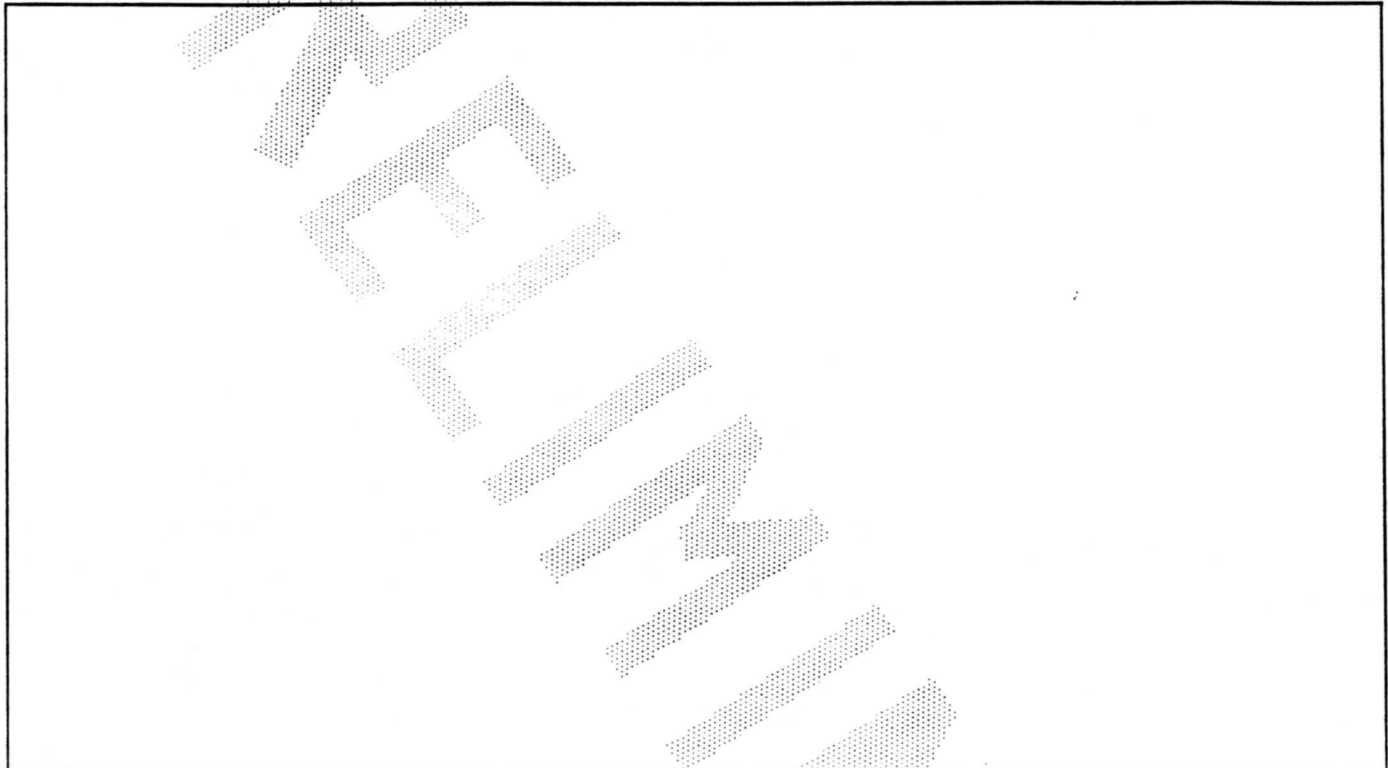
Figure 1-1 System Block Diagram

C3SYS8.DRW
900828
CENTRAL PROCESSING UNITS
CROSSBAR
MEMORY BOARDS
INTERFACE ADAPTER
CPU UTILITIES BOARD
EXPANSION PORT BUS
PBUS 0
CCU 00
CCU 01
PBUS 1
CCU 02
CCU 03
PBUS 2
CCU 04
CCU 05
PBUS 3
CCU 06
CCU 07
SPU WORKSTATION INTERFACE
PERIPHERALS
BAY POWER CONTROLLERS
I/O SUBSYSTEM
HIGH SPEED CHANNEL
SPU WORKSTATION

The processor cabinets for the C3800 Series are physically configured around a central cabinet that contains the crossbar (see Figure 1-2). There can be from three-to-six cabinets per system: one central crossbar cabinet, up-to-four processor cabinets and one I/O interface cabinet.

Each processor cabinet may contain two heads. A head is defined as a single processor and one memory board.

Figure 1-2 Cabinet Configuration (Top View)



PRELIMINARY

1.2 Description of Major Units

The following major units comprise a C3800 Series computer system:

- Central Processing Unit
- Memory Subsystem
- Crossbar
- CPU Utilities Hardware
- Service Processor Unit
- Clock and Diagnostic Hardware
- I/O Subsystem
- Power Subsystem

1.2.1 Central Processing Unit (CPU)

Each CPU consists of two boards: a scalar processor board and a vector processor board.

1.2.1.1 Scalar Processor

The scalar processor executes scalar instructions, performs address generation and translation, and dispatches the vector processor. The scalar processor contains the following major functional units:

- Address Scalar (AS) Unit
- Instruction Processor (IP)
- Data Cache (DC)
- Return Control (RC)

The AS unit executes scalar instructions under microcode control. Decoded instructions are dispatched to the AS from the instruction processor (IP). Each scalar instruction has a corresponding set of microcode that resides in the control store located in the AS.

The AS also receives vector instructions from the IP. These instructions are passed on to the vector processor for execution.

The instruction processor has its own cache memory where it stores all instruction data received from main memory. The instruction processor maintains the program count and executes instruction cache read/write operations. When data is read from the instruction cache, the instruction processor decodes and parses the information and sends it to the address scalar unit for execution.

The data cache arbitrates memory request priorities, controls access to the data and PTE caches, and handles all requests that involve accessing the crossbar (that is, all memory and communication register transfers).

Return control retains the control information for all read requests that go out on the crossbar to memory or the communication registers. When the requested data arrives, the RC matches the incoming data with the correct control information and forwards the data to its destination.

1.2.1.2 Vector Processor

The primary function of the vector processor is to execute all vector instructions. Vector instructions are dispatched from the scalar processor and input by way of the vector dispatch interface. Operands are input from either the scalar processor (if they are cache resident) or from memory. Inside the vector processor, operands

PRELIMINARY

are stored in the vector register file until needed. Data produced from vector operations may be sent to either the scalar processor or memory.

1.2.2 Memory Subsystem

A C3800 Series system may have up to 4 GB of addressable memory implemented using up-to-eight memory boards. Each memory board has a capacity of from 128 MB-to-512 MB, depending on the size of the DRAM (1MB or 4MB) being used.

Each memory board can accept requests and return memory data to the requesting processors at the rate of 64 bits of data per clock cycle. Thus, each board has a bandwidth of 500 MB per second; a fully configured, eight-board memory system has a bandwidth of four GB per second.

The 64 bits of data that a memory board can handle is divided into two 32-bit words, called even and odd. Each even and odd word can be addressed independently, and each one has a separate send and return datapath.

1.2.3 Crossbar

The crossbar provides the link between the processors, the memory boards and the CPU utilities board. By definition, a processor may be either a CPU or an I/O processor.

Each processor communicates with the crossbar on a separate point-to-point bus. No multidrop buses are used. The same is true for the memory boards and the CPU utilities board.

The crossbar routes data to and from the memory boards and arbitrates between requests to identical memory boards. The memory boards can accept one even and one odd request and return one even and one odd request per clock cycle. The crossbar determines which processor wins access to a particular memory board and to which processor any returning data must be sent.

The crossbar has a total of nine processor ports, eight memory board ports and one port for the CPU Utilities board. Each 64-bit (plus parity) port is divided into an even and an odd half. The 32-bit (plus parity) odd and even halves function completely independent of each other.

1.2.4 CPU Utilities Hardware

The CPU utilities hardware resides on the CPU utilitites (CU) board and consists of the following major functional units:

- Communication Registers
- Control Register Address Space
- ASAP
- Microtrap and Interrupt Structure
- Deadlock Detection Logic

1.2.4.1 Communication Registers

The communication registers provide a mechanism for multiple thread execution across more than one CPU. A communication register is a 64-bit addressable register with an associated lock bit that is maintained by hardware. The lock bit is used by software and microcode as a semaphore for the contents of the individual communication register.

1.2.4.2 Control Register Address Space

The control register address space contains special purpose registers that require system-wide accessibility.

1.2.4.3 ASAP

Multiprocessing requires a method for scheduling the use of the available processors in the system. The goal is to divide and conquer the workload. The C3800 Series processors use a scheduling scheme called Automatic Self-Allocating Processors (ASAP). A fundamental characteristic of ASAP is that each CPU within the complex is solely responsible for scheduling itself. There is no master process for finding idle CPUs and scheduling work for them.

The CPU Utilities Board includes much of the hardware that is used to detect work (processes or threads) as well as various mechanisms that permit a processor to associate and disassociate itself from a given process.

1.2.4.4 Microtrap and Interrupt Structure

1.2.4.5 Deadlock Detection Logic

1.2.5 Service Processor Unit

The service processor unit (SPU) initializes the computer system, boots the CONVEX operating system, monitors system operation, and executes system diagnostics during maintenance. These functions are performed by SPU-based programs that execute unter the SPU operating system.

The SPU resides in a stand-alone cabinet separate from the processor and I/O cabinets. The SPU contains the following functional units:

- SPU Workstation
- Key Switches
- Workstation Interface Board
- Peripherals

The SPU workstation is a Hewlett Packard product that is based on the Motorola 68030 microprocessor.

The workstation interface board provides the interconnecting link between the SPU workstation and the various functional units of the C3800 System with which the SPU must interact. These interconnecting links include a parallel interface to the CPU Utilities board for access to both system memory and the clock and scan control circuits. There is also a serial interface through which control can be exercised over the Bay Power Controllers located in each cabinet. In addition, there is an interface for the SPU key switches and an RS-232 printer interface.

The peripherals include a disc drive, tape drive, modem, and a serial printer.

A key switch mounted on the workstation cabinet determines the operating mode of the SPU Workstation. Operating modes include: OFF, LOCAL, CPU ONLY, and SECURE. A second key switch determines the access mode of the SPU modem. The access modes are: OFF, SPU, and CPU.

Tables 1-1 and 1-2 define the switch positions for the SPU and modem key switches, respectively.

1.2.6 Clock and Diagnostic Hardware

Operations performed by the clock and diagnostic hardware are based on commands received from the SPU through the workstation interface board. Clock and scan operations are performed through use of scan control modes, which control the individual boards in the system, and the contents of the clock and scan control registers. These control registers reside in memory located on the CU board and are part of the clock generator and scan engine.

The clock and diagnostic hardware is partitioned across the CU board, the crossbar control board, and the crossbar backplane. Functionally, there are five areas:

- Clock generator
- Scan engine
- Scan memory
- Workstation-to-CU interface
- XP interface

1.2.7 I/O Subsystem

The basic components of a C3800 Series Input /Output Subsystem consist of an Interface Adapter (IA), a high speed channel, the PBUS, and the Channel Control Units (CCUs). See Figure 1-1.

The I/O chapter in this book focuses primarily on the interface adapter and the PBUS. Information about the other components in the I/O Subsystem can be found in the User Guide for that product.

The Interface Adapter (IA) serves as a memory interface for input/output controllers in the C3800 Series. The IA connects PBUS resident Channel Control Units to memory through the Crossbar. Both data and interrupts are handled by the IA. Interrupt handling for the C3800 Series computer systems differs significantly from the C100 and C200 Series computers.

Four separate PBUS interfaces and a newly defined I/O interface, called the Expansion Port, are supported by the IA. The PBUS is the general purpose input/output interface bus. The Expansion Port (XP) is a fast, special purpose input/output channel that provides a higher performance alternative to the PBUS.

To the CCUs, the IA looks like the Peripheral Interface Adapter (PIA) used on other CONVEX C-Series computers. To the Crossbar and memory, the IA looks like a processor.

1.2.8 Power Subsystem

The Power Subsystem uses a distributed power architecture. Input AC power is rectified into unregulated DC and then distributed to DC-to-DC converters that are physically located close to the load. The control and monitoring of power and environmental functions is also distributed.

PRELIMINARY

Table 1-1 SPU Operating Mode Key Switch Defintions

Switch Position	Definition
OFF	Disables SPU and CPU access. This signals a power shutdown to the CPU. Hardware on the workstation interface board immediately resets the Bay Power Controllers (BPCs), which in turn reset their PPCs and removes power from each board in the system. Moving the keyswitch to one of the other positions does not cause power to return until the SPU software commands the BPCs to return power.
LOCAL	Allows both CPU and SPU access. Same as using ^P and ^D on the C100 or C200 SPU Console.
CPU ONLY	Allows only CPU access. This makes the SPU Workstation the equivalent of just another terminal tied to the operating system.
SECURE	SPU keyboard and mouse are mechanically disabled. This permits the SPU to continue with output, but user input is not possible. SPU software ignores the modem even if the modem key switch is in the SPU position.

Table 1-2 SPU Modem Key Switch Definitions

Switch Position	Definition
OFF	No modem access permitted.
SPU	Modem physically connected to SPU's serial port. Current SPU operating mode applies to SPU access via modem.
CPU	Modem physically connected to AUX serial port on workstation. External cable from AUX port to SYSTECH or VASYNC port will enable remote access to operating system regardless of SPU operating mode.

PRELIMINARY

Architecture Overview

2.1 Contents

- Register Sets
- Memory Management
- Multiprocessing
- Automatic Self-Allocating Processors
- Communication Registers
- Deadlock Detection
- CPU Timers

2.2 Overview

This chapter provides an introduction to the system architecture for readers wh have no prior experience with CONVEX computers. More knowledgeable readers will find it useful as summary or refresher material. This is not intendec to be an exhaustive, in-depth treatment of the subject. It is only an overview of some of the most important aspects of the system arhitecture. Key among them are memory management, the concept of automatic self-allocating processors, and the role of the communication registers.

2.3 Register sets

There are four register sets and two status registers. The four register sets are:

- address registers - eight 32-bit registers, called A0-A7.
- scalar registers - eight 64-bit registers, called S0-S7.
- vector registers - three registers and eight accumulators.
- communication registers - 4,096 registers that are either 32 or 64-bits wide.

The address, scalar, and vector registers are sometimes referred to as general registers; as their respective names indicate, these registers have been partition according to the type of data they handle. Each CPU has a set of general registe

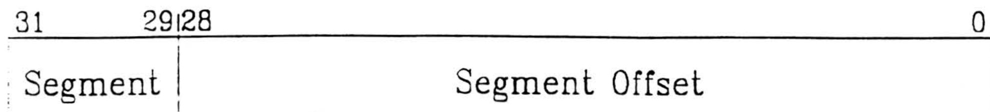
The fourth register set, the communication registers, provide communication a synchronization *between* processors during multiprocessing operations. The communcation registers physically reside on the CPU Utilities board. Thus the have a central location outside the individual processors.

Each CPU has two 32-bit status registers. They are:

- Program Counter (PC)
- Processor Status Word (PSW)

The 32-bit Program Counter (PC) contains the virtual address of the cur instruction. The PC format is shown below in Figure 2-1. The segment specifies one of eight 512 MByte segments of virtual memory; the offset field specifies the instruction address within the 512MByte segment.

Figure 2-1 Program Counter Format



The 32-bit Processor Status Word (PSW) contains flags that enable or disable exception processing and show the result of numerical operations. Appendix A contains a brief definition of the individual PSW bits. For a more detailed description of each bit, refer to the Architecture Reference Manual.

2.4 Memory management

The system architecture defines a four gigabyte virtual address space that is divided according to use. Memory management includes access control, virtual-to-physical address translation, and various mechanisms to assure efficient transition from one software operating context to another.

2.4.1 Virtual address space

Each processor can access up to four gigabytes of virtual memory. This address space is divided into eight segments, each of which is 512 Megabytes. The eight equal-sized segments are allocated to five memory partitions called rings. Rings provide a structure for specifying where different types of software and data must reside and makes it possible to control access where needed.

Table 2-1 shows how the rings and segments are allocated across the four gigabytes of virtual memory. As the table shows, four segments (0-3), which is one half the total available address space, are assigned to Rings 0-3, respectively. Rings 0-3 are for use by the operating system and its associated data. The other half of the address space, which is comprised of segments 4-7, are assigned to Ring 4. Ring 4 is for user software and data.

The virtual memory rings provide the means for a simple memory protection structure that assures easy detection and handling of protection violations. The ring structure is arranged so that the operating system kernel is located in the innermost ring (Ring 0), operating system data structures are located in Rings 1, 2, and 3; all user processes are located in the outermost ring (Ring 4). The privilege-level of a ring is inversely proportional to the ring number. Thus the operating system (Ring 0) has the highest privilege level.

2.4.2 Process structure and control

A *process* is one or more instruction streams (threads) that reside within a single virtual address space. Every process is defined by a *context*. There is both a hardware and a software context. The hardware context consists of the contents of all the registers, including the Program Counter and the Program Status Word. The software context includes all the program variables and other data within the user program as well as the operating system variables that support execution of the user program.

Every process also has *state* associated with it. The state of a process is the condition of the process at any given time. The state of a process changes in response to system events.

Table 2-1 Virtual Memory Rings and Segments

Ring	Segment	Owner
0	0 (512 MBytes)	Operating System
1	1 (512 MBytes)	Operating System
2	2 (512 MBytes)	Operating System
3	3 (512 MBytes)	Operating System
4	4 (512 MBytes)	User
	5 (512 MBytes)	User
	6 (512 MBytes)	User
	7 (512 MBytes)	User

Process control involves the use of *stacks*, *stack frames*, and process *return blocks* to manage process activity. Stacks contain the hardware and software context information as well as the current process state. All of this information is contained within the stack in units called *stack frames*. Return blocks, of which there are several different types, are simply data structures that are used to manage hardware context.

In general, stacks are used as dynamic storage that is allocated and deallocated during the execution of a user program. *Push* and *Pop* instructions are used to store and remove stack data.

2.4.3 Terminology

It is essential that some of the basic elements of the system be understood before a description of address translation will be meaningful. Below is a list of essential terms with definitions that have been reduced to their simplest form. For a more comprehensive description, refer to the Architecture Reference manual.

Thread - a single stream of instructions.

Process - a collection of instruction streams (threads) that reside within a single virtual address space and that share the same SDRs (defined below).

Page - a contiguous 4-Kbyte block of memory; both the virtual and physical address of a page are contiguous.

Page Frame - a page stored in physical memory.

Segment - a contiguous block (512 Megabytes) of virtual memory addresses.

Segment Descriptor Register (SDR) - contains address translation and validity information used in the first level of virtual-to-physical address translation.

Page Table - a memory resident (or, cache resident) table of address translation and validity information; the information is contained in 4-byte words that are called page table entries.

Page Table Entry (PTE) - contains address translation and validity information for one page frame; a minimum of two page table entries must be looked up to

complete the address translation process; a third PTE is required in some instances.

Communication Index Register (CIR) - defines which subset of the communication registers is being used by the program running on a CPU. Each CPU has one CIR, and each process has a different CIR value.

Thread Identifion (TID) Register - used to subdivide a process into disjoint threads. Up to 32 threads may be running in the same process (i.e., have the same CIR). The TID makes it possible to have a unique identifier for each thread. The TID is used primarily for operations that involve unshared memory. Unshared memory, in this case, is defined as one or more threads in a process using the same logical address to access different physical locations in memory.

2.4.4 Segment descriptor registers

A Segment Descriptor Register (SDR) is used in the first level of virtual-to-physical address translation. The SDR specifies what segment of virtual memory a process is associated with and whether or not the segment is valid.

The four gigabytes of addressable memory is divided into eight 512 Megabyte segments. There are eight SDRs, one for each of the segments. Each SDR is 32-bits long.

The SDRs are physically located in the communication registers. When a process is loaded for execution, the appropriate segment descriptors are loaded into the CPU's SDRs.

2.4.5 Page table entries

A PTE is a 32-bit word. It indicates the validity of a reference and is used in determining the physical memory location of a valid reference.

The second and third stages of virtual-to-physical address translation are accomplished using Page Table Entries (PTEs). These are similar in function to segment descriptors, which form the top-level of the translation tree.

If there are multiple threads within a process and the individual threads require unshared memory, then an additional level of translation is invoked using the Thread Identifier (TID) register and a third PTE. If the threads within a process all share the same physical memory, then no additional thread-level translation is needed.

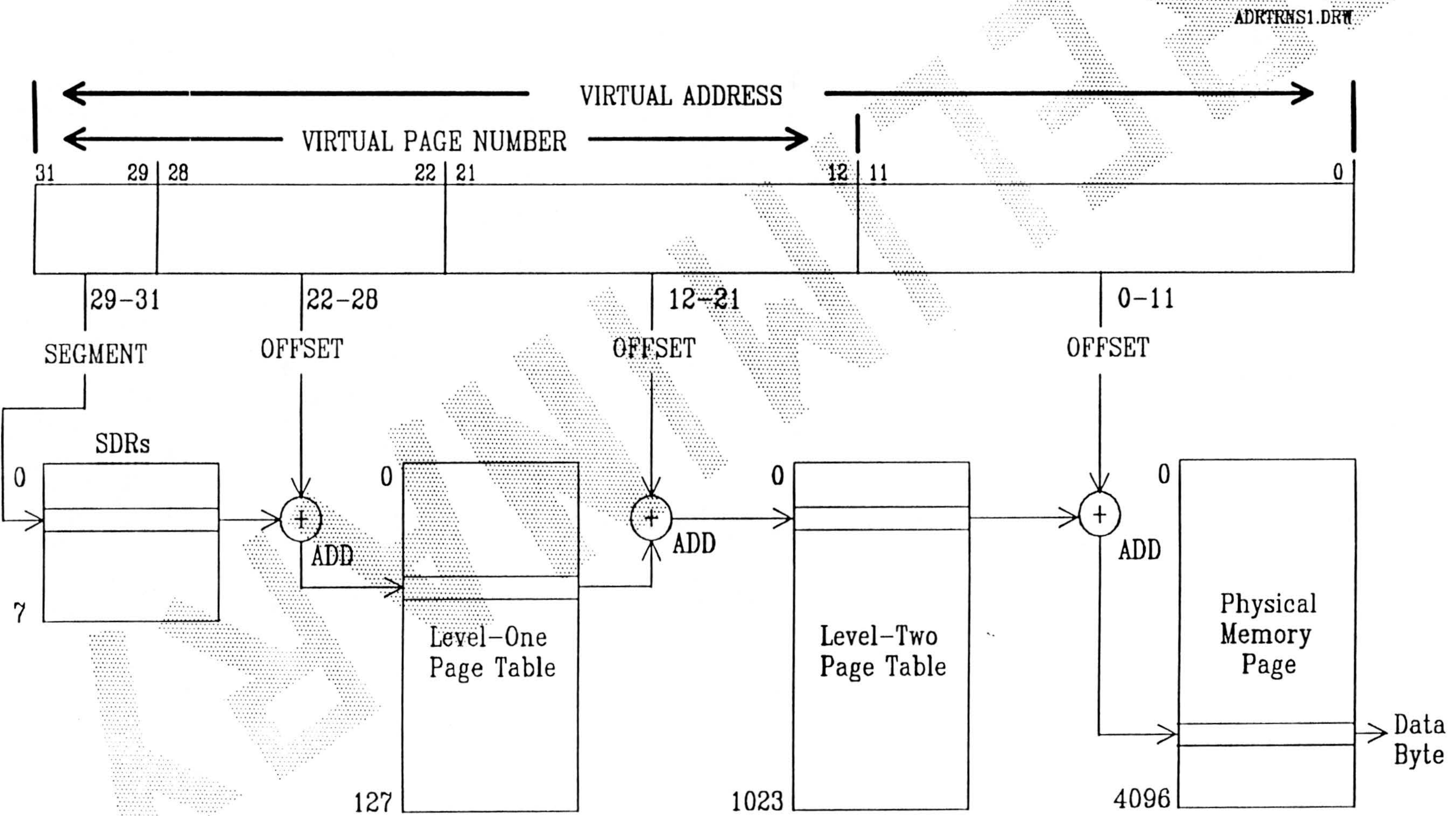
The C3800 series uses a scratch RAM and a PTE cache to speed up the address translation process. Both are physically located on the Scalar Processor board. The scratch RAM is used to store a copy of the SDRs and, in some cases, the first level PTE.

2.4.6 Address Translation

The basic steps involved in virtual-to-physical address translation are illustrated in Figure 2-2. The process involves accessing the appropriate SDR followed by a series of page table lookups. Each lookup is pointed to by the contents of the previous table entry and an offset value from the virtual address. Figure 2-2

PRELIMINARY / DO NOT COPY CONVEX COMPANY PRIVATE

Figure 2-2 Address Translation



shows that two levels of PTE access are required to complete the address translation and arrive at a physical memory address.

When a multi-threaded process contains threads that need to have their own unique region of unshared memory, an additional level of translation is required beyond those shown in Figure 2-2. In such cases, the contents of the Thread Identification (TID) register is used as an offset value, and a third PTE access is performed to make the virtual-to-physical memory address translation complete down to the individual thread level.

The address translation process is described in more detail in the Scalar Processor chapter of this manual.

2.4.7 Shared and Unshared Memory

Shared memory means that more than one thread uses the same virtual address to access the same physical locations in memory.

Unshared memory means that each thread uses the same virtual address to access different physical location in memory. The Thread Identification (TID) Register makes this possible. The TID modifies the virtual-to-physical address translation for the thread to allow each thread of a process to have private physical memory for variables that need to be unshared.

2.4.8 Reference and Modified Bits

Reference and Modified (R and M) bits are used by the operating system to track memory usage by the CPUs. One reference bit and one modified bit exists for each physical memory page (four KBytes). Unlike previous CONVEX computer systems, the R and M bits in the C3800 series physically reside in main memory.

The R and M bits are altered whenever a successful memory access occurs. Successful memory accesses are defined as ones that do not cause a PTE violation. Read, write, or execute sets the reference bit. A write access sets both the reference bit and the modified bit. In addition, the R and M bits can be directly accessed through read and write memory byte operations.

The R and M bits only reflect succesful memory access from the CPUs; they do not record memory access from the Input /Output subsystem.

2.5 Multiprocessing introduction

The C3800 Series permits a tightly-coupled set of processors to allocate and deallocate parallel code streams (threads) automatically at the hardware level without operating system intervention. No CPU ever has to wait for another CPU to become available; thus, full use is made of every available CPU execution cycle. This combination of multiprocessing and parallel processing is completely flexible and permits the software to run without regard for the number of available processors.

The following definitions apply:

- CPU — One physical Central Processing Unit
- Complex — The entire set of one or more physical CPUs in a configuration

- Process — A collection of one or more threads executing within a single virtual address space
- Thread — Any single instruction stream executing within a process
- Multiprocessing — The creation and scheduling of processes on a complex or any subset of a complex

2.6 Automatic self-allocating processors

Multiprocessing requires a method for scheduling use of the available processors in the system. The goal is to divide and conquer the workload. For example, a process that takes ten seconds of CPU time will run in five seconds if two processors are available and the work is equally divided.

The C3800 Series processors use a scheduling scheme called Automatic Self-Allocating Processors (ASAP). The fundamental characteristic of ASAP is that each CPU within the complex is solely responsible for scheduling itself (that is, associating and disassociating itself from an executing process). There is no master process for finding idle CPUs and scheduling processes or threads.

The ASAP mechanism also allows the operating system to schedule threads. Both the operating system and the processor microcode can create and terminate threads independently.

ASAP makes it possible for CPUs to migrate automatically between processes so any combination of serial and parallel processs may execute simultaneously.

All CPU events generated locally, such as system calls or page faults, are processed within the CPU that initiated those events. Externally generated events, such as interrupts, are delivered to any available CPU that is currently accepting them. This approach eliminates the need for a master CPU and permits each CPU to function independently.

Figure 2-3 illustrates the significant advantage that the use of automatic self-allocating processors has over conventional forms of parallel processing.

2.6.1 CPU states

A CPU always operates in one of two states:

- **Allocated**
- **Idle**

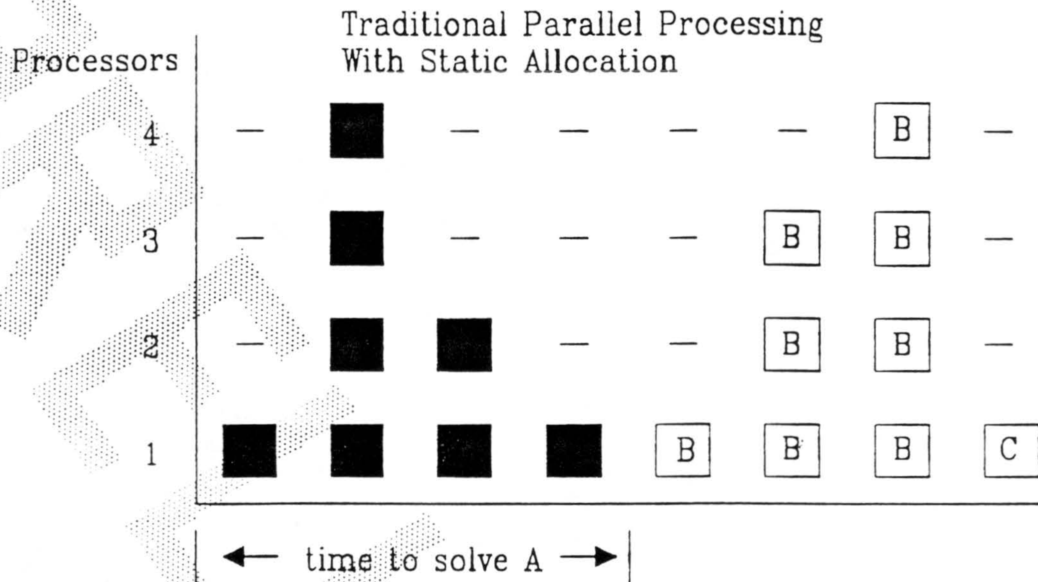
An allocated CPU is one that is currently executing a thread within a process. An idle CPU executes a microcode idle loop in search of a process or thread in need of service. Only two events cause the processor to exit the microcode idle loop: (1) detecting a process or thread that requires service or (2) an interrupt.

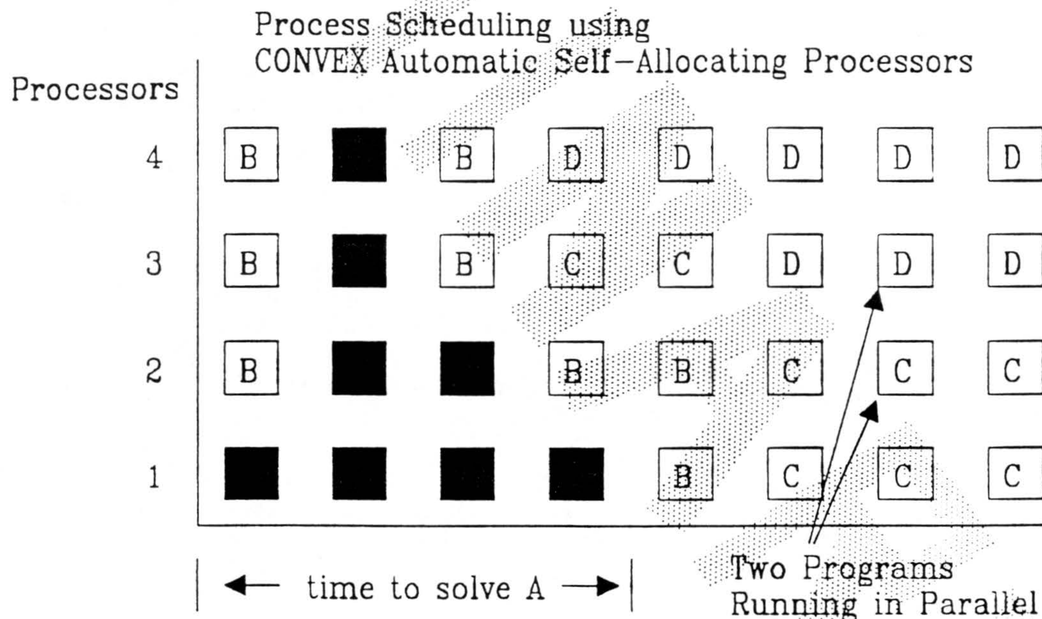
2.6.2 CPU scheduling

Two different types of scheduling are used to start CPU execution on behalf of a process. One occurs when the CPU changes from the idle to allocated state (*called thread creation*). The other type occurs when the CPU stops executing one process and starts executing a new one (*called context switching*).

Both transitions involve the hardware communication registers and the communications index register. All process context necessary for a threads

Figure 2-3 Processing Comparison





PRELIMINARY

execution becomes available to a CPU when a communication register set is bound to it. The act of binding a CPU to a communication register set establishes a process context for thread execution. A communication register set is bou loading the Communication Index Register (CIR) located in the CPU with a communication register set index. Once a CPU's CIR is loaded, it immediately shares all process context with any other CPU whose CIR contains the same index.

PREL

2.6.3 Forking

Allocating a CPU to a new thread involves a process called forking. A fork is an event with 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 it would like to have more CPUs assist in the work. This is a request, not a demand; that is, if there are no available CPUs the posting CPU does not wait until there is one; it just continues with its thread of execution. This satisfies the goal that programs written to exploit available multithreading must work if only one CPU is available.

User software can use the following isntructions to manage CPU processing functions without operating system intervention.

All user CPU management functions are supported by the following five instructions:

- pfork — Post a *fork* event (request allocation of a CPU)
- spawn — Post a *fork* event for multiple CPUs (request allocation of CPUs)
- cfork — Clear a *fork* event (clear need for CPU allocation).
- wfork — Wait for a *fork* event (deallocate CPU and wait for a *pfork*)
- *join* — Wait for a *fork* event if the executing thread is not the last thread (conditional deallocation of a CPU)

Additional instructions are available for use only by the operating system. They include the *idle* instruction (briefly described later in this section) and instructions for loading, storing, and moving process context information. For more information regarding the use of these instructions, consult the Architecture Reference Manual chapter, "Multiprocessing Management."

There are two types of forks that may be posted: (1)request for a single CPU to initiate a thread or (2) a request for as many CPUs as are available to initiate threads. This posting is done by executing a *pfork* or *spawn* instruction. These instructions differ only in the number of CPUs they request. The pfork instruction requests a single CPU, while spawn requests all available CPUs. These instructions load a group of communication registers (fork event registers) with enough information to start a thread. This information consists of a program count to execute from, an initial Program Status Word (PSW) value, and stack, frame, and argument pointers to define local memory structures. The addressing of communication registers here is CIR based, so the fork is posted relative to a particular process. Idle processors scan through the fork event registers in each CIR, looking for a posted fork. If one is found, the idle processor enters that CIR and loads the information from the 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 processors to take. In addition, the *cfork* instruction is prov to clear a fork.

At the end of a thread of execution, each CPU may terminate its thread (relinquish and deallocate the CPU). There are three instructions to do this:

- wfork
- join
- idle (a privileged instruction)

The *wfork* terminates a single thread of execution initiated through *pfork*. After a thread is terminated, 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. The last CPU to reach the join instruction continues to execute instructions after the join. Thus, the process continues executing as a single thread after the join.

The operating system uses the *idle* instruction to reschedule the CPU. Executing this instruction sends the CPU into an idle loop where the CPU searches for posted forks in the fork event registers of each CIR. The CONVEX Architecture Reference Manual describes these instructions in greater detail.

2.6.4 Parallel processing

The ASAP mechanism provides for two types of parallel processing. They are called:

- Symmetric
- Asymmetric

In symmetric parallel processing, all threads in a process execute the same instruction stream. Upon completion, each thread executes a join instruction, which forces a multi-threaded process back to a single thread. Symmetric parallel processing is what the compiler typically uses to parallelize loops. For example, a loop that requires ten iterations can be spread across two CPUs (using *spawn* and *join*). The loop iteration count is shared between the two CPUs, and each CPU computes five of the iterations.

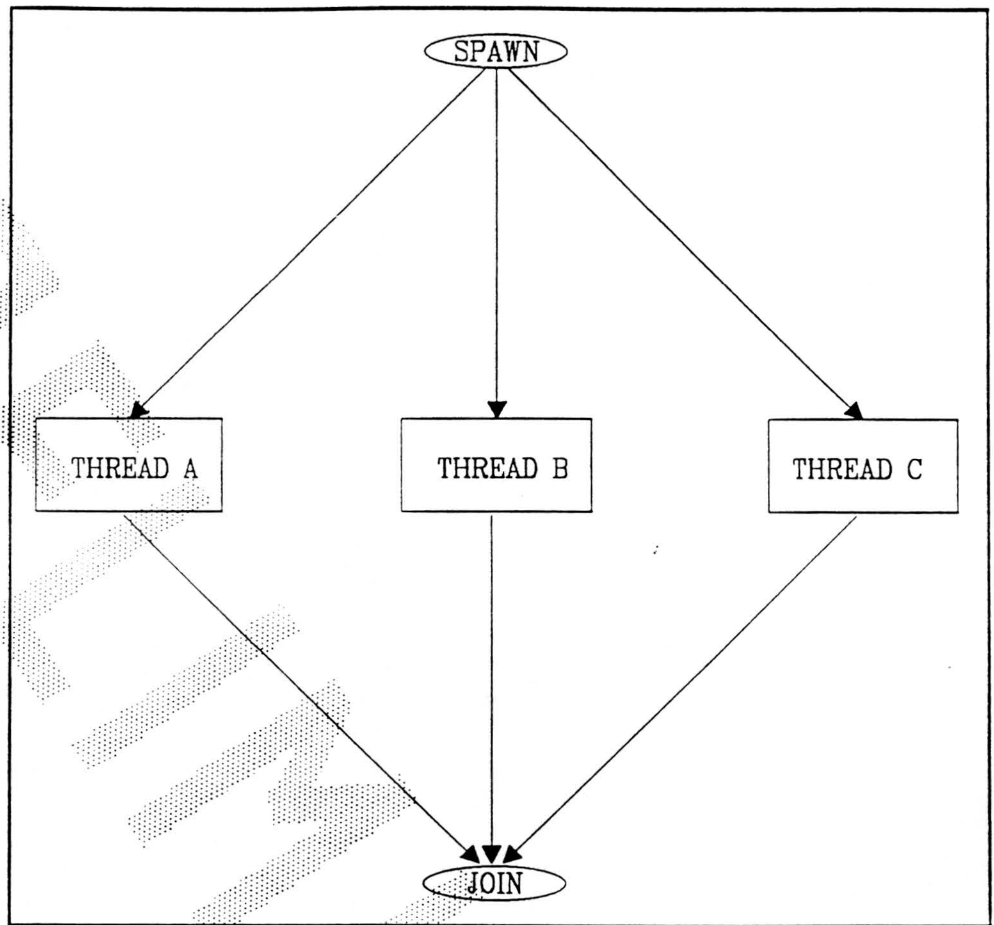
Figure 2-4 provides a symmetric parallel processing example.

In this figure, a single process changes from single-threaded to parallel and back again. The initial single thread posts a need for more threads by executing a spawn. Each thread that enters the process in response to the spawn leaves the fork posted. When the first thread completes its job, it executes the join instruction; join 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 analogous to the *fork* system call construct under UNIX. The child thread may or may not communicate with the parent; the parent thread may terminate leaving the child thread in execution; and either thread may fork additional threads.

In general, asymmetrical threads are disjoint, executing different code streams with different data, but within the same process address space. Asymmetric

Figure 2-4 Symmetric Parallel Processing



PRELIMINARY

processing differs from the multi-threaded 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. An asymmetrical thread is initiated with a CPU requesting the execution assistance of ***one*** other CPU as opposed to a CPU requesting execution assistance of ***all*** available CPUs as implemented by the *spawn/join* instruction pair.

With asymmetric parallel processing, the pfork instruction is used to post a fork for a another single CPU to execute. Normally, this will be a separate instruction stream that can be carried to completion independent of the posting CPU. The key point is that the multiple threads within the process execute different functions. The accepting CPU terminates the additional single thread with the wfork instruction. If the fork is not taken by another CPU, then the posting thread (CPU) should clear the fork with *cfork.*

Normally, the compiler does not use asymmetric parallel processing. However, it may be used by the operating system and some user applications.

Figure 2-5 shows an asymmetric parallel process with three threads.

Thread B posts the need for an independent thread to be run concurrently. Thread A is started, and notifies B that it is running. B finishes its work and terminates. Thread A posts the need for a new thread C to perform another task. Thread C starts, and notifies thread A that it is executing; Thread A completes and terminates. Thread C determines all tasks have been completed and clears the need for another thread and continues on as a single threaded process.

2.7 Communication registers

The communication registers are a single set of registers shared by the entire complex for communication between the threads of a process. All CPUs in the complex have equal access to the communication registers. Threads within a process communicate by sending and receiving data through these registers. A special set of instructions control communication register access.

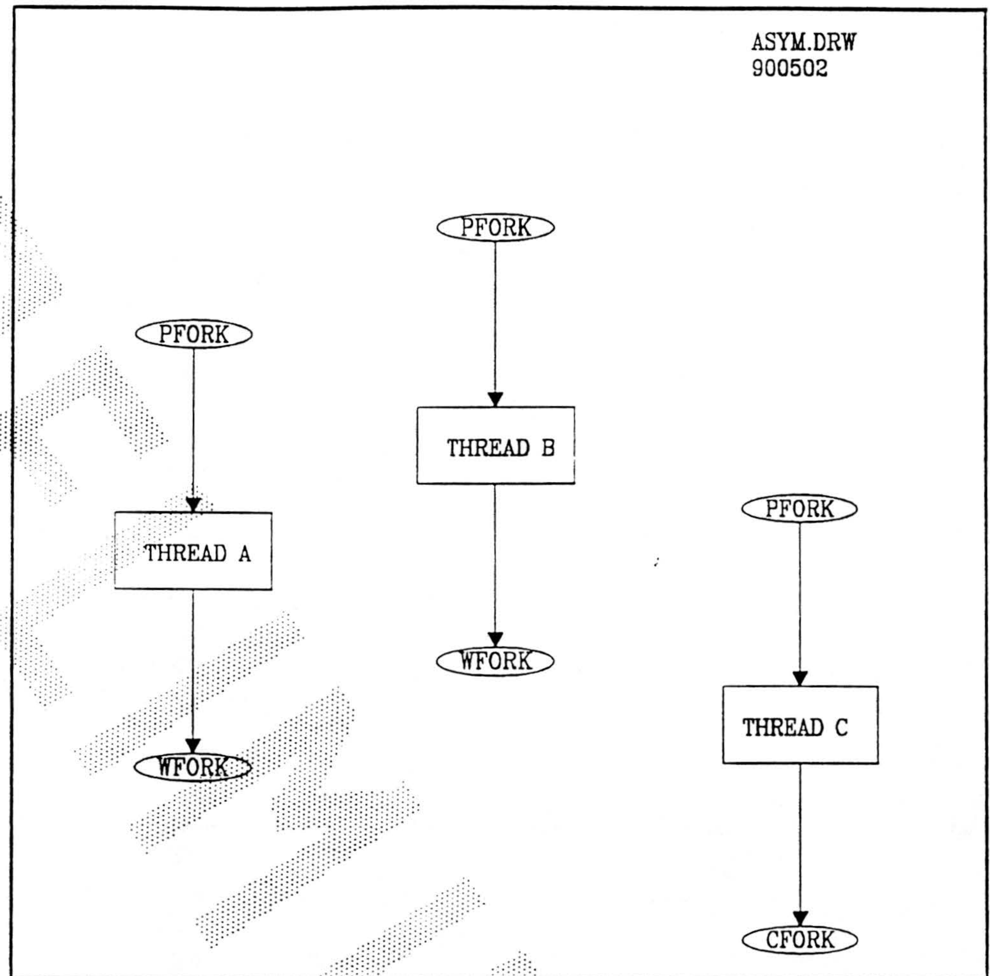
A communication register is visible from software as an addressable 32-bit or 64-bit register with an associated lock bit. The data portion may be manipulated by some instructions that do not examine the state of the lock bit. Lock bits are: (1) used by both software and hardware as a binary semaphore and (2) manipulated by the communication register instruction set to control and synchronize access by multiple CPUs to each communication register.

2.7.1 Communication index registers

The communication registers are divided into groups, called partitions. There is one partition for each process, and each partition contains information, called context information, specifically related to a particular process.

The Communication Index Register (CIR) specifies which partition is associated with a particular process. The CIR contains an index value that binds a particular set of physical communication registers to a process. Each executing process is associated with a different CIR index. Figure 2-6 provides an example of CIR-to-Register Set binding. This example is only one of many possible binding combinations.

Figure 2-5 Asymmetric Parallel Processing



PRELIMINARY

Figure 2-6 CPU-to-Register Set Binding

PRELIMINARY

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.

There is one CIR per CPU. Each CIR is a five-bit register and may represent 32 different index values. It should be noted that the five-bit CIR does not limit a complex to managing and executing a total of only 32 processes; 32 is simply the maximum number of processes that can be mounted at any given time.

A process whose state is currently represented by a partition of the communication register set is said to be *mounted* on the CPU complex. Any CPU can execute any mounted process by changing the index value in its CIR to reference the partition describing the process. This action binds a communication register set to a CPU when a CPU mounts and begins executing a thread. When multiple CPUs in the complex are executing multiple threads of a process, these threads use the communication registers within the partition to communicate with and synchronize operations with each other.

Multiple CPUs can be bound to the same communication register partition; this occurs when each CPU loads its CIR with the same index value.

2.7.2 Communication register partitions and rings

There are 4,096 communication registers divided among 32 partitions. Each partition contains a total of 128 communication registers. Each partition is further divided into protection rings. The rings are designated for specific use and labeled: Ring 0 Hardware, Ring 0 Software, andRing 4 Software. Figure 2-7 illustrates the protection rings for register set two. Each register set contains the same protection ring division as that shown in the illustration for register set two.

The communication registers are divided among the rings within each partition:

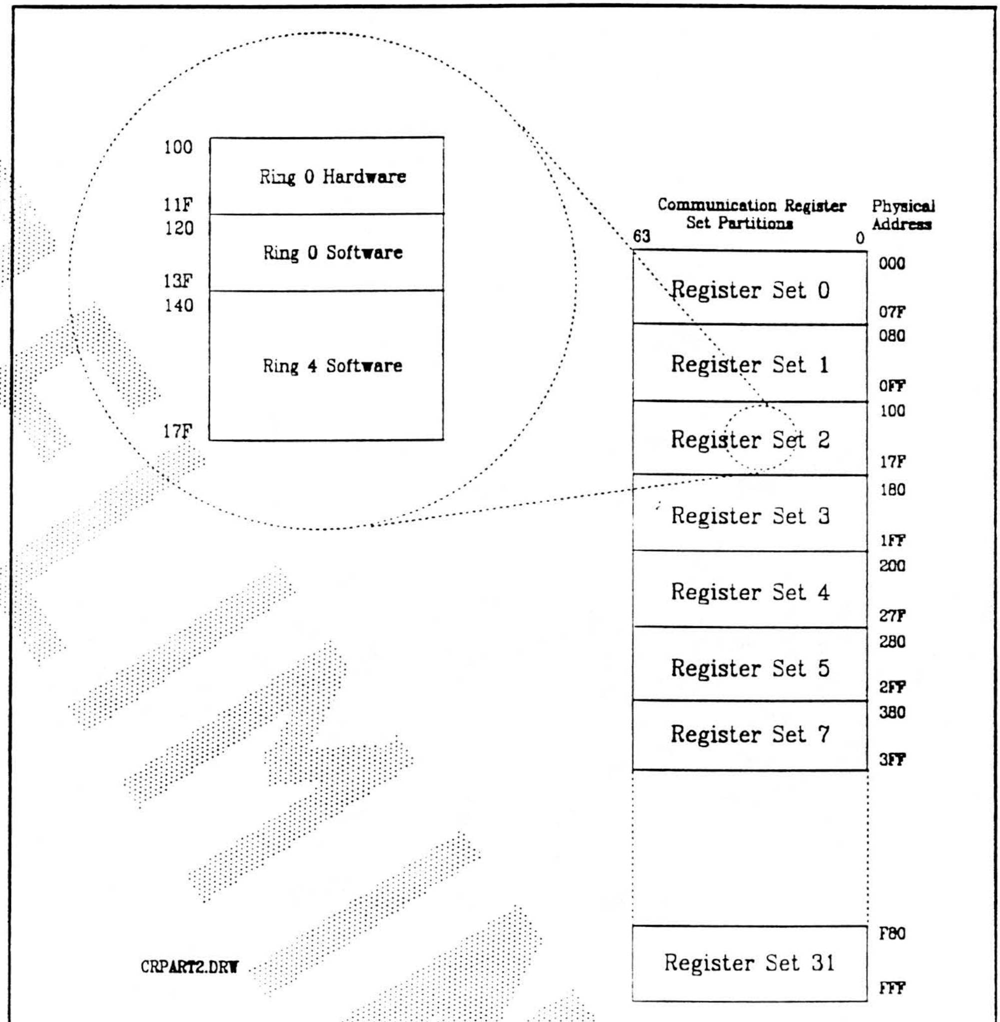
- Ring 0 Hardware - contains 32 communication registers (each partition)
- Ring 0 Software - contains 32 communication registers (each partition)
- Ring 4 Software - contains 64 communication registers (each partition)

Across all 128 partitions, Ring 0 Hardware and Ring 0 Software each have 1,024 communication registers; Ring 4 Software has 2,048 communication registers. The total equals 4,096.

Every communication register has a specific physical address. Each ring is assigned specific communication register address ranges within the partition. Figure 2-8 illustrates the virtual address ranges for each ring, and Table 2-2 lists the physical address ranges divided among the different rings and partitions. A description of communication register addressing is provided later in this chapter.

Ring 0 programs have access to all communication registers and can use any virtual communication address with no protection checking. However, an *invalid communication address* system exception occurs if a process executing in Ring 4 violates its ring protection boundary.

Figure 2-7 Communication Register Partitions



PRELIMINARY

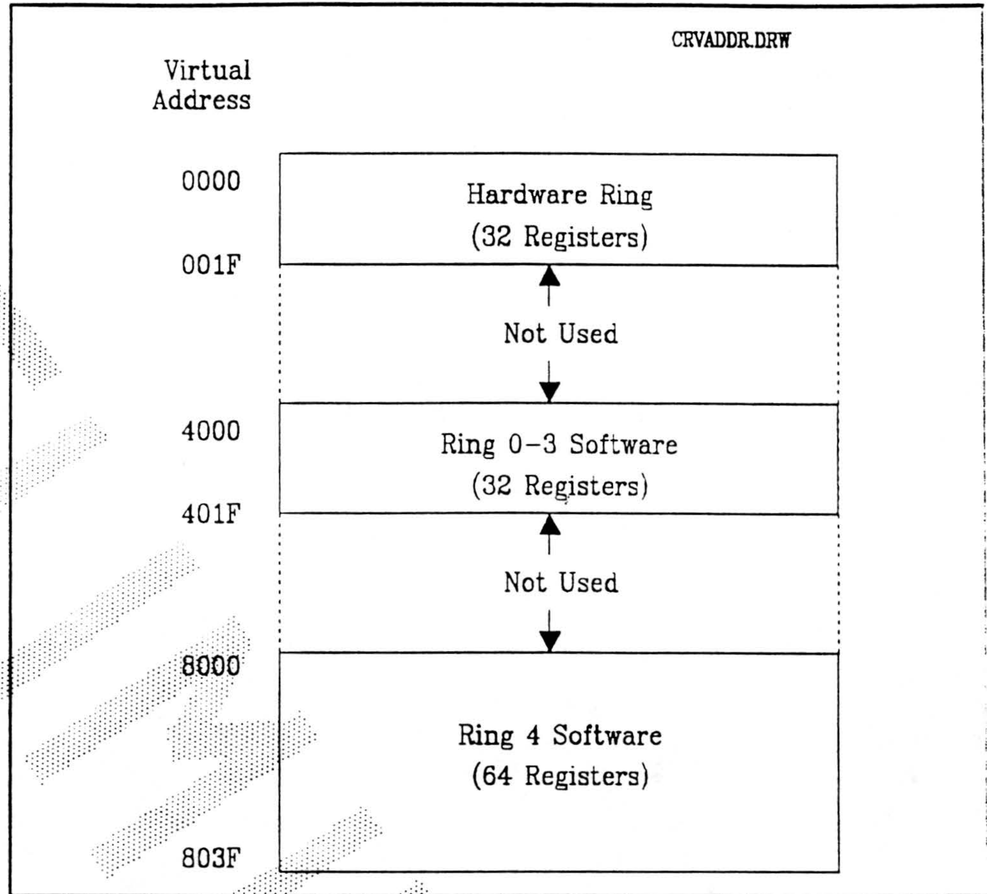


Figure 2-8 Comm Registers Virtual Address Range

PRELIMINARY

Table 2-2 Ring Physical Address Ranges

Communication Register Set Partition	Ring 0 Hardware Physical Address Ranges	Ring 0 Software Physical Address Ranges	Ring 4 Physical Address Ranges
00	000-01F	020-03F	040-07F
01	080-09F	0A0-0BF	0C0-0FF
02	100-11F	120-13F	140-17F
03	180-19F	1A0-1BF	1C0-1FF
04	200-21F	220-23F	240-27F
05	280-29F	2A0-2BF	2C0-2FF
06	300-31F	320-33F	340-37F
07	380-39F	3A0-3BF	3C0-3FF
08	400-41F	420-43F	440-47F
09	480-49F	4A0-4BF	4C0-4FF
10	500-51F	520-53F	540-57F
11	580-59F	5A0-5BF	5C0-5FF
12	600-61F	620-63F	640-67F
13	680-69F	6A0-6BF	6C0-6FF
14	700-71F	720-73F	740-77F
15	780-79F	7A0-7BF	7C0-7FF
16	800-81F	820-83F	840-87F
17	880-89F	8A0-8BF	8C0-8FF
18	900-91F	920-93F	940-97F
19	980-99F	9A0-9BF	9C0-9FF
20	A00-A1F	A20-A3F	A40-A7F
21	A80-A9F	AA0-ABF	AC0-AFF
22	B00-B1F	B20-B3F	B40-B7F
23	B80-B9F	BA0-BBF	BC0-BFF
24	C00-C1F	C20-C3F	C40-C7F
25	C80-C9F	CA0-CBF	CC0-CFF
26	D00-D1F	D20-D3F	D40-D7F
27	D80-D9F	DA0-DBF	DC0-DFF
28	E00-E1F	E20-E3F	E40-E7F
29	E80-E9F	EA0-EBF	EC0-EFF
30	F00-F1F	F20-F3F	F40-F7F
31	F80-F9F	FA0-FBF	FC0-FFF

2.7.3 Communication register addressing

Each communication register is addressable with two communication addresses; one is CIR based (virtual) and one is CIR independent (physical).

Virtual communication register addressing uses the address (Ceffa) contained in the assembly language instruction, the base address for the particular ring, and the CIR index value to generate the correct physical communication register address. Figure 2-9 illustrates the components of this virtual-to-physical addressing scheme. Virtual-to-physical translation of communication register addresses is a function of the processor's address generation logic. Note that Ceffa is assembly language syntax; it means Communication Register effective address. Ceffa originates from the communication register instruction that is about to be executed by one of the processors. As Figure 2-9 illustrates, Ceffa is used as an offset value for the physical base address of the particular ring and register set involved.

CIR-independent access to any physical communication register set is possible through the Ring 0 virtual address space. This is possible regardless of (and in addition to) the current communication register set mapping. Physical addressing is accomplished by defining a fixed virtual-to-physical translation for a portion of the virtual address space assigned to the hardware communication registers. This mapping allows Ring 0 software to access all communication register sets regardless of which communication register set is currently bound to the CPU through its CIR index.

2.7.4 Communication registers modified bits

The communication registers contain information about the current process. The communication registers are saved and restored by the operating system when a process is rescheduled. Rescheduling is defined as terminating the current process and loading information for a different process into the communication registers.

The communication registers have a structure, called *modified bits*, to facilitate save and restore operations for the communication registers. The communication registers modified bits are similar in function to the memory reference and modify bits. The hardware uses the communication register bits to save and restore only those communication registers that have been modified. Each individual register does not have a modified bit; instead, a modified bit covers a contiguous region of the communication register address space. Any time a communication register or lock bit in the particular region is altered, the modified bit corresponding to that region is set.

2.7.5 Hardware communication registers

Half of the Ring 0 communication address space is allocated to hardware. These registers are used by the hardware and Ring 0 software to implement multithreaded execution.

The hardware communication register set contains all process-specific states necessary to schedule a process and create or terminate executing threads. This register set is only accessible from Ring 0 and is the primary structure for process scheduling. Figure 2-10 illustrates the hardware communication registers in relation to the overall communication registers hierarchy.

Figure 2-9 Comm Reg Virtual Addressing Scheme

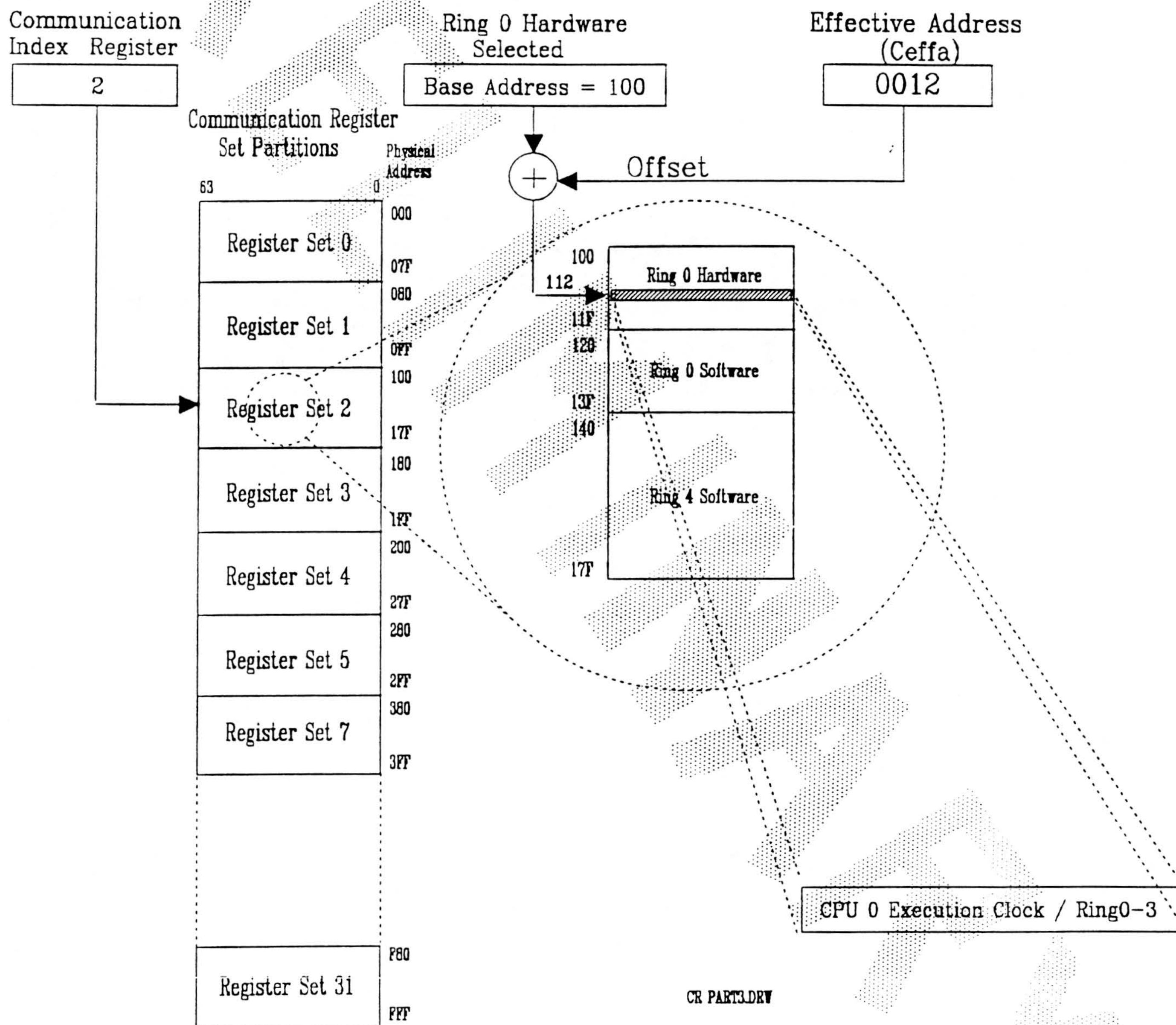
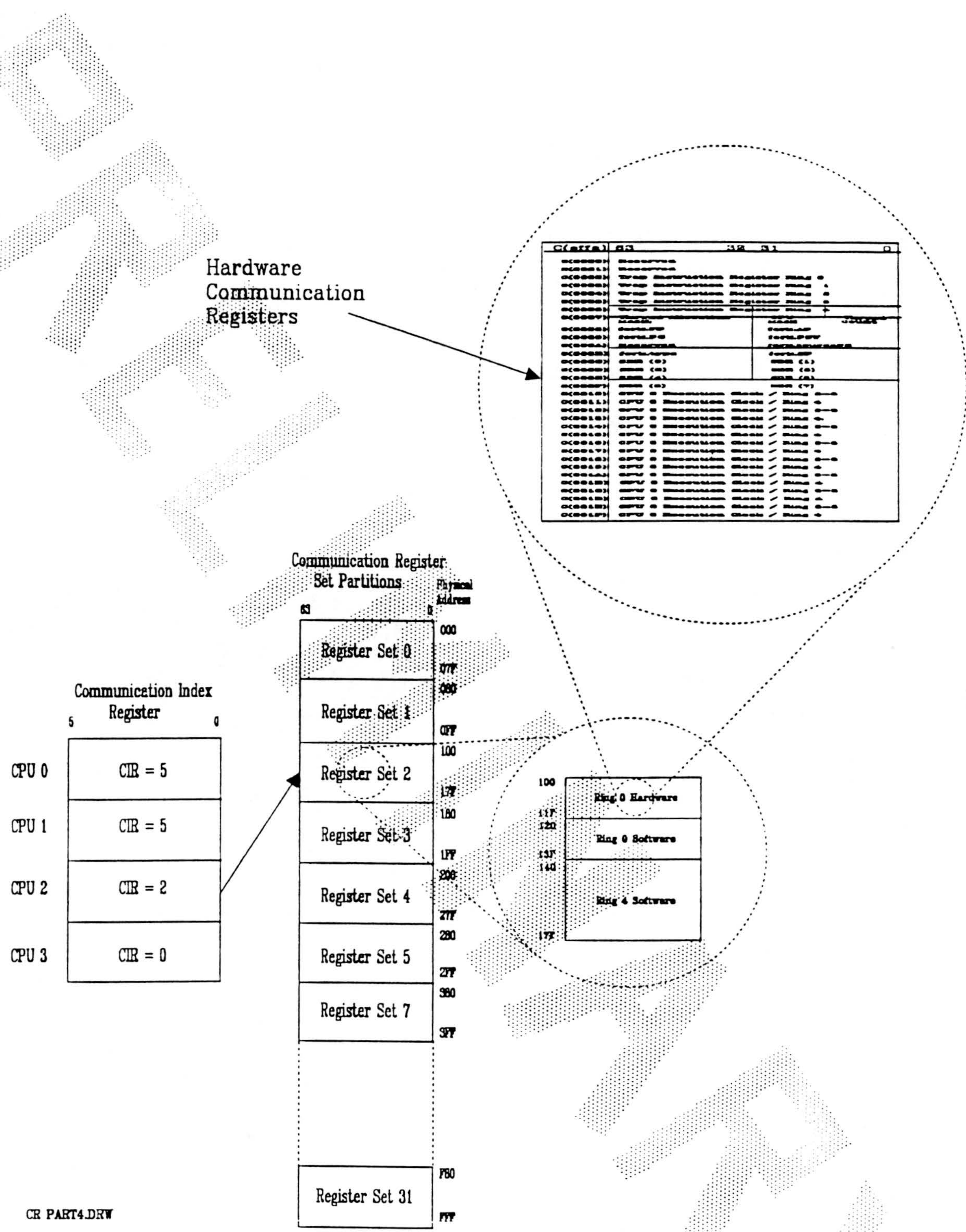


Figure 2-10 Communication Register Hierarchy



The text that follows provides a brief description of the hardware communication registers. The purpose is to give the reader a very general overviewof their functions. If you need more detailed descriptions of the hardware communication registers, refer to the *Architecture Reference Manual.*

Table 2-3 lists the hardware communication registers along with the effective address (Ceffa) of each register. Ceffa is assembly language syntax; it means Communication Register effective address. For more information regarding communication register addressing, refer to that topic title in this chapter.

2.7.5.1 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* instructions. The *trap* instruction can selectively force all threads sharing the same communication register set to enter the exception handler. The *pbkpt* instruction is used to force all threads within the current process to stop execution.

The lock bits on the TIRs are ignored.

For more definitive information, refer to the "Exceptions and Interrupts" and "Instruction Set" chapters in the Architecture Reference Manual.

2.7.5.2 Thread allocation mask and count

The *thread allocation mask* is a 32-bit mask and is the primary means for defining how many active threads exist for a particular process. Each bit position in the thread allocation mask represents a unique thread identification (ID), which allows a process to create up to 32 unique threads.

To create a thread (CPU transition from idle to allocated), a unique thread ID is generated by clearing a single bit in the thread allocation mask; this is performed by the CPU idle loop (the idle loop searches the communication register sets for a posted fork event). The CPU Thread ID (TID) register is then loaded with an allocated thread ID to identify the new thread throughout its existence. When a CPU changes from allocated to idle, it sets the bit associated with the CPU's TID register in the thread allocation mask; this is a function of the CPU idle loop.

The *allocated thread count* is a 16-bit integer that specifies 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 is terminated, the thread count is decremented. The thread count is used in conjunction with the *join* instruction; it can also be used to determine the current multithreading extent of a process. The lock bit for this register is shared with the thread allocation mask and is governed by the protocol defined for it.

The lock bit for the thread mask and thread count (thread allocation register) is interpreted as a "valid" bit, so this register can be manipulated with *snd* and *rcv* operations. This lock bit is the central synchronization point for all fork operations. An idle CPU waits until it can successfully receive this register to ensure that a valid fork is taken, and then allocates a thread to the fork. By locking the thread mask/count (i.e., making the thread count/mask unreceivable), software can ensure that no forks are accepted in that communication register set.

2.7.5.3 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. Basically, thread creation begins when one process executing on a CPU requests the

Table 2-3 H/W Communication Register Allocation

C(effa)	63 32	31	
C(0000)	Reserved		
C(0001)	Reserved		
C(0002)	Trap Instruction Register Ring 0		
C(0003)	Trap Instruction Register Ring 1		
C(0004)	Trap Instruction Register Ring 2		
C(0005)	Trap Instruction Register Ring 3		
C(0006)	Trap Instruction Register Ring 4		
C(0007)	Thread Allocation Mask	CPU Mask	Thread Count
C(0008)	fork.FP	fork.AP	
C(0009)	fork.PC	fork.PSW	
C(000A)	Reserved	fork.source_PC	
C(000B)	fork.type	fork.SP	
C(000C)	SDR (0)	SDR (1)	
C(000D)	SDR (2)	SDR (3)	
C(000E)	SDR (4)	SDR (5)	
C(000F)	SDR (6)	SDR (7)	
C(0010)	CPU 0 Execution Timer Register / Ring 0-3		
C(0011)	CPU 0 Execution Timer Register / Ring 4		
C(0012)	CPU 1 Execution Timer Register / Ring 0-3		
C(0013)	CPU 1 Execution Timer Register / Ring 4		
C(0014)	CPU 2 Execution Timer Register / Ring 0-3		
C(0015)	CPU 2 Execution Timer Register / Ring 4		
C(0016)	CPU 3 Execution Timer Register / Ring 0-3		
C(0017)	CPU 3 Execution Timer Register / Ring 4		
C(0018)	CPU 4 Execution Timer Register / Ring 0-3		
C(0019)	CPU 4 Execution Timer Register / Ring 4		
C(001A)	CPU 5 Execution Timer Register / Ring 0-3		
C(001B)	CPU 5 Execution Timer Register / Ring 4		
C(001C)	CPU 6 Execution Timer Register / Ring 0-3		
C(001D)	CPU 6 Execution Timer Register / Ring 4		
C(001E)	CPU 7 Execution Timer Register / Ring 0-3		
C(001F)	CPU 7 Execution Timer Register / Ring 4		

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 listed and defined as follows:

- fork.FP — initial frame pointer for the thread
- fork.AP —initial argument pointer for the thread
- fork.PC — program count to begin execution of the thread
- fork.PSW — initial program status word for the thread
- fork.source_PC —program count for the thread posting the fork
- fork.SP — initial stack pointer for the thread
- fork.type — defines the fork type of a posted fork to prevent mixing *pfork, spawn,* and *join* instructions in a multithreaded process; this parameter is passed from posting to acceptance of the fork; fork types include: *pforked, spawned, and stopped.*

When a fork is posted with *pfork* or *spawn*, the program count of the instruction following the *pfork* or *spawn* instruction is loaded into *fork.source_PC* (located in the fork event registers). When a fork is taken, the value in *fork.source_PC* is loaded into the an idle CPU's program counter to establish a current ring of execution as that CPU changes to the active state. A current ring of execution must be established since an idle CPU has no state.

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, posted-to-taken, and taken-to-cleared.

Forklck is a lock bit on the *fork.FP* and *fork.AP* register combination. When this lock bit is a one, the hardware is changing the fork from clear-to-posted or from posted-to-taken.

Forkposted is a lock bit on the *fork.type* and *fork.SP* register combination. When this lock bit is a one, there is a valid fork posted that is ready to be taken.

2.7.5.4 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. These registers are accessed with *put* and *get* operations.

2.7.5.5 CPU execution timer registers

The CPU execution timers (CTRs) are 64-bit microsecond timers used to track execution time per CPU within each ring. These clocks are updated on ring crossings, CIR changes, and when communication register state is saved. Execution of the *ctrsg* instruction forces an update of the timer. Software must ensure that these timers are updated to obtain an accurate reading.

More information on this subject is provided later in this chapter.

2.8 CPU deadlock detection

The hardware can detect when the currently executing threads within a process have reached a deadlock condition. By definition, a process deadlock occurs when all the currently executing threads of a process are doing a synchronization instruction followed by a branch back to that instruction. Deadlocks are considered system exceptions and are passed to the process deadlock handler for resolution.

Synchronization instructions attempt to change the value of a lock bit and return status on the success or failure of the lock/unlock operation. Examples include the *tas*, *snd*, *rcv*, and *inc* instructions. For a complete list of the*synchronization* instructions refer to the Architecture Reference Manual. It should be noted that these instructions are sometimes referred to as deadlock detection instructions.

These instructions all perform some sort of semaphore or synchronization operation and return status to the Program Status Word.

When a deadlocked process is detected, each thread within the process immediately enters the Ring 0 process deadlock handler. The process deadlock handler schedules other threads within the process to resolve the deadlock condition.

The concept of deadlock also extends to certain cases of thread termination. For example, if the last thread in a process executes a *wfork* instruction and no other fork is posted in the CIR, a *last thread termination* deadlock has occurred.

Another instance of improper thread termination that will cause a deadlock involves execution of a *wfork* instruction in place of a *join* instruction.

The fundamental rule here is that 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.

2.9 CPU timers

Each CPU contains ___ timers, which are listed here and described in the text that follows.

- Execution Timer
- Thread Timer
- Other

2.9.1 Execution Timer

To provide accurate accounting information to the operating system, CPU Execution Timers (CTR) are included for each process; the timers track (in microseconds) the time each CPU spends in Ring 0 (system time) and Ring 4 (user time). The CTRs are visible to the operating system in the hardware communication registers. There is a set of these timers for each CIR.

The CPU microcode maintains the CTRs on a demand basis. For example, if a CPU is running in Ring 0 and executes a *get* of the Ring 0 CTR for the CPU, the time will not be current.

The events that cause the microcode to update a particular timer are primarily ring crossings and execution of the *ctrsg* instruction. The operating system uses the *ctrsg* instruction to instruct all CPUs to update their CTRs. For example, if

one thread of a process is the operating system kernel executing on CPU0 (in Ring 0) and the other thread is a user program in Ring 4, the operating system thread can execute *ctrsg* and then read the Ring 4 or Ring 0 CTR and know that the time is accurate.

The CPU execution timer includes a single hardware *delta timer* on each CPU that counts by microseconds and is clearable. An update of the CTR is performed by reading the current CTR value from the communication register, adding the current delta time to the CTR value, and writing the sum back to the communication register. The delta timer is cleared immediately afterward.

2.9.2 Thread timer

Each CPU contains one 64-bit microsecond timer per thread . The *Thread Timer (TTR)*, which is implemented in microcode and accessed by nonprivileged instructions, 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 whenever a CPU is executing a thread. It can be read or written at any time by the currently executing thread.

The thread timer is updated by the same delta timer used for the CTRs. The delta timer is a microsecond timer that exists on each CPU. It is used to time intervals between accesses to the Thread Timer(TTR) or CPU Execution Timer (CTR). The thread timer is updated by adding the delta timer count to the current TTR's value and clearing the delta timer.

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.

2.9.3 CTR and TTR timer updating

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.

PRELIMINARY

Instruction Processor

3

Contents

3.1 Overview

The instruction processor performs the following major functions::

- Fetches intstructions for the instruction cache
- Maintains the program counter
- Dispatches the scalar processor
- Decodes instrutions???
- Generates memory fetch addresses?? (for instructions only???)

The instruction processor can be restarted from three different address sources: (there are more than three; see block diagram)

The instruction processor hardware consists of the following major functional units:

- Instruction Cache
- Branch History Cache
- Lookahead Address Logic
- Pre-crack Logic
- Memory Interface
- Instruction Valid Logic
- Instruction Parsing Logic
- Parse Address Logic
- Branch Tags
- Instruction Dispatch Information
- Instruction Queue

3.2 Instruction Cache

PRELIMINARY

Scalar Processor

4

Contents

- Overview
- Major Functions
- Major Units
- Scalar Data Structure
- Address Generation Logic
- Data Return Queue
- Scalar Processor Microsequencer
- Scalar Processor Operations

4.1 Overview

The information in this chapter provides a brief description of all the major hardware units that comprise the scalar processor. The last section in the chapter describes the major operations performed by the scalar processor, including sequence of events and data flow.

The Neptune Scalar Processor physically resides on the scalar board along with the instruction processor. The instruction processor is described in a separate chapter.

The vector processor is dispatched from the scalar processor instead of from the instruction processor. Two separate 64-bit buses are used to transfer data to/from the scalar processor (or memory) and the vector processor. A 32-bit bus is used to transfer load and store addresses to the vector processor.

4.1.1 Major Functions

The Scalar Processor performs the following major functions:

- Executes Scalar Instructions
- Performs Address Generation and Translation
- Dispatches the Vector Processor

4.1.2 Major Units

The major units that comprise the scalar processor are listed below and illustrated in a simplified block diagram (see Figure 4-1).

- Data Return Queue
- Scalar Data Structure
- Address Generation Logic
- Scalar Processor Microseqencer

A description of each functional unit is provided in the sections that follow.

4.2 Data Return Queue

The memory return data queue accepts read data from the memory system and transfers it to the appropriate destination. A 24-entry queue exists for memory

Figure 4-1 Scalar Processor Simplified Block Diagram

FROM MEMORY

SPSIMPL2
900618

INSTRUCTION PROCESSOR

ADDRESS GENERATION LOGIC

DATA RETURN QUEUE

MUX

DATA STRUCTURE

ADDRESS

DATA

TO MEMORY

MICRO-SEQUENCER

ADDRESS

DATA

DATA

TO VECTOR PROCESSOR

FROM VECTOR PROCESSOR

PRELIMINARY

return data. The queue allows the processor to accept all memory return data independent of whether the final destination (i.e.,vector processor, instruction processor, scalar register file, or data cache) is ready.

4.3 Scalar Data Structure

Scalar Data Structure includes the following major components. Refer to Figure 4-2.

- Register File
- ALU
- Data Cache
- Floating-Point and Miscellaneous Integer Function Units

4.3.1 Register File / ALU Gate Array

The RF/ALU gate array contains a two-stage register file and the ALU data path. The register file has thirty-two 36-bit registers organized as:

1. Eight 32-bit Address (A) registers

2. Eight 64-bit Scalar (S) registers

3. Eight 32-bit temporary registers

The register file has three 64-bit input ports (A, B and C) for writing results from the A, B, and C buses. The A bus inputs results from the integer ALU logic. This bus requires no arbitration since writes are controlled by the microcode. The floating-point / miscellaneous integer unit writes results to the register file using the B bus. The gate arrays within the floating point unit must arbitrate for access to the B bus. The C port is used to input cache read data, vector-to-scalar processor transfers, and memory return data. Transfers from these three sources must arbitrate for use of the C bus.

4.3.2 Data Cache

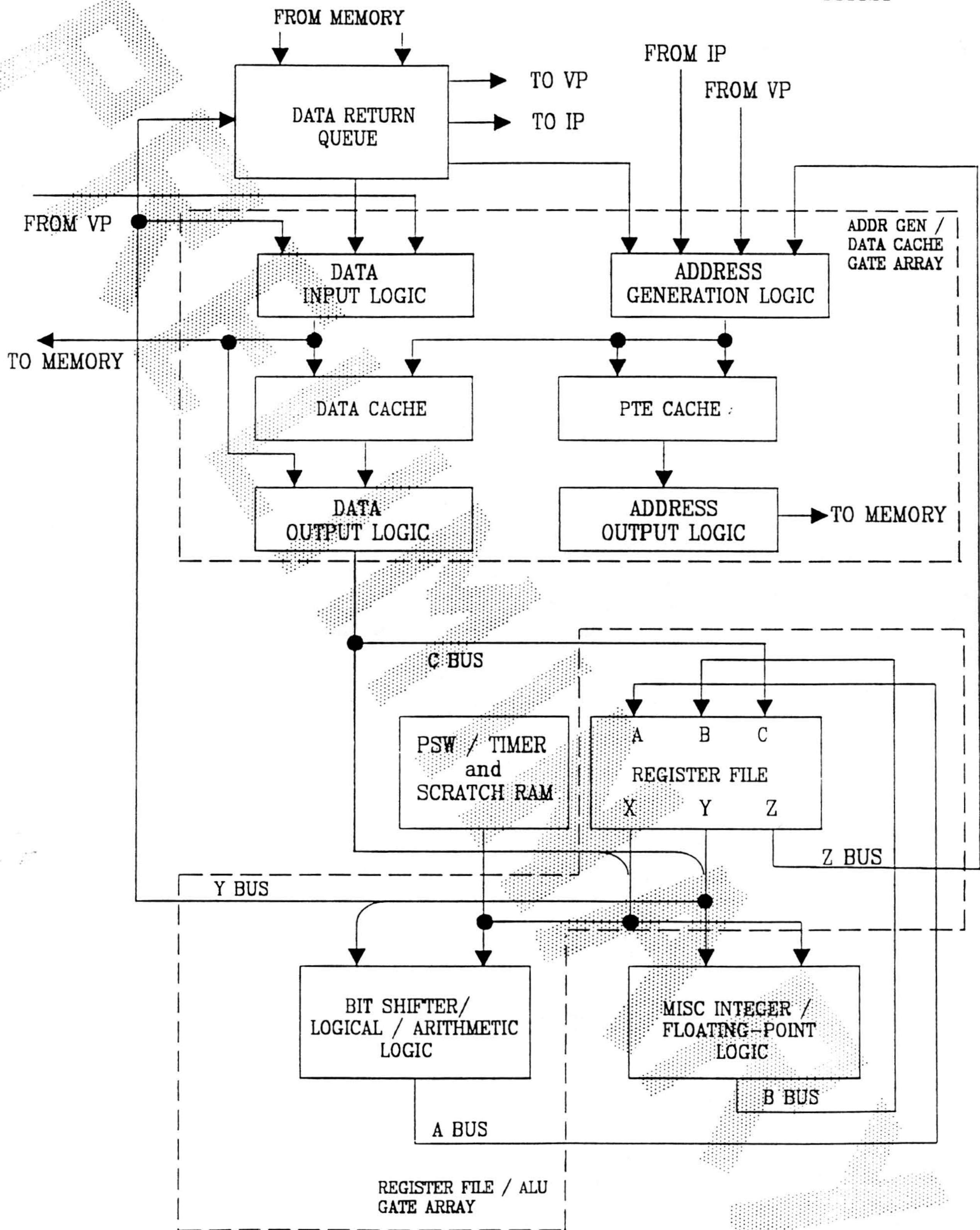
The data cache contains operand data that is used during the execution of scalar and vector instructions.

The data cache has a 16K Byte capacity; thus four pages of data may reside in the cache simultaneoulsy. The cache is organized as an even and an odd bank of 2K-locations by 32 bits each. The odd and even cache banks have their own tags and validity bits and function independent of each other.

4.3.3 Floating-Point and Miscellaneous Integer Function Unit

PRELIMINARY

Figure 4-2 Address and Scalar Data Structure



4.4 Address Generation Logic

The address generation logic accepts memory requests from the scalar processor and the instruction processor. The address generation logic is contained in the Address Generation / Data Cache gate array. See Figure 4-2.

4.4.1 PTE Cache

The Page Table Entry (PTE) cache contains physical memory addresses that are used during logical-to-physical address translation. Each PTE cache consists of a logical PTE tag, two validity bits, a physical PTE translation address, and reference and modified bits. The physical page address (address bits xx-yy) is read from the PTE cache and then combined with the offset within the page(address bits aa-bb) to form the complete physical address that is sent to memory.

4.4.2 Scratch RAM

The scratch ram contains a copy of the segment descriptor registers (SDRs) and serves as a first-level PTE cache.

4.5 Scalar Processor Microsequencer

The scalar processor's microsequencer controls the order in which operations inside the scalar processor are performed. The microsequencer includes the following primary components:

- 4K-location control store
- Microprogram counter logic
- Eight-deep stack for the microprogram counter
- Test condition and branch select logic
- Microinstruction Register

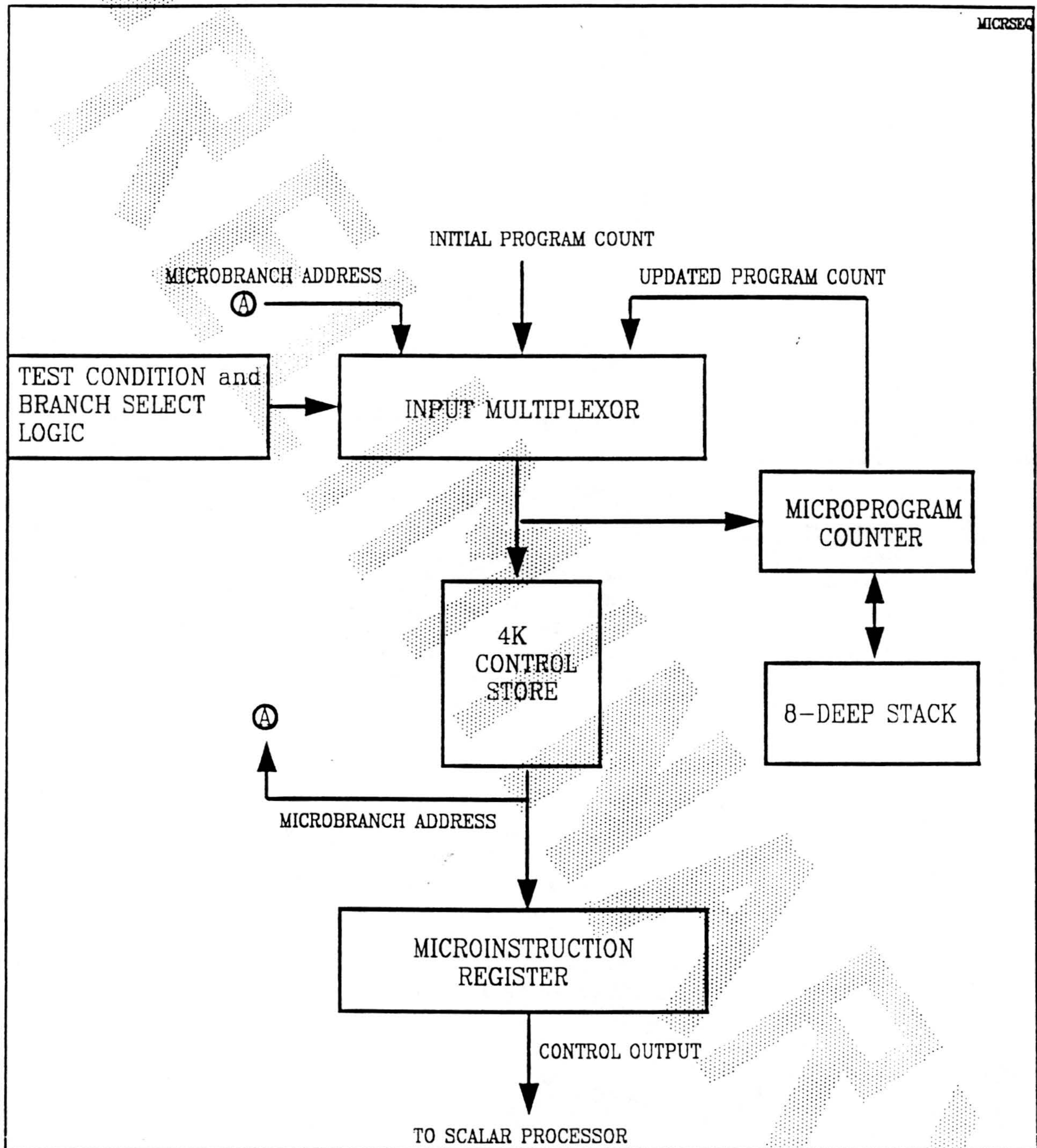
A block diagram of the scalar processor microsequencer is provided in Figure 4-3.

Microcode for each scalar instruction executable by the scalar processor resides in the control store. Each time the instruction processor decodes a scalar instruction, it will send the microcode entry point for that particular instruction to the scalar processor where it will be used to access the appropriate control store starting location. That same value is also input to the microprogram counter logic. That initial program count value is incremented once each machine cycle until the operation is completed or until a branch or interrupt occurs.

4.5.1 Source Hazard Logic

A hazard is defined as a condition that will temporarily halt execution of an instruction until the hazard is cleared. A hazard condition is not always an error condition. In some cases, for example, a hazard is simply a flag that is set to indicate that something must happen (an operand returning from memory) before execution of a given instruction can continue. Operations unrelated to the hazard conditon continue without interruption.

Figure 2-3 Microsequencer



The purpose of the hazard logic in the scalar processor is to assure that all resources required to successfully execute a given instruction are available. Items checked include the following:

1. Microinstruction register is valid.

2. If a macro-PC branch occurred previous to this instruction, assure that it branched correctly.

3. The operands are ready.

4. If this instruction specifies writing to a register file, assure that the register file is not hazard locked.

5. If this instruction requires accessing the PSW, assure that it is not hazard locked.

6. Assure that all selectable hazard conditions are cleared.

4.6 Scalar Processor Operations

Scalar Processor operations include address translation, cache read, indirect cache read, cache write, memory read, and scalar instruction execution. This section describes the basic steps involved in each of these operations.

4.6.1 Address Translation

This section describes how virtual-to-physical address translation is performed.

4.6.1.1 Memory Management Terminology

It is essential that the basic elements of the memory management system be understood before a description of address translation will be meaningful. Below is a list of essential terms with definitions that have been reduced to their simplest form. For a more comprehensive description of memory management refer to the Architecture Reference manual.

Thread - a single stream of instructions.

Process - a collection of instruction streams (threads) that reside within a single virtual address space and that share the same SDRs (defined below).

Page - a contiguous 4-Kbyte block of memory; both the virtual and physical address of a page are contiguous.

Page Frame - a page stored in physical memory.

Segment - a contiguous block (512 Megabytes) of virtual memory addresses.

Segment Descriptor Register (SDR) - contains address translation and validity information used in the first level of virtual-to-physical address translation.

Page Table - a table (memory resident?) of address translation and validity information; the information is contained in 4-byte words that are called page table entries.

Page Table Entry (PTE) - contains address translation and validity information for one page frame; a minimum of two page table entries must be looked up to

complete the address translation process; a third PTE is required in some instances.(see thread-level PTE).

Communication Index Register (CIR) - defines which subset of the communication registers is being used by the program running on a CPU. Each process has a different CIR value.

Thread Identifier (TID) - used to subdivide a process into disjoint threads. Up to 32 threads may be running in the same process (i.e., have the same CIR). The TID makes it possible to have a unique identifier for each thread. The TID is used primarily for operations that involve unshared memory. Unshared memory, in this case, is defined as one or more threads in a process using the same logical address to access different physical locations in memory.

4.6.1.2 Basic Steps

There are either five or seven basic steps involved in the address translation process. Figure 4-4 provides a flow chart of the basic steps, and figure 4-5 gives a more detailed view of the same process. The text that follows outlines the seven basic steps.

1. Access SDR. The segment number is used to access the appropriate segment descriptor register (SDR). The segment number is specified in the three most significant bits (31 .. 29) of the virtual address. Go to step 2.

2. Access first-level PTE. The SDR contains a base address (called page frame base). A seven-bit offset value taken from the virtual address is added to the page frame base to arrive at the effective address of the first page table entry. Go to step 3.

3. Access second-level PTE. The first page table entry also contains a page frame base. The page frame base and a ten-bit offset value taken from the virtual address are added together to find the effective address of the second page table entry. Go to step 4.

4. Is the thread-level PTE flag set? If no, go to step 5. If yes, go to step 6.

5. Access physical memory. The second page table entry also contains a page frame base. This page frame base and a 12-bit offset value taken from the virtual address are added together. The result is a 32-bit physical memory address. Go to step 8.

6. Access thread-level PTE. The second page table entry contains a 24-bit page frame base. This page frame base and a five-bit offset value taken from the Thread Identification (TID) register are added together to create the effective address of the thread-level page table entry. Go to step 7.

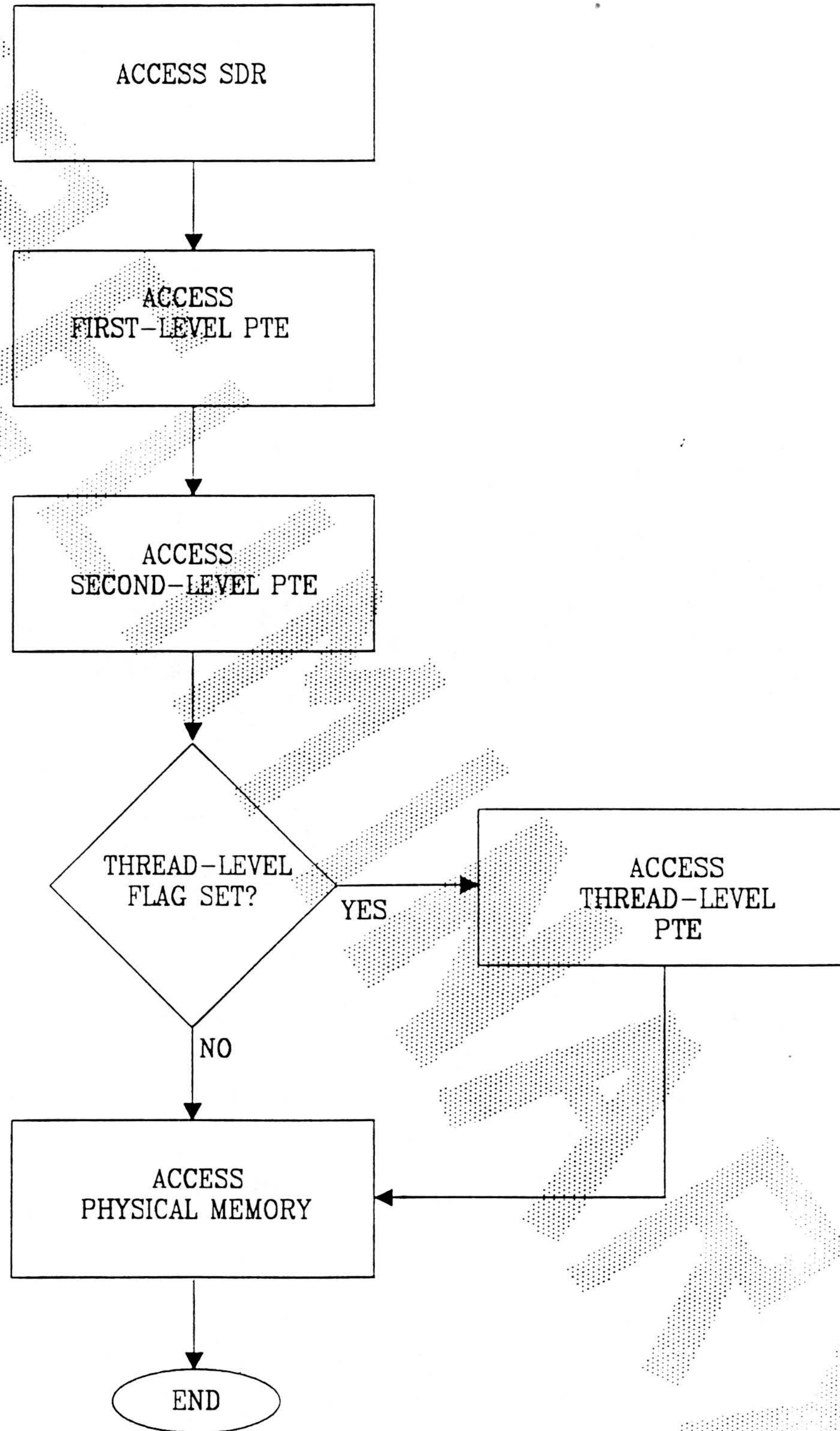
7. Access physical memory. The thread-level page table entry contains the physical memory, base address for a specific thread. This base address and a 12-bit byte offset value taken from the virtual address are added together. The result is a 32-bit physical memory address. Go to step 8.

8. Address translation completed.

4.6.1.3 Segment Descriptor Registers

The 4 Gigabytes of addressable virtual memory is divided into eight 512 Megabyte segments. There are eight SDRs, one for each of the segments. Each SDR is 32-bits long.

Figure 2-4 Address Translation Basic Steps



PRELIMINARY

Figure 1-4, Address Translation Basic Steps

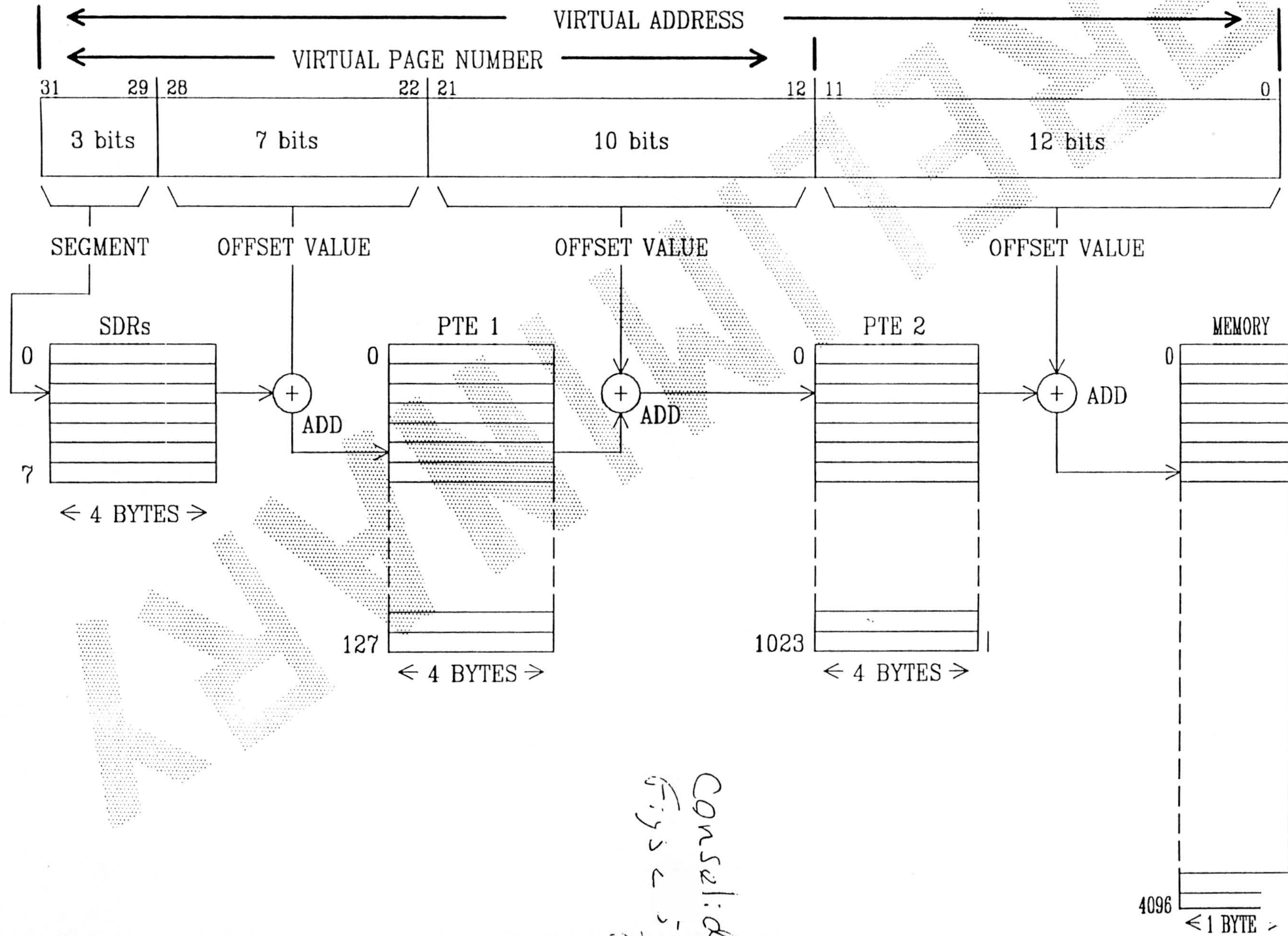


Figure 2-5 Address Translation (w/o Thread)

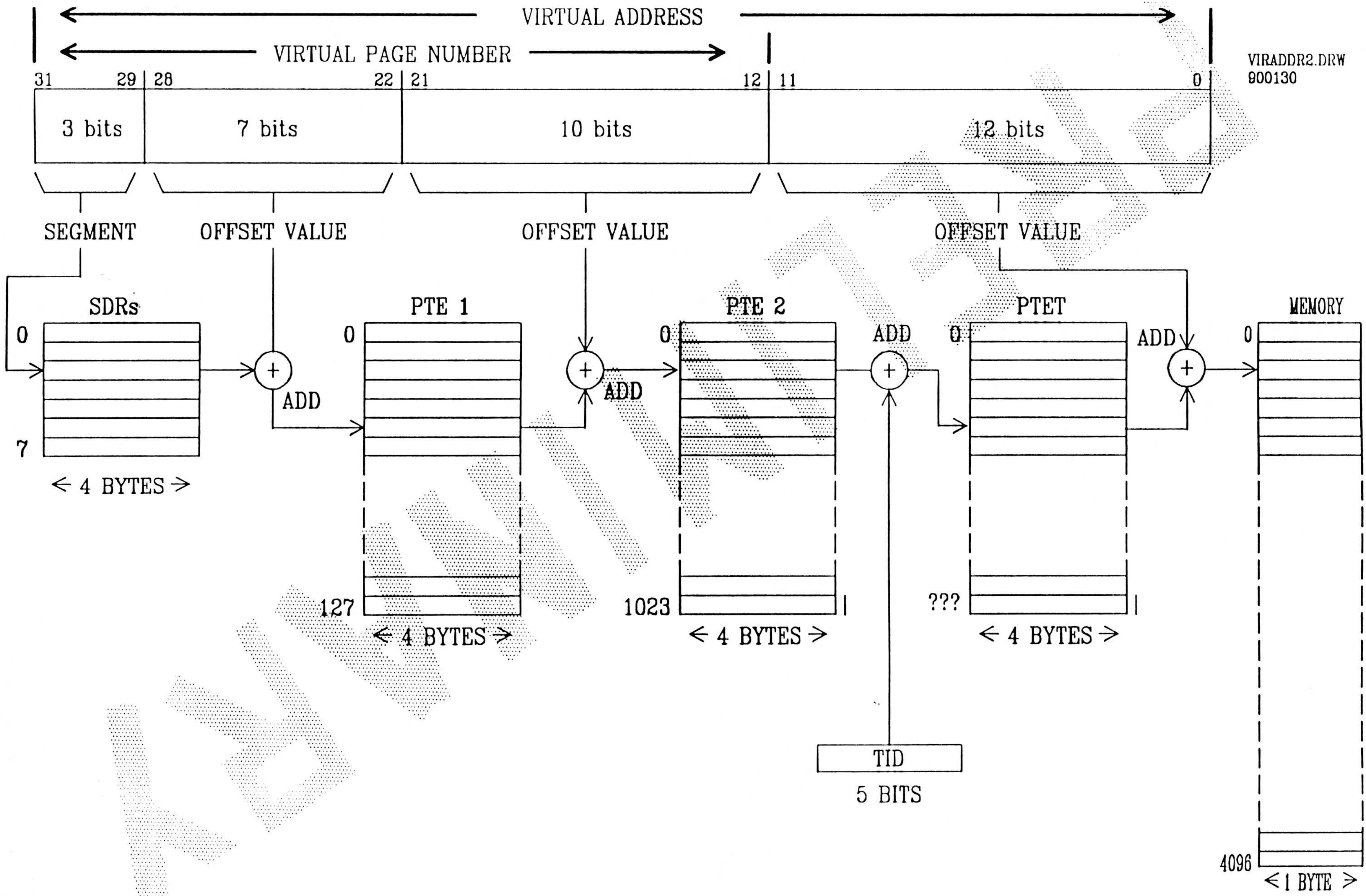


Figure 2-8 Address Translation (with Thread)

When a process is loaded for execution, the appropriate segment descriptors are loaded into the CPU's SDRs.

Each segment descriptor register points to the beginning of a first-level page table, and contains protection information as well.

4.6.1.4 Page Table Entries

The second and third stages of virtual-to-physical address translation are accomplished using Page Table Entries (PTEs). These are similar in function to segment descriptors, which form the top-level of the translation tree.

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

An additional level of lookup tables is needed in order to accommodate multiple threads within a process. The PTE is one of 128 entries for the first index-level, one of 1,024 entries for the second index-level, or one of 32 entries for the thread index-level.

In order for each thread within a process to have a unique address space, a thread-level PTE is indexed by the Thread Identification (TID). Refer to Chapter 5, "Multiprocessor Management," for more information regarding thread identification.

If the level T bit (bit) of the second-level PTE is set, the page frame base in t second-level entry is expanded to bits .8 and is used as the base address of a TID-indexed table of thread-level entries. These thread-level entries provide the final-level of translation to the data page.

If the level T bit is clear, only bits .12 of the second-level entry are used as the page frame base of the data page. The TID does not enter into the translation since these pages do not require any thread-level translation.

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

4.6.1.5 Address Translation Hardware Operation

The scalar processor has a scratch ram that contains a copy of the segment descriptor registers and serves as a first level PTE cache. An associated validity bit indicates whether the first level PTE has been encached.

The first level PTE valid bit will be used as a test condition. On a PTE miss, first the scratch ram will be accessed to obtain the first level page table entry. If the validity bit indicates that it has not been encached yet then the appropriate SDR in the scratch ram will be accessed and used to access main memory for the first level PTE. The first level PTE will be written in the scratch ram with its validity bit set. Once a valid first level PTE is obtained, main memory is accessed for the second level PTE (and possibly the thread level PTE).

(C3 Spec page 2-7 is source for paragraph that follows.) Each PTE Cache (there is one odd and one even PTE cache) consists of a logical PTE tag, validity bits, a physical PTE translation address, and reference and modify bits. The physical page address is staged after being accepted from the ram and then combined with the offset within the page to form a complete physical address.

[The C200 Series architecture has one 64-bit microsecond timer per thread implemented in microcode that is accessed by nonprivileged instructions. The thread timer (TTR) allows each thread to determine the CPU execution time of any code region without the overhead of a system call. This register only reflects the CPU time on a ring-specific basis and cannot be used to time inner ring calls. (For more see Arch Ref manul page 5-39)

Virtual Address Format			
Ring	Virtual Address (bits 31 - 0)		Effective Virtual Address
	Segment (bits 31 - 29)	Segment Byte Offset (bits 28 - 0)	
0	0	0000 0000 - 1FFF FFFF (512 MBytes)	0000 0000 - 1FFF FFFF
1	1	0000 0000 - 1FFF FFFF (512 MBytes)	2000 0000 - 3FFF FFFF
2	2	0000 0000 - 1FFF FFFF (512 MBytes)	4000 0000 - 5FFF FFFF
3	3	0000 0000 - 1FFF FFFF (512 MBytes)	6000 0000 - 7FFF FFFF
4	4	0000 0000 - 1FFF FFFF (512 MBytes)	8000 0000 - 9FFF FFFF
	5	0000 0000 - 1FFF FFFF (512 MBytes)	A000 0000 - BFFF FFFF
	6	0000 0000 - 1FFF FFFF (512 MBytes)	C000 0000 - DFFF FFFF
	7	0000 0000 - 1FFF FFFF (512 MBytes)	E000 0000 - FFFF FFFF

4.6.2 Cache Operations

Cache operations include ....

4.6.2.1 Cache Read

Cache read operations...

Table x-y Cache Read

Micro Program Count Level	Microinstruction Register 1 Level	Microinstruction Register 2 Level	
Cycle 1	Cycle 2	Cycle 3	Cycle 4
Read register file	Generate effective address	Access PTE / DC	Enable C bus write
Output data through ZBus port	Enable address select	Generate hit	Enable read bypass

4.6.2.2 Indirect Cache Read

Indirect cache read operations ....

Table x-y. Indirect Cache Read

Microprogrm Level	Microinstruction Register 1 Level	Microinstruction Register 2 Level				
Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7
Read register file	Generate effective address	Access PTE / DC	Enable C bus bypass	Generate effective address	Access PTE / DC	Enable C bus write
Output data through ZBus port	Enable address select	Generate Hit		Enable address select	Generate Hit	Enable read bypass

4.6.2.3 Cache Write

Cache write operations ....

Table x-y. Cache Write

Cycle 1 Micro Program Count Level	Cycle 2 Microinstruction Register 1 Level	Cycle 3 Microinstruction Register 2 Level	Cycle 4	Cycle 5
Read register file	Generate effective address	Access PTE / DC	Merge zone/data	Issue memory request
Output data through ZBus port	Enable address select	YBus ROT ????	Write data cache	Crossbar transfers data to memory
			Generate physical address	

4.6.2.4 Cache Purge

Cache purge is used to ....

4.6.3 Memory Operations

Memory operations include ....

4.6.3.1 Memory Read

Memory read operations ....

Table x-y. Memory Read

Micro Program Count Level	Microinstruction Register 1 Level	Microinstruction Register 2 Level					
Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8
Read register file	Generate effective address	Access PTE / DC	Generate physical address	Issue memory request	Crossbar transfers data from memory	Data enters return queue	Bypass Cbus write
Output data through ZBus port	Enable address select	Generate miss		Crossbar transfers memory request to memory	Issue memory ready	ROT ???	

4.6.3.2 Memory Write

Memory write operations ....

4.6.4 Scalar Instruction Execution

Scalare instruction executions are ....

Table x-y. Scalar Instruction Execution

Instruction Dispatch		Micro PC Level	Microinstruction Register Level 1	Microinstruction Register Level 2	
Board Transition	Select Control Store Address	Access Control Store	Check for Branch or Hazard	Perform ALU Operation	Write Result to Register File and update PSW
		Fanout from Control Store	Register File Read Bypass		

Extra Table - Not Currently Used

	Virtual Address (bits 31 - 0)	
Ring	Segment (bits 31 - 29)	Segment Byte Offset (bits 28 - 0)
0	0	0000 0000 - 1FFF FFFF (512 MBytes)
1	1	0000 0000 - 1FFF FFFF (512 MBytes)
2	2	0000 0000 - 1FFF FFFF (512 MBytes)

PRELIMINARY

	Virtual Address (bits 31 - 0)		
3	3	0000 0000 - 1FFF FFFF	(512 MBytes)
4	4	0000 0000 - 1FFF FFFF	(512 MBytes)
	5	0000 0000 - 1FFF FFFF	(512 MBytes)
	6	0000 0000 - 1FFF FFFF	(512 MBytes)
	7	0000 0000 - 1FFF FFFF	(512 MBytes)

PRELIMINARY

5 Vector Processor

Contents

- Overview
- Input Staging
- Vector Dispatch Interface
- Function Pipes
- Pipe Control
- Vector Register File
- Output Staging

5.1 Overview

The information in this chapter provides a brief description of all the major hardware units that comprise the vector processor.

The vector processor resides on the vector processor board, which is one of the two circuit boards that comprise the central processing unit.

5.1.1 Major Functions

The primary function of the vector processor is to execute all vector instructions. These vector instructions are dispatched from the scalar processor and input by way of the vector dispatch interface. Operands are input from either the scalar processor (if they are cache resident) or from memory. Internal to the vector processor operands are stored in the vector register file until needed. Data produced from vector operations may be sent to either the scalar processor or memory.

5.1.2 Major Units

The major units that comprise the vector processor are illustrated in Figure 5-1 and briefly defined in the text that follows.

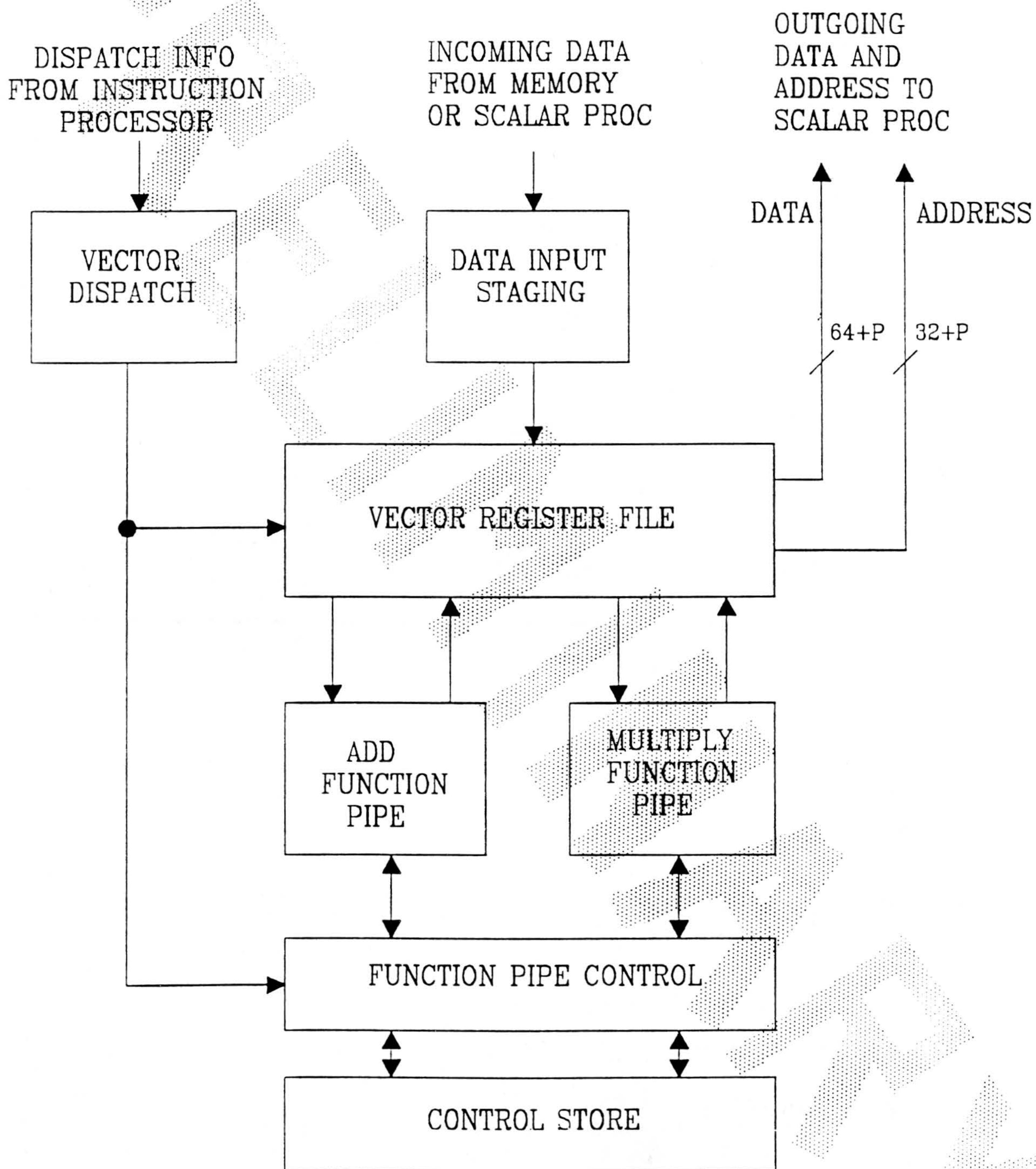
5.2 Input Staging

The input staging(IS) logic receives and queues data from the scalar processor and memory. Memory data(MXV) and scalar data(SXV) are queued in separate three-level deep queues.

5.3 Vector Dispatch Interface

The vector processor receives instructions from the scalar processor through the vector dispatch (VD)interface. The VD selects the function pipe to execute the instruction, determines whether the vector register file ports and other required resources are available, and determines whether chaining or accelerated execution may occur. The VD waits for a seed from the scalar processor if one is required. When all required conditions are met, the VD dispatches the instruction to the selected function pipe controller. This dispatch includes an entry-point microaddress, vector register and port selects, source and destination data sizes, execution rate, and the scalar seed. The pipe controller then may initiate execution of the instruction without needing to perform resource checks.

Figure 5-1 Vector Processor Simplified Block Diagram



PRELIMINARY

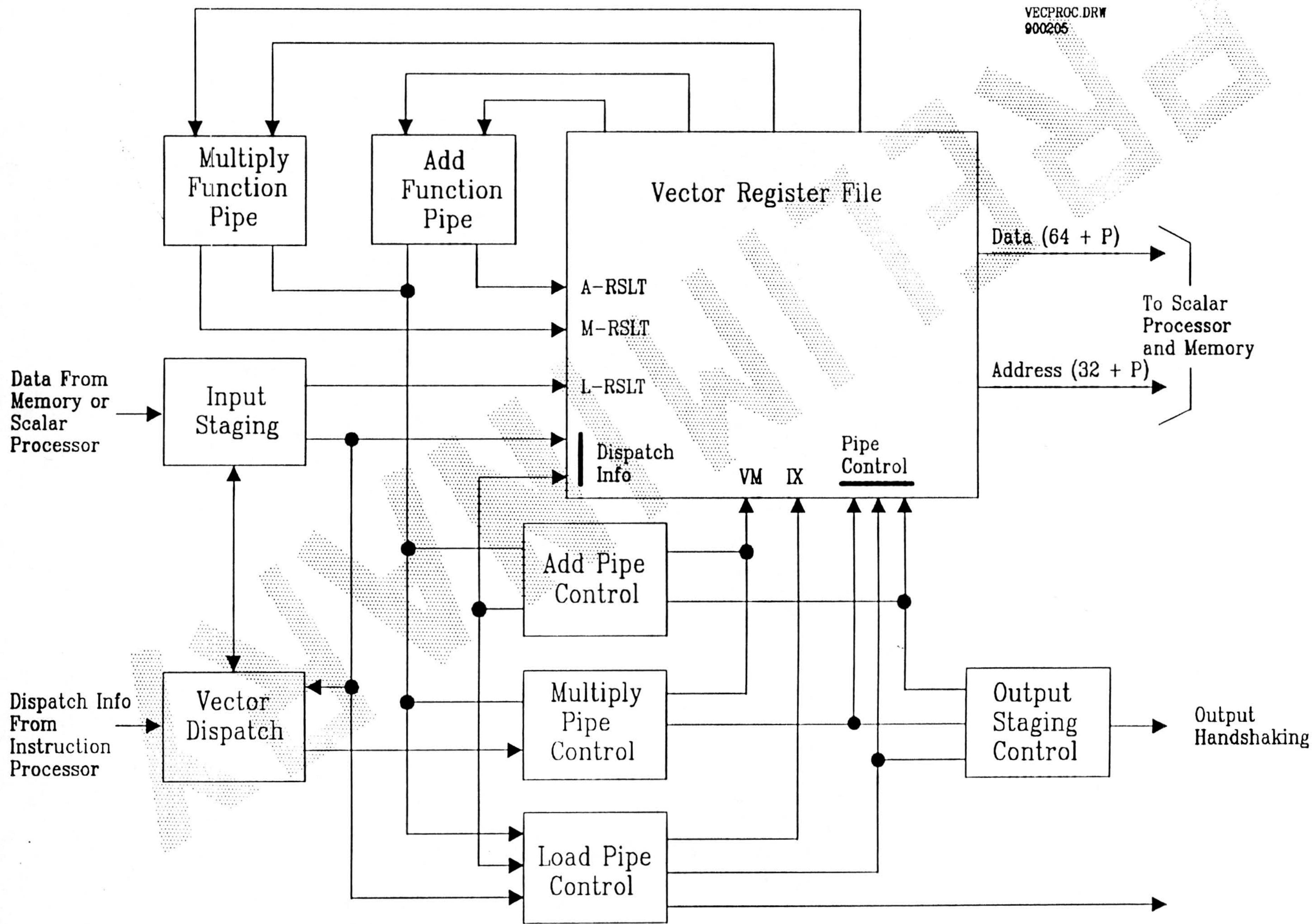
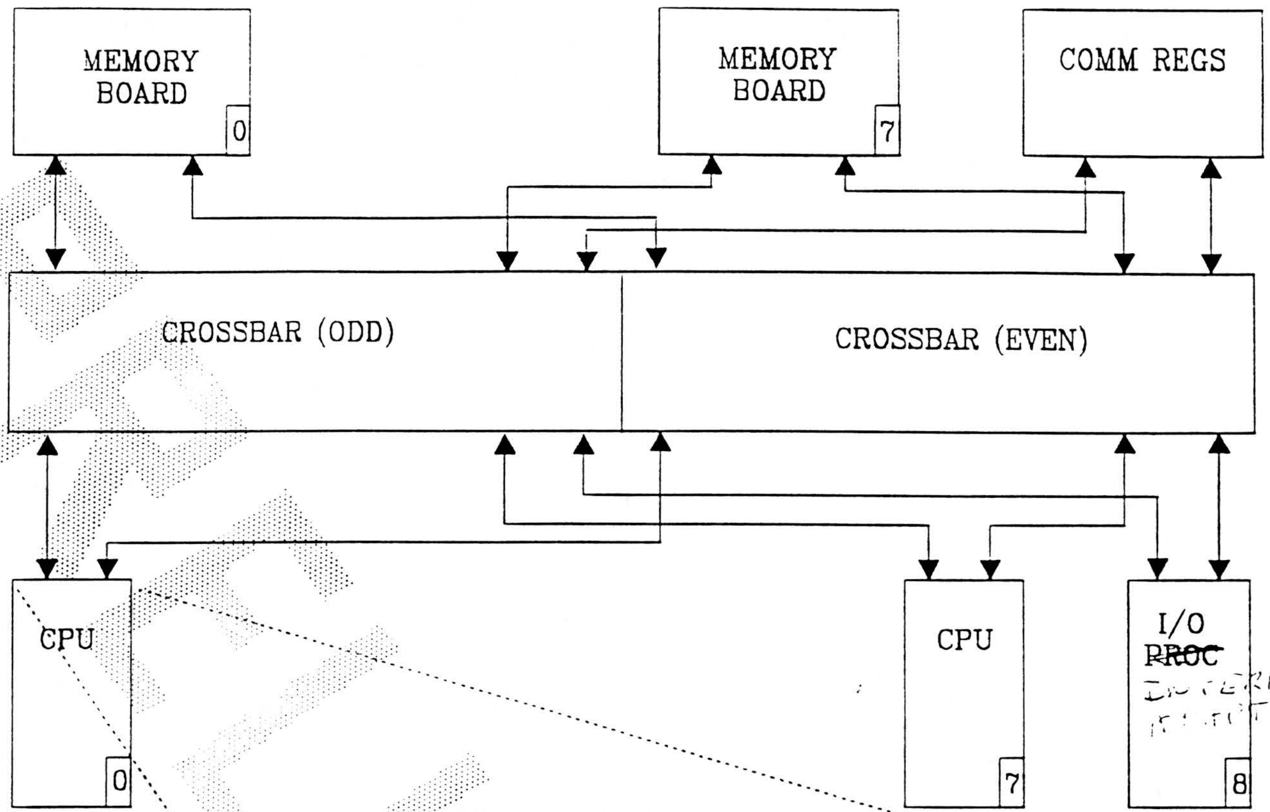
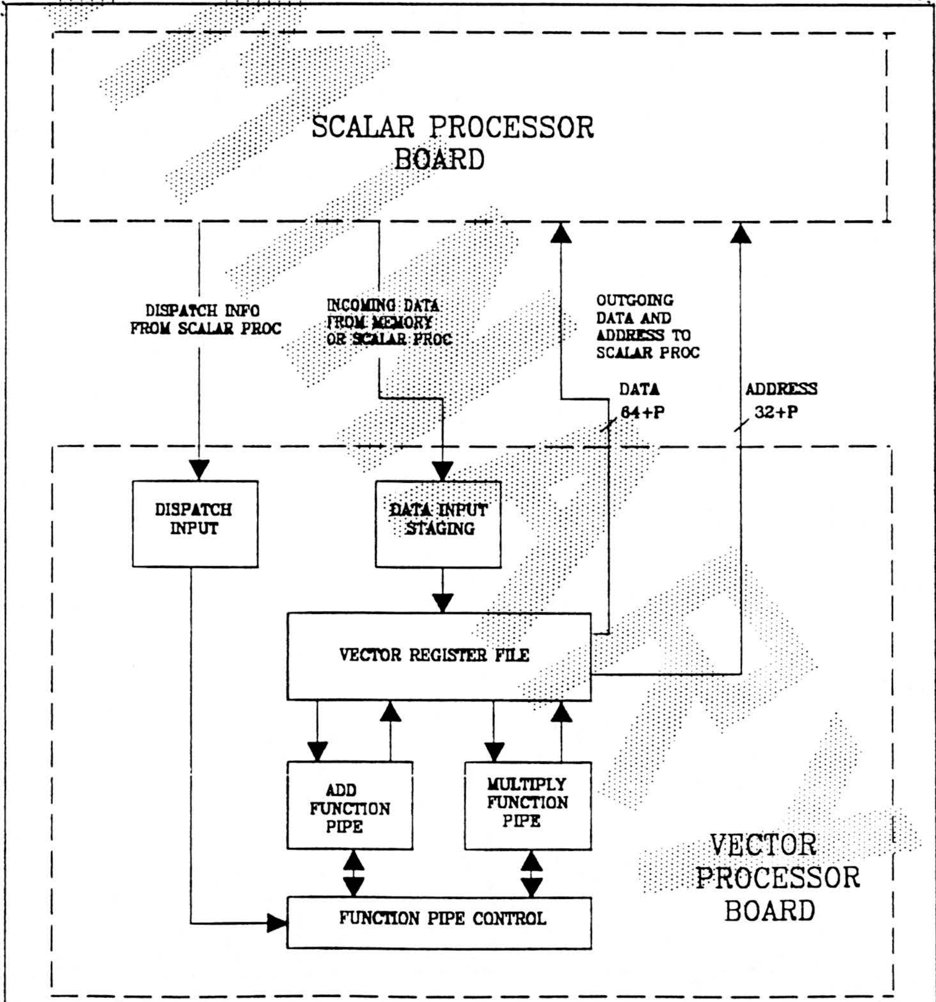


Figure 5-2 Vector Processor Block Diagram





VPSIMPL3.DRW
900214

5.4 Function Pipes

Three functional units, called function pipes, make up the heart of the vector processor. All the instructions executed by the vector processor are divided among these three function pipes.

The add function pipe executes the following vector instructions:

- Add
- Subtract
- Compare
- Convert
- Logical
- Shift
- Min and Max
- Integerize
- Trailing zeroes
- Leading ones

The multiply function pipe executes the following vector instructions:

- Multiply
- Divide
- Square Root

The load function pipe is used to perform all load and store operations. These include vector register load and stores, vector length register loads, moves between a scalar register and a vector register element, and load and stores to the vector merge register. **(last two sentences from C2 Maint doc.)**

5.5 Pipe Controllers

The add and multiply pipe controllers are responsible for reading vector elements from the Vector Register file, sending those elements through a function pipe, and writing the results back into the register file. The load pipe controller is responsible for all load and store operations. All three pipe controllers may send data to the scalar processor and memory.

Each pipe controller contains the following:

- Microsequencer
- Vector Element Counter
- Vector Merge Register
- Control Queue
- Backdoor Controller

In addition, the load pipe controller has a copy of the vector stride register.

5.6 Vector Register File

The vector register file contains the memories for the vector accumulators (V0-V7), along with data selection and staging registers. Each vector accumulator may contain from 0 to 128 64-bit operands called vector elements.

It should be noted that vector accumulators are sometimes referred to by the more general term, vector register. There are four types of vector registers. Specifically, they are: vector accumulators, the vector merge register, the vector stride register, and the vector length register. The vector register file being described here contains only vector accumulators.

The vector register file consists of four RAM banks, labeled for the two vector accumulators each bank contains: V0/V4, V1/V5, V2/V6, and V3/V7. Each bank is 256-locations by 72-bits and is divided into two halfs. Each half bank is 256-locations by 36-bits and is separately addressed.

5.7 Output Staging

The output staging ....

PRELIMINARY

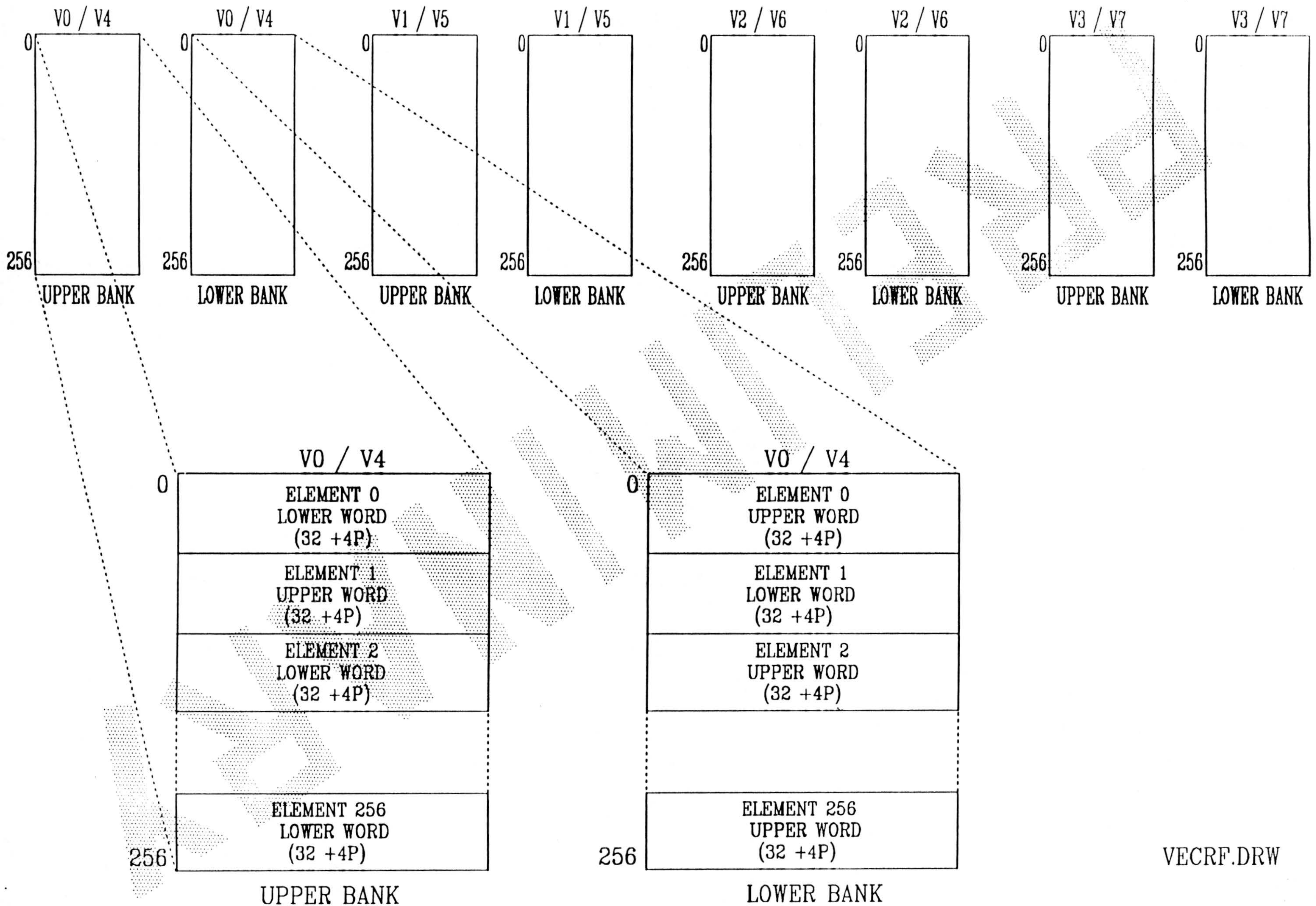


Figure 5-5 Vector Processor Register File

9-24-90

Crossbar

6

6.1 Overview

All communication between memory, the CPU utilities (CU) board, and the processors is through the crossbar. The crossbar has a total of nine processor ports, eight memory board ports, and one port for the CU board. Figure 6-1 provides an overview of the relationship between these major components.

A processor may be either a CPU or an I/O processor. Each processor communicates with the crossbar on a separate point-to-point bus. No multidrop buses are used. Point-to-point buses eliminate bus contention and significantly increase overall throughput.

The memory boards and the CU board also communicate with the crossbar over point-to-point buses.

Physically, the crossbar consists of seven circuit boards that reside in the centrally-located crossbar cabinet.

Functionally, the crossbar is divided into three areas. They include two 32-bit data paths for transferring odd and even data words between the processors, memory and the CU board. The third path is used exclusively to transfer control and status information between the processors and the CU board.

6.2 Crossbar operations

By definition, send transfers go from a processor through the crossbar to either memory or the CU board. Return transfers, on the other hand, originate from either memory or the CU board and go through the crossbar to a processor.

CU board transfers include read, write, status, and read/modify/write operations. These transfers may involve the communication registers or other hardware located on the CU board. Most transfers to and from the CU board use the crossbar's odd and even data paths. However, some operations bypass the odd and even data paths and use the more direct crossbar control path; these include deadlock status from the CPUs, as well as communication register status, and trap dispatch transfers to the processors.

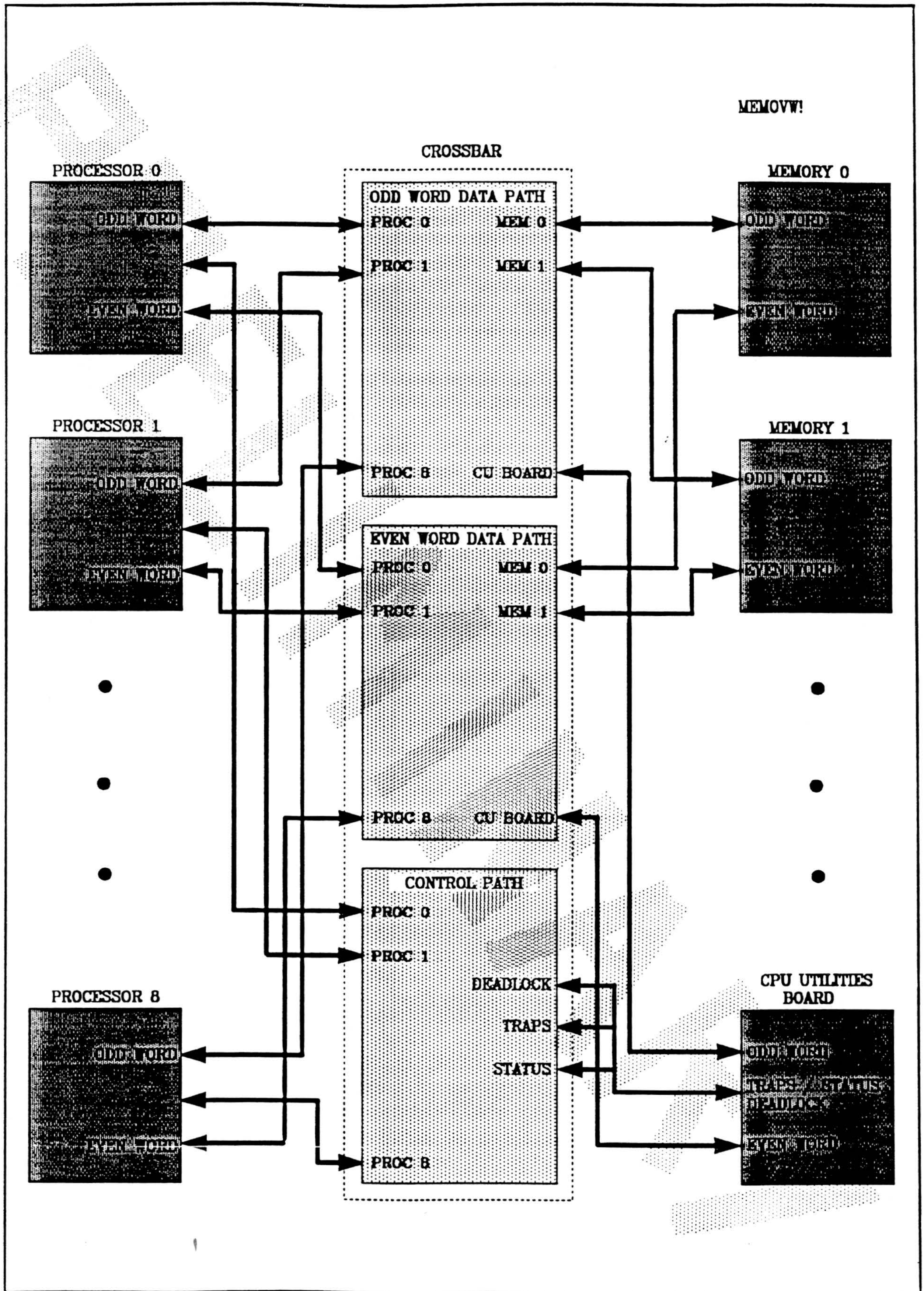
The crossbar routes data to and from the memory boards and arbitrates between requests to identical memory boards. The memory boards can accept one even and one odd request and return one even and one odd request per clock cycle. The crossbar determines which processor wins access to a particular memory board. Similarly, the crossbar must know which processor should receive data being returned from memory.

6.2.1 Memory transfers

The following memory transfer operations are handled by the crossbar:

- Memory read request
- Data return

Figure 6-1 Interconnection overview



- Memory write request
- No operation
- Test and modify (TAS or TAC)

6.2.1.1 Processor-to-crossbar handshaking

When the crossbar is able to accept a memory request from a processor, it asserts the request next signal. If a processor needs to make a memory request, it will issue a ready to the crossbar the following clock cycle along with the necessary request information.

6.2.1.2 Memory read or write request

The crossbar handles memory read and memory write requests the same except for one significant difference. For memory read requests, the crossbar must track which processor originated the request and in what order the request was made. This is necessary to ensure that the returning data goes to the correct processor and that it does so in exactly the same order as the orignal request was made.

To perform a memory request, the processor waits for the crossbar to issue a request next signal. The following clock cycle the processor will initiate the actual transfer by issuing a ready signal to the crossbar along with a cycle code that specifies the memory transfer type. At the same time, the processor also sends the memory address; the memory address contains a board select code that identifies the memory board the memory request should be sent to, the memory bank on the board, and the memory word address. The XARB uses the memory board and bank codes to determine the destination of the memory request. Of course, for memory write requests, data, parity, and zone information will also be included. The zone information is applicable only to byte write operations; the zone bits specify which byte location in the memory word will be the recepient of the data byte being sent.

The memory request waits in the XARB send path until XXXX. The length of the wait is indeterminate. Once the request is actually delivered to the memory board, the requested data (for memory read requests) will return within a fixed period of time. (which is ????

When the crossbar sends a memory request to a particular memory bank, it sets an internal flag to note the fact that the bank is busy. The crossbar will not send any additional requests to that memory bank until a bank done signal is received by the crossbar.

6.2.1.3 Data return

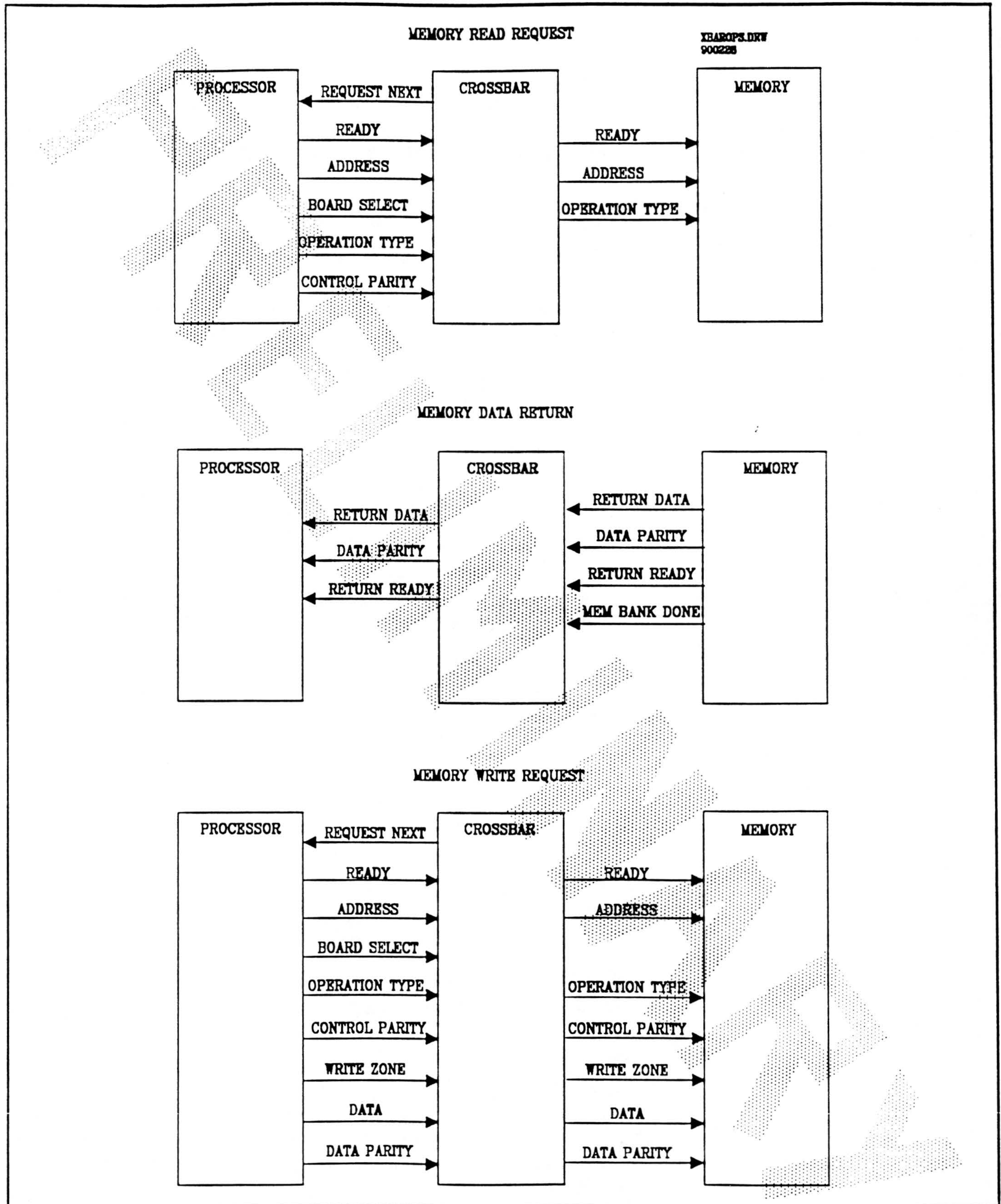
Figure 6-2 is a simplified illustration of the data return interfacing between memory, the crossbar, and the processor.

When return data is valid for a processor, the crossbar will assert the ready signal and transfer the returning data along with valid parity to the appropriate processor.

There is no handshaking between the crossbar and the receiving processor for returning data. The processor must always accept the data during the same clock cycle that the ready signal is asserted. Since the processor will always accept returning data, there are no overrun registers in the return data path. Return data flows without interruption from the memory board through the crossbar to the processor.

The memory bank done signal is issued by the memory bank controller from which the return data originates. Memory bank done informs the crossbar that this bank is returning the requested data and is now free to accept other requests.

Figure 6-2 **Memory transfers**



6.2.1.4 No operation

A no operation is used for test purposes only. However, it fully exercises the crossbar.

6.2.1.5 Test and modify

Test and Modify operations are used in the execution of the two CONVEX assembly language instruction, Test and Set Byte (TAS) and Test and Clear Byte (TAC). They provide a means for passing information between processors and manipulating semaphores in the communication registers.

6.2.2 CPU utilities board transfers

Transfers between the CU board and the processors use the same two 32-bit, odd and even crossbar data paths that are used for memory transfers. However, there are some differences in the way the data transfers are handled. In addtion, there is a third path that is used exclusively to transfer control and status between the CPUs and the CU board.

6.2.2.1 Data transfers and status request

The following CU board transfers are handled using the crossbar's odd and even data paths:

- Read data
- Write data
- Read modify write
- Status request

CU transfers include the following information:

- Two-bit code that specifies the transfer type (also called a cycle type)
- 26-bit address
- 32-bits of read or write data
- Control and parity

The read, return, write, and read-modify-write data requests involving the CU board are handled by the crossbar even and odd data paths much the same as memory transfers (see Figure 6-3). However, there are some significant differences; those differences are describied in the text that follows.

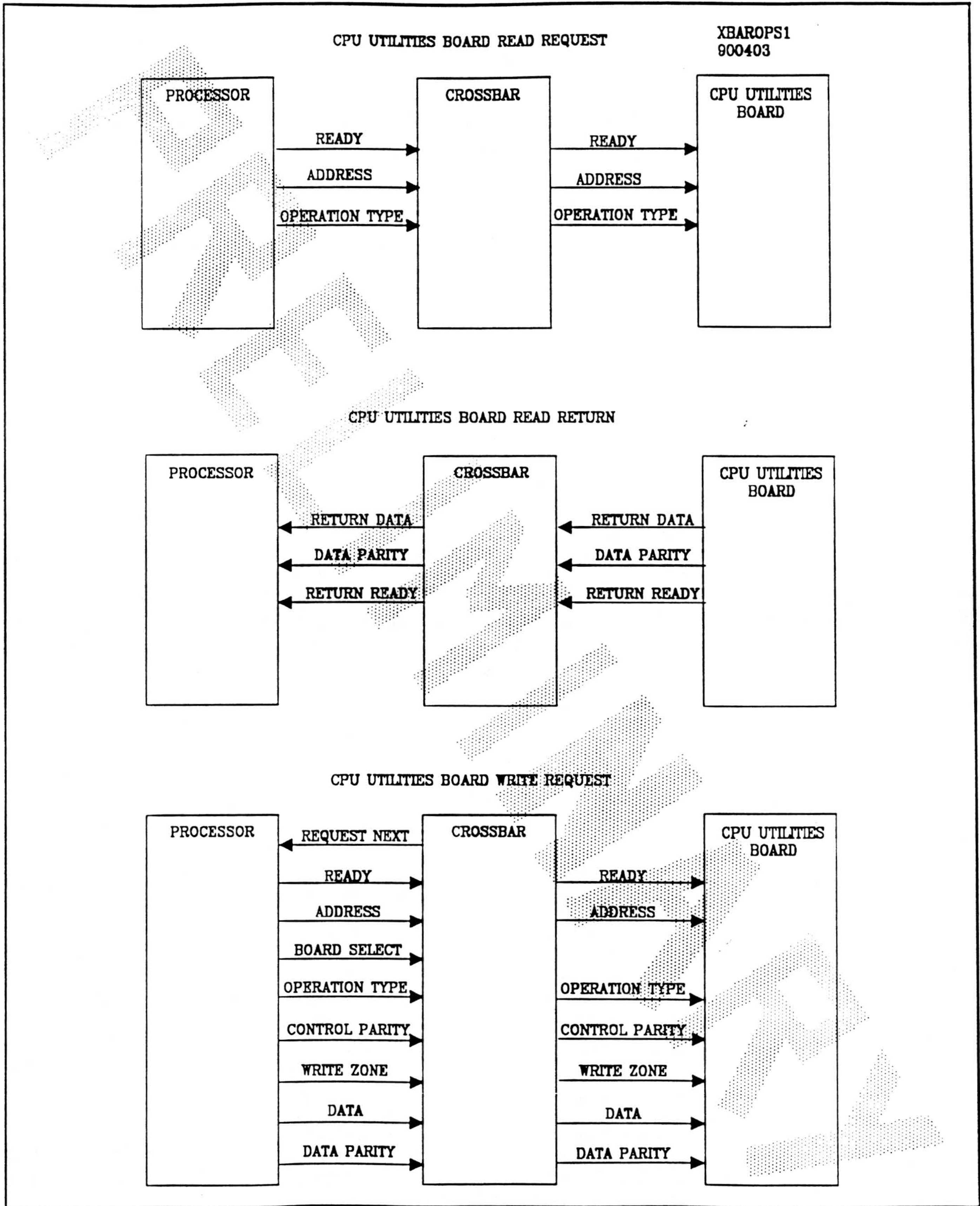
6.2.2.2 CU board and memory transfer differences

To the crossbar, the CU board appears as a fixed-latency, non-blocking memory board. Fixed latency means data requests to the CU return within a fixed period of time. Both memory boards and the CU board have fixed latency periods, but they differ in length. The CU board has a latency of either two or three cycles, whereas for memory boards it is XXX cycles. Non-blocking means that the CU board (unlike a memory board) is never too busy to accept a transfer request. The CU board can accept one new request every cycle.

Other differences between memory and CU board operations are noted in the text that follows.

It is important that both words of a longword request arrive at the CU board at the same time. To assure this, processors make only longword requests and do so only if the processor queue on both the even and odd siders of the crossbar are empty (i.e., request pending not activated).

Figure 6-3 Utility board transfers



For CU operations only, the processors pack the 26-bit odd word address with command and status information. The CU board interprets the command and status information it receives in the odd word address field and applies the command to both the even and odd word data streams.

6.2.2.3 Crossbar-to-utilities board handshaking

The crossbar-to-CPU utilities board interface uses only a ready signal to inform the CPU utilities board that a request from the crossbar is being sent; there is no other handshaking between the crossbar and the CPU utilities board.

6.2.2.4 Status request and return

Status request and return is used for interprocessor communication during semaphoring operations.The status request is sent by way of the odd and even crossbar data path. The status information being requested is the state of the lock bits for a particular communication register. Returning lock-bit status bypasses the odd and even data path and is sent to the requesting processor by way of the XCL board.

6.2.2.5 Status and control transfers

The following transfer types pass through the XCL board:

- Deadlock status from the CPU to the CU board
- Traps dispatched from the CU board to the processors
- Communication register status from the CU board to the processors
- Microtraps

6.2.3 Request pending and stores pending

Request pending and stores pending are asserted if the crossbar queues for a processor are not empty. The following signals are defined for this function:

REQ_PEND - Pending status active

ST_PEND - Pending stores requests active

6.2.4 Processor request overflow

Overflow occurs whenever the processor continues to send requests to the crossbar after access to the required memory board cannot be obtained.

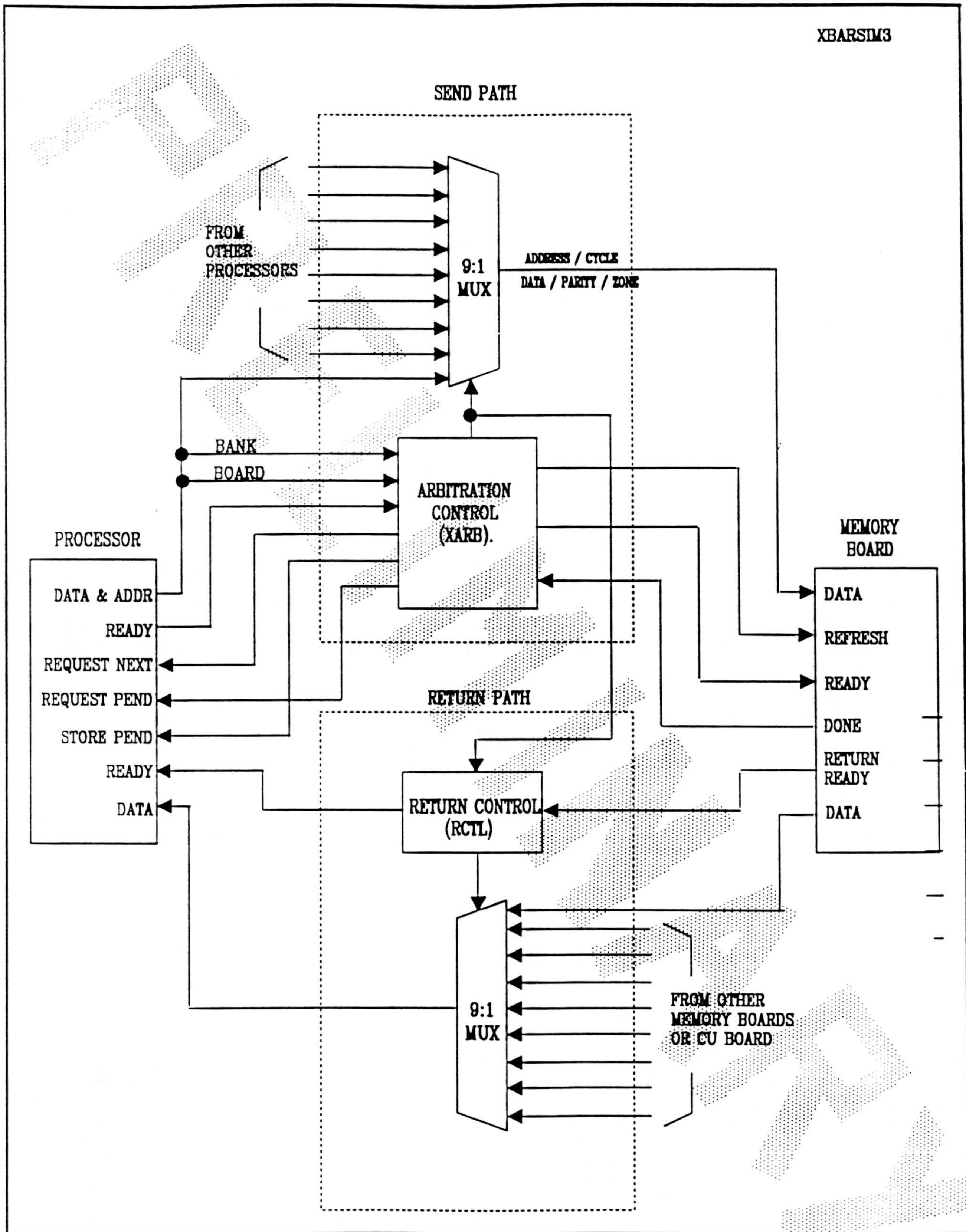
When a processor port cannot get access to a requested memory board, it is said to be blocked and the crossbar informs the processor by deasserting the Request Next signal. The processor may send two additional requests before realizing that the requests are blocked. This causes an overflow condition. A three-deep queue exists in the crossbar to store requests from the processor when overflow occurs.

6.3 Crossbar functional units

The crossbar has a total of nine processor ports, eight memory board ports and one port for the CPU utilities board board. Each 64-bit (plus parity) port is divided into an even and an odd half. The 32-bit (plus parity) odd and even halves function completely independent of each other.

Figure 6-4 provides a simplified view of the crossbar send and return data paths and their respective arbitration and return control circuits. This block diagram depicts only one half (odd or even) of the crossbar.

Figure 6-4 Crossbar data paths



6.3.1 Crossbar boards

Seven circuit boards comprise the crossbar. One board, labeled XCL, handles only control transfers.The remaining six boards handle data.

The six boards involved with data transfers can be further divided into two sets of boards, one board set for even, 32-bit data words and the other set for odd, 32-bit data words. Each even and odd board set contains three boards, labeled XS0, XS1, and XRT.

Figure 6-5 gives a more detailed illustration of the functional units within the crossbar, including board partitioning and the primary gate arrays.

6.3.2 Gate array functions

The crossbar data path boards contain four different gate array types. They are:

XARB - arbitration gate array

SXBR - send from processor to memory (via crossbar)

RCTL - return control

RXBR - return data to the processor from memory (via crossbar)

If more than one processor is in contention for a particular memory board, the arbitration (ARB) gate array determines which processor wins access.

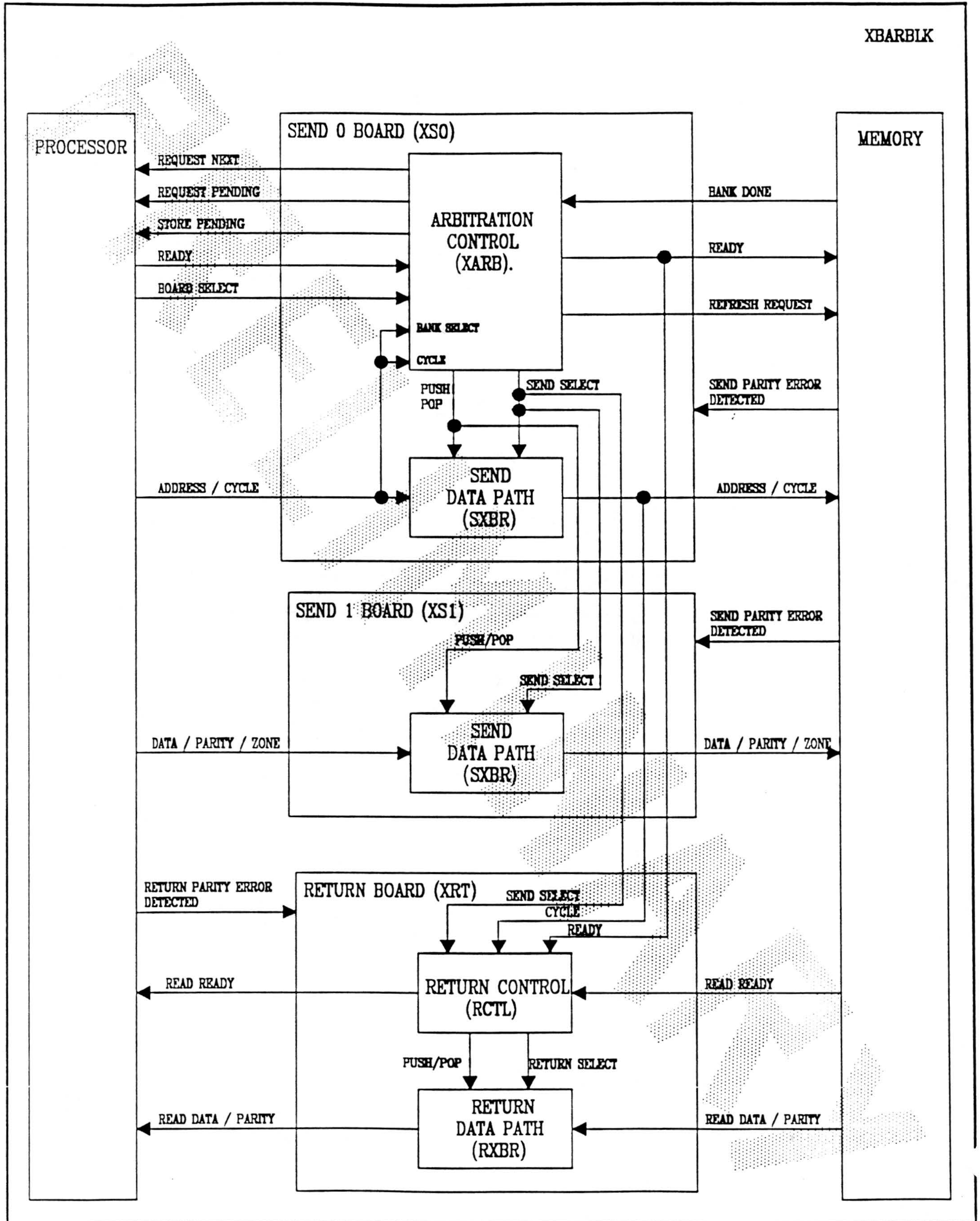
The send crossbar (SXBR) consists of input queues for processor data and a multiplexer to select data from these queues for each of the memory ports. The multiplexer selects are controlled by the XARB gate arrays as are the queue controls.

The return control gate array (RCTL) notes when a read request is issued to a memory board so that it can generate the appropriate mulitplexor selects and a read ready (RD_RDY) for that data at the proper time.

The return crossbar (RXBR) gate arrays, like the send crossbar gate arrays, simply multiplex data. They select data being input from one of the eight memory boards or the commreg board and route it to the appropriate processor's return data port.

PRELIMINARY

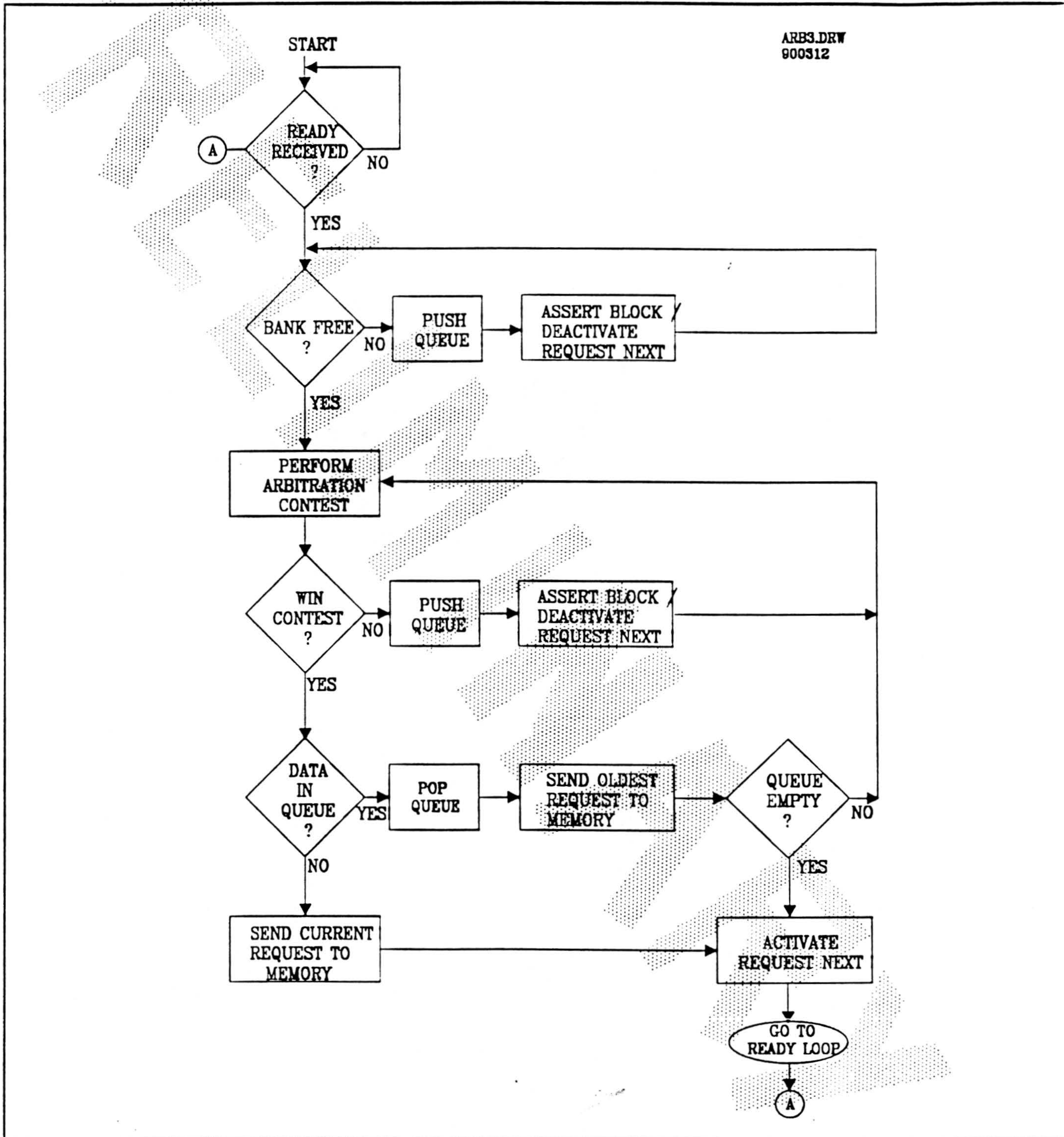
Figure 6-5 Crossbar functional block diagram



6.3.3 Arbitration

The sequence involved in arbitrating processor transfer requests is illustrated in the arbitration flow chart (see Figure 6-6) and described in the text that follows. It should be noted that this process applies to send transfers only. Send transfers go from a processor through the crossbar to either a memory board or the CPU utilities board.

Figure 6-6. Arbitration flow chart



When a ready is received from a processor, the arbitration logic performs two tests. One test is to determine whether the requested memory bank is free. The other test is to see if any other processors are currently contending for access to the same memory board.

If multiple processors are requesting access to the same memory board, then the arbitration logic will decide who wins. The win mechanism is based on a *modified round robin* approach. The term *round robin* means that each processor has its turn at having the highest priority. However, it is modified slightly from a pure round robin approach to permit a processor to retain its position of highest priority if it encounters a busy bank. Whenever a busy bank is encountered, the processor will retain its position of highest priority until the bank is not busy and the memory transfer is successfully initiated. In addition, once a processor wins access to a particular memory board, it is permitted to retain access for up to 16 clocks provided the requests are made at the rate of one per clock cycle.

If the processor does not win the arbitration contest, the crossbar deactivates the request next signal to the processor. This action tells the processor that it is blocked and that it should refrain from sending additional reqests. In the interim, the data and/or address information associated with the processor's request is put (PUSH) in the crossbar queue. If the processor continues to send requests, an overflow condition occurs.

How the processor handles an overflow condition is described in the next section.

6.3.4 Overflow queue

Overflow occurs whenever the processor continues to send requests to the crossbar after access to the required memory board cannot be obtained.

Both the XARB and SXBR gate arrays are designed to handle overflow. A five-deep queue exists in the SXBR gate array to store requests from the processor when overflow occurs.

The push signal from the Arbitration gate array informs the SXBR that a processor port is blocked and that its requests should be queued. The pop signal tells the SXBR that the port is no longer blocked and that it should empty its queue to memory.

6.3.5 Physical configuration map

A software-loadable table, the PCM, resides in the arbitration gate array. This table contains a definition of the memory boards present in this particular computer system. If a processor requests access to a non-present memory board, this will be detected in the arbitration gate array and a PCM Error will be generated.

6.3.6 Parity checking

The crossbar performs no parity checking. However, a copy of the last three transfers are kept at the output staging registers for both the send and return data paths. If a parity error is detected by a processor or memory after a crossbar transfer, the data in the crossbar's three-deep queue can be scanned to determine whether the data was bad before or after it reached the crossbar.

6.3.7 Control crossbar

Memory Subsystem 7

7.1 Contents

- Overview
- Memory Boards
- Major Functional Units
- Interface
- Error Detection and Correction

7.2 Overview

The memory subsystem includes the following features and characteristics:

- Four gigabyte address range provided by 32-bit address
- Up-to four gigabytes of physical memory
- Up-to-eight memory boards
- 128MB-to-512 MB per board (depends on DRAM size)
- 1MB or 4MB DRAMS
- Seven-bit error correction code
- Single and multi-bit error detection
- Single-bit error correction

A C3800 Series system may have up to 4 GB of addressable memory implemented using up-to-eight memory boards. Each memory board has a capacity of from 128 MB-to-512 MB, depending on the size of the DRAM (1MB or 4MB) being used.

7.3 Memory Boards

Memory boards reside in the the processor cabinet(s); one memory board is physically paired with each processor.

Each memory board can accept requests and return memory data to the requesting processors at the rate of 64 bits of data per clock cycle. Thus, each board has a bandwidth of 500 MB per second; a fully configured, eight-board memory system has a bandwith of four GB per second.

The 64 bits of data that a memory board can handle is divided into two 32-bit words, called even and odd. Each even and odd word can be addressed independently, and each one has a separate send and return datapath.

Each memory board has two independent, 32-bit sides of 16 banks each. One side of the board is addressed as even memory (address bit two equal to 0), the other as odd memory (address bit two equal to 1).

A memory board physically consists of the motherboard and the following major components:

- 32 daughter boards (called memory cards; 16 on each side of the motherboard)
- Four bank control gate arrays
- One read EDC (error detection and correction) gate array
- 32 write EDC gate arrays (one on each daughter board)
- Input staging and multiplexing logic

7.4 Major Functional Units

The major functional units of a memory board are listed below and described in the text that follows:

- Input Staging
- Bank Control
- Read Error Detection and Correction (REDC)
- Write Error Detection and Correction (WEDC)
- Memory Card (MC)

7.4.1 Input Staging

7.4.2 Bank Control

Bank control generates address and control strobes for the memory banks. Each memory board contains four Bank Control Gate Arrays (BCGAs). Every BCGA controls eight memory banks.

7.4.3 Read EDC

There is one Read EDC gate array on each memory board. The RDEDC gate array handles error detection and correction for all return (read) data, including both even and odd data words.

7.4.4 Write EDC

The Write EDC performs parity checking and generates Error Correction Codes (ECC) for all memory writes and partial writes that are destined for a particular memory bank. Each memory board contains 32 Write EDC gate arrays, one for each memory bank. The Write EDC gate arrays are physically located on the memory cards. Each memory card contains one Write EDC gate array.

7.4.5 Memory Card

The Memory Card (MC) is a small daughter board mounted on the memory board. Each memory card represents one memory bank and contains 39 RAMS, 32 for data bits and 7 for ECC. With 1MB RAMs, this corresponds to 4MB of data per memory card or 128 MB for all 32 banks. With 4MB RAMS, there are 16MB per memory card and 512 MB for the entire 32 bank memory board.

7.5 Interface

Memory boards interface only to the Crossbar. No other board in the system can access a memory board without going through the Crossbar. Each memory board has a single, even-side crossbar interface and a single, odd-side crossbar interface.

Table 7-1 lists and defines the Crossbar to Memory board interface signals.

7.6 Error Detection and Correction

The memory system in the C3800 Series systems use the same error detection and correction code as the C200 Series systems. This code requires seven bits of ECC for 32 bits of data.

For read operations, error detection and correction are performed in the same clock cycle. Thus single-bit error correction does not delay returning data; single bit errors do not affect the fixed return rate feature of the memory boards. When a correctable error is detected, the data and its address are registered in a log ring. The Service Processing Unit can scan this information while the memory board is still functioning in the system.

Multi-bit memory errors cannot be corrected, and they will generate a hard error.

PRELIMINARY

Table of Contents

CPU Utilities Hardware

Overview 8
The Communication Registers 8
Control Register Address Space 8
ASAP Accelerator 8
Interrupts and Microtraps 8
  Interrupts 8
  Microtraps 8
Process Deadlock Detection 8
Hardware 8

PRELIMINARY

CPU Utilities Hardware

8

PRELIMINARY

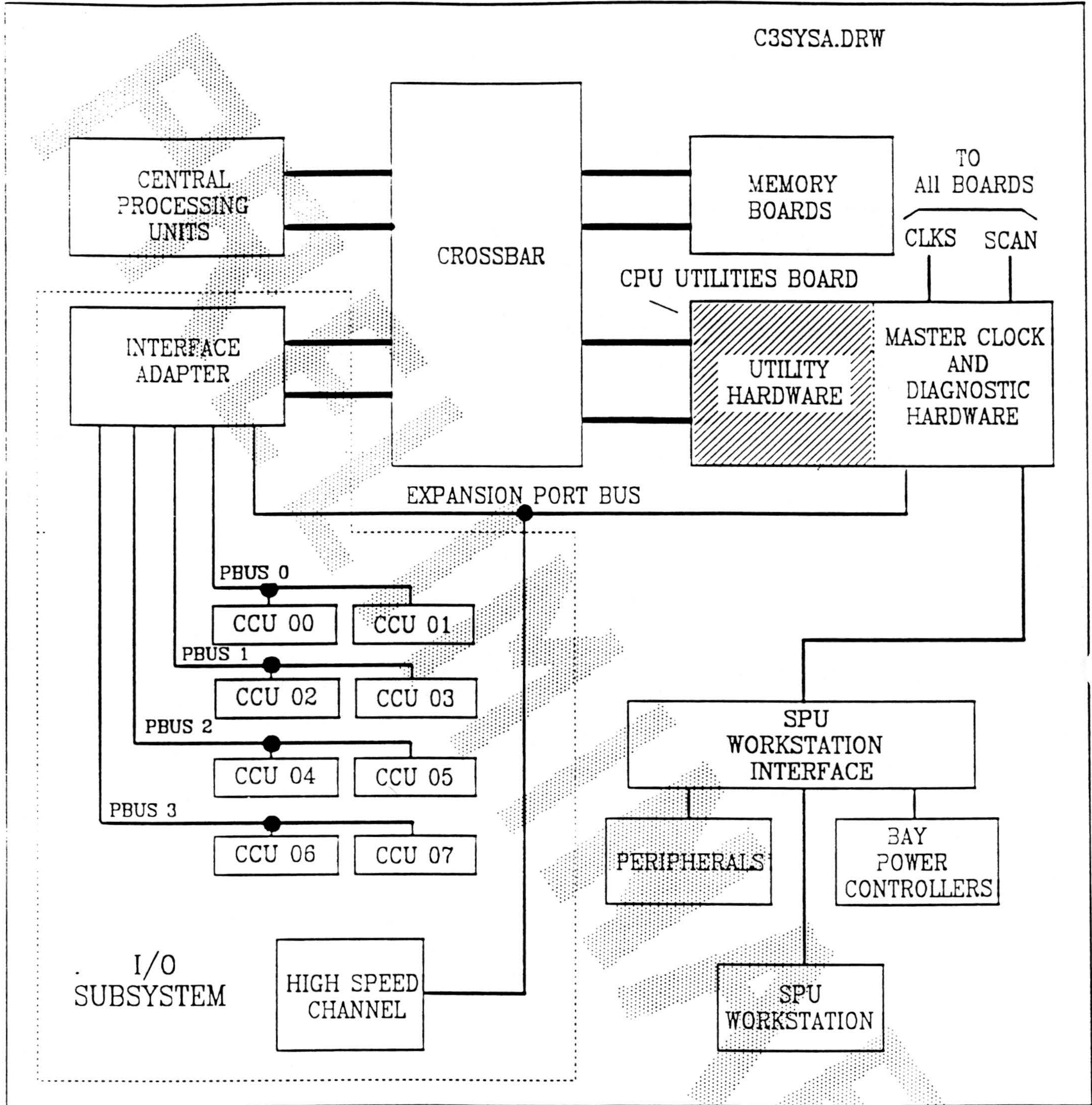
8.1 Overview

The CPU Utilities (CU) board contains the master clock and both diagnostic anc utility hardware. This chapter describes only the utility hardware portion of the CU board. The utility hardware includes the following:

- Communication Registers
- Control Register Address Space
- ASAP Accelerator
- Microtrap and Interrupt Structure
- Deadlock Detection Logic

Figure 8-1 illustrates the relationship between the CPU utility hardware and the rest of the system.

Figure 8-1 Utility Hardware Simplified Block Diagram



8.2 The Communication Registers

The communication registers provide a mechanism for multiple thread execution across more than one CPU. A communication register is a 64-bit addressable register with an associated lock bit that is maintained by hardware. The lock bit is used by software and microcode as a semaphore for the contents of the individual communication register.

The communication registers are physically located on the CU board.

8.3 Control Register Address Space

The control register address space contains special purpose registers that requir system-wide access. Access to these registers is limited to the processor microengines; they are not directly accessible from the processor's assembly language instruction set. The microcode commands used to access the control register address space include get_x, put_x, snd_x, and rcv_x.

Table 8-1 illustrates the control register address space allocation.

Table 8-1 Control Space Address Allocation

Control Address	63 32	31	
000	Read / Write Communication Registers Lock Bits		
001	Time of Century (TOC)		
002-003	Reserved		
004	All Zeroes		NITC
005	All Zeroes		ITC
006	All Zeroes		ITSR
007	All Zeroes		ITIN
008	All Zeroes	Type / Purge Address / Int Vec	
009	All Zeroes		I/O Install
00A	All Zeroes		CPU Install
00B-00F	Reserved		
010	All Zeroes		P0 CIR
011	All Zeroes		P1 CIR
012	All Zeroes		P2 CIR
013	All Zeroes		P3 CIR
014	All Zeroes		P4 CIR
015	All Zeroes		P5 CIR
016	All Zeroes		P6 CIR
017	All Zeroes		P7 CIR
018-01F	Reserved		
020	All Zeroes		P0 IDLE
021	All Zeroes		P1 IDLE
022	All Zeroes		P2 IDLE
023	All Zeroes		P3 IDLE
024	All Zeroes		P4 IDLE
025	All Zeroes		P5 IDLE
026	All Zeroes		P6 IDLE
027	All Zeroes		P7 IDLE

Table 8-1 Control Space Address Allocation (continued)

Control Address	63 32	31 0
028-02F	Reserved	
030	All Zeroes	GP
031	All Zeroes	GE
032	All Zeroes	TCPU
033-037	Reserved	
038	All Zeroes	P0 LE
039	All Zeroes	P1 LE
03A	All Zeroes	P2 LE
03B	All Zeroes	P3 LE
03C	All Zeroes	P4 LE
03D	All Zeroes	P5 LE
03E	All Zeroes	P6 LE
03F	All Zeroes	P7 LE
040-047	Reserved	
048	All Zeroes	L0 BE
049	All Zeroes	L1 BE
05A	All Zeroes	L2 BE
05B	All Zeroes	L3 BE
05C	All Zeroes	L4 BE
05D	All Zeroes	L5 BE
05E	All Zeroes	L6 BE
05F	All Zeroes	L7 BE
060-067	Reserved	
068	All Zeroes	PCIR
069-FFF	Reserved	

Control address 0 refers to a 64-bit shift register that is manipulated by the snd_x and rcv_x commands. This register supplies the lock bit portion of the data during a write communication registers (wrcmr) command and receives the lock bits during a read communication registers (rdcmr) command. CPUs executing either a load cmr (ldcmr) or store cmr (stcmr) assembly language instruction vie for access to this register using the snd_x command and relinquish access with the rcv_x command.

Control address 01 contains the Time of Century (TOC) counter. The *movTOC*, S. assembly language instruction is executed by using the get_x command. The TOC keeps wall clock time, not user time. Normally, initialization is performed immediately after power up, initialization involves loading a value into the counter;. The TOC is not saved or restored during a context switch.

Control addresses 04-07 reference the next interval timer count (NITC), the interval timer counter (ITC), the interval timer status register (ITSR), and the interval timer interrupt number (ITIN). Brief definitions of these registers are provided in the text that follows; for more complete information refer to the Architectural Reference Manual.

The interval timer is used to interrupt the processor at a programmable rate. Th ITC is loaded with an initial count value from the NITC. The ITSR controls both the operation of the ITC and interrupt generation. The ITIN contains the numbe of the virtual channel that is to receive the next timer interrupt.

Control address 08 is the trap command register. CPUs and I/O interface adapters initiate firmware generated traps and interrupts by contending for anc depositing commands in this register using the snd_x command. The trap type defines the action to be performed, qualified by a purge address or an interrupt vector.

Control address 09 contains a bit mask that defines the crossbar ports accessed I/O interface adapters. This mask determines the destinations of I/O interrupts that are intended for all interface adapters.

Control address 0A contains a bit mask that defines the crossbar ports accessed by CPUs. This mask determines the destinations of traps that are intended for a CPUs.

Control addresses 10-17 contain the current CIR of each CPU.

Control addresses 20-27 specify the idle state of the associated CPU. A one indicates the CPU is idle; a zero indicates it is not idle.

Control addresses 30-5F contain registers associated with CPU interrupt arbitration. This includes a global pending register (GP), global enable register (GE), target CPU register (TCPU), local enable registers for each CPU (PxLE), ar a set of eight broadcast enable (BE) registers. The Architectural Reference manu defines the purpose of these registers.

Control address 68 contains the CIR of the next available fork for the CPU accessing this register.

8.4 ASAP Accelerator

The ASAP Accelerator provides a fast mechanism for polling the communication registers for posted forks and spawns. A requesting CPU initiates the activity to determine whether or not an available process has been posted. For more information about ASAP (Automatic Self-Allocating Processors), the role of the communication registers, a definition of processes and threads, and other related information, please read Chapter 2 of this manual or refer to the Architectural Reference Manual.

The ASAP Accelerator consists of a state file, a barrel shifter, and the PCIR register.

The state file, which is implemented using a set of latches, contains 32 entries, which is the maximum number of processes that may be mounted at any given time. A process is mounted when its state is currently represented by a partition of the communication registers. Each mounted process is represented by an index value. When this value is loaded into the CIR of a particular CPU, that CPU can begin executing the associated process (or its thread). Each CPU has a five-bit CIR that may hold the index value of any one of the 32 processes that may be mounted.

Each state file entry is associated with one mounted process and consists of 12-bits. The 12 bits contain information that tells a requesting CPU whether it can load and execute a thread for this process. Specifically, the 12-bit entry indicates whether the fork is accessible (locked or unlocked), posted (requesting service), and whether or not the fork is stopped. It also specifies the CPU mask and indicates whether the thread allocation mask (TAM) is equal to zero or not. The CPU mask specifies which CPUs can take the fork, and the thread allocation mask indicates the number of active threads that exist for this particular process. The actual number of active threads is not important provided the number is greater than zero.

The barrel shifter and a trailing zero count is used to determine which one of the available forks has the highest priority. The index value for the highest priority fork is placed in the *priority communications index register* (PCIR).

8.5 Interrupts and Microtraps

Interrupts result from asynchronous events. For example, incoming data from an I/O device is an asynchronous event. When an interrupt is issued it is first sent to the CU board, the CU board then forwards the interrupt to the appropriate destination. The processor that receives the interrupt from the CU board will also receive the address of the appropriate interrupt handler. The processor then vectors to that address and performs the interrupt service routine.

Microtraps result from situations where one CPU needs to tell the other CPUs in the complex to perform a particular operation. For example, the *ctrsg* instruction causes a microtrap to occur. This command, which tells the CPU timers in the entire complex to update their current time, is executed by first notifying the CU board. The CU board then coordinates sending the microtrap to the rest of the complex.

The CU board provides the following trap and interrupt functions:

- receives all interrupts initiated by CPUs and Interface Adapters
- notifies the appropriate CPU(s) and I/O devices to perform interrupt service
- maintains all interrupt registers
- receives all microtraps that originate from CPUs

- coordinates firmware microtrap service by notifying the CPUs in the complex
- continuously monitors process deadlock conditions
- issues a deadlock trap if a process deadlock condition is detected

8.5.1 Interrupts

To initiate an interrupt, a CPU or Interface Adapter executes a snd_x command to the CU board's trap control register (control space address 08; see Table 8-1). An eight-bit interrupt vector address must be included with the command and a transmit interrupt (xmti) must be specified as the trap type.

If the snd_x command is successful, the CU will test the interrupt vector to determine whether the destination is a CPU (vector less than eight) or Interface Adapter (vector greater than seven). If it is a CPU, a bit corresponding to the interrupt vector will be set in the global interrupt pending register (GP). This register is part of the Control Address Space contained on the CU board (see Table 8-1). If the destination is an Interface Adapter, an interrupt trap with a vector composed of the vector from the trap control register will be sent to all I/O ports. Control space address 09 defines which ports are I/O ports. Upon receipt of the vector, each Interface Adapter must match the vector against a set of bounds registers to determine the I/O device that is the final destination of th interrupt. If the comparison results in a match, then the IA must contend for access to its local interrupt bus and forward the interrupt to the appropriate I/C destination. The IA that has a match, sends a trap complete (TRAP_COMP) signal back to the CU board as soon as the interrupt is forwarded to its destination. IAs that do not have a match, send a trap complete signal back to th CU board immediately. After the respective IAs issue the trap complete signal, they resume normal operation.

8.5.2 Microtraps

When a CPU issues a firmware microtrap or interrupt, it is sent to the CU board by way of the crossbar. The CU board then assumes responsibility for coordinating the trap or interrupt sequence by issuing traps to the appropriate target CPU(s) or I/O devices and monitoring trap completion.

To initiate a microtrap, a CPU executes a *snd_x* command to the CU board's trap control register (control space address 08; see Table 8-1) specifying the appropriate trap type. If the snd_x-x command is successful (status = 1), the originating CPU can assume the trap sequence has begun and proceed with its portion of the trap operation. Meanwhile, the CU will issue a trap request to all required CPUs (except the originator) according to the priority sequence shown in Table 8-2. Each CPU (including the originator) must acknowledge acceptance or completion of the trap using the trap complete (TRAP_COMP) signal; afterward they may independently resume their previous operation. For *ctrsg* type traps only, the CU notifies the microtrap originator (with a MT_COMP signal) as soon as all CPUs involved complete their trap operations and responc with a trap complete signal.

Table 8-2 Trap Priorities

Priority	Trap Type
0	CPU interrupts
1	Firmware microtraps or I/O interrupt:
2	Process deadlock trap

8.6 Process Deadlock Detection

A deadlock occurs when all the currently executing threads of a process are executing a *synchronization* instruction followed by a branch back to the same instruction. Synchronization instructions are defined as instructions that attempt to change a lock bit and then return status on the success or failure of the lock/unlock operation. Examples of synchronization instructions include the *tas, snd, rcv,* and *inc* instructions.

Deadlocks are defined as system exceptions and are passed to the process deadlock handler for resolution.

The CU board monitors process deadlock conditions and issues a deadlock trap if a process deadlock condition is detected. To facilitate deadlock detection, each CPU sends the CU board a deadlock status signal. The CU board also receives the current CIR for each CPU and stores this information in control space address 10-17. If all CPUs that are executing the same process (i.e., they have the same CIR value) indicate a deadlock condition, the CU assumes the process is deadlocked. However, this same symptom can occur for simple synchronization delays. Even if the deadlock is legitimate, it may clear if a latent arithmetic exception occurs. To make sure the deadlock is legitimate and not about to clear due to a latent arithmetic exception, the CU will wait for 32 clock cycles before issuing a deadlock trap.

8.7 Hardware

The major hardware components on the CPU Utilities board are listed as follows:

- Random access memory
- NDAT gate arrays
- NADR gate array
- Master clock generator gate array
- Scan engine gate array

Eighteen self-timed RAMs contain the communication registers and their associated lock bits. Both the interrupt data structure and the access control logic for the communication register data structure are sliced across two identical NDAT gate arrays. A single NADR gate array contains the address structure, lock bit data path, crossbar address and control interface registers, and the ASAP fork accelerator data structure. The NDAT and NADR gate arrays each contain part of the control register address space. The master clock generator and scan engine reside on separate (TBD) gate arrays.

PRELIMINARY

Table of Contents

Clock and Diagnostic Hardware

Overview 9
Operations 9
Scan Writing to System Boards 9-
Scan Reading and Ring Verifcation 9-
SST and CAST 9-
Scan Control Modes 9-
Clock and Scan Control Registers 9-
Command/Status Register 9-
Clock Generator Commands 9-
Command Enable Registers 9-
Clock Frequency Control Registers 9-
Disabled Clocks Registers 9-
Burst Counter Register 9-
Command/Status Register 9-
Board Reset Registers 9-
Scan Control Enable Registers 9-
Scan Counter Register 9-
Error Location Register 9-
Hard Error Registers 9-
Output Buffer Register 9-
Input Buffer Register 9-
Scan Mask Register 9-
Scan Compare Register 9-
I/O Address Register 9-
Clock Generator 9-
1x and 2x Clock Generator 9-
Burst Counter 9-
Single/Micro Step Engine 9-
Clock Control Engine 9-
Scan Engine 9-
Scan Memory 9-
WC Interface 9-
XP Interface 9-
Memory Map 9-
Map Registers 9-
Memory Test Registers 9-
Memory Test Operating Modes 9-

PRELIMINARY

Clock and Diagnostic Hardware

PRELIMINARY

9.1 Overview

This chapter describes the basic scan operations and the clock and diagnostic hardware contained on both the CU board and the control crossbar board (XCL)

Operations performed by the clock and diagnostic hardware are based on commands received from the SPU through the workstation interface board. Clock and scan operations are performed through use of scan control modes, which control the individual boards in the system, and the contents of the clock and scan control registers. These control registers reside in memory located on the CU board and are part of the clock generator and scan engine. Topics related to clock and scan opeartions located in this chapter include:

- Operations
- Scan control modes
- Clock and scan control registers

The clock and diagnostic hardware is partitioned across the CU board, the crossbar control board, and the crossbar backplane. Functionally, there are five areas:

- Clock generator
- Scan engine
- Scan memory
- Workstation-to-CU interface
- XP interface

See Figures 9- and -. Figure - depicts the clock and diagnostic hardware in relation to the rest of the system, and Figure - provides a simplified block diagram of the major units.

Figure 9-1 Clock and Diag H/W - System-Level View

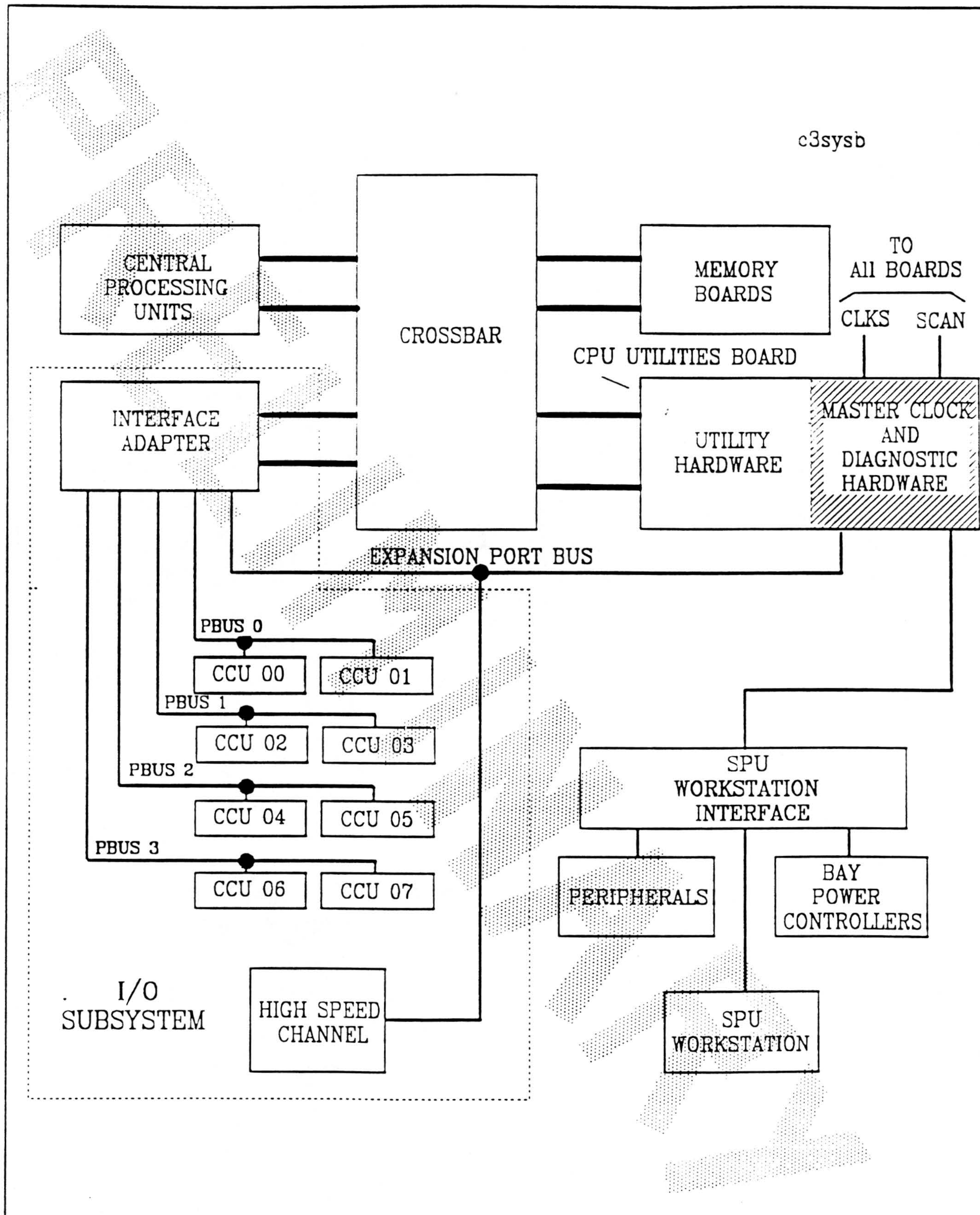
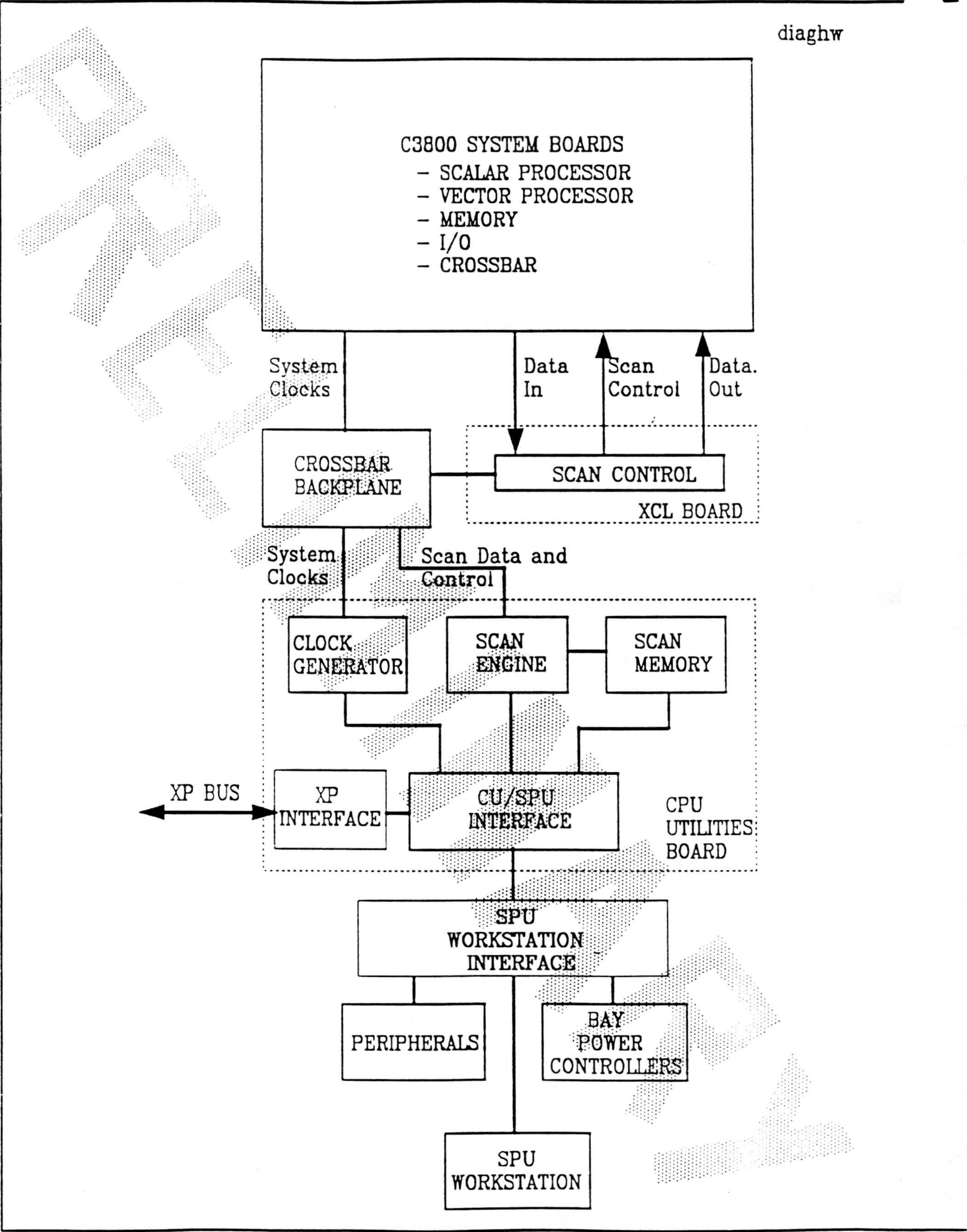


Figure 9-1 Clock and Diagnostic Hardware



9.1.1 Operations

The clock and diagnostic hardware is used to initialize the scalar and vector processors, perform scan-based testing of the boards in the system, and operate the system boards at different clock speeds for testing and troubleshooting. Operations described in this chapter include the following:

- Writable control store initialization
- Writable control store verification
- Scan-based testing
  - Scan writing to system boards
  - Scan reading and ring verification
  - SST and CAST
- Single and microstep operation
- Burst mode operation

9.1.2 Scan Control Modes

The scan control modes are the most basic scan operations. There are eight modes; these eight modes are implemented by way of three scan control lines that go to each board in the system.

9.1.3 Clock and Scan Control Registers

The clock and scan control registers reside in memory located on the CU board. They are central to all clock and scan operations. There are two sets of registers: one set for the clock generator and another set for the scan engine. Some of the registers between the two sets have identical names.

9.1.4 Clock Generator

The clock generator creates and controls the system clocks. A master oscillator produces a 2 nanosecond master clock. From this clock, three controllable clock rates of 4, 6, and 12 nanoseconds are derived. Each of these three clocks may be sent to any board in the system, and each may be operated in one of four modes: free run, burst run, single/micro step, and disabled. Commands issued from the SPU determine the clock rate and the operating mode for each individual board slot.

9.1.5 Scan Engine

The scan engine controls the transfer of scan data to and from the boards in the system. The scan engine has three major functions:

- Control scan lines
- Provide bidirectional, parallel-to-serial conversion of scan data
- Mask and compare received data

The scan engine interfaces with the scan memory, the SPU, and all the boards in the C3800 system.

9.1.6 Scan Memory

The scan memory contains the data the scan engine uses to perform scan operations. The scan memory consists of a 4K-location by 36-bit memory array that contains the data used for all scan operations. The 36-bit data word consists

of 32 data bits and four parity bits. The memory is divided into four 1K by 36-bit pages, each of which can support a scan ring of up to 32,768 bits.

9.1.7 Workstation-to-CU Interface

The workstation-to-CU (WC) interface, which is physically located on the CU board, connects the SPU workstation to the clock and diagnostic hardware. The WC interface is connected to the SPU workstation through the workstation interface board.The SPU controls the WC interface and uses it to write and read data to and from the clock generator, scan engine, scan memory, and main memory.

The WC interface receives address, data, and control signals from the SPU. It synchronizes these signals to the system clock on the CU board and distributes them to logic on the board. The WC interface decodes addresses received from the SPU and generates the appropriate select signals during data transfer operations.

9.1.8 XP Interface

The XP interface contains three functional areas. All are related to the SPU. Those three functional areas provide:

- Part of the SPU's memory access data path
- Hardware to initialize and pattern test main memory
- The means to send and receive system interrupts

PRELIMINARY

9.2 Operations

The clock and diagnostic hardware is used to initialize the scalar and vector processors, perform scan-based testing of the boards in the system, and operate the system boards at different clock speeds for testing and troubleshooting. Initialization of the scalar and vector processors is performed by downloading information contained on the SPU disk drive to the writable control stores located in both the scalar and vector processors. The scan hardware is capable of not only downloading (writing) WCS information to the SP and VP, it can also read back and verify what was written.

Scan-based testing involves a process of cycling known data through the system boards and examining the data upon return to the scan engine. If the returning data does not match what was sent out, an error indication is generated.

Control of the system clocks makes it possible to operate system boards in single step, microstep, or busrt mode.

The text that follows describes WCS initialization, WCS verification, and scan-based testing.

9.2.1 Writable Control Store Initialization

The scalar and vector processors are initialized by writing data from the SPU disk to the writable control store located in each processor. The steps involved in this WCS download operation are explained in the text that follows and illustrated in Figures 9-2 and 9-3.

The scalar and vector processors require different initialization data. However, because the scan engine can write to more than one board at a time, all the scalar processors in the system or all the vector processors in the system (but not both scalar and vector) can be initialized simultaneously.

Writing to a WCS location requires three steps:

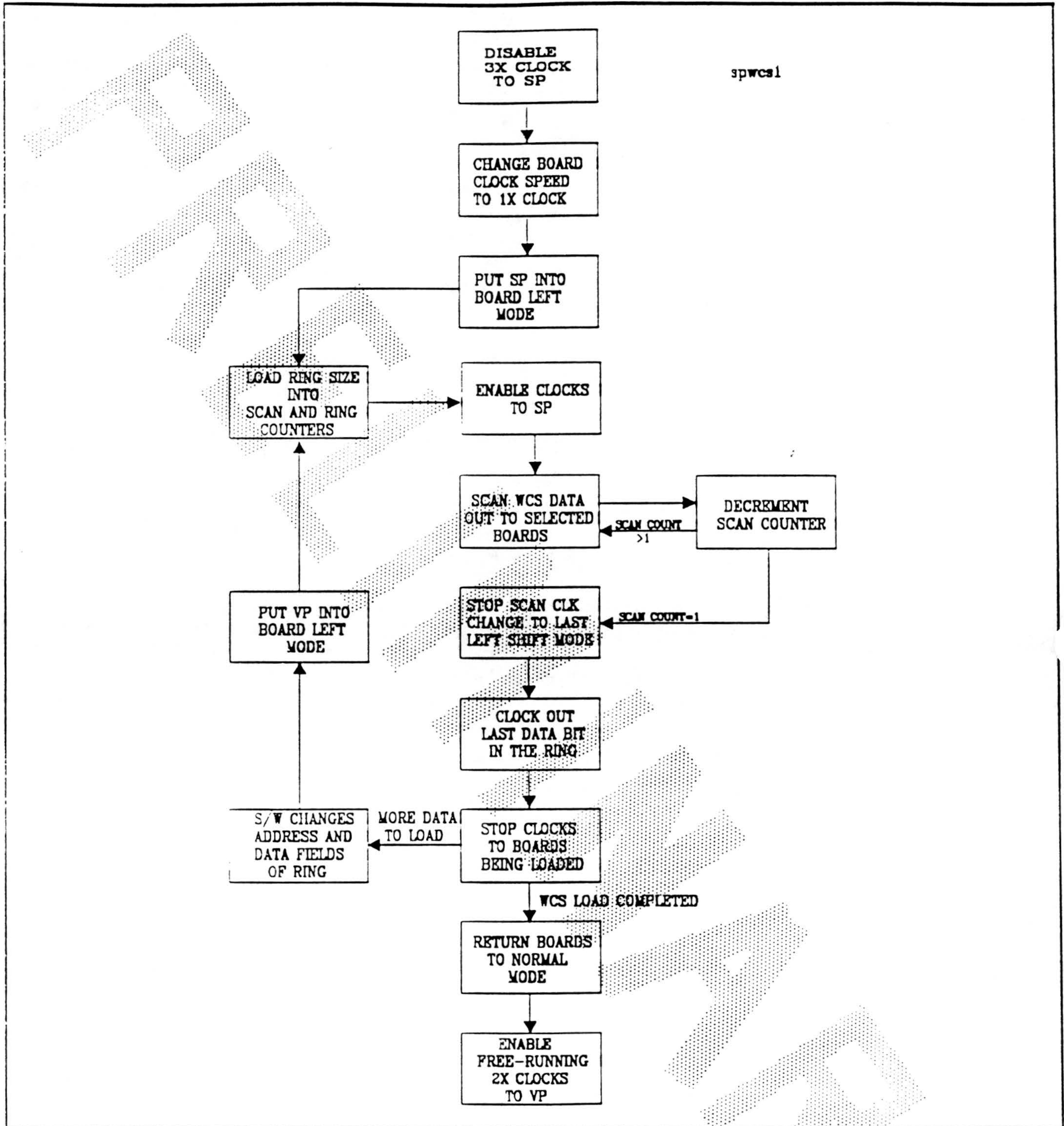
1. Load the write data into scan memory.
2. Shift the address, write enable, and write data into the selected boards.
3. Clock the write data into the memory chips.

The process of writing to the control store begins by filling the scan memory outgoing data page with the data that will be scanned out to the selected boards. The clocks to the selected boards are disabled and their scan control lines are set to the correct mode for writing scan data (board left). For vector processors, load the scan and burst counters with the ring size. For scalar processors, load the scan and ring counters with the ring size. Scan the WCS data (address, data, and control information) out to the selected board(s).

At this point, the data, address and control information are present at the control store input on the respective processor boards. The method required to clock the WCS information into the control store differs between scalar and vector processors and is described separately in the next two paragraphs.

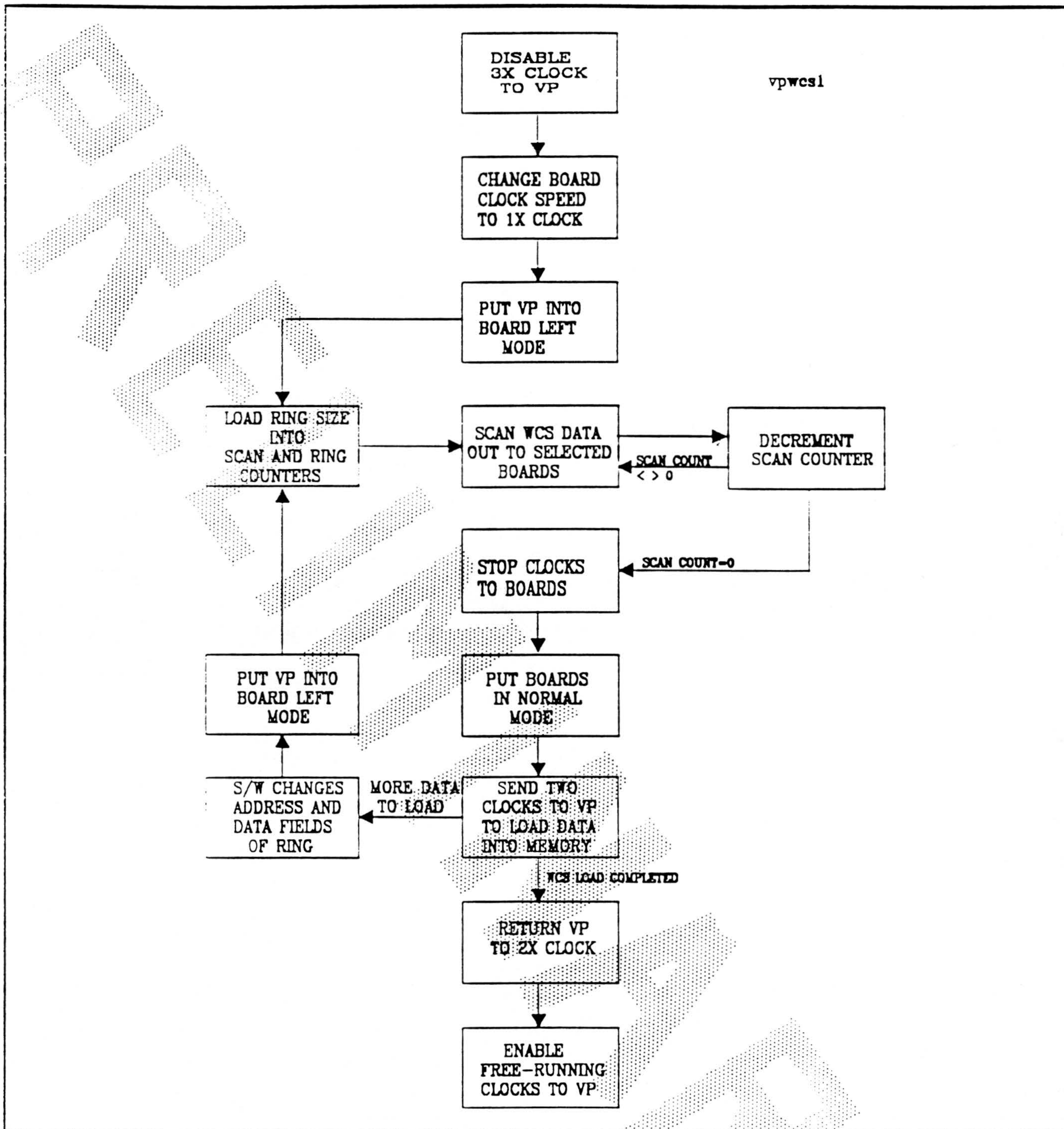
For scalar processors, the scan clock is stopped and the scan control mode is changed from *board left* to *last left shift*. This automatically executes whatever command is at the control store input. In this instance, a control store write operation is performed.
For vector processors, after the entire scan ring is shifted into the board, the scan control mode is changed from *board left* to *normal*. Then two 1x clocks are issued to the board. The clocks cause the data to be written into the control store.

Figure 9-2   Scalar Proc WCS Download Flowchart



PRELIMINARY

Figure 9-3   Vector Proc WCS Download Flowchart



PRELIMINARY

One iteration of the WCS download is complete. Now the SPU fetches new data for the scan memory so the download operation can continue. However, there is a difference. For the remainder of the download the process is sped up by changing only those fields in the scan ring that change from location to lo[illegible]. Since only the address and data fields change, and they occupy only a sma[illegible] portion of the scan ring, the time to load data into scan memory decreases significantly.

Once the data and address are reloaded into the outgoing data page of the scan memory, the SPU reloads the scan engine control registers and enables a new scan write operation. This iterative process of writing scan memory, scanning th data out to the boards, and clocking it into the respective control stores continue until the entire WCS download is complete.

9.2.2 Writable Control Store Verification

WCS verification requires much more time than a WCS write. A WCS write can be performed on all boards of the same type simultaneously; whereas, only one board at a time can be read and verified. Additional time is also required becaus each verification includes both a control store write and read operation.

The steps required to perform WCS verification are explained in the text that follows and illustrated in Figures - and -.

The WCS verification process occurs in five steps:

1. The scan ring containing the address and read enable for the control store location being accessed is loaded by the SPU into the outgoing data page of scan memory.
2. The ring is scanned into all of the processors whose WCS was loaded previously.
3. The SPU changes the scan mode and clocks the memory data into the scan ring.
4. The processors are scan read one at a time and verified against expected data.
5. Between each scan read, the SPU polls the scan engine error register to see if an error was detected. After all of the boards have been verified, the SPU increments the address and repeats the process for the next location of the framestore.

Verification of the scalar and vector processor WCS begins by transferring the address and read enable ring data from the SPU disk to the outgoing data page of the scan memory. The clock to the selected boards is changed to 1x, the scan and burst counters are loaded with the ring size, and a scan write is enabled. When the scan write completes, the address and read enable will be at the contr[illegible] store input, but the read will not have been executed.

The scalar and vector processors use different methods for reading data out of the control store and into the scan ring. The scalar processor uses a two-step process to get the data from the control store to the scan ring. Before the last bit [illegible] scanned into the board, the scan control lines are changed to put the board into the *last left shift* mode. This tells the control store RAM that the scan is ending and that a read operation should begin. The scan control lines are changed agai[illegible] This time to the *normal* mode. The board then receives a single clock which transfers the data out of the control store RAM and into the scan ring. Only one step is used for reading a vector processor's control store. After the entir[illegible] is scanned into the board, the scan control lines are changed to the normal [illegible]e. One clock is issued to transfer the data out of the control store RAM and into th[illegible] scan ring.

Figure 9-4 Scalar Proc WCS Verification Flowchart

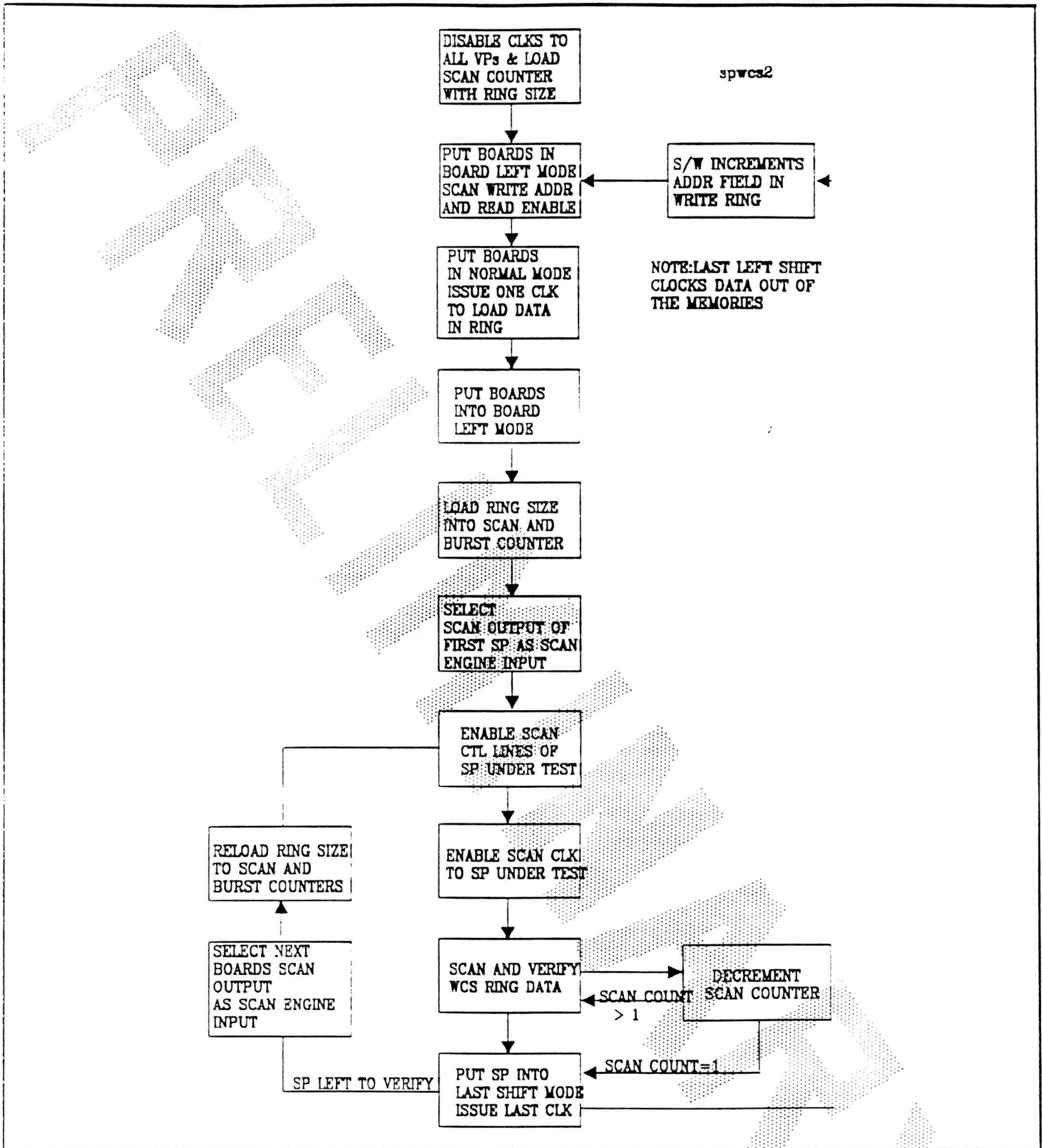
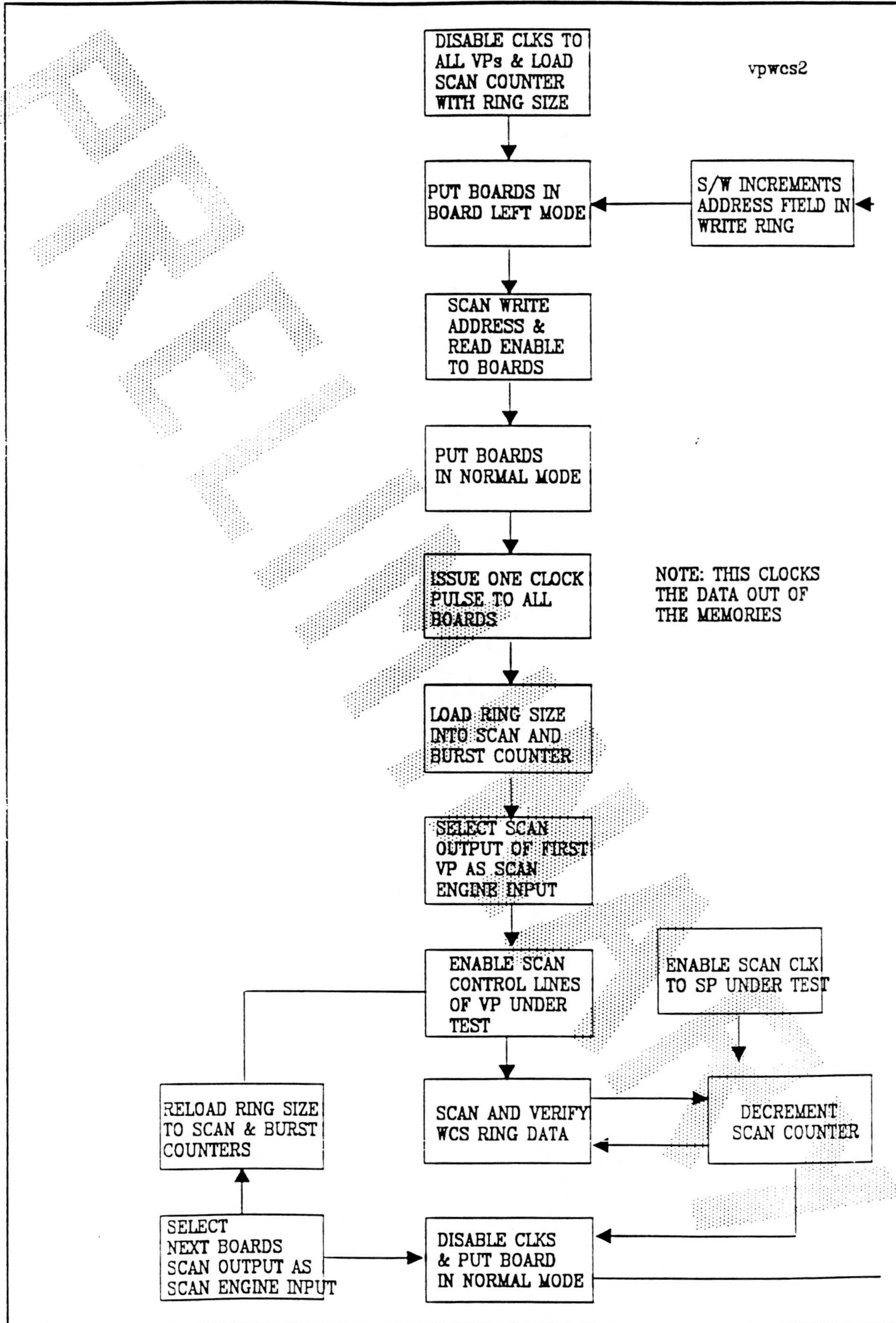


Figure 9-5 Vector Proc WCS Verification Flowchart



PRELIMINARY

Before the ring data is read and verified, the SPU loads the mask and compare data into the appropriate scan memory pages. Once this occurs, scan verification can begin. However, unlike a scan write, the scan engine can only read one board at a time. The software determines which board is read by programming the select lines on the scan engine's input multiplexer. Since the data contained in the WCS is identical for all boards of the same type, the same mask and compare data is used to verify all eight boards. After each board is verified, the SPU checks for an error. If no error is detected, the next board verification is enabled. If an error is found, the error location register is read and the data that contains the error is read from the result data page of scan memory.

After all eight boards of a particular type have been verified, the SPU updates the address and expected data fields in the scan memory for the next control store location. The same mask ring may be used for all control store locations.

9.2.3 Scan-Based Testing

Scan-based testing involves cycling known data through the system boards and examining the data upon return to the scan engine. If the returning data does not match what was sent out, an error indication is generated. Scan-based testing can be executed in several modes, including single-step and burst modes.

9.2.3.1 Scan Writing to System Boards

Scanning data out to the system boards, or scan writing, is the process of transferring board data contained on the SPU disk to the boards in the system. This function is basically the same as WCS downloading. The data transfer occurs in two stages. First, the data is transferred from the SPU disk to the scan memory under control of the SPU workstation microprocessor. Then the scan engine executes a serial transfer of the data from the scan memory to the selected board(s).

The transfer of data from the SPU disk to the selected boards is controlled by the software in a five-step process.

1. The outgoing ring data is transferred from a file on the SPU disk to the outgoing data page of scan memory.
2. Disable the clocks and activate scan enables for all the boards under test.
3. Program the boards for a scan write. This is done by changing the scan control code to board left mode and resetting the recirculation line.
4. The clocks are set to the 1x rate and the scan and burst counters are loaded.
5. When the data has been loaded into scan memory and the command registers set up properly, the scan engine is enabled and the data transfer to the boards begins.

The scan engine is enabled by setting the burst command code in the clock generator's command/status register. This causes clocks to be issued to the board(s) under test, and to the scan engine's output shift register.

During the actual process of scanning, the scan engine does not require any interaction with the SPU. The scan engine will detect when its shift register is becoming empty and automatically fetch the next data from scan memory. While the scan engine is writing, the SPU is preparing the next ring data and polling the status of the scan engine's busy bit. The busy bit is located in the scan engine command/status register. When the scan and burst counters reach zero, all of the ring data will have been transferred to the board(s). The clocks to the board(s) are disabled and the busy bit is reset. Resetting the busy bit tells the polling SPU that the scan write has completed.

9.2.3.2 Scan Reading and Ring Verifcation

Scan reading and ring verification includes three basic steps:

1. Reading data from a board under test
2. Masking off unwanted data
3. Comparing the results against the expected data

While the data is being read from the board, the scan engine fetches the ring mask and compare data from scan memory. As soon as 32 bits (one word) of dat have been read in from the board, it is loaded into the input buffer register. The contents of the input buffer are then masked and compared with the data from scan memory. The result is checked for errors and loaded into the resultant data page of scan memory. If an error is detected, the current number of the scan counter is stored in the error location register.

The mask, comparison data, and result all occupy the same address, but in different pages of scan memory.

When the scan engine completes the read and verification process, it disables th clocks to the board under test and resets the busy bit in the scan engine control register. This indicates to the polling SPU that the process is complete. The SPU then reads the error register. If there are no errors, the SPU can continue with th next test. If there is an error, then the SPU reads the error location register to get the scan memory address of the ring data that contains the error.

9.2.3.3 SST and CAST

SST (System Scan Test) and CAST (CONVEX At-Speed Test) are the primary methods for testing boards. SST checks the interconnects and logic on the boarc (DC testing), while CAST verifies that the board functions properly at nor--l operating speeds. Both SST and CAST use the scan write and read proce previously described. Basically, it is a three-step process that includes:

1. Loading scan memory
2. Writing the scan data to the system boards
3. Reading and verifying the scan data

SST and CAST differ in the way data is scanned into the system boards. SST usι the board left mode and CAST uses the load mode. These modes are described elsewhere in this chapter.

9.2.4 Single and Microstep Operation

Single and microstep allow the user to walk through individual events in the system by controlling the flow of clock pulses to a single board or group of boards in the system. In between the clock pulses, software can perform a non-destructive scan out of the state of the boards under test to determine whether they are operating properly.

Microstepping is a single-board test where the phase relationship of the steppec clock with respect to the other clocks in the system is not important. Software has the capability to send one or two clock pulses to the selected board(s). Each board has a clock pulse rate register that determines the clock pulse rate for tha board. 1x, 2x, or 3x are the supported step clock rates. The 1x, 2x, and 3x clocks pulses are sent out on the same edge. If the microstep occurs more than once th phase relation between the clocks will change. That is why microsteppir ou be limited to testing a single board. When microstep mode ends, the har e automatically re-synchronizes the step and free-running clocks.

Single stepping allows one system clock (1x) period worth of clocks to be issued to the boards under test. For example, a board receiving a 3x clock would receive three clocks at speed: two 2x clock pulses and one 1x clock pulse. In this mode, boards operating at different clock rates maintain the same phase relationship that they have when free running.

The contents of the command/status register determines whether single or microstepping is selected, and the contents of the command enable register determines which board(s) will be involved in the operation.

9.2.5 Burst Mode Operation

Burst mode is a variation of single stepping. In burst mode the clock generator operates at full speed for a specified number of system clock periods and then stops. This provides a function equivalent to setting breakpoints. A 16-bit counter determines the number of clock periods per burst. A count of one is the same as executing in single-step mode.

To execute in burst mode, the clock generator command and status register must contain the burst mode command code. This will cause the clock generator to free run for the number of clock periods specified in the burst counter register.

When execution begins, the busy bit sets in the command and status register. During execution of burst mode, all of the clocks maintain the correct phase relationship with each other.

When the burst count reaches zero, the busy bit is reset and all of the selected clocks are disabled in the high state. The reset condition of the busy bit indicates to a polling SPU that the operation has been completed. The clocks will remain high until another clock command is issued. If the restart command is given, the clock generator will resynchronize the stopped clocks with the free running clocks. Once synchronized, the clocks will free run at the rate specified in the clock frequency control register.

PRELIMINARY

9.3 Scan Control Modes

Each board in the system receives four scan control signals. The four signals consist of a three-bit scan control code and a recirculation signal. The scan contrc code determines the scan mode that will be executed and the recirculation s al determines the source of the scan data that is input to the board under test.

Of the eight possible scan modes, six are currently defined. Three of the six modes are general purpose modes that are used by all the boards in the system. They are called:

- Normal mode
- Load mode
- Board left mode

The other three modes are used for boards that have unusual scan requirements. They are called:

- Shift left log
- Shift left sys
- Last left shift

Table xx shows the scan control codes and their corresponding functions.

The recirculation signal determines the source of the scan data that is input to th board under test. This signal selects either data from the scan engine or the data from the board's scan data out line. When the scan engine's output is selected, the scan operation will either be a scan write or a destructive scan read. When the control signal is in this state, it is possible to perform a simultaneous scan read and write to a selected board. When the board's data output is select 1e ring data is recirculated back into the board. As the data is read out of the . ring it is clocked back into the input of the scan ring. After the ring is completel clocked out of the board, the state of the board will have returned to the state it was in before the read occurred. This is referred to as a non-destructive scan rea

The following paragraphs describe the operation of the different scan modes.

9.3.1 Normal

This is the scan disable mode. The system boards are in this mode during norm system operation. In this mode, the free-running data paths and clocks will be connected in the selected board.

The code for normal mode is present at the scan control input of every board th does not have its corresponding scan control enable bit set in the scan control enable registers, which are located on the control crossbar.

The normal mode is used whenever a board is operated in single step or in burs mode.

9.3.2 Load

The load mode is used as the second step in the three step CAST test. Placing a board in this mode forces all the registers on the board to clock data in on each rising edge of the input clock. The registers will clock the data in regardl of tl clock enables that are in place during normal operation. This mode will ed after the CAST test pattern has been loaded into the board(s) under test. Two clocks (normal board frequency) will be issued to the board and after changing scan modes, the resultant data will be scanned back into the scan engine.

Table 9-1 Scan Control Codes

Scan Control Code	Scan Mode	Purpose
000	Normal	Scan disabled
001	Shift Left Log	Special memory scan
010	Shift Left System	Special memory scan
011	Undefined	
100	Load	Used during CAST only
101	Last Left Shift	Used during scan of self-timed RAMS
110	Undefined	
111	Board Left	Primary scan mode

PRELIMINARY

9.3.3 Board Left

This is the primary mode used to scan data in and out of boards in the system. By entering board left mode, all of the registers in the board under test are connected into a large shift register with each stage being clocked by the input clock. By recirculating the output data to the input, the software may scan read the contents of a selected board without disrupting the state of the board. If the data is not recirculated during a scan read, the state of the board is lost. However, if the scan engine output is selected as the scan input to the board under test during a scan read, it is possible to perform a simultaneous scan read and write.

9.3.4 Shift Left Log

The shift left log mode is used to scan out the contents of the *memory log* from a memory board.

This scan is a non-destructive read. To assure that the scan is non-destructive, the set-up, execution, and exit from this scan mode must occur in sync with the board's 2x clock. This is essential because memory refresh requires a 2x clock.

The scan control lines will be clocked out of the scan engine and into the memory board on the rising edge of the 1x (system) clock. Data will immediately begin shifting out on the rising edge of the 1x clock. When the last bit is shifted out of the log ring, the memory board's scan control lines will be changed to normal mode, and the board will be returned to normal operation.

9.3.5 Shift Left Sys

The shift left system mode is used to scan out the *system log* from a memory board. It operates the same as the shift left log mode. This scan is a non-destructive read.

9.3.6 Last Left Shift

This mode is used to read and write self-timed RAMS with the scan ring. The last left shift mode is used for boards, such as the I/O interface adapter and the scalar processor board, that have self-timed RAMS as part of their on-board memory.

Whenever the scan engine's command/status register indicates that a self-timed RAM is being accessed during a scan operation, the scan engine automatically enters the this scan mode for the last bit of the scan. This mode tells the self-timed RAMS to execute the contents of their input register.

9.4 Clock and Scan Control Registers

The clock and scan control registers reside in memory located on the CU board. They are central to all clock and scan operations. There are two sets of registers: one set for the clock generator and another set for the scan engine. Some of the registers between the two sets have identical names. For example, there are two separate command/status registers, one for the clock generator and another for the scan engine.

The text that follows provides a map of the scan and clock memory followed by brief description of each control register.

9.4.1 Scan and Clock Memory Map

The scan engine, scan memory, and clock generator are accessed as a block of memory locations. The memory block consists of 8,224 locations of 32-bits each. The scan memory occupies the first 8,192 locations, over half of which is reserve for future scan memory expansion. Of the remaining 32 locations, 16 are allocated to the clock generator control registers and 16 to the scan engine contr registers . Table xx defines the scan and clock memory map.

PRELIMINARY

Table 9-2 Scan and Clock Memory Map

Address	Register Name	Group
0x8080	Reserved	Not Allocated
0x8078	Hard Error Register II	Scan Engine Control Registers
0x8074	Hard Error Register I	
0x8070	Scan Control Enable Register II	
0x806C	Scan Control Enable Register I	
0x8068	Board Reset Register II	
0x8064	Board Reset Register I	
0x8060	Scan Compare Register	
0x805C	Scan Mask Register	
0x8058	Input Buffer Register	
0x8054	Output Buffer Register	
0x8050	I/O Address Register	
0x804C	Scan Counter Register	
0x8048	Error Location Register	
0x8044	Command/Status Register	
0x8040	Reserved	Not Allocated
0x8024	Burst Counter Register	Clock Generator Control Registers
0x8020	Disabled Clocks Register II	
0x801C	Disabled Clocks Register I	
0x8018	Clock Frequency Control Register III	
0x8014	Clock Frequency Control Register II	
0x8010	Clock Frequency Control Register I	
0x800C	Command Enable Regster II	
0x8008	Command Enable Regster I	
0x8004	Command/Status Register	
0x8000	Reserved for Scan Memory Future Expansion	Not Allocated
0x4000		
0x3000	Result Data	Scan Memory
0x2000	Comparison Data	
0x1000	Ring Mask	
0x0000	Outgoing Data	

9.4.2 Clock Generator Control Registers

The clock generator control registers are briefly described in the text that follows

9.4.2.1 Command/Status Register

Address = 0x8000

The 32-bit command /status (CS) register contains the clock generator command and its execution status. A three-bit command select code specifies the command and a single busy bit indicates its execution status. The remaining 28 bits in the register are reserved.

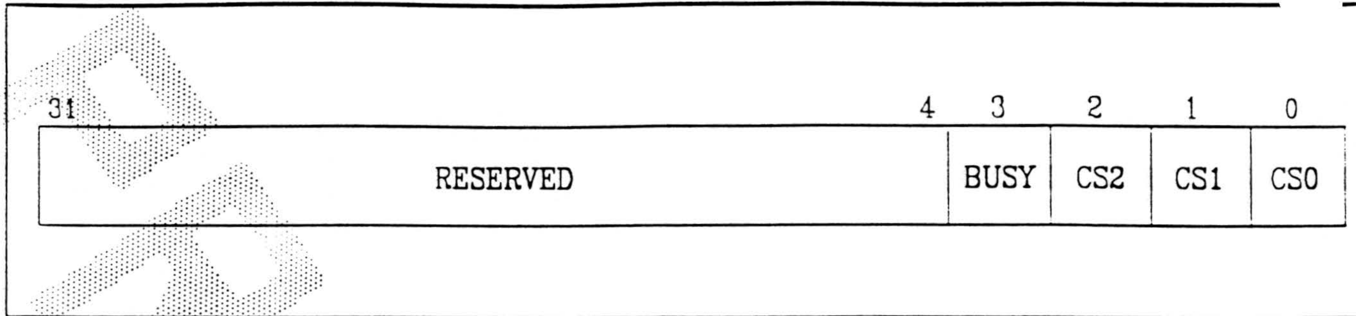
The command/status register receives commands from the SPU. The act of writing to the command/status register causes the command specified in the CS bits to be executed immediately. Figure illustrates the format of the command/status register.

The busy bit is a one bit read-only field in the CS register that indicates the command execution status of the clock generator. After the SPU writes a command to be executed to the CS register, the clock control engine will set the busy bit. When set, this bit indicates to the SPU that the command is in the process of being executed. The busy bit remains set until the command is completed; when execution is completed, the control engine resets the busy bit.

The three-bit command select field contains the encoded clock generator command. The command originates from the SPU. The eight available commands are listed in Table xx.

PRELIMINARY

Figure 9-7 Clock Generator Command/Status Register



PRELIMINARY

Table 9-3 Clock Generator Command Codes

Code	Command
000	Disable clocks
001	Burst operation
010	Scan
011	Restart clocks
100	Step one clock
101	Step two 2x or 3x clocks
110	Step two 1x clocks
111	System step

9.4.2.2 Clock Generator Commands

The *disable clocks* (stop) command disables all the clock drivers specified by the command enable register. The selected clocks remain disabled until a restart command is issued (provided the drivers are still selected).

The *burst* command instructs the clock generator to free run for the number of clock periods specified in the burst counter register. During execution, all of the clocks maintain the correct phase relationship with each other. See burst mode description (9.8.6). When the burst counter reaches a count of zero, the busy bit is reset and the selected clocks are disabled on the last rising edge of the 1x clock.

The *scan* command instructs the clock generator to issue a programmable number of 1x clocks to the board slots specified by the command enable register. The burst counter register specifies the number of clocks. The scan command overrides (but does not alter the contents of) the clock frequency register. Once the command is executed, the control engine sets the busy bit in the CS register and enables the clock drivers. When the burst counter reaches a count of zero, the busy bit is reset and the selected clocks are disabled on the last rising edge of the 1x clock.

The *restart* command causes the disabled clocks specified by the command enable register to be resynchronized and to begin free running.

The *step one clock* command instructs the clock generator to issue a single clock to boards selected by the command enable register. Each board selected will receive simultaneously a low-going pulse equal to the period of the 6x clock. This is equivalent to issuing the clock that occurs just before the rising edge of the 1x clock. Clocks on boards that receive different clock rates will lose their phase relationship if the command is executed more than once. Therefore, use of this command should be limited to boards that have the same clock rate.

The *step two 2x or 3x clocks* command causes the clock generator to issue two 2x or 3x clocks to the boards selected by the command enable register. The contents of the clock frequencey register determines whether the clocks will be 2x or 3x. If this command is used on a board receiving a 1x clock, only one clock period will be issued. The two clock periods occur immediately before and after the rising edge of the 1x clock. Clocks on boards that receive different clock rates will lose their phase relationship if the command is executed more than once. Therefore, use of this command should be limited to boards that have the same clock rate.

The *step two 1x clocks* command causes the clock generator to issue two 1x clocks to the boards selected by the command enable register. The control engine assumes that this command will only be executed on boards that have 1x specified as their clock rate in the clock frequency command register. Boards with 2x or 3x specified as their clock rate will receive three and four clock periods, respectively.

The *system step* command causes the clock generator to issue clocks for one system clock period to boards selected by the command enable register. The number of clocks issued is determined by the clock frequency control register. 3x boards will receive three clocks, 2x boards two clocks, and 1x boards one clock. The phase relationship between the three clock rates is preserved with the use of the system step command. This command may be used repeatedly on boards that have different clock rates.

9.4.2.3 Command Enable Registers

Address - 0x8004 and 0x8008

The two 32-bit command enable registers activate the diagnostic clock moc specified by the clock generator's command/status register. One enable bit is assigned to each board slot in the system. When a board slot enable bit is reset, the free-running clock (specified in the clock frequency control register) is enabled to the board. If the enable bit is set, the slot will be placed in the diagnostic mode specified by the command/status register. Due to clock disabling and resynchronization restrictions, it is important that boards enter anc exit a diagnostic mode at the same time. Therefore, it is essential that the command enable register be set up correctly before the command/status register is loaded. The organization of the command enable registers is illustrated in Figure -.

[Insert Figure.... Command Enable Registers]

The command enable register only affects operation of the clock generator. For example, setting up the command enable register to enable the appropriate boarc slot(s) and coding the clock generator's command/status register with a scan command is not enough to invoke scanning. To completely set up a diagnostic function, such as scanning, the SPU must program the control registers in both the scan engine and the clock generator.

There are separate command enables for the scan engine and clock generator because there are times when they must be set differently for the same board slot For example, for single step and burst mode operations the board functions normally, except that the clock rate can be altered in a variety of ways. Thus, the clock generator control registers are set up to accomplish the desired operation, but the scan generator is set up for normal (non scan) operation. The most efficient way to do this is by simply resetting the scan control enable register.

9.4.2.4 Clock Frequency Control Registers

Address = 0x800C, 0x8010, & 0x8014

The three 32-bit clock frequency control registers determine the frequency of the clocks going to each board slot in the system. Two bits are assigned to each slot in a bay. The two bits allow the software to select one of four available clock frequencies: 3x, 2x, 1x, and disabled. These clock rates are relative to the master oscillator frequency, which is 2 nanoseconds or 6x. The rate specified in the register determines the clock frequency output to the selected board during both free-running and diagnostic modes. It should be noted that in diagnostic scan mode, the clock to the selected boards is forced to 1x, regardless of the contents of the frequency control register. The clock frequency control codes are defined in the Table xx.

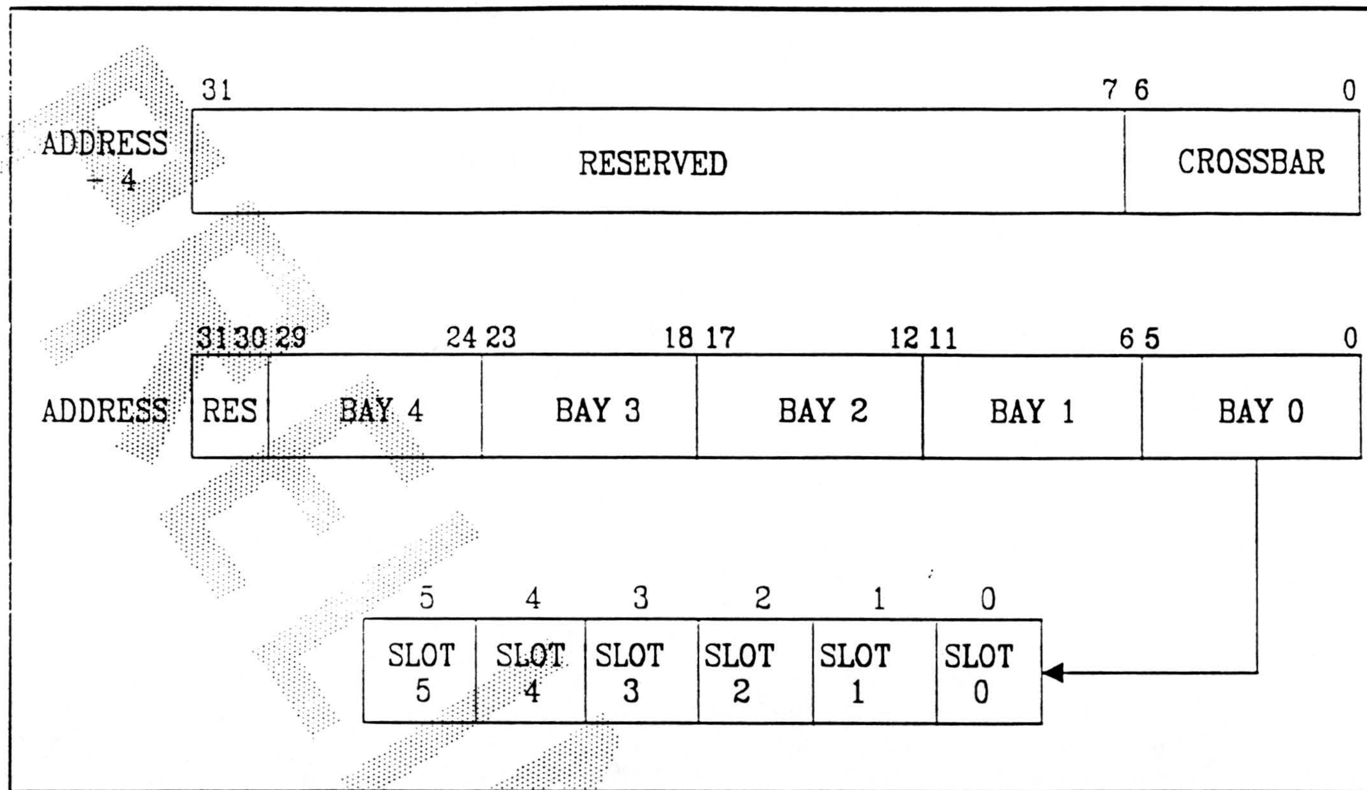
[Insert Table... Clock Frequency Control Codes]

Before attempting to change the clock frequency of any given board slot, the clocks to that board slot should be disabled. This is accomplished by first setting the applicable board slot enable bit and then issuing a disable clocks command.

To bring the clocks back up properly after they've been disabled, first write from the SPU the desired clock rate into the clock frequency register. Then initiate a restart by writing a restart command to the command/status register from the SPU. The clock frequency control registers are illustrated in Figure.

[Insert Figure.... Clock Frequency Control Registers]

Figure 9-7 Command Enable Register

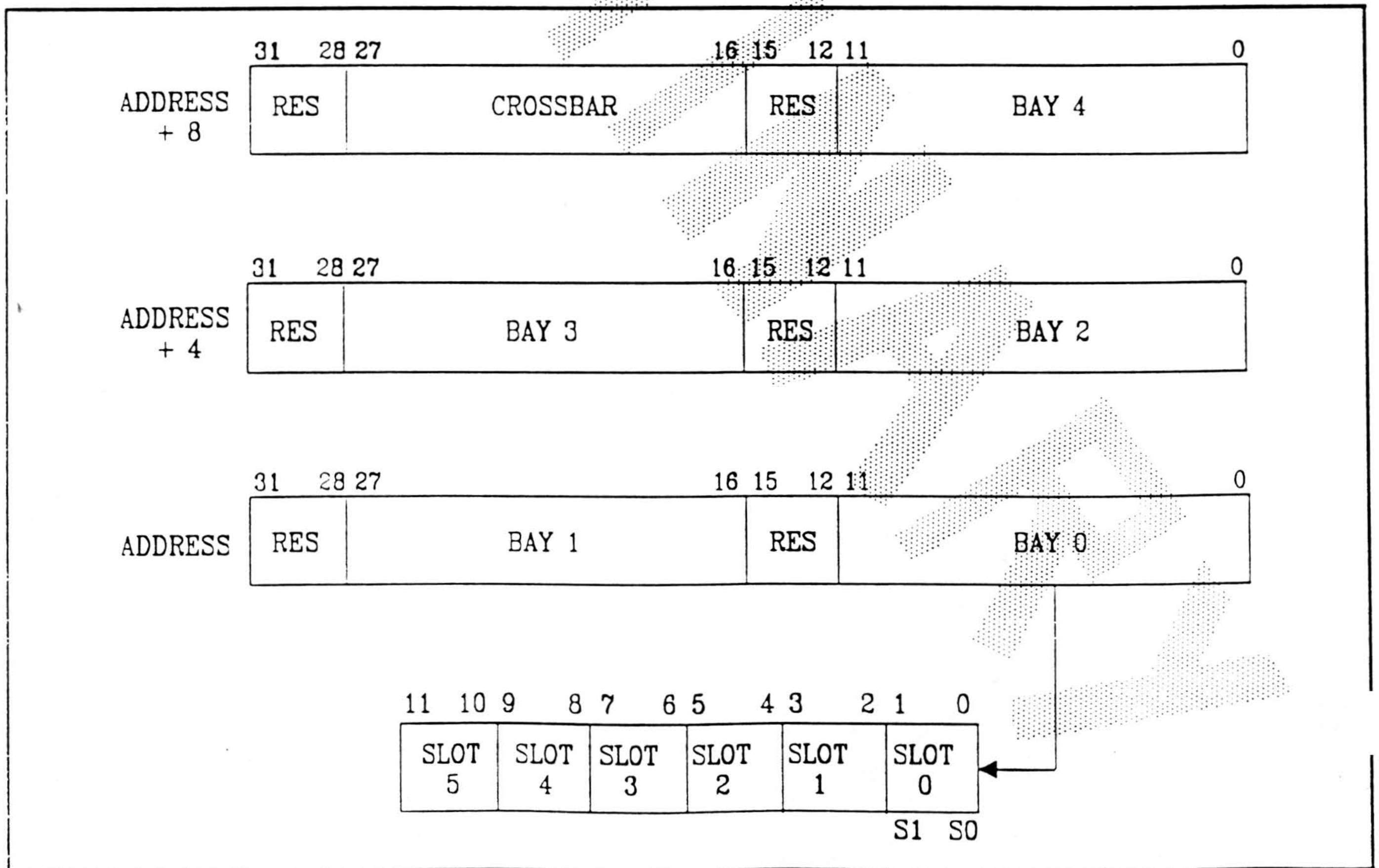


PRELIMINARY

Table 9-4 Clock Frequency Control Codes

Code	Frequency
11	3x Clock (Osc/2)
10	2x Clock (Osc/3)
01	1x Clock (Osc/6)
00	1x Clock (Osc/6)

Figure 9-8 Clock Frequency Control Register



PRELIMINARY

9.4.2.5 Disabled Clocks Registers
Address = 0x8018, 0x801C

The two 32-bit disabled clocks registers indicate which clock drivers are disabled from sending clocks to the system boards. The registers contain one bit for every board slot in the system. Bits are set for board slots whose clocks are disabled. The registers are updated whenever clocks in the system are enabled or disabled. When a clock disable command is issued, the register is loaded with the logical OR of its current contents and the contents of the command enable register (clocks about to be disabled). When a restart clocks command is issued, the active bits of the command enable register will reset the corresponding bits in the disabled clocks register. The organization of the disabled clocks register is illustrated in Figure.

[Insert Figure.... Disabled Clocks Register]

9.4.2.6 Burst Counter Register
Address = 0x8020

The 32-bit burst counter register contains the contents of the clock generator's 16-bit burst counter. Writing to this register causes the burst counter to be loaded with the number of clocks that will be issued to the system boards currently enabled (by the command enable register). Reading the burst counter register enables output of the current contents.

Burst mode and scan use this counter to control the number of clock periods that are issued during a test. Once the test is enabled, the counter will decrement by one on each rising edge of the 1x clock. After the counter reaches zero, the busy bit is reset in the clock generator's command/status register.

The format of the burst counter register is illustrated in Figure x-x.

[Insert Figure.... Burst Counter Register]

9.4.3 Scan Engine Control Registers
The scan engine control registers are briefly described in the text that follows.

9.4.3.1 Command/Status Register
Address = 0x8040

The 32-bit command/status register contains sixteen bits for scan operation commands and one busy bit that indicates whether the current command has completed. The busy bit also determines who can access scan memory.

Writing to the command/status register causes the written command to be immediately executed. Therefore, it is important that all enable and counter registers required for the operation are written prior to writing the command/status register. The organization of the register is illustrated in Figure.

[Insert Figure.... Scan Engine Command/Status Register]

The busy bit is directly driven by the scan engine's master controller. Thus, from the SPU, the busy bit is read-only. When a scan engine command is executed, the master controller will immediately set the busy bit. The SPU can poll the bit to determine whether the scan engine has completed the command. When the command is completed, the master controller will reset the busy bit; this indicates that the scan engine is ready to execute another command.

Figure 9-10 Disabled Clocks Register

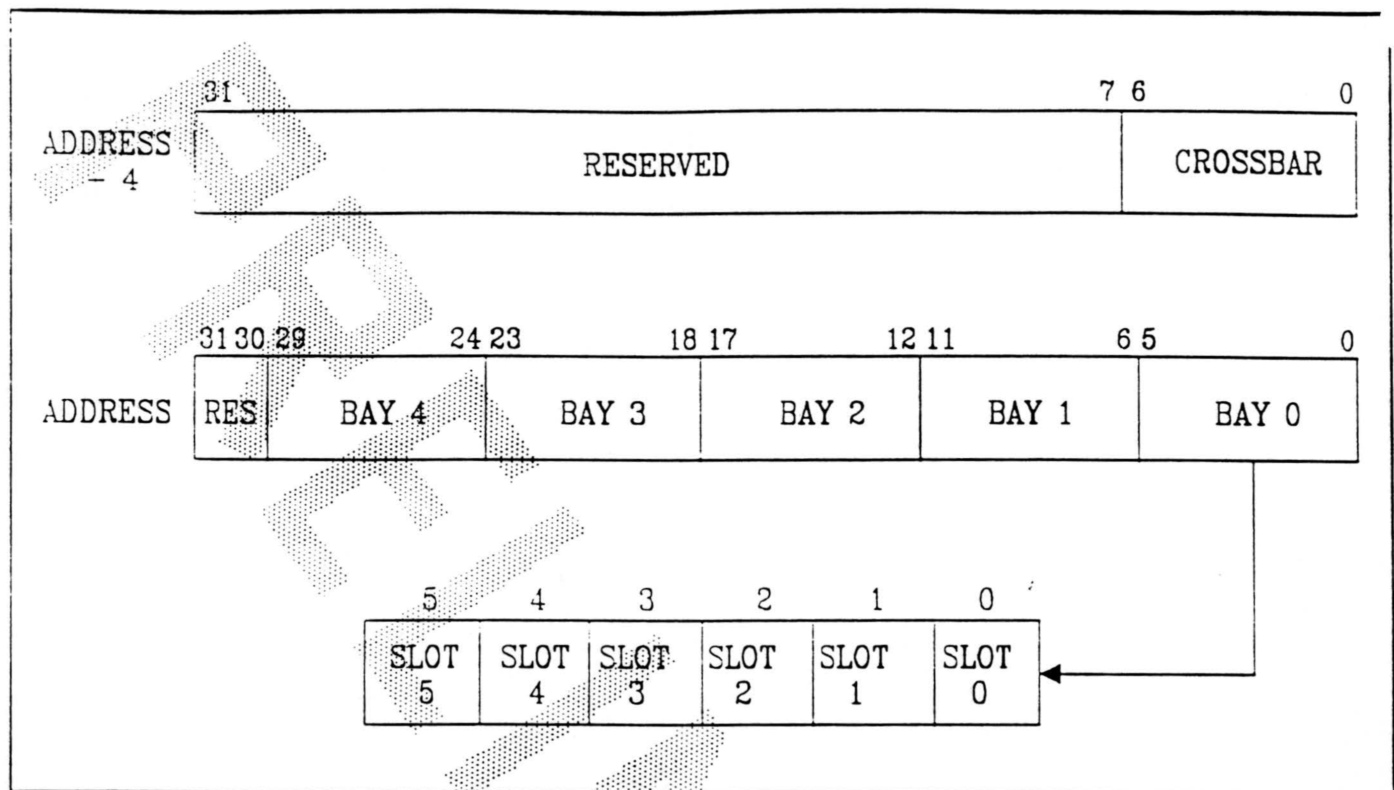


Figure 9-11 Burst Counter Register

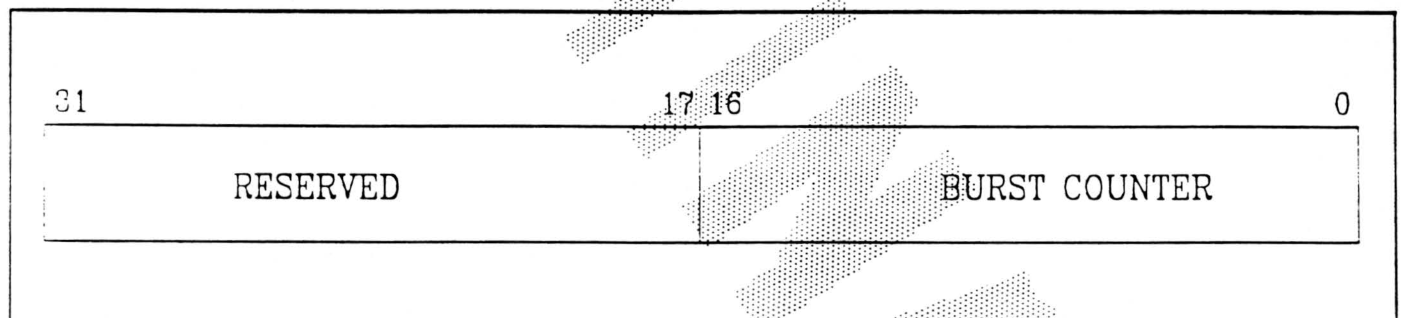
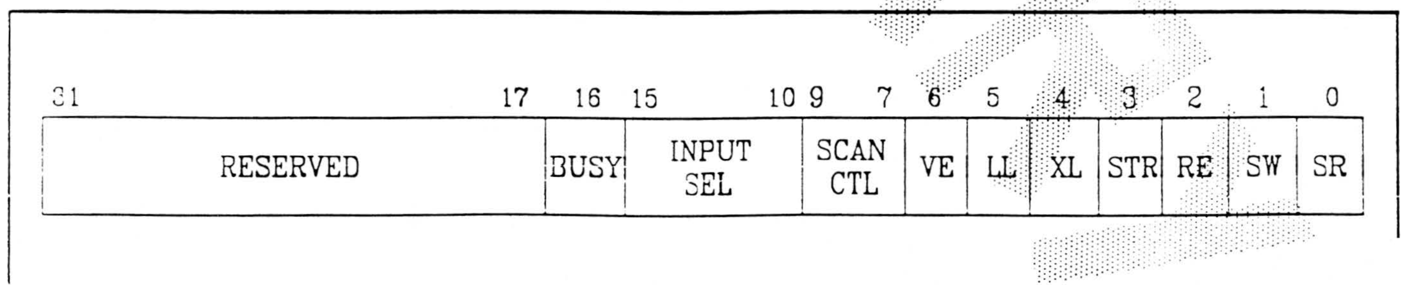


Figure 9-12 Scan Engine Command/Status Register



The busy bit determines who can access scan memory because the busy bit drives the scan memory multiplexer select line.When the busy bit is set, the scar engine is master of the bus and may transfer data to scan memory. When the bit is reset, the SPU is master of the bus and may load or analyze the scan memory contents.

The six-bit *input select field* specifies the board whose scan output will be input t the scan engine. The field directly drives the scan engine input multiplexer, which is location on the control crossbar board.

The three-bit *scan control field* contains a scan control code that will be output to boards selected by the scan control enable register. The scan control codes are described elsewhere in this chapter.

The *VE bit* contains the logical result of ORing the error register bits. By testing the state of this bit, the SPU can determine if an error has been detected during the verification of a ring, without having to perform a separate read from the error register. Resetting the error register automatically resets this bit.

The *LL bit* forces the scan engine to perform a local loopback. The signal drives the select line on the two-input multiplexor that is attached to the front of the input shift register. When the signal is set, the data output of the output shift register is selected and all data shifted out of the scan engine can be scanned ba in. When the signal is reset, the output of the scan line multiplexor is connected to the input of the scan engine.

The *XL bit* forces the scan engine to perform a scan loopback at the data multiplexor on the crossbar control board. The XL bit overrides the input select field of the control register and forces the multiplexor to select the input that is connected to buffered version of the scan data output. This can be used to chec: scan data path integrity across the two bays. When the bit is reset, the output selected by the inut select field is connected to the input scan engine.

The *STR bit* indicates to the scan engine that a self-timed RAM is being accesse in the current scan operation. The STR bit enables special hardware that chang the scan control lines for the last bit of the shift. This causes the control scannec into the memory chips to be executed (see the *last left scan mode* described in thi chapter).

The *RE bit* controls the recirculation of scan ring data during a scan read operation. If data is not recirculated back to a board that is being read, the ring data (and, therefore, the board state) is lost. The bit will be transmitted to boarc selected in the scan control enable register. When the bit is set, data from the output of a board's scan ring will be fed back to its input (on the selected boarc When this bit is reset, the read is destructive or a simultaneous scan read and write will occur.

The *SW bit* enables the ouptut scan engine, thereby enabling the scan writing function.

The *SR bit* enables the input scan engine, thereby enabling the scan reading(verification) function. If both the SW and SR fields are set, the scan engine will simultaneously perform a scan read and write operation with the board(s) selected in the input select field. It should be noted that if both fields a set, the RE field must be reset or else new data will not be written to the board.

9.4.3.2 Board Reset Registers

Address = 0x8060, 0x8064

The 32-bit board reset registers contain the individual board reset lines. Each bit directly drives the reset line for each board in the system. When the bit associated with a board is set, the reset line will become active and the board will be reset. The selected board will remain reset until the SPU clears the boards's reset bit in the register. Figure illustrates the organization of the register.

[Insert Figure.... Board Reset Registers]

9.4.3.3 Scan Control Enable Registers

Address = 0x8068, 0x806C

The 32-bit scan control enable registers contain enable bits for the scan control lines of each individual system board.

If a board is not enabled, the scan control lines default to the normal (scan disabled) mode. The organization of the scan control enable registers is shown in Figure.

[Insert Figure.... Scan Control Enable Registers]

9.4.3.4 Scan Counter Register

Address = 0x8048

The 32-bit scan counter register contains the output from the scan engine's 16-bit scan counter. Reading from this register provides the current contents of the scan counter. Writing to this register loads the scan counter with a value equal to the number of bits in the scan ring under test.

Once a scan operation begins, the contents of this register will decrement whenever a data bit clocks into or out of one of the scan engine shift registers.

The scan counter value is contained in the lower 16 bits of the register as shown in Figure.

[Insert Figure..... Scan Counter Register]

9.4.3.5 Error Location Register

Address = 0x8044

The 32-bit error location register contains the scan memory address of the first verification error detectedby the input scan engine. When the first error is detected in a scan ring, the contents of the input address counter are written into the error location register. This captures the scan memory address of the first incoming data word that contains an error. This record makes it possible for the software to quickly find the faulty data within scan memory. As Figure illustrates, the lower 16 bits of the register contain the error location. The SPU is responsible for clearing the contents of this register.

[Insert Figure.... Error Location Register]

PRELIMINARY

Figure 9-13 Board Reset Register

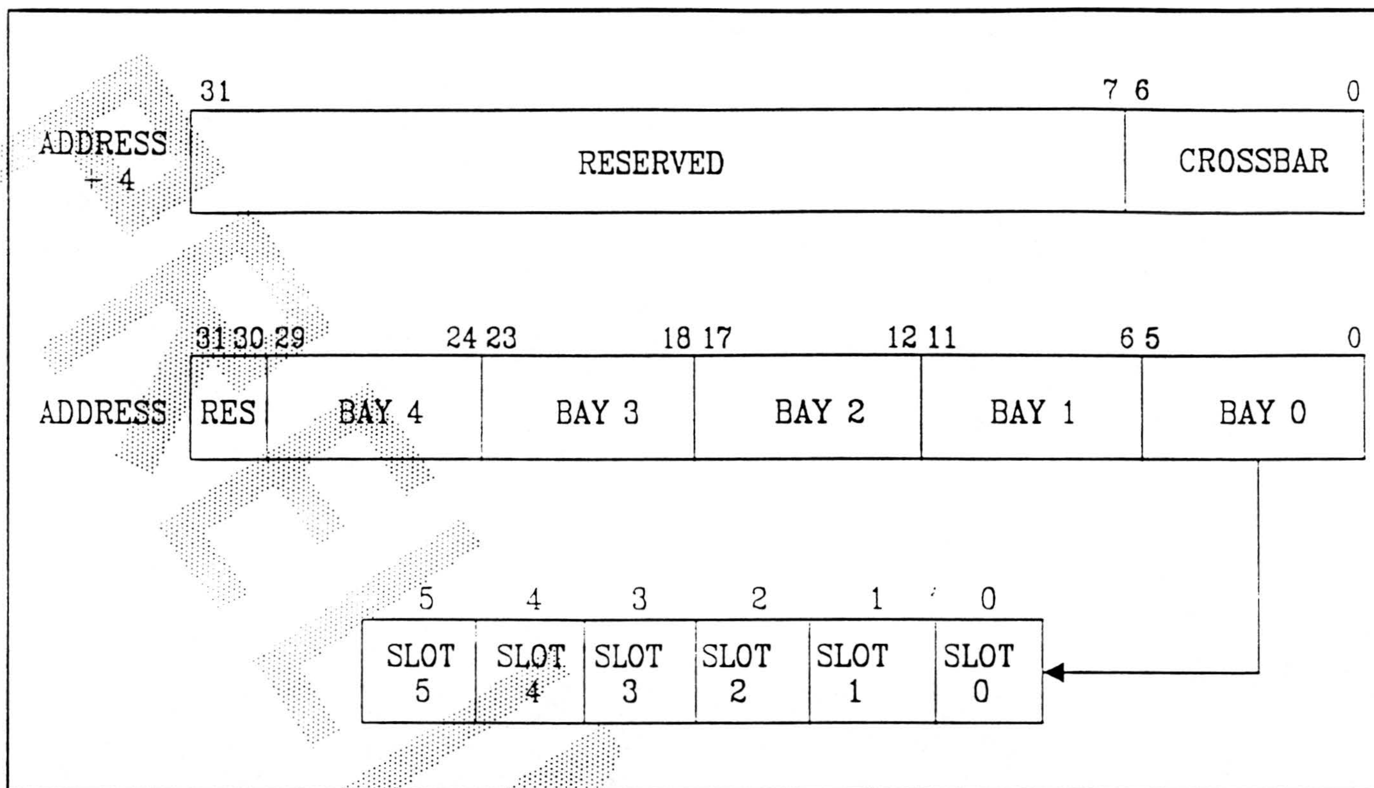
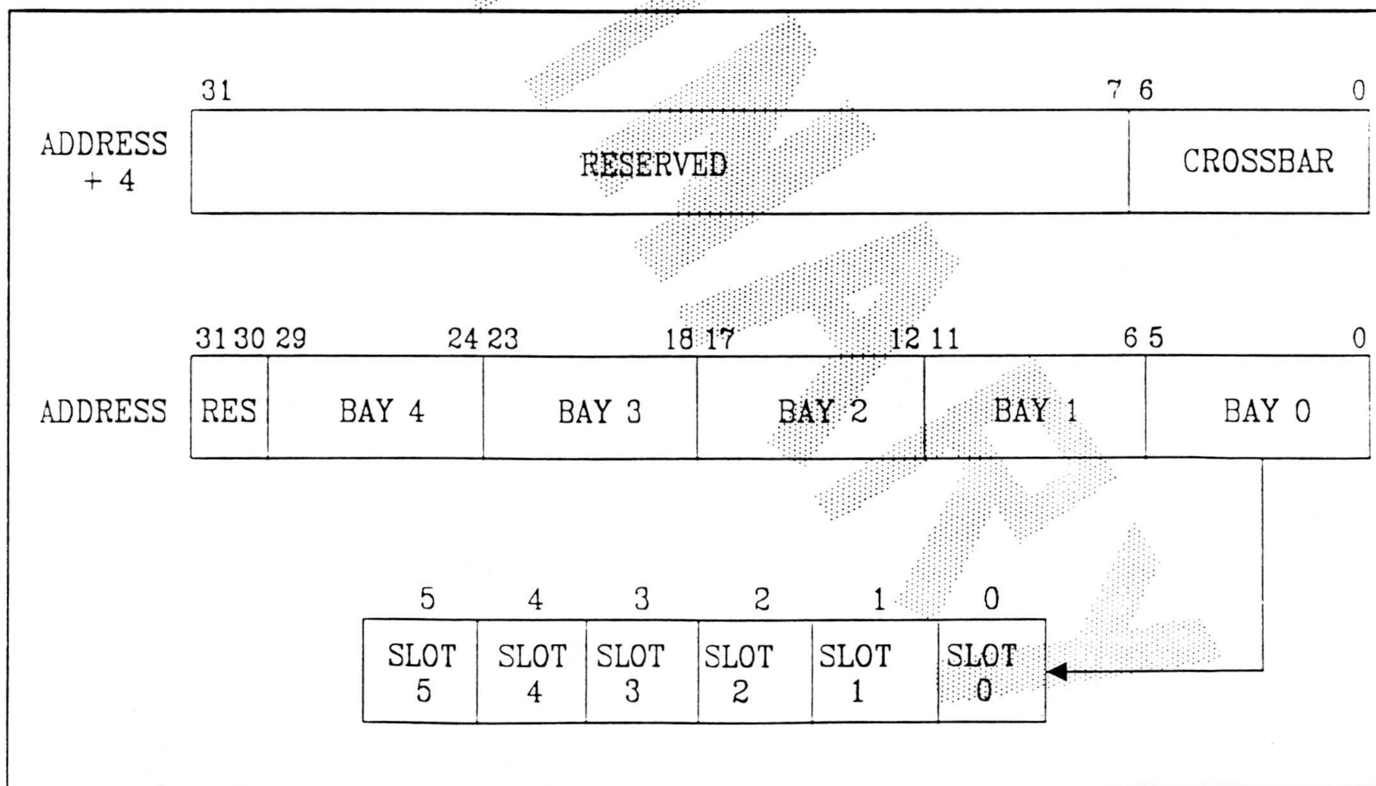


Figure 9-14 Scan Control Enable Register



PRELIMINARY

Figure 9-15 Scan Counter Register

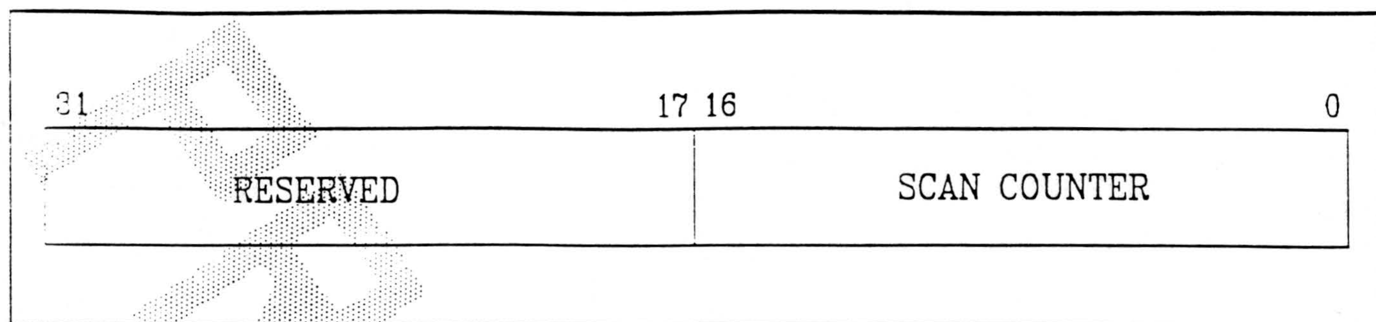
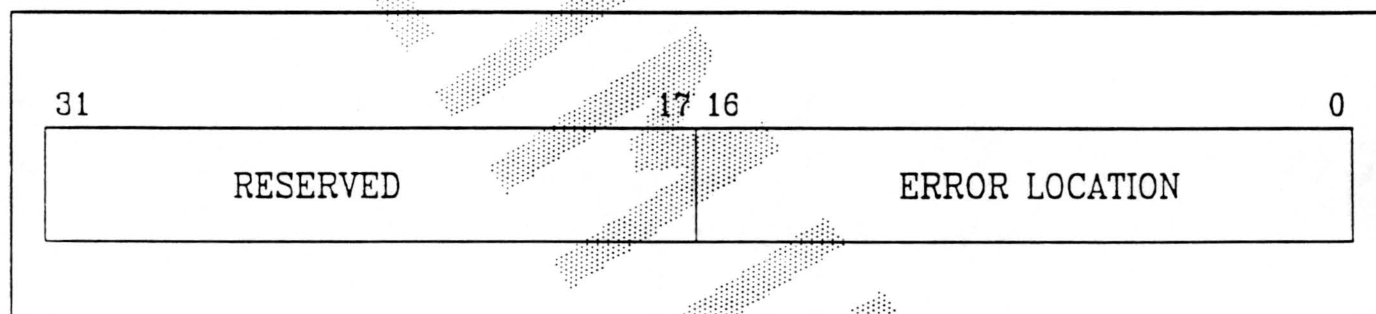
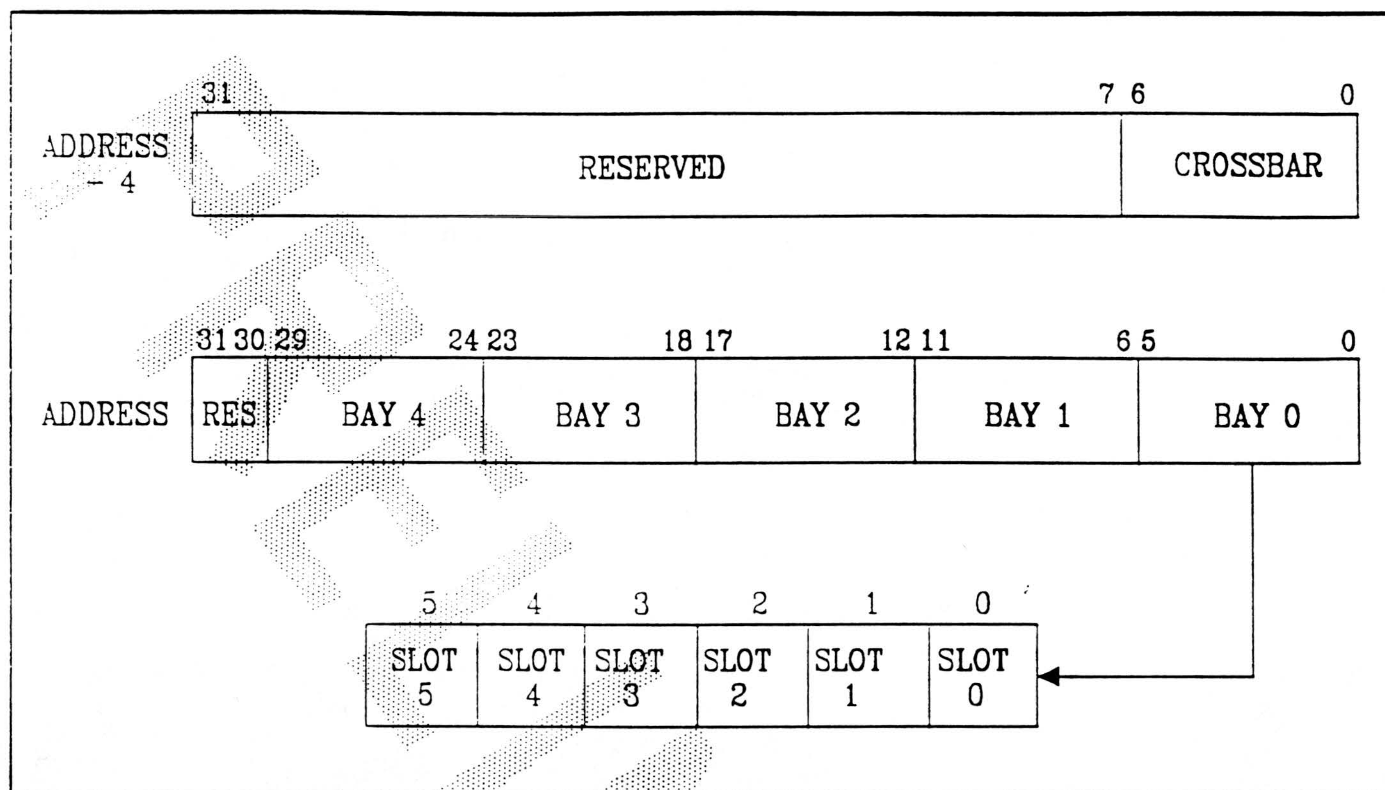


Figure 9-16 Error Location Register



PRELIMINARY

Figure 9-16 Hard Error Register



PRELIMINARY

9.4.3.6 Hard Error Registers

Address = 0x8070, 0x8074

The 32-bit hard error registers capture the hard error signals output from t' individual system boards. The hard error registers continually load the sta the hard error lines on each tick of the system clock (1x).

A hard error means that a fatal error has been detected and that the system must stop. Thus a hard error causes the clock generator to disable the clocks to all boards in the system. To accomplish this, the output of the hard error registers are logically ORed together and the result is input to the clock generator. If this hard error input becomes active, the clock generator will disable the clocks. In addition, the contents of the hard error registers will be held until cleared by the SPU. Holding the data in the registers gives the SPU an opportunity to read the data and determine which system board(s) caused the error. Figure illustrates th organization of the two registers.

[Insert Figure.... Hard Error Registers]

9.4.3.7 Output Buffer Register

Address = 0x8050

The 32-bit output buffer register contains the scan data that will be shifted out to the boards in the system. During scan write operations, data from the output buffer register will be transferred in parallel to the output shift register. Howeve the new data is not loaded into the output shift register until the last bit of previously loaded data (if any) has been serially shifted out to the board(s) in th system. The most significant bit is shifted out first from the output shift register.

After data has been transferred from the output buffer register to the shif register, a flag will be set indicating that the output scan pipeline is no lon full. Then the output scan engine will reload the output buffer register from the outgoing page of scan memory, increment the outgoing address counter, and reset the not full flag.

9.4.3.8 Input Buffer Register

Address = 0x8054

The 32-bit input buffer register contains the raw scan data that was serially clocked into the input shift register from the system board(s). During scan operations, the input buffer register, under control of the scan engine, loads the 32-bit data word in parallel from the input shift register. Since the raw scan data may contain data that is not of interest to the software, the contents of this register is transferred to the mask register where unwanted data is stripped.

The least significant bit of the input buffer register is the last bit that was clocke into the input shift register from the system board(s).

PRELIMINARY

9.4.3.9 Scan Mask Register

Address = 0x8058

The 32-bit scan mask register contains mask data that is used for scan ring verification. During scan ring verification, the scan engine loads the scan mask register with mask data from scan memory. The inverse of the contents of the scan mask register is then logically ANDed with the contents of the input buffer register. Loading a one into a given bit position in the scan mask register will force the corresponding bit in the input buffer register to zero. This masks off unwanted data. The data output from the mask register is input to the scan compare register. The mask register can be bypassed by loading it with all zeros. In this case, the scan data will pass directly from the input buffer register to the scan compare register.

9.4.3.10 Scan Compare Register

Address = 0x805C

The 32-bit scan compare register contains the comparison data for scan ring verification. This register is ordinarily loaded from the scan memory by the scan engine. The contents of this register are exclusive ORed with the input scan data after the unwanted fields have been masked off (see mask register). The output of this register is checked for errors and loaded into the scan memory.

The scan compare register can be bypassed by loading it with all zeros. In this case, the scan data will pass directly from the scan mask register to scan memory.

9.4.3.11 I/O Address Register

Address = 0x804C

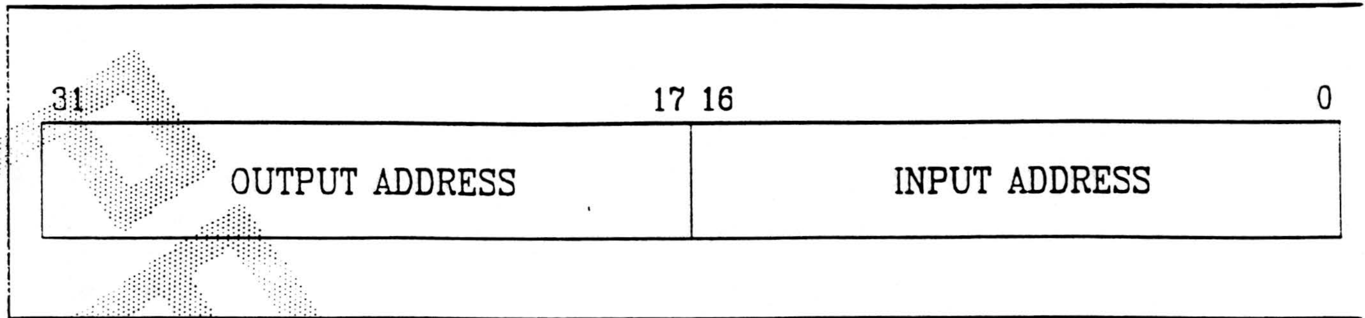
The 32-bit I/O address register contains the contents of the scan engine's input and output address counters. The output address counter provides the scan memory address of data that is to be loaded into the output scan buffer and shifted out to boards in the system. The input address counter provides the scan memory address used to retrieve mask and compare data from scan memory. The input address counter also provides the scan memory address for result data that is destined for scan memory.

The register is divided into two 16-bit sections. The output counter occupies the upper 16 bits of the register, and the input counter occupies the lower 16 bits. Writing data to this register causes the input and output counters to be loaded with input data. Reading this register outputs the current value of both counters. Reading the counters before enabling them will allow the software to verify proper loading of the register. The I/O register is illustrated in Figure.

[Insert Figure..... I/O address register]

PRELIMINARY

Figure 9-17 I/O Address Register



PRELIMINARY

9.5 Clock Generator

The clock generator creates and controls the system clocks. A master oscillator produces a 2 nanosecond (500 Mhz) master clock. From this clock, three controllable clock rates of 4, 6, and 12 nanoseconds(ns) are derived. The 12 ns, 6 ns, 4ns, and 2ns clocks are referred to as 6x, 3x, 2x. and 1x, respectively. The system clocks are illustrated in Figure x-x.

The 1x, 2x, and 3x clocks may be sent to any board in the system, and each may be operated in one of four modes: free run, burst run, single/micro step, and disabled. Commands issued from the SPU determine the clock rate and the operating mode for each individual board slot.

Commands are received from the SPU through the workstation-to-CU interface The transfers are synchronous and occur at the 1x clock rate. Nine control registers determine what function is being executed, and which boards are involved. Once a command is initiated (by writing to the control register), the clock control engine determines when to disable, switch, and enable the selecte clocks based on the state of the free running engines. The control engine determines clock operation by controlling the state of the individual clock drive enable and select lines. Figure provides a block diagram of the clock generator.

The clock generator contains three functional areas. They are:

1. I/O interface
2. Clock control
3. Clock drivers

The I/O interface connects the clock generator's internal control registers to the workstation-to-CU interface and performs internal address decoding.

The clock control section creates the free running 1x and 2x clocks and generate the clock driver control signals . The clock control section also performs phase detection on the free running clocks, which is used to determine when the cloc can be switched.

The clock driver section controls and switches the individual board-slot clocks based on inputs from the clock control section.

[Insert Figure.... Clock Generator Functional Block Diagram]

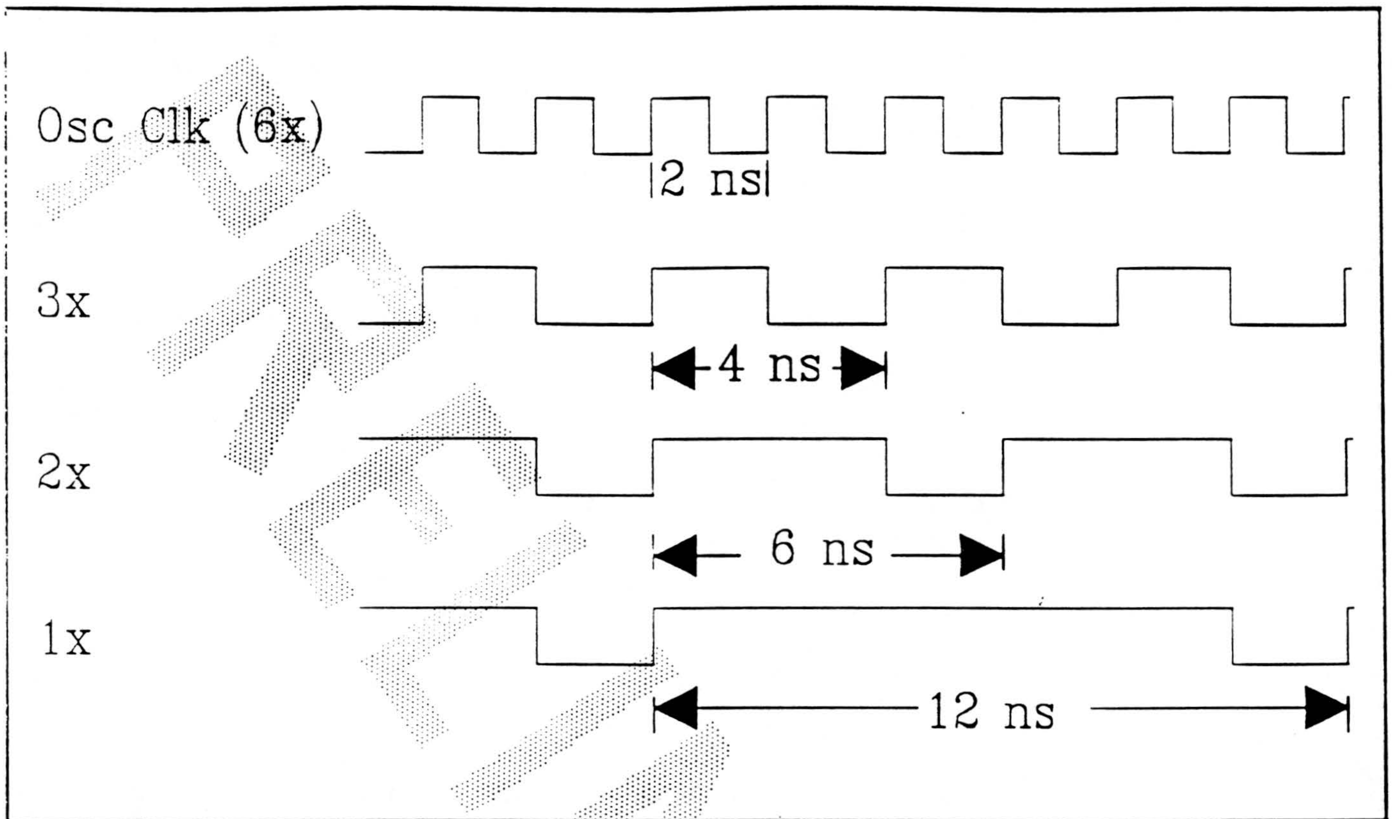
9.5.1 I/O Interface

The clock generator's I/O interface controls the exchange of data between the internal generator control registers and the workstation-to-CU (WC) interface. The I/O interface receives 32 data, four address, and four control lines from the WC interface and sends out eight data lines.

Two of the input control lines provide a byte select function. To read out all fou bytes of the 32-bit word, the WC interface steps the byte select signals through four states (00, 01, 10, and11) and one byte at a time is read out. This is done for each clock generator read requested by the SPU.

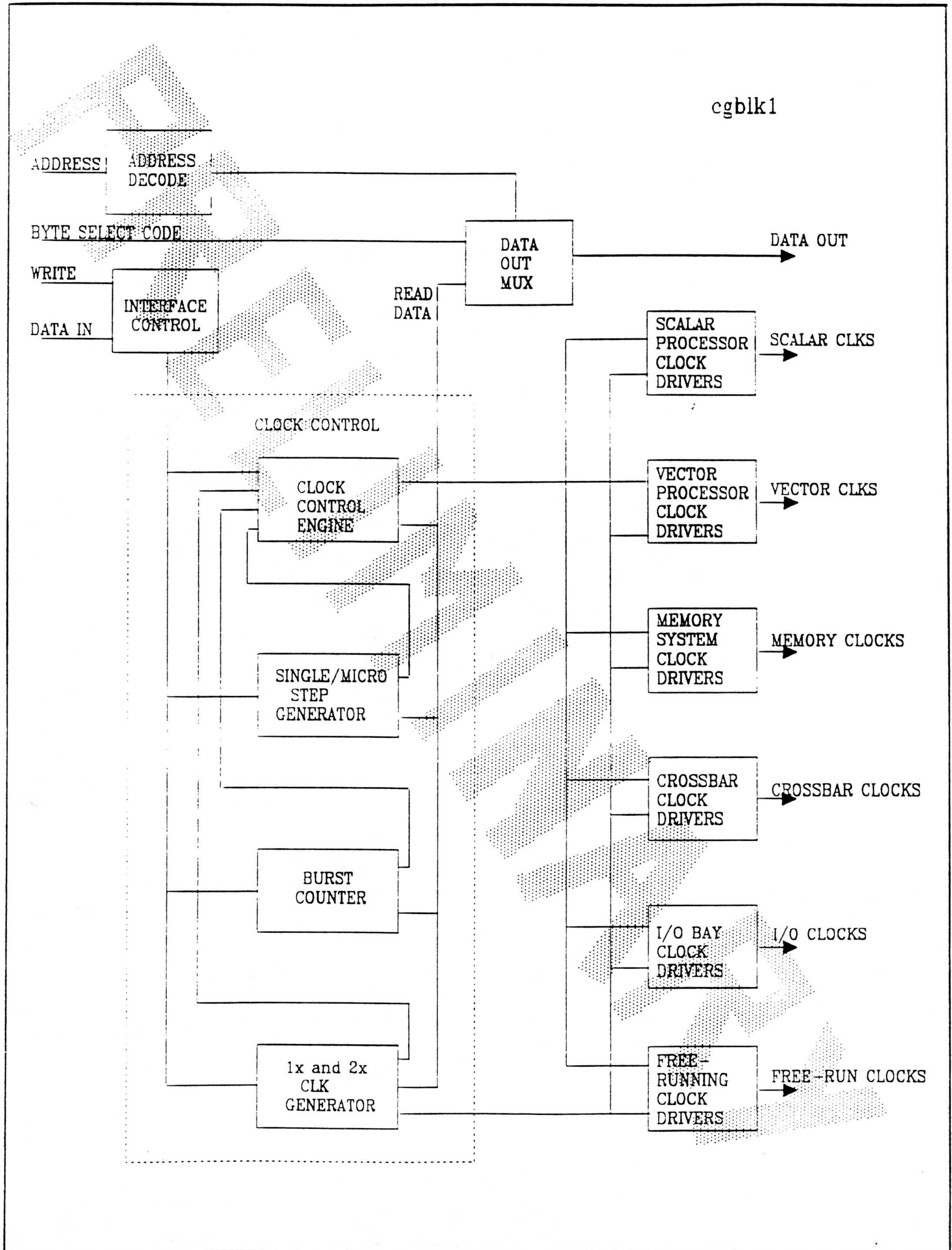
A write to the clock generator requires three system clocks, and a read requires five system clocks.

Figure 9-18 System Clocks Timing Diagram



PRELIMINARY

Figure 9-20 Clock Generator Block Diagram



9.5.2 Clock Control

The clock control section is responsible for generating the 1x and 2x master clocks, and the control lines for the individual board-slot clock drivers. Frc these free running clocks and control lines, the clock generator creates cloc the entire system that are both free running and controllable.

The clock control contains four functional areas. They are:

1. 1x and 2x clock generator
2. Burst counter
3. Single/micro step engine
4. Clock control engine

The 1x and 2x clock generator creates the free running, 1x and 2x clocks that are used by the board-slot clock drivers. The burst counter is a 16-bit binary counte that is used for scan and burst run operations. The single/micro step engine contains the logic required to single or double step the 3x, 2x, or 1x clocks. The control engine generates the clock driver select and enable lines based on input from the control registers and the other clock control blocks.

9.5.2.1 1x and 2x Clock Generator

The 1x and 2x clock generator creates the free running 1x and 2x clocks that are used by the clock drivers. The 1x and 2x clocks are created from ring counters operating at the 3x clock rate.

The 1x and 2x clocks are 1/6 and 1/3 the rate of the master (6x) oscillator clock. The low portion of each clock period is equal to one period of the 6x clock. Clocks are always enabled or disabled on the rising edge of the 1x clock when a four clocks are high.

9.5.2.2 Burst Counter

The 16-bit burst counter counts 1x clock periods during scan and burst mode operations. The counter is a 16-bit binary down counter that is decremented (when enabled) on the rising edge of the 1x clock. Prior to enabling the counter, is parallel loaded with the number of 1x clock periods to be counted. The count is enabled by writing the burst command to the clock generator's command/status register. After the counter is loaded with the desired clock count and enabled, the counter decrements to zero. When the counter reaches zero, decrementing is stopped. The zero count causes the control engine to disable the burst counter, and to disable the clocks to the boards selected by the command enable registers.

9.5.2.3 Single/Micro Step Engine

The single/micro step engine contains the logic responsible for single and doub stepping the different clock rates. The principle use for this logic is CAST, wher two at-speed clocks must be issued to a board after the test pattern is scanned i

The single/micro step engine includes a four-bit parallel load shift register that operates at the 3x clock rate. When the engine is selected by setting the step bit the command/status register, the output of the shift register drives the clock driver enable lines selected by active bits in the command enable register. The pattern output from the shift register enables or disables the clocks specified by the clock frequency register.

9.5.2.4 Clock Control Engine

The clock control engine directly controls both the clock frequency and the clock mode of every clock driver.

The clock control engine receives clock information from the clock generator's control registers and the other clock control functional units (i.e., the 1x and 2x clock generator, the burst counter, and the single/micro step engine). The control registers dictate what clock function (see clock generator commands) is performed and which boards are involved(see command enable register). The other functional units in the clock control section supply information as to when the clock functions can start and stop.

Writing a command to the command/status register enables the clock control engine. Once enabled, the clock control engine immediately begins executing the desired function on the clock drivers specified in the command enable register.

After the free running clocks are disabled and the diagnostic operation has begun, the clock control engine will monitor the operation. When the operation is completed, the clock control engine will disable the clock driver(s) for the selected board(s) and reset the busy bit in the command/status register. The clocks will remain disabled until the SPU executes a restart command for the selected board slots.

9.5.3 Clock Drivers

The clock drivers output clocks to the system boards based on inputs from the free running clock generators and the clock control engine. The clock generator contains a total of 40 clock drivers. There is one clock driver for each of the thirty-seven board slots in the system. These clock drivers are fully controllable. The remaining three clock drivers handle fixed-rate, free running clocks used by the scan engine and the interface adapter.

Each board slot clock driver receives enable and select inputs that determine whether the driver is enabled or disabled and which clocks are selected for output.

Each board slot clock driver receives a 6x, 2x, and 1x clock. The 6x clock is used inside the clock generator to clock the output buffers.

Two select inputs control the 1x and 2x clocks. Protective logic prevents both clocks from being selected at the same time. If neither the 1x or 2x clock is selected, clock selection defaults to the 3x clock.

9.6 Scan Engine

The scan engine controls the transfer of scan data between the scan memory and the boards in the system. The scan engine has three main functions:

1. Controls the scan control interface
2. Provides a bidirectional parallel-to-serial conversion of the scan data
3. Verifies received scan data

Commands from the SPU are input to the scan engine by way of the workstation-CU interface. The scan engine connects to each board in the system through a six-line interface. The six lines consist of five discrete lines for scan control and data input and one common line for buffered-data output. In addition to interfacing with the SPU and the system boards, the scan engine must also interface with scan memory. This interfacing involves transfering data between scan memory and the scan engine control registers.

addition to interfacing with the SPU and the system boards, the scan engine mus also interface with scan memory. This interfacing involves transfering data between scan memory and the scan engine control registers.

The scan engine is based on a state machine and contains five functioanl a... They are:

1. Master controller
2. Output scan engine
3. Input scan engine
4. Scan Memory Multiplexer
5. Scan control distribution

The master controller is responsible for generating control signals based on commands from the SPU, arbitrating memory requests from the input and output scan engines, and generating the control signals for the scan memory multiplexer.

The output engine converts scan data into a serial data stream that is shifted intc selected boards. The input engine is responsible for serial-to-parallel conversion of incoming scan data, masking and comparing that data with expected values, and sending the data to scan memory for storage.

The scan memory multiplexer connects the workstation-CU interface and the scan engine to the scan memory.

Scan control distribution handles the I/O intensive task of distributing and receiving scan control, scan data, and hard error signals. Figure ... illustrates the block diagram of the scan engine.

9.6.1 Master Controller

The master controller responds to commands from the SPU, arbitrates scan memory requests from the input and output scan engines, and generates the appropriate signals required to control the scan scan memory multiplexer.

The master controller initiates scan operations based on commands from the SPU. The SPU communicates those commands to the master controller by executing a write to the scan engine's command/status register.

The master controller handles scan memory read and write requests from both the input and output scan engines. Scan memory requests from the output engine have priority over requests from the input engine.

After the input or output scan engine's request has been recognized, the master controller activates the appropriate signal lines to permit the scan memory acces to occur. The master controller enables the selected address counter onto the memory bus and generates the required read or write control signals to perform the operation. When the operation is completed, the master controller will set th done signal of the selected input or output scan engine. The selected scan engin acknowledges the completion of the operation by resetting its request line for at least one system clock period.

During execution of a scan operation, the master controller monitors the ... counter. When the count reaches zero, which indicates that the scan ope... has completed, the master controller disables the counter, resets the two scan engine enables, and resets the busy bit in the scan engine's command/status register.

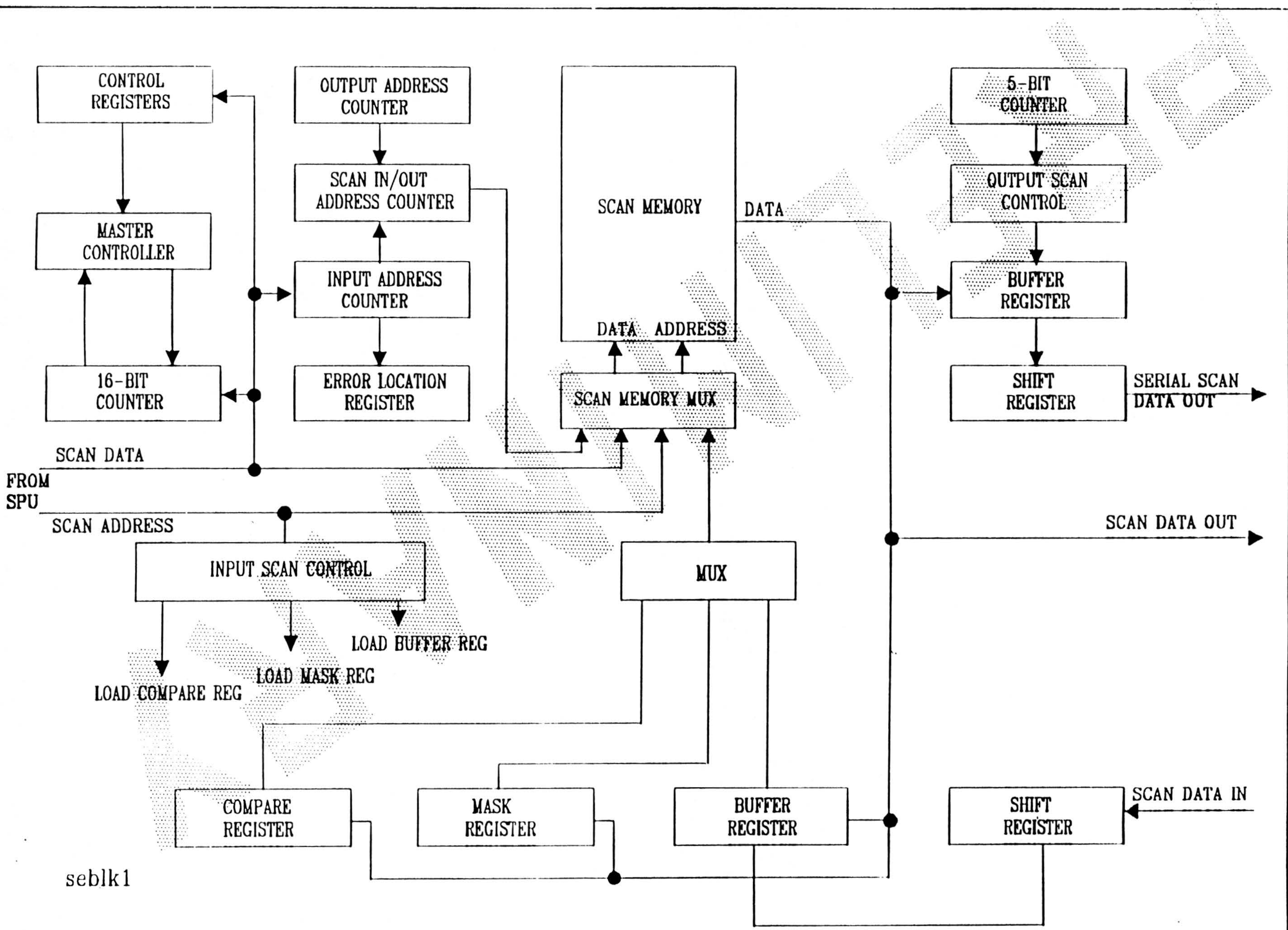


Figure 9-20 Scan Engine Block Diagram

9.6.2 Output Scan Engine

The output scan engine converts scan data into a serial data stream that is shifte out to selected boards. The output scan engine consists of a buffered, parallel-load shift register coupled to a small state controller. The state cor and the shift register are both clocked off the controlled scan clock. Once the engine is enabled, the rising edge of the scan clock will cause data to shift out. A five-bit counter in the state controller keeps track of how many bits have been shifted out and is used to indicate when data must be written to avoid underrur The state controller also keeps track of whether the buffer register contains valic data. The state controller will try to keep both registers loaded with valid data a all times. When either register is regarded as not full, the controller will request data write by the master controller.

To initiate a write, the scan engine controller will set its data request line to the master controller. The controller will then wait until the master controller responds with the asserted done signal. When the done signal is asserted, the controller will clock the valid data into the buffer register, and then the shift register (on the next clock) if necessary. The controller will then reset its request line for at least one clock to acknowledge receipt of the done signal and increment the address counter for the next access.

PRELIMINARY

9.6.3 Input Scan Engine

The input scan engine handles data that is being scanned back from boards in the system. The input scan engine performs the following steps:

1. Converts the incoming serial scan data into parallel data words
2. Masks out unwanted data
3. Compares the masked output against the data expected
4. Transfers the result of the comparison to scan memory

During the process of scanning data in, masking it, comparing it, and saving the result, three scan memory accesses are required, one each to get the mask and compare data words and a third to store the result. The word address remains the same throughout each of these scan memory accesses, only the page select code is modified each time.

If the incoming scan data does not match what was expected, an error is generated. The occurrence of an error is recordeded by setting the VE bit in the scan engine's command/status register. In addition, for the first error detected during any given scan operation, the contents of the input address counter will be written into the error location register. Writing to this register is inhibited for the remainder of the scan operation, regardless of the number of errors detected.

Regardless of whether the compare produces an error indication or not, the input scan engine will initiate a write operation to store the result from the compare in scan memory. Upon completion of this last step, the input controller will increment the input address and begin fetching the mask and compare data for the next scan word.

9.6.4 Scan Memory Multiplexer

The scan memory multiplexer is a two-to-one multiplexer that steers data and address bits to scan memory from either the SPU or the input scan engine. The SPU input to the scan memory multiplexer is by way of the CU board's workstation-CU interface. The multiplexer is a passive logic circuit under control of the busy bit in the scan engine's command/status register. When set, the busy bit causes the multiplexer to select the scan engine input; when reset, the SPU input is selected. The multiplexer is designed so that a switch from one input source to the other cannot occur in the middle of a data transfer.

PRELIMINARY

9.6.5 Scan Control Distribution

The scan control distribution logic, which is located entirely on the control crossbar board (XCL), is responsible for the distribution and reception of scan data, scan control, and scan hard error signals. Scan data and control lines discrete, point-to-point connections; they comprise 250 pins on the control crossbar board.

The control crossbar board contains two 32-bit scan control enable registers and the logic used to gate out the master control signals. Each board in the system that is enabled by the scan control enable registers receives four scan control line that determine the nature of the scan operation performed on that board. The four lines consist of a three-bit scan control code and a recirculation select signal.

Each board in the system outputs both a serial scan data line and a hard error line to the clock generator and scan engine. The serial scan data line from each board is input to a 37:1 multiplexer, which is located on the crossbar board. The multiplexer steers the serial data output from one of the system boards to the input scan engine. The system board whose scan data is selected by the multiplexer for input to the scan engine is determined by the six-bit input select field in the scan engine's command/status register.

A hard error signal indicates that a serious error has been detected on one of the boards in the system and that all of the system clocks must be disabled. The stat of each of these signals is loaded on the rising edge of each system clock. The logical OR of these signals is input to the clock generator and used to disable clocks.

The scan engine has the ability to scan write multiple boards simultaneously. Therefore, the scan data output line from the scan engine is buffered and output individually to each of the boards in the system.

PRELIMINARY

9.7 Scan Memory

The scan memory contains the data the scan engine uses to perform scan operations. The scan memory consists of a 4K-location by 36-bit memory array that contains the data used for all scan operations. The 36-bit data word consists of 32 data bits and four parity bits. The memory is divided into four 1K by 36-bit pages, each of which can support a scan ring of up to 32,768 bits.

Each one of the four pages contains a unique type of scan data. The four data types are outgoing (write) data, ring mask data, compare data, and result data. The outgoing data page contains data that will be loaded into the output scan engine and transmitted to the boards in the system. The mask data page contains data that will be logically anded with data scanned in from system boards; the purpose of this is to mask off unwanted information. The comparison page contains data that will be compared with the data output from the masking operation, and the result page contains the output that this comparision produces.

The SPU and the scan engine both have the ability to transfer data to and from the scan memory through the memory's 36-bit data interface. The bus master is determined by the control/data/address mux in the address decode section of the scan engine. The mux select line determines whether the SPU or the scan engine has access to the memory and is controlled by the state of the scan engine. If the scan engine is in idle mode, the interface is controlled by the SPU. When the scan engine is executing a diagnostic (scan) operation it is master of the bus and scan memory read or write commands from the SPU are ignored.

9.8 WC Interface

The workstation-to CU (WC) interface connects the workstation interface board to the clock generator, scan engine and scan memory. The WC interface performs the control handshake with the workstation interface board and is responsible for synchronizing the address, data and control with the system clock. All accesses through the WC interface are 32-bit word accesses. Once the address is synchronized, the WC interface will perform the address decode, and create the required control signals for access to the memory, scan engine, and clock generator.

PRELIMINARY

9.9 XP Interface

The XP interface contains three functional areas. All are related to the SPU. Those three functional areas provide:

- Part of the SPU's memory access data path
- Hardware to initialize and pattern test main memory
- The means to send and receive system interrupts

9.9.1 Memory Access

The SPU must communicate with main memory. The hardware to facilitate memory transfers between the SPU and main memory is spread across both the WI and CU boards. The XP interface, located on the CU board, is one part of the memory path the SPU uses for transfers to and from memory.

Transfers between the SPU and main memory include the following:

- Memory read
- Memory write
- Test and set
- Test and clear
- Memory scrub

Figure illustrates the hardware involved in the SPU-memory data path.

9.9.1.1 Memory Map

The SPU is allocated a 4 MB address space that is mappable to the main memory in 4KB chunks. Main memory access from the SPU is accomplished using a windowing technique and memory mapping. The SPU performs memory reference operations as if they were destined for the SPU's local memory. However, for addresses that fall within a certain range, an address conversion is performed. Figure provides a simplified illustration of SPU-to-main memory mapping.

9.9.1.2 Map Registers

A set of map registers are used to accomplish the conversion. There are 1,024 map registers. Each map register contains 32 bits. Figure shows the map register word format, and Table provides a definition of the map register bits.

9.9.2 Memory Diagnostic Functions

The XP interface includes hardware that provides the SPU with main memory initialization and pattern testing capability. This diagnostic hardware consists of a pattern generator / comparator that is linked to the XP data path and a state machine that controls the operation of the XP interface and the pattern logic.

9.9.2.1 Memory Test Registers

Four 32-bit registers control the hardware. They include:

- Main memory address register
- Odd word pattern register
- Even word pattern register
- Test control register

The main memory address register contains the current PBUS transfer address. Since the interface only transfers longwords, the least significant three bits of this register are always forced to zero.

Figure 9-21 SPU-to-Memory Data Path

C3SYS10.DRW

CENTRAL PROCESSING UNITS

CROSSBAR

MEMORY BOARDS

INTERFACE ADAPTER

UTILITY HARDWARE

MASTER CLOCK AND DIAGNOSTIC HARDWARE

CPU UTILITIES BOARD

EXPANSION PORT BUS

CLOCK GENERATOR

SCAN ENGINE

SCAN MEMORY

XP INTER-FACE

CU/SPU INTERFACE

CPU UTILITIES BOARD

SPU WORKSTATION INTERFACE

PERIPHERALS

BAY POWER CONTROLLERS

SPU WORKSTATION

Note:

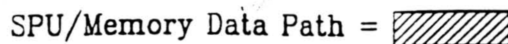
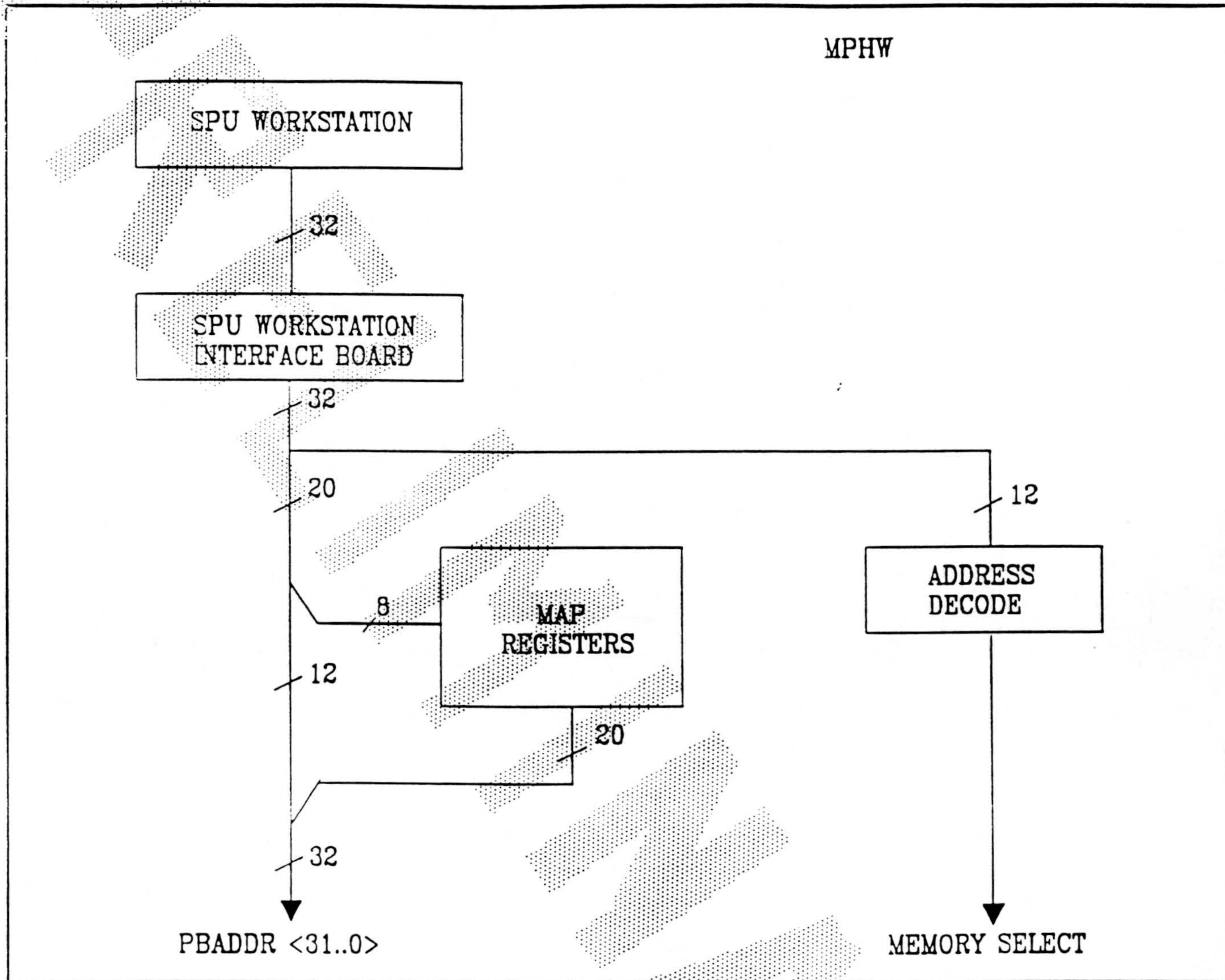
SPU/Memory Data Path = 

Figure 9-23 SPU Memory Mapping Scheme



PRELIMINARY

Figure 9-24 Map Register Format

31 - 29	28	27	26	25	24 - 23	22	21	20	19 - 0
RES	VAL	RES	SCR	TAC	RES	I/O	TAS	RES	PAGE FRAME

Table 9-5 Map Register Bit Definitions

Bit	Name	Description
31-29	Reserved (RES)	
28	Valid	When set, indicates that the information in the register is valid and that the SPU may access the corresponding page. An attempt to access a page that does not have the valid bit set causes a DIO bus error.
27	Reserved (RES)	
26	Scrub Bit (SCR)	When set, causes the Interface Adapter to scrub the referenced memory location to correct a single-bit ECC error. IA performs a word write with no zones. This causes memory to read the corrupted word, correct it, and write the corrected word back into memory without merging in any new data. SPU read or write word operations with the scrub bit set cause an XP transaction that is word aligned and four bytes long.
25	Test and Clear (TAC)	When set, initiates test and clear operations on semaphores in main memory. TAC operations are only allowed on bytes. Any access larger than a byte to a page with the TAC bit set causes a DIO bus error. A TAC operation is performed by reading the desired byte with the TAC bit set in the map register. The memory location's contents are returned and the location is set to 0x00.
24 - 23	Reserved (RES)	
22	I/O	When set, causes SPU XP reads and writes to access PBUS I/O space instead of PBUS memory space.
21	Test and Set (TAS)	When set, initiates test and set operations on semaphores in main memory. TAS operations are only allowed on bytes. Any access larger than a byte to a page with the TAS bit set causes a DIO bus error. A TAS operation is performed by reading the desired byte with the TAS bit set in the map register. The memory location's contents are returned and the location is set to 0xFF.
20	Reserved (RES)	
19 - 0	Page Frame	

Note

Scrub, TAC, I/O, and TAS bits define mutually exclusive operations. A memory access that uses a map register with more than one of these bits set will cause a DIO bus error.

Odd and even word pattern registers contain the next pattern to be written to memory on a write or the next pattern expected on a read.

Figure illustrates the format of the four memory test registers, and Table xx defines the purpose of the individual bits in the test control register.

9.9.2.2 Memory Test Operating Modes

There are four memory test operating modes: hold, increment, shift left, and complement. Each mode affects the test pattern.

In the hold mode, the word patterns contained in the even and odd word registers remain unchanged for the length of the transfer. This operating mode can be used to initialize main memory to a constant value (such as zero).

In the increment mode, each pattern word is incremented by two at the end of every longword transfer. This mode is used to load an *address-equals-data* pattern in memory on a word boundary.

In the shift left mode, the two pattern words are linked together to form one 64-bit left shift register. This mode is used to generate a pattern of walking ones and zeroes.

In the complement mode, each pattern word is complemented at the end of every longword transfer on the XP bus.

9.9.3 System Interrupts

The XP interface and the XP interrupt bus provide the SPU with the means to send and receive system interrupts 0x08, 0x09, 0x0A, and 0x0B.

The interrupt control register has an integral part in handling interrupts. To initiate an interrupt, the SPU does a write operation to the interrupt vector byte in the interrupt control register. Writing to this byte of the register causes the XP interface to request the XP interrupt bus. After the interrupt bus is granted, the XP interface sends the interrupt vector out on the bus.

An interrupt is defined as pending from the time the SPU writes the vector to th interrupt control register until the interrupt bus cycle completes. Another interrupt vector byte should not be written to the interrupt control register whil an interrupt is pending. Doing so will cause a bus error indication to be sent to the SPU.

The format of the interrupt control register is illustrated in Figure xx and the bit definitions for the register are provided in Table xx.

PRELIMINARY

Figure 9-25 Memory Test Registers

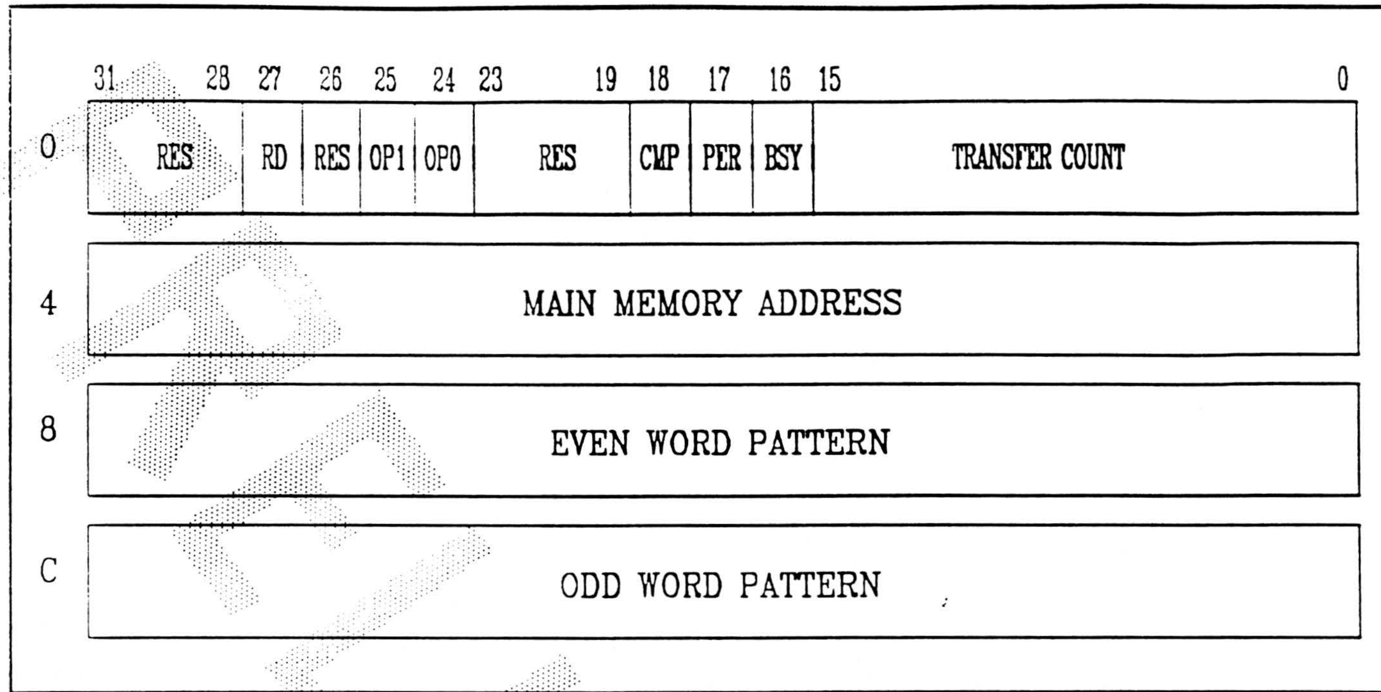


Table 9-6 Test Control Register Bit Definitions

Bit	Name	Description
31 - 28	Reserved (RES)	
27	Read /Write (RD)	Enables memory write or memory read and compare operation.
26	Reserved (RES)	
25 - 24	Opcode (OP)	Specifies one of four pattern types. See *Memory Test Operating Modes* for more information.
23 - 19	Reserved (RES)	
18	Compare Error (CMP)	When set, indicates the 64 bits of data read from XP bus did not match even and odd pattern registers.
17	PBUS Error (PER)	When set, indicates that memory has reported a PBUS error.
16	Busy (BSY)	When set, indicates that a write or read/compare operation is in progress. Set by state machine when transfer count is loaded; reset by state machine when transfer completes or terminates due to error.
15 - 0	Transfer Count	Byte count used in XP header.

Figure 9-26 Interrupt Control Register Format

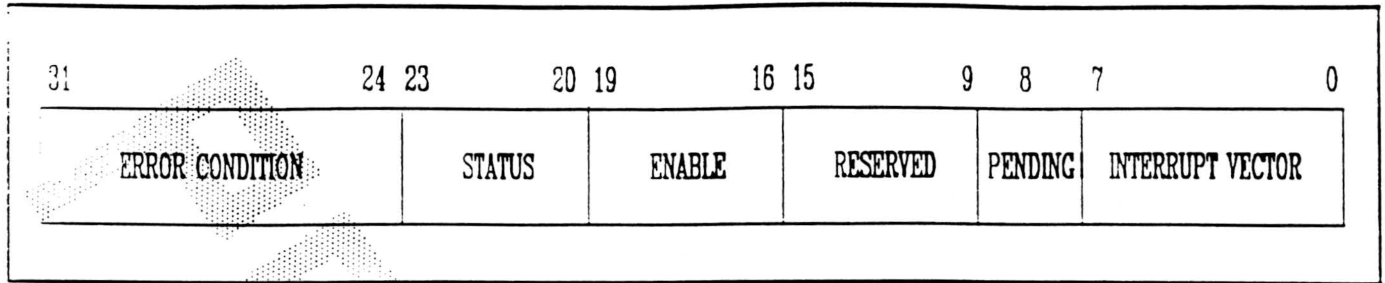


Table 9-7 Interrupt Control Register Bit Definitions

Bit	Name	Description
31 - 24	Error Conditions	Each bit represents an error condition interrupt that is non-maskable on the CU. They are requests for immediate action due to an error condition detected by the clock generator or scan logic. Each individual error condition interrupt may be cleared by writing a one to the appropriate bit position. The error condition interrupts for each bit are listed as follows: 31 - Hard error 30 - Interface adapter soft error 29 - Memory board soft error 28 - Scan memory access error 27 - Scalar halt 26 - Scan memory parity error 25 - Clock generator parity error 24 - XCL parity error
23 - 20	Status	Each bit represents the status of a system interrupt. Bits 20, 21, 22, and 23 provide status for system interrupts 0x08, 0x09, 0x0A, and 0x0B, respectively. When a status bit is set, the corresponding interrupt has been recognized by the CU logic. Once set, a status bit remains set until a one is written to it.
19 - 16	Enable	Each individual bit is an enable for a system interrupt. Bits 16, 17, 18, and 19 enable system interrupts 0x08, 0x09, 0x0A, and 0x0B, respectively.
15 - 9	Reserved	
8	Pending	Indicates the most recent interrupt vector written by the SPU that is still pending on the interrupt bus. Pending means either the CU has not yet been granted access to the XP interrupt bus or that there is an interrupt cycle in progress that has not yet completed.
7 - 0	Interrupt Vector	The number of the current or most recent interrupt broadcast by the SPU/CU.

Table of Contents

Workstation Interface

Overview 10
DIO Bus Interface 10
  Address Maps 10
  Select Code Switches 10
  Card ID / Reset Register 10
  Card Size Register 10
  Card Status and Control 10-
Parallel Interface 10-
  Data Transfer Sequence of Events 10-
  Error Conditions 10-
BPC Interfaces 10-
  Cable Interlock 10-
  Serial Interface 10-
  Reset Control 10-
Key Switch Interface 10-
Printer Interface 10-
Utility and Diagnostic Hardware 10-
  Miscellaneous Control Register 10-
  Parity Error Force Register 10-
  Loopback Data Register 10-
  CU Address Register 10-
Interrupt Register Logic 10-

PRELIMINARY

Workstation Interface 10

PRELIMINARY

10.1 Overview

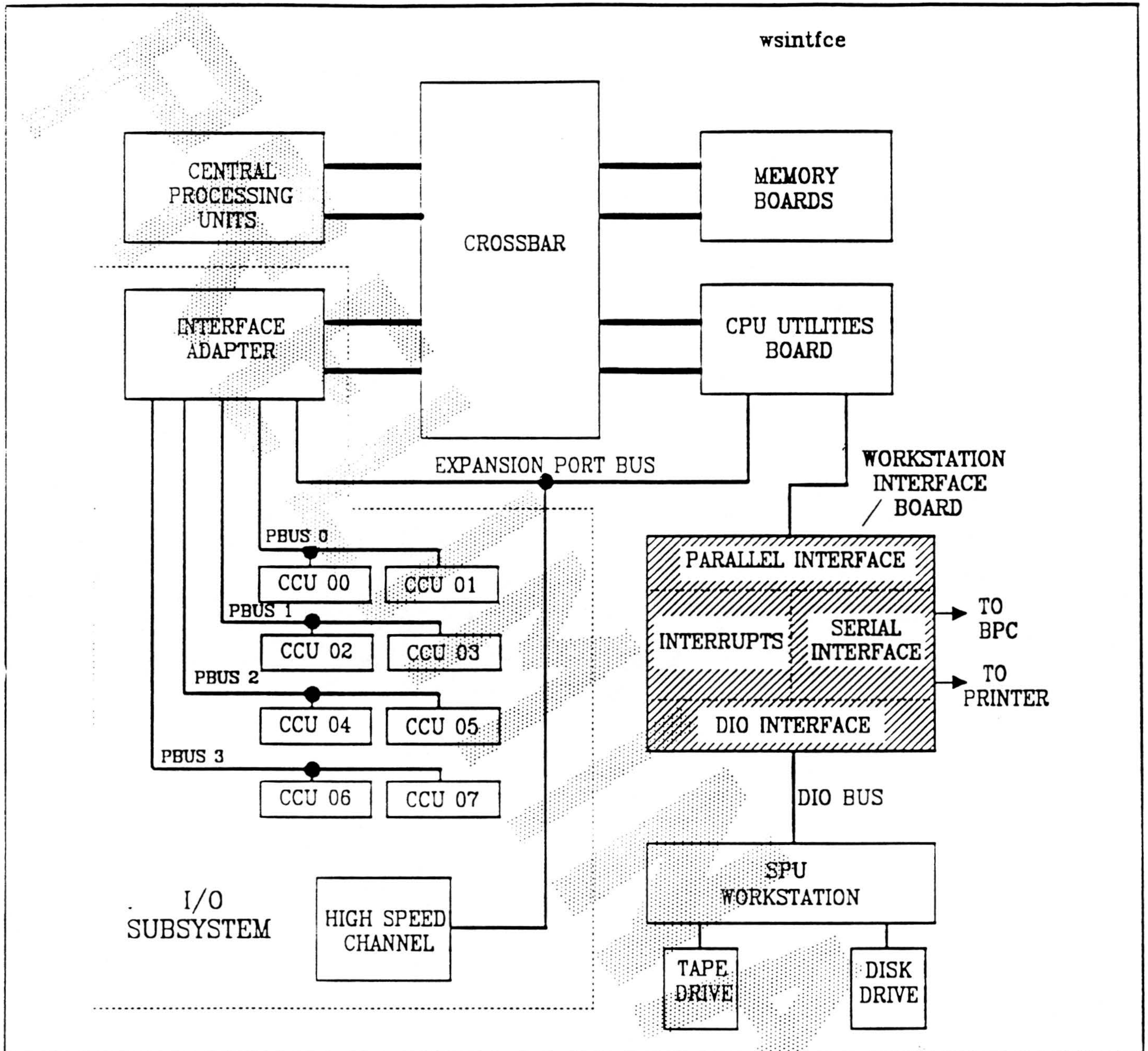
The workstation interface (WI) board is the link between the SPU workstation and the other functional units in the system. The WI board physically resides with the SPU workstation and its associated hardware in a standalone cabinet that is apart from the processor and I/O cabinets. A DIO bus cable links the WI board to the workstation. The DIO bus is a proprietary Hewlett Packard (HP) bu used for communicating with the SPU workstation. The workstation interface board connects to the other functional units in the system by way of a cable from the WI board to the backplane of the I/O bay cabinet.

The workstation interface board consists of the following functional areas:

- DIO Bus Interface
- Parallel Interface
- Serial BPC and Printer Interfaces
- Key Switch Interface
- Utility and Diagnostic Hardware
- Interrupt Request Logic

Figure 10-1 illustrates the workstation interface board and its relationship with the SPU workstation and the rest of the C3800 System.

Figure 10-1 Workstation Interface



10.2 DIO Bus Interface

The DIO Bus links the SPU Workstation with the workstation interface board. The interface board is designed to work with either one of two different DIO bus versions (DIO I or DIO II). The HP332 workstation uses the DIO I bus. The DIO I bus works with other similar Hewlett Packard workstations and provides h performance. The capability to work with either DIO I or DIO II buses has b designed into the workstation interface to provide greater flexibility in the type of workstation that can be used in the future. Currently, the HP 332 is the SPU workstation and DIO I is the applicable bus.

Table 10-1 lists and defines the bus signals for both DIO I and DIO II.

PRELIMINARY

Table 10-1 DIO Signals

Signal Name	DIO I	DIO II	WI	Comments
Address / Data Buses and Bus Control Signals				
BA<31..24>	x	x	Input	Buffered Address
BA<23..1>	x	x	Input	Buffered Address
BD<15..0>	x	x	Bidirectional	Buffered Data
XD<15..0>		x	Bidirectional	Extended Data
BAS24*	x		Input	24-bit Address strobe
BAS32*		x	Input	32-bit Address Strobe
ADACK*		x	Output	Address Acknowledge
BLDS*	x	x	Input	Buffered Lower Data Strobe
BUDS*	x	x	Input	Buffered Upper Data Strobe
LWORD*		x	Input	Long-word access
BR/W*	x	x	Input	Buffered Read/Write
BLK*		x	Input	Block Mode
RMC*		x	Input	Read/Modify/Write Cycle
DTACK16*		x	Output	16-bit Data Transfer Acknowledge
DSACK32*		x	Output	32-bit Data/Size Acknowledge
BERR*	x	x	Output	Bus Error
Interrupt Signals				
IACK32*	x	x	Input	Interrupt Acknowledge Cycle
IR3*	x	x	O	Level 3 Interrupt Request
IR4*	x	x	O	Level 4 Interrupt Request
IR5*	x	x	O	Level 5 Interrupt Request
IR6*	x	x	O	Level 6 Interrupt Request
Utility Signals				
RESET*	x	x	I	Bus Reset

10.2.1 Address Maps

The SPU workstation includes a memory map that defines a variety of configuration parameters for the SPU workstation. The address map for a workstation that uses a DIO I bus differs from one with a DIO II bus. Fig' '0 and 10-3 illustrate the two types of address map. Both types of address m. include space allocated to the workstation interface.

The total DIO I address space is 16 megabytes; the total for DIO II is four gigabytes. The DIO II address map includes a 16 megabyte region labeled DIO I Compatibility Address Space. As its name indicates, the contents of this region are identical to that shown in Figure 10-2 for the DIO I address space.

Within the 16 megabyte DIO I address space is a two megabyte area allocated fo external I/O. From the SPU workstation's point-of-view, the workstation interface is external I/O. The two megabyte space is divided into 32 sections of 64 kilobytes each. At boot time, the workstation interface responds within a single 64 kilobyte section. After the SPU initializes the workstation interface, an additional one megabyte interface window becomes available.

Figure 10-2 DIO I Address Map

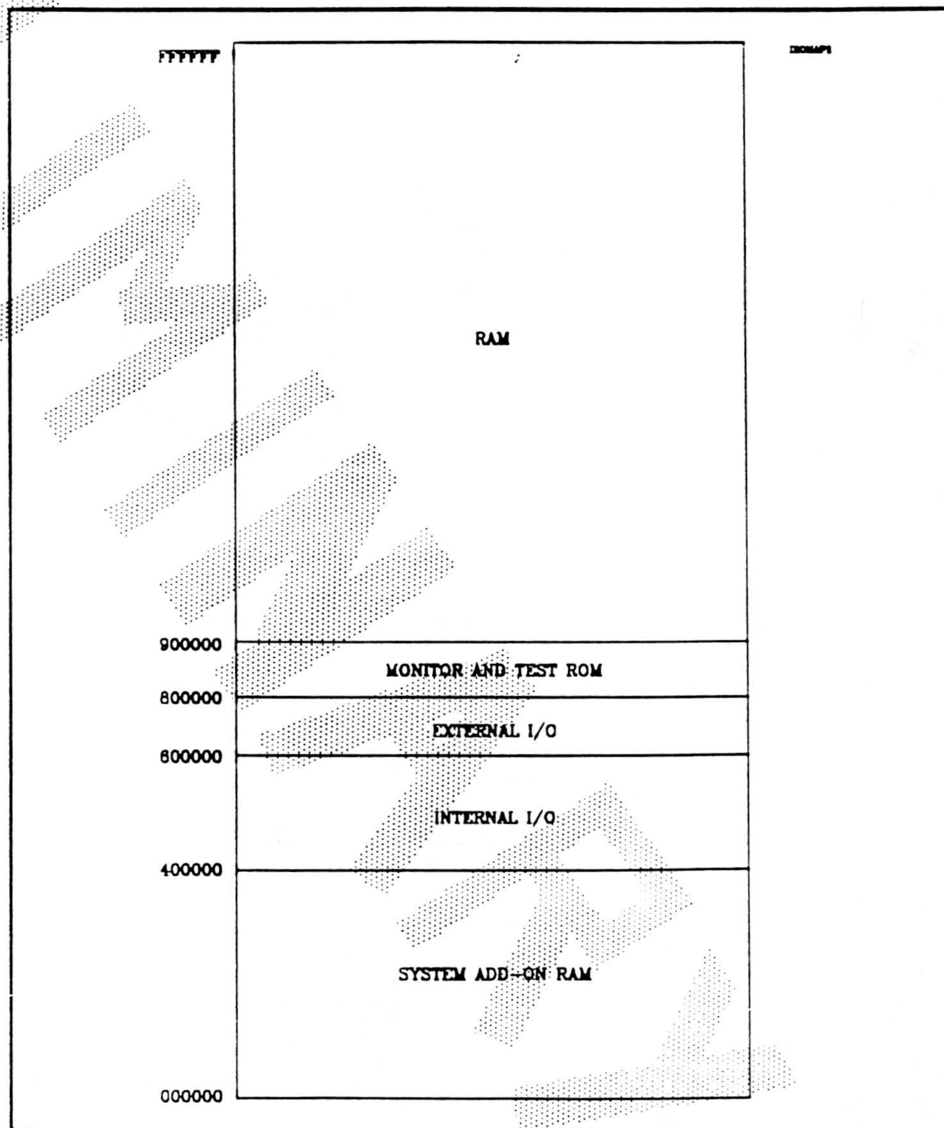
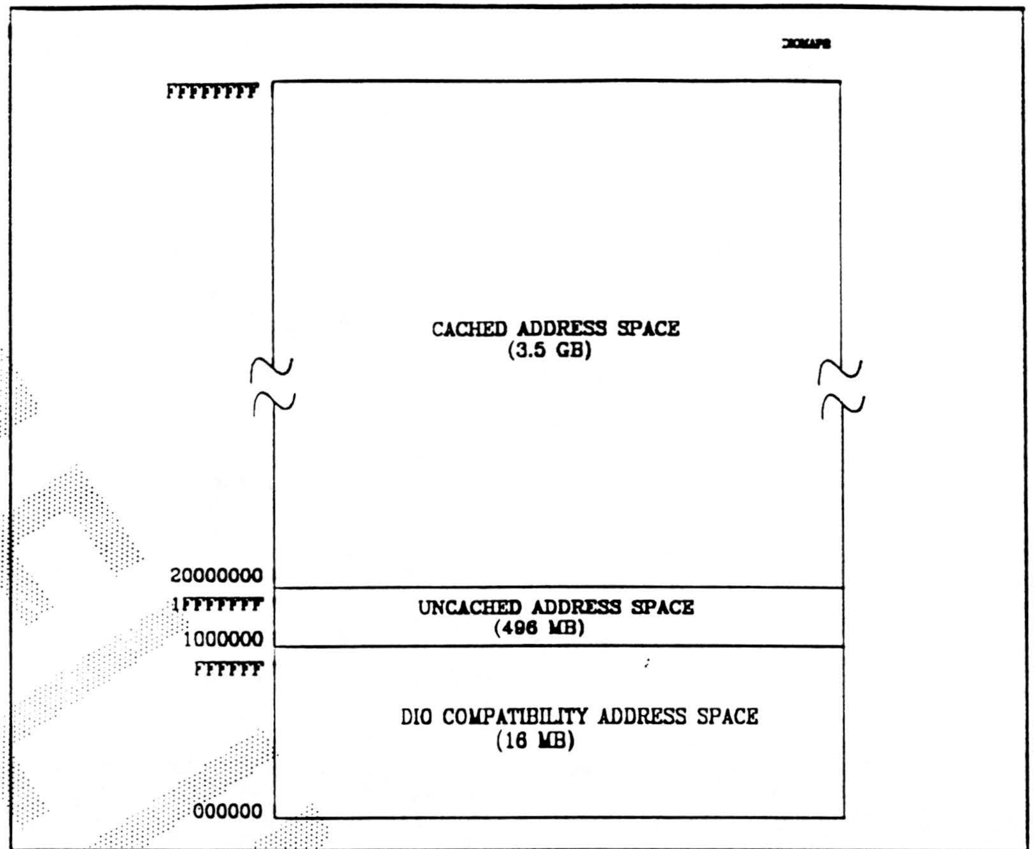


Figure 10-3 DIO II Address Map



PRELIMINARY

10.2.2 Select Code Switches

Eight select code switches determine the physical address of the workstation interface board. If the SPU uses a DIO I bus, then the most significant sele d switch should be set to the logic zero position. If a DIO II bus is being usec a the most significant select code switch should be in the one position.

Whenever the most significant switch is in the zero position, the least significant five switches will be read to determine the physical address of the board. If the most significant switch is in the one position, the least significant seven switches will be read to determine the board's physical address. DIO I bus interfaces require a physical address range of 0 through 31, and DIO II bus interfaces require a physical address range of 132 through 255.

When the SPU workstation asserts an address on the DIO bus, part of the address is compared with the setting of the select code switches. If they match, then the workstation interface board is the intended destination.

For DIO I bus implementations, the address comparator on the workstation interface board has the most significant three bit positions hardwired for a value of a 011. For an address match to occur, the three most significant bits (23-21) of the eight bit address must be 011 and the least significant five bits (20-16) must match the select code switches.

Tables 10-2 and 10-3 show the select code switches and the corresponding address bits with which the switches are compared for both DIO I and II bus interfaces.

Table 10-2 Select Code Switches for DIO I Bus

	Address Bits							
	23 (MS)	22	21	20	19	18	17	16 (LS)
Switch Setting	0	1	1	x	x	x	x	x
	x = variable MS = most significant LS = least significant							

Table 10-3 Select Code Switches for DIO II Bus

	Address Bits							
	29 (MS)	28	27	26	25	24	23	22 (LS)
Switch Setting	0	x	x	x	x	x	x	x
	x = variable MS = most significant LS = least significant							

PRELIMINARY

10.2.3 Card ID / Reset Register

The ID register specifies the card type. The register contains a primary and secondary identification value. For the NWI, the primary ID is 15 and the secondary ID is zero. Figure 10-4 illustrates the format of this register.

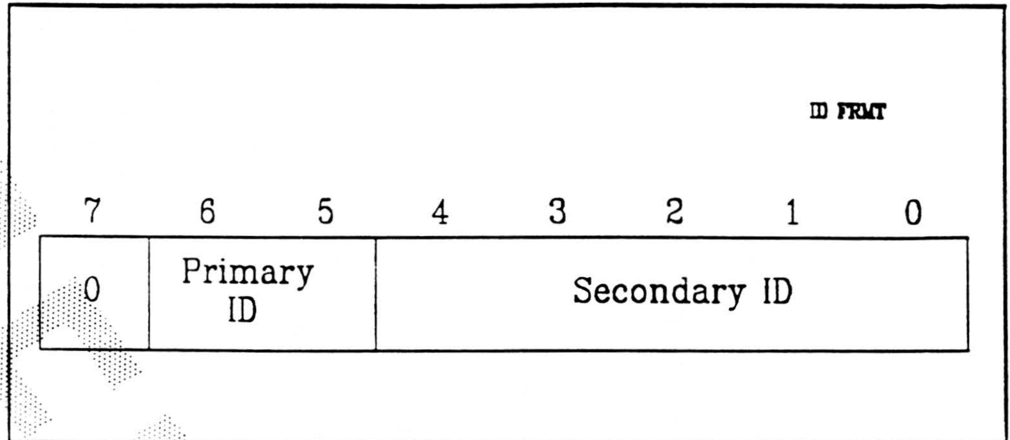


Figure 10-4 Card ID and Reset Register Format

If the SPU executes a write transfer to the Card ID / Reset register, the workstation interface board is reset to its initial power on condition.

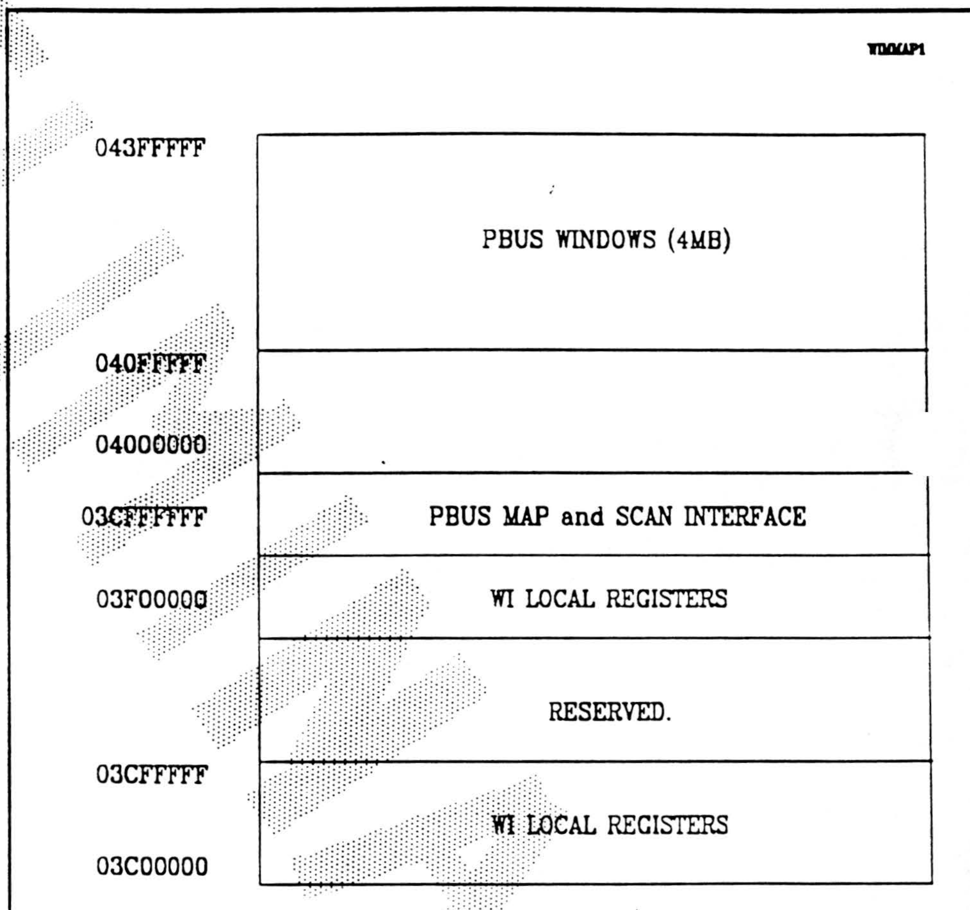
PRELIMINARY

10.2.4 Card Size Register

The Card Size Register (offset 101H) specifies the total address space occupied by the WI board. This one byte, read-only register will always contain a hexadecimal value of 07 for the WI board. This value specifies that a total c megabytes is reserved in the DIO II address space. This register is ignored I-based systems.

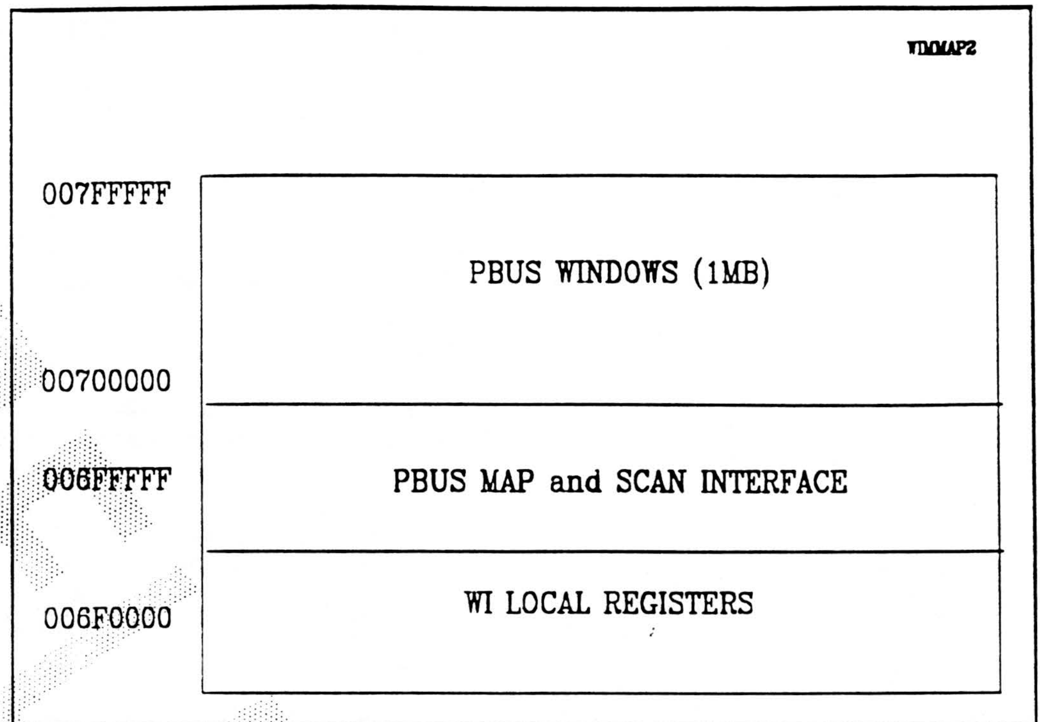
The address space allocated to the workstation interface board differs for DIO I and DIO II-based systems. For DIO I-based systems, the WI board is allocated 32 kilobytes of memory at boot time and an additional one megabyte after initialization. For DIO II-based systems, the WI board's allocated address space occupies eight megabytes at all times. Figures 10-5 and 10-6 show the WI memory map for both DIO I and DIO-II-based systems. The memory map assumes default values for the select code switch settings.

Figure 10-5 WI Memory Map (DIO II Configurations)



PRELIMINARY

Figure 10-6 WI Memory Map (DIO I Configurations)



PRELIMINARY

10.2.5 Card Status and Control

The Card Status and Control register contains interrupt enable, interrupt request and interrupt level select bits. Figure 10-7 illustrates the register format.

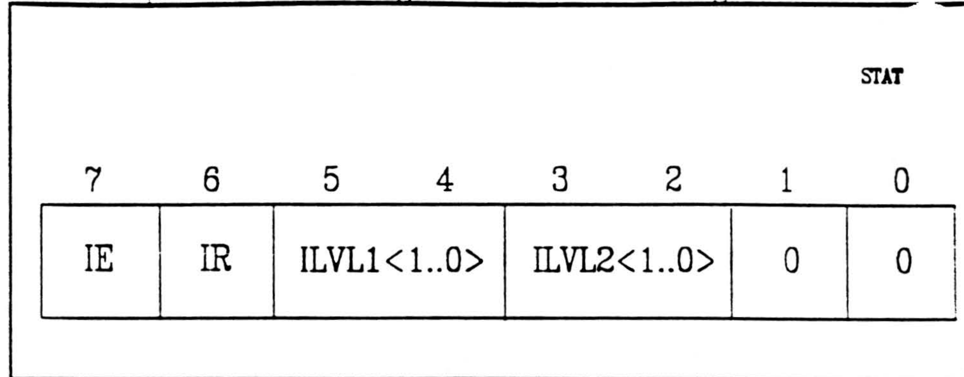


Figure 10-7 Card Status and Control Register Format

The interrupt enable bit serves as a global interrupt enable for all interrupt requests destined for the DIO II bus from the workstation interface.

The interrupt request is a read only bit which reflects the state of the WI's internal interrupt request. If both the interrupt request and interrupt enable bits are set, the WI asserts one of the DIO II bus interrupt request lines, IR3 through IR6. The particular interrupt request line asserted is determined by the interrupt level select bits.

To clear an interrupt requires writing to the register that originated the interrupt and not by accessing the Card Status and Control register.

The register contains two pairs of interrupt level select bits. Each pair can be coded separately and represents one interrupt level. Bits 5-4 contain interrupt level one (ILVL1) and bits 3-2 contain interrupt level two (ILVL2). Interr[illegible] ve[illegible] one (ILVL1) is used for the UARTS; interrupt level two (ILVL2) is used fo[illegible] C[illegible] board and key change interrupts.

Table 10-4 lists the two-bit binary code for each interrupt level.

Table 10-4 Interrupt Level Binary Codes

ILVL Binary Code	Interrupt Level
00	IR3
01	IR4
02	IR5
03	IR6

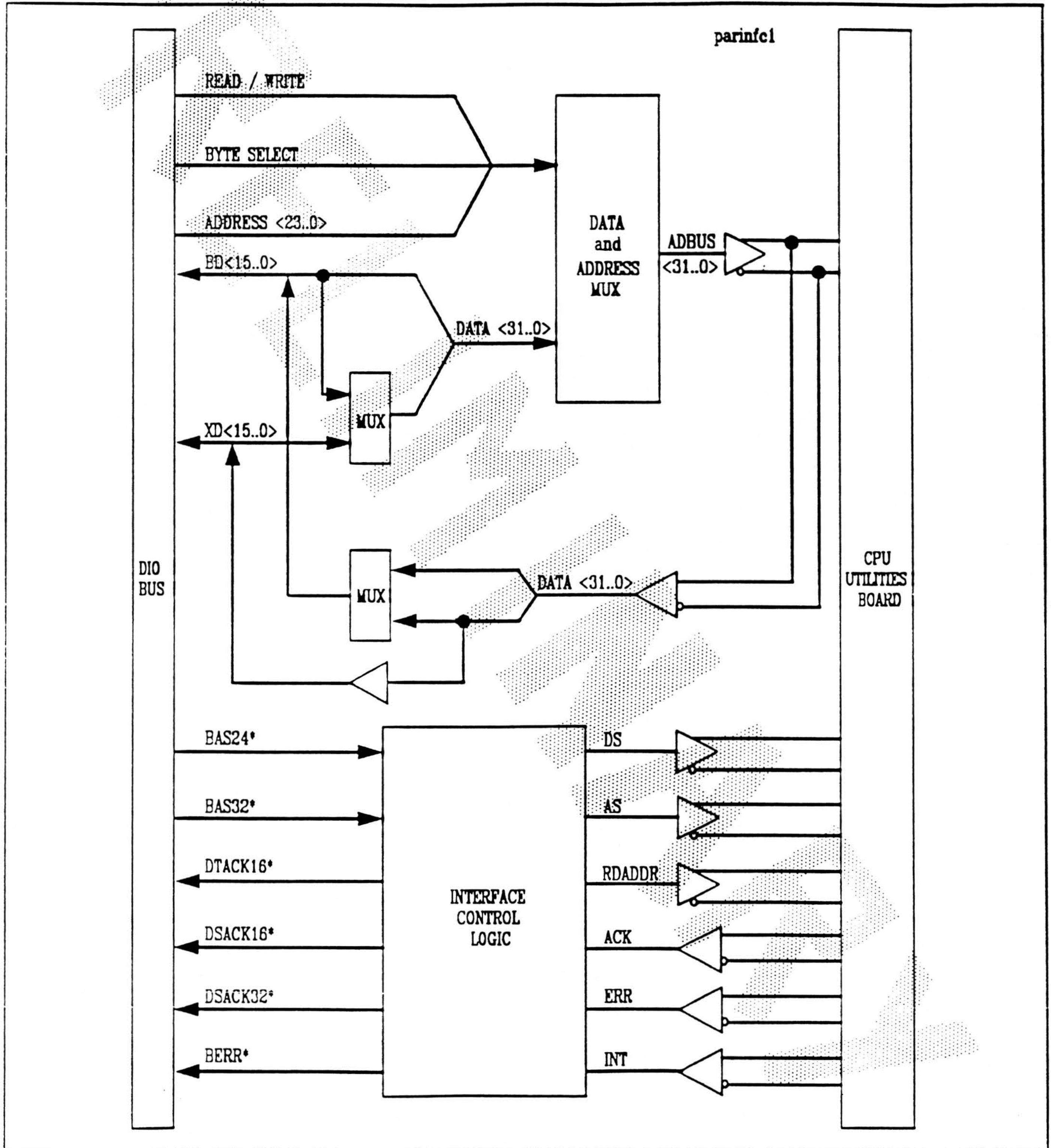
PRELIMINARY

10.3 Parallel Interface

The parallel interface physically resides on the WI board and serves as an interface between the SPU workstation and the CU board. On the CU board, the parallel interface splits into two interfaces, one for the XP bus and the other for the diagnostic hardware or scan logic.

Figure 10-8 provides a block diagram of the parallel interface.

Figure 10-8 Parallel Interface Block Diagram



Differential drivers and receivers are used for each of the interface signals between the WI and CU boards. Data and address lines are multiplexed, which reduces the overall number of wires.

Only the least-significant 24-bits of the address are passed to the CU board. least significant two address bits are always zero, indicating an address on a longword address boundary. Byte select and read/write control signals are supplied along with the 24-bit address.

Parity is checked on the incoming data read transfers and sent on outgoing data write transfers. Parity is also included with the address transfers.

10.3.1 Data Transfer Sequence of Events

The sequence of events for a data transfer are listed as follows:

1. Transfer 24-bit address, byte select signals, and read /write control signal.
2. Send Address Strobe (AS signal). This serves as an address latch control in the CU board
3. Transfer the data (write operation) or tri-state the bus (read operation).
4. Assert the Data Strobe (DS).
5. The CU acknowledges receipt of the data (ACK signal).

AS24* for DIO I or AS32* for DIO II

DS24* for DIO I or DS32* for DIO II

PRELIMINARY

10.3.2 Error Conditions

The CU board interface includes three error detection mechanisms:

- WI-to-CU Board Cable Interlock
- CU Error Reporting
- Parity Checking

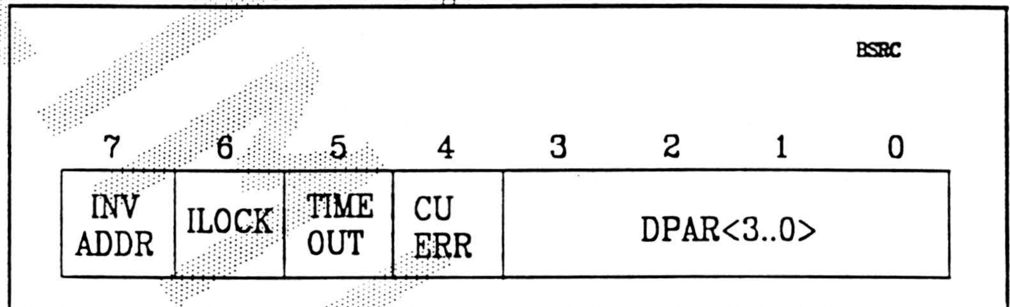
An interlock signal is used to assure the integrity of the WI-to-CU board cable. The absence of the interlock signal during a transfer will result in a DIO bus error.

The CU reports errors it detects to the WI board by means of the ERR signal. The types of errors that cause the CU to assert an ERR signal are described in the Bus Error Source Register (BSRC) bit definitions. Transfers that occur when the ERR signal is being asserted result in a DIO bus error.

Four parity bits accompany the data, address and control information transferred through the parallel interface to the CU board. If the CU detects a parity error when it receives a data or address transfer, a ERR signal will be issued. During read cycles, detection of bad parity by the CU board will result in a DIO bus error.

Whenever a DIO bus error occurs, the SPU microprocessor must read the Bus Error Source Register (BSRC) to determine the source of the error. Figure 10-9 illustrates the format of the BSRC register.

Figure 10-9 Bus Error Source Register Format



Bit 7, INV ADDR, indicates that the address supplied on the DIO II bus corresponded to a reserved (unused) section of the WI address space.

Bit 6, ILOCK, indicates a bad parallel interface cable connection between the WI board and the CU board. This bit is set during parallel transfers if the CU interlock signal is absent.

Bit 5, TIME-OUT. This bit is set if the CU fails to assert either an acknowledge (ACK) or error (ERR) within TBD seconds.

Bit 4, CU ERR. This bit is set if the CU asserts an ERR signal at any time during a data transfer. The CU will assert the ERR signal if the CU detects:

- a parity error in a transfer it receives from the WI board
- an invalid address
- an incorrect transfer size (some transfers must be on a word boundary)
- an aborted transfer due to a time out

For more specific information about an error, the software must poll the CU board.

Bits 3-0, DPAR, are set if a parity error is detected in data received from the CU during a read cycle. Note that data parity errors detected by the CU on write cycles cause the CU ERR bit to set. Parity errors can be simulated via the Parity Error Force Register at offset 0x203.

10.4 BPC Interfaces

There is a total of five Bay Power Controllers (BPCs), one in the bottom of each cabinet bay. Each BPC is responsible for controlling the Bay Power Supplies, monitoring the bay-level environment, and handling communications with the Workstation Interface board.

A single cable links the BPC interfaces on the WI board to the I/O Bay. Inside the I/O Bay, the cable is broken into individual cables for each BPC.

There are five BPC interfaces located on the WI board. Each BPC interface consists of three functional units:

- Cable Interlock
- Serial Interlock
- Reset Control

10.4.1 Cable Interlock

The cable interlock assures the integrity of both the bundled BPC interface cable that runs from the WI board to the I/O Bay and the individual BPC cables that interconnect the I/O Bay with each BPC. Each BPC has its own individual interlock signal. The signal loops back at the BPC and returns to an octal UART on the WI board. If an interlock signal changes state, an interrupt is generated on the WI board and sent to the SPU workstation.

10.4.2 Serial Interface

The serial interface allows the SPU to send commands to any BPC and to receive BPC status information. UARTs control the serial channels, and each UART has an assigned interrupt for initiating service requests to the SPU.

10.4.3 Reset Control

BPC reset signals may be used to reset the microprocessor located in each BPC. After being reset, the BPC microprocessor executes its power-up sequence.

PRELIMINARY

10.5 Key Switch Interface

SPU and modem key switches on the C3800 SPU are used to select the operating mode for the SPU workstation and its attached modem. A definition of the positions for these two key switches is provided earlier in this chapter in Tables 10-6 and 10-7. Execution of the operating modes of these two switches is a function of the SPU software.

Table 10-5 **SPU Operating Mode Key Switch Defintions**

Switch Position	Definition
OFF	Disables SPU and CPU access. This signals a power shutdown to the CPU. Hardware on the workstation interface board immediately resets the Bay Power Controllers (BPCs), which in turn reset their PPCs and removes power from each board in the system. Moving the keyswitch to one of the other positions does not cause power to return until the SPU software commands the BPCs to return power.
LOCAL	Allows both CPU and SPU access. Same as using ^P and ^D on the C100 or C200 SPU Console.
CPU ONLY	Allows only CPU access. This makes the SPU Workstation the equivalent of just another terminal tied to the operating system.
SECURE	SPU keyboard and mouse are mechanically disabled. This permits the SPU to continue with output, but user input is not possible. SPU software ignores the modem even if the modem key switch is in the SPU position.

Table 10-6 **SPU Modem Key Switch Definitions**

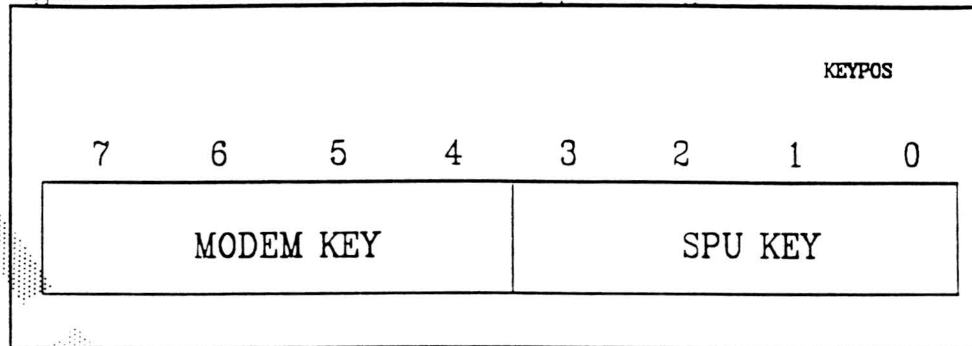
Switch Position	Definition
OFF	No modem access permitted.
SPU	Modem physically connected to SPU's serial port. Current SPU operating mode applies to SPU access via modem.
CPU	Modem physically connected to AUX serial port on workstation. External cable from AUX port to SYSTECH or VASYNC port will enable remote access to operating system regardless of SPU operating mode.

PRELIMINARY

The workstation interface board has a key position register that contains the current position of both the SPU and the modem key switches. A key change interrupt notifies the SPU if there is any change in the contents of the key position register. If a key change occurs, the appropriate key change interru[illegible]t (bit 4 or 5; see Tables 10-5 and 10-9) is set in the Interrupt Status Register. A the SPU detects the key change interrupt, the SPU software will read the key position register to determine the new switch position.

Figure 10-10 illustrates the format of the key position register.

Figure 10-10 Key Postion Register Format



Bits 7-4 contain a four-bit code that specifies the current position of the modem key switch. Each bit represents one switch position; the bit with a zero indicates the current switch setting.

Bits 3-0 contain a four-bit code that specifies the current position of the SPU key switch. Each bit represents one switch position; the bit with a zero indicates the current switch setting.

Table 10-7 defines the four-bit key position codes for both the SPU and modem halves of the key position register.

Table 10-7 Key Position Codes

Key Code	Modem Key Switch Position	SPU Key Switch Position
1110	OFF	OFF
1101	SPU	LOCAL
1011	CPU	CPU Only
0111	Not Used	Secure

10.6 Printer Interface

The workstation interface board includes a serial printer interface to support a local SPU printer. A UART channel is provided for this purpose.

10.7 Utility and Diagnostic Hardware

The Utility and Diagnostic Hardware consists of four registers that provide utility and diagnostic capabilities to the workstation interface. The four register are listed below and described in the text that follows.

- Miscellaneous Control Register
- Parity Error Force Register
- Loopback Data Register
- CU Address Register

10.7.1 Miscellaneous Control Register

Figure 10-11 illustrates the format of the Miscellaneous Control Register.

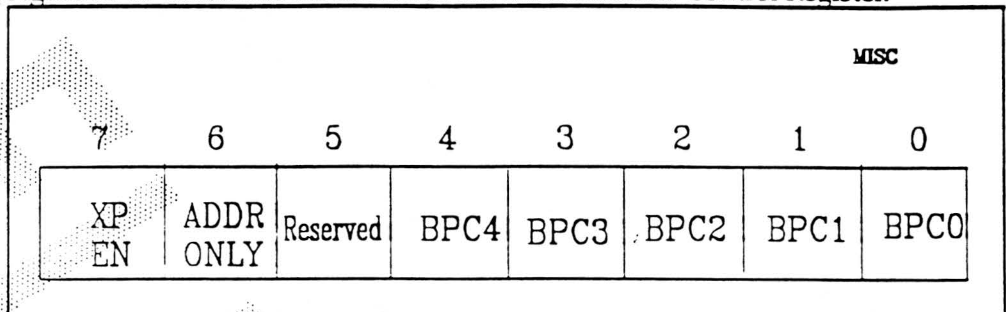


Figure 10-11 Miscellaneous Control Register Format

The XP bit (bit 7) determines whether or not the WI board will respond to DIO Bus cycles that map to the PBUS address space. When set, the WI board will initiate a transaction with the CU during such DIO bus cycles. When cleared, these DIO bus cycles will time-out without affecting the WI board. This is a requirement of the SPU system software to correctly configure the system at boot time. This bit is cleared by reset.

The Address Only bit (bit 6) is a diagnostic mode control bit. When cleared (normal operation), all accesses to the CU include both an address and data phase. These accesses use a large portion of the SPU-to-CU interface logic.

When set, which is considered the diagnostic mode, CU accesses have only an address phase. The CU address can subsequently be read back and verified. This mode of operation uses a minimum of CU logic.

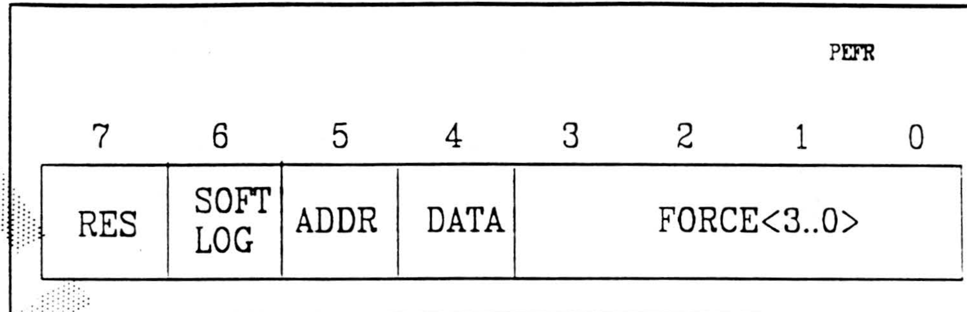
Bit 5 is not used and always will be a logic zero.

The BPC Reset bits (bits 4-0) drive the reset lines to each BPC. A logic one in a BPC bit position will reset (and hold reset) that particular BPC. A logic zero will release the respective BPC. Currently, all of these bits are set when a WI reset occurs.

10.7.2 Parity Error Force Register

For diagnostic purposes, parity errors associated with the CU parallel interface can be forced using the Parity Error Force Register (PEFR). Both address and dat parity errors can be forced separately (or both simultaneously). In additio parity errors can be soft logged. Soft logging results in an entry in the BSR without generating a bus error. Thus, system software can force parity errors (and verify their occurrence) without generating entries in the SPU error log. Th format of the PEFR is illustrated in Figure 10-12.

Figure 10-12 Parity Error Force Register Format



Bit 7 is not used and always reads a logic zero.

Bit 6 is the Soft Log bit. When set, parity errors set the appropriate PERR bits in the BSRC, but do not cause a bus error on the DIO bus. When cleared, parity errors also cause bus errors. This bit is cleared by reset.

Bit 5, the address force bit, controls the forcing of address parity errors. When se the force bits are XORed into the address and control parity on the CU interface

Bit 4, the data force bit, controls the forcing of data parity errors. When s force bits are XORed into the address and control parity on the CU interface.

Bits 3-0 are the force bits. When either one (or both) of bits 5-4 are set, the parity error force bits are XORed into the parity of the address and data paths of the C interface.

10.7.3 Loopback Data Register

The 32-bit Loopback Data Register (LBDR; offset 0x210) provides a loopback point in the data path of the CU parallel interface. Writing and reading this register tests the complete data path of the CU parallel interface, including the differential drivers and receivers.

Every write to the CU parallel interface causes selected data bytes and parity in the LBDR to be updated. This register always contains the last data written. If a bus error occurs, the contents of the LBDR may be retreived for analysis by executing an LBDR read operation.

10.7.4 CU Address Register

The CU Address Register (offset 0x214) is a 32-bit register located in the address path. Reading this register returns the address and control information from the previous read or write transaction to/from the CU board. This occurs without affecting the CU board logic. Physically, this register resides on the CU board.

Typical diagnostic use of this register involves performing address-only transfers to the CU board followed by a read of the CU Address Register at offset 0x214. This register always contains the address and control information of the last transaction involving the CU board. If a bus error occurs, the contents of this register may be retreived for analysis by executing a read operation.

10.8 Interrupt Register Logic

The CPU Utilities board can issue an interrupt to the workstation interface board. The interface board is then responsible for asserting one of the interrupt request lines, IR <6..3> *, on the DIO II bus. These interrupt lines are controlled by the interrupt enable (IE) bit and interrupt level (ILVL) bits in the card status and control register. The IE bit serves as a global interrupt enable for all interrupt requests destined for the DIO II bus from the workstation interface.

In additon to the CU board interrupts, there are interrupts that originate locally from within the workstation interface board. Among these are: UART service requests, BPC cable interlock connection errors, printer interrupts, and key switch changes.

Individual interrupts can be enabled or disabled by means of the Interrupt Enable Register located at offset 0x209. Upon receipt of an interrupt from the workstation interface board, the workstation processor reads the Interrupt Request Register (ISR; offset 0x20B) to determine the origin of the interrupt request.

The Interrupt Force Register (IFR) at offset 0x20D serves as a diagnostic aid. The force interrupt bits are ORed together with status bits to generate interrupt requests. Of course, the appropriate interrupt enable bit must also be set in the IE register.

The bit assignments across the three interrupt registers are identical. This is evident in Table 10-5, which defines the format of the WI board's three interrupt registers.

Table 10-9 briefly defines what each interrupt means and shows the corresponding interrupt register bit and the interrupt level associated with each interrupt.

The UART interrupts (bits 0-3) are requests from the octal UART. The UART status registers must be read to determine what service is required. Such services include transmit and receive operations, error reporting, and detection of a bad cable interlock.

The SPU key change interrupt (bit 4) is asserted whenever the SPU key switch position is altered.

The modem key change interrupt (bit 5) is asserted whenever the modem key switch position is altered.

Interrupt register bit 6 is reserved for future expansion.

Assertion of the CU interrupt (bit 7) indicates that the CU requires service. In response, the CU Interrupt Control Register will be polled to determine the cause of the interrupt. Other than the corresponding interrupt force register bit, there is nothing in the workstation interface that can initiate this interrupt.

Table 10-8 Interrupt Registers

Interrupt Registers	Interrupt Register Bits							
	7	6	5	4	3	2	1	0
Interrupt Enable Register	CU Board	Reserved	Modem	SPU	UART3	UART2	UART1	UART0
Interrupt Status Register	CU Board	Reserved	Modem	SPU	UART3	UART2	UART1	UART0

Table 10-9 Workstation Interface Interrupt Map

Interrupt Register Bit	Interrupt	Level	Comment
7	CU Interrupt	ILVL2	Check CU interrupt status
6	Reserved	ILVL2	Not used
5	Modem Key Change	ILVL2	Modem key switch has changed
4	SPU Key Change	ILVL2	SPU key switch has changed
3	UART3	ILVL1	Request for service, ports and 7
2	UART2	ILVL1	Request for service, ports and 5
1	UART1	ILVL1	Request for service, ports and 3
0	UART0	ILVL1	Request for service, ports and 1

PRELIMINARY

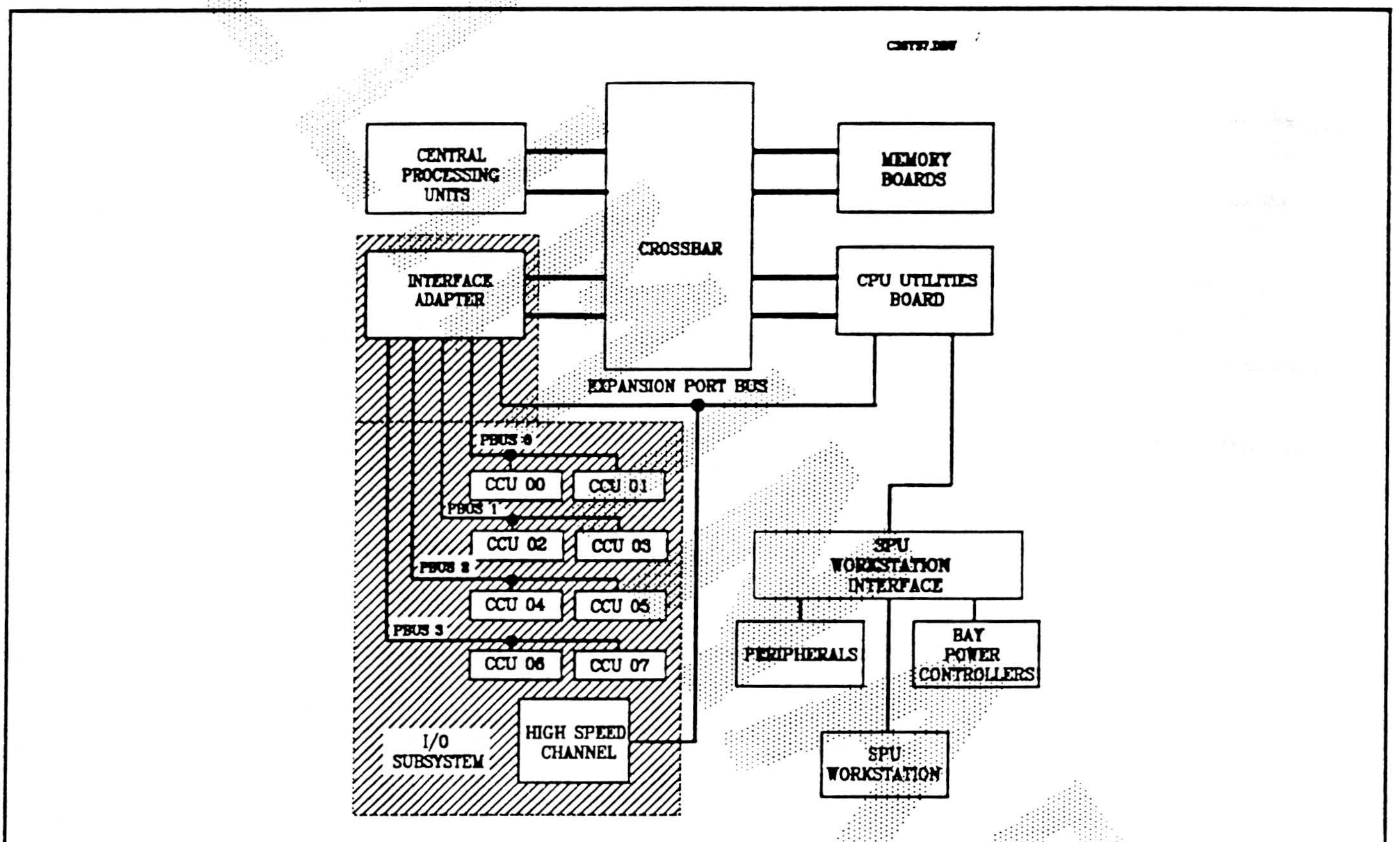
Input / Output Subsystem

11

11.1 Overview

The basic components of a C3800 Series Input /Output Subsystem, which are illustrated in Figure 11-1, consist of an Interface Adapter (IA), a high speed channel, the PBUS, and the Channel Control Units (CCUs). This chapter focuses primarily on the interface adapter, the PBUS, and the expansion port. Information about the other components of the I/O Subsystem can be found in the User Guide for that product.

Figure 11-1 I/O subsystem



The Interface Adapter (IA) serves as a memory interface for input/output controllers in the C3800 Series. The IA connects PBUS resident Channel Control Units to memory through the Crossbar. Both data and interrupts are handled by the IA. Interrupt handling for the C3800 Series computer systems differs significantly from the C100 and C200 Series computers.

Four separate PBUS interfaces and a newly defined I/O interface, called the Expansion Port, are supported by the IA. The PBUS is the general purpose input/output interface bus. The Expansion Port (XP) is a fast, special purpose input/output channel that provides a higher performance alternative to the PBUS.

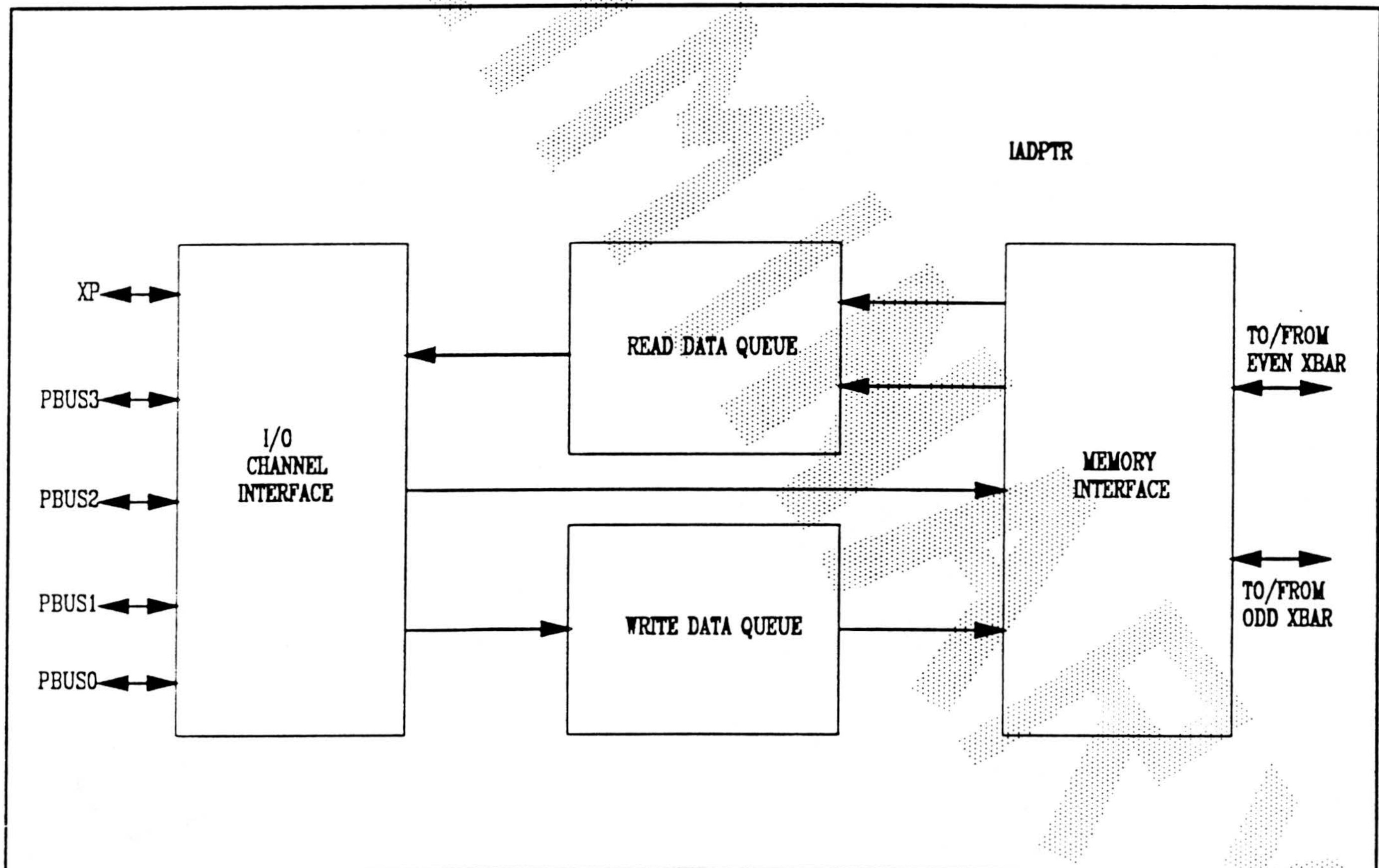
To the CCUs, the IA looks like the Peripheral Interface Adapter (PIA) used on other CONVEX C-Series computers. To the Crossbar and memory, the IA looks like a processor.

11.2 Interface adapter

The Interface Adapter (IA) contains four major functional unts that are defined below. See Figure 11-2 for a simplified block diagram of the IA.

1. I/O Channel Interface - contains the data path and control logic for interfacing with the four PBUS channels and the XP channel.
2. Read Data Queue - provides temporary storage for data coming from memory whose destination is an I/O channel.
3. Write Data Queue - provides temporary storage for I/O data destined for memory.
4. Memory Interface - includes the Crossbar interface, read and write data staging registers, and memory read request identification and tracking.

Figure 11-2 Interface adapter simplified block diagram



11.2.1 Input/output channel interface

The I/O Channel Interface contains the data path and control logic for interfacing with the four PBUS channels and the XP channel. Most of the data path resides on eight Channel Data Slice (CDS) gate arrays. Bus arbitration, state machine, and control logic are implemented external to the gate arrays.

Each I/O channel requires a 64-bit (plus an 8-bit parity) data path. There are eight CDS arrays. Each CDS array contains an eight-bit (plus one parity bit) slice of the data path for each of the five I/O channels.

The CDS array receives a system clock from the master clock logic located on the CPU Utilities board. The CDS array divides down the system clock to produce PBUS and XP clocks.

Parity checking is performed on all incoming write data as it enters the CDS array.

The data path in each CDS array consists of three sections for each I/O channel: write data buffering, read data buffering, and a partial header holding register.

Write data buffering

There is only one level of buffering for write data from PBUS channels. As write data is received from a PBUS channel, it is immediately written into the write data queue. Each PBUS interface is guaranteed one write cycle access to the write data queue during each PBUS cycle.

The XP channel interface has two levels of write data buffering; its write data queue access is restricted to cycles that have no PBUS write data.

Read data buffering

Two levels of buffering are provided for the PBUS interface channels. The first level, which is clocked by a system clock, sources read data from the read data queue. The second level is PBUS clocked and sources read data to the PBUS.

The XP interface channel has three levels of read data buffering. The first two registers are system clocked, and the third is XP clocked. XP read data is sourced from the read data queue during cycles in which no PBUS read data is available.

Partial header register

The partial header register holds the I/O channel's memory request (header) until the request can be forwarded to the Interface Adaper's memory interface. For small requests (read or writes of 16 long words or less), the entire request is forwarded to the memory interface at one time. For memory requests greater than 16 long words, the request is split into multiple 16 long word requests. The partial header register holds the transfer type, the starting addrss and the remaining byte count of the memory request. After the last request is forwarded to the memory interface, the partial header register becomes available to receive a new header from the I/O channel interface.

11.2.2 Read data queue

The Read Data Queue (RDQ) provides temporary storage for memory read data that is destined for an I/O channel. Read queue arbitration logic controls the flow of data from the RDQ to the PBUS and XP channels.

The RDQ contains the following functional units:

- Two independent 2K-location by 36-bit queues (called MS and LS RDQs)
- Multiple address counters for writing data into MS and LS RDQ
- Multiple address counters for reading data from the MS and LS RDQs
- Multiple fill-level counters for tracking the degree of fullness of the MS and LS RDQs

The RDQ's structure is similar to that of the C3800 Series memory subsystem. Like the memory subsystem, the RDQ has two independent, 36-bit halves (32 data bits and 4 parity bits). In the memory subsystem these two halves are referred to as even and odd; in the RDQ they are labeled MS RDQ (for even memory data words) and LS RDQ (for odd memory data words).

There is one MS RDQ and one LS RDQ for each I/O channel. Since there are five I/O channel interfaces, there are five MS RDQs and five LS RDQs. The five MS RDQs are implemented using 2K-location by 9-bit self-timed RAMS, as are the LS RDQs. The 2K-location RAMs are divided into eight sections, each containing 256 locations. Each I/O channel is allocated one section.

Separate addresses (called head pointers) are required for writing data received from memory into each MS and LS RDQ. These head pointers are derived from eight-bit counters. There is a total of ten head-pointer address counters (five MS and five LS head-pointer address counters). As read data is received from memory, the head pointers are used to address the RAMs when writing data into the RDQ. After each write access, the head pointers are incremented in preparation for the next write.

For reading data out of the RDQs, one address is used for each MS RDQ and LS RDQ pair. Referred to as a tail pointer, this address is also derived from a set of eight-bit counters. There is one tail-pointer address counter for each I/O channel. Each time data is read froman RDQ pair (MS and LS) the appropriate counter is incremented in preparation for the next read.

Both MS and LS RDQs are read at the same time in order to return read data in long word (72-bits) transfers to the I/O channel device.

Fill-level counters are used to track the fullness of the RDQs. The fill level of the MS and LS RDQs are kept in separate eight-bit counters. Each I/O channel interface has an MS RDQ fill-level counter and an LS RDQ fill-level counter. The fill-level counter is incremented when data is written into the RDQ and is decremented as data is read from the RDQ.

The RDQ has one read and one write cycle for each system cycle. Reads take place during the first half of a system cycle and writes occur during the second half.

11.2.3 Write data queue

The Write Data Queue (WDQ) provides temporary storage for I/O data that is destined for memory. Write queue arbitration logic controls the flow of data from the PBUS and XP channels into the WDQ.

The WDQ contains the following functional units:

- One WDQ for both even and odd word data
- Five address counters for writing data into the WDQ
- Five address counters for reading data from the WDQ
- Five fill-level counters for tracking the WDQ's degree of fullness

Both the PBUS and XP channels send write data in long word transfers (72-bits). The WDQ is organized to accept long word writes into the RAMs. Unlike the RDQ, which is implemented as two 36-bit wide buffers, the WDQ is one 72-bit wide buffer.

Each I/O channel interface is allocated its own WDQ for buffering I/O write data destined for memory. Thus, there are five WDQs implemented in 2K-location by 9-bit self-timed RAMs. Within the 2K-location RAMs, each I/O channel is assigned one 256K-location section as its WDQ.

Each I/O channel uses a separate eight-bit write address counter to access its WDQ. Write addresses are called head pointers. As write data is received from the I/O channel device, the head pointer is used to address the next sequential RAM location in the WDQ. After each WDQ write, the head pointer is incremented by one in preparation for the next write operation.

Each I/O channel has a separate eight-bit read address counter that is used for WDQ read access. Read addresses are called tail pointers. The memory interface initiates WDQ read operations of the I/O data stored there. These are 72-bit (long word) read operations. After each read operation, the tail pointer is incremented by one in preparation for the next WDQ read.

As with the RDQ, fill-level counters track the fullness of the WDQ. Each I/O channel interface has its own WDQ fill-level counter. The WDQ fill-level counters are incremented as data is written into the WDQ and decremented as data is read from the WDQ.

Like the RDQ, the WDQ has one read and one write cycle for each system cycle. Reads take place during the first half of a system cycle and writes occur during the second half.

11.2.4 Memory interface

The memory interface contains two 36-bit wide data paths that serve as an even and odd word interface to the Crossbar. Most of the memory interface resides on four Crossbar Data Slice (XDS) gate arrays; however, the memory interface does include some read data staging registers and control logic that are external to the XDS arrays.

The memory interface has a total of four XDS gate arrays. They function as two pairs. One pair for the 36-bit even word interface and one pair for the 36-bit odd word interface.

In addition to the data paths, the XDS gate arrays also contain even and odd word address and control Crossbar interfaces.

Each XDS gate array includes a 24-location by four-bit FIFO for staging I/O channel IDs and memory error /status flags. An ID tag is generated for each memory read request that originates from an I/O channel. The ID tag is stored in the FIFO and used to identify returning memory read data.

The XDS FIFO also stages error status and flags associated with each read request. The XDS gate array checks for PCM address violations. If a PCM violation is detected, the read request associated with the violoation will not be sent to the Crossbar. The PCM error indication is stored in the FIFO and used to terminate the I/O channel handshake after all previously requested read data has returned from memory.

The address and fill counters used by the Read and Write Data Queues are physically located in the XDS gate arrays. Refer to the previous sections that describe the Read Data Queue and Write Data Queue for a functional description of the address and fill counters.

Outside the XDS gate arrays, the memory interface provides staging and control of read data returning from memory. Two staging registers buffer incoming read memory data that is enroute to the Read Data Queue (RDQ). The memory interface also exercises control over writing this data into the RDQ. Specifically, the memory interface controls selection of the correct RDQ head pointer (write address) and generates increment controls for both the head pointer counters and the fill-level counters.

The memory interface also contains other logic external to the XDS gate arrays that control reading I/O data destined for memory from the Write Data Queue. This control includes a word counter for generating the appropriate number of read requests to the WDQ, logic for controlling data flow through the XDS gate array, and responsibility for issuing requests to the Port Arbiter for the next header.

11.3 Arbitration

The Interface Adapter contains a variety of different arbiters. The most obvious ones are the arbiters located in the I/O Channel Interface that arbitrate access to PBUS and Expansion Port data and interrupt buses. In addition, the Read Queue, Write Queue, and the Memory Interface contain arbitration logic. These arbiters are listed below and described in the text that follows.

- Read Queue Arbiter
- Write Queue Arbiter
- Port Arbiter

11.3.1 Read queue arbiter

The memory interface controls writing incoming memory data into the RDQ, and the read queue arbiter controls reading that data from the RDQ. Once data is written into the RDQ, the read queue arbiter is notified. The read queue then controls read access of the RDQ for subsequent transfer of the data to the appropirate I/O channel. PBUS I/O channels are granted read access every PBUS cycle. The Expansion Port is granted access only when read access is not required by one of the PBUS I/O channels.

The purpose of the read queue arbiter is summarized as follows:

- Control RDQ read access and assure that read data can be returned on all four PBUS I/O channels every PBUS cycle.
- Minimize read data return latency through the Interface Adapter.

- Maintain an adequate throughput rate at the Expansion Port even when all four PBUS I/O channels have active data returning.

11.3.2 Write queue arbiter

The memory interface controls reading I/O data destined for memory from the WDQ, and the write queue arbiter controls writing that data into the WDQ. The write queue arbiter allocates write access cycles as write data is received by the five I/O channels. Priority is given to PBUS I/O channels. Data from a PBUS channel is always accepted for WDQ write access, unless the WDQ is full.

The purpose of the write queue arbiter is summarized as follows:

- Control WDQ write access so that write data can be accepted from all four PBUS I/O channels every PBUS cycle.
- Minimize the number of write data staging registers needed between the I/O channel interface and the WDQ.
- Maintain an adequate throughput rate at the Expansion Port even when all four PBUS I/O channels are generating write transfers.

11.3.3 Port arbiter

The port arbiter controls the selection of I/O channel interrupts and memory requests for service by the memory interface. Interrupts have a higher priority than I/O channel memory requests. The port arbiter employs a round robin arbitration scheme where the most recently selected I/O channel assumes the lowest priority during the next arbitration period.

I/O channel memory requests are not presented to the port arbiter until they are ready for service. For memory write operations, the request is not posted until a sufficient amount of write data is in the Write Data Queue. Memory read operations are not posted until there is sufficient buffer space available in the Read Data Queue.

11.4 Interrupt handling

A C3800 Series system may have multiple Interface Adapters. Each IA supports two interrupt buses, one for the PBUS (and its associated CCUs) and the other for the expansion port.

Interrupt handling for the C3800 Series differs from that of prior C-Series computers. In the earlier systems, the posting and acknowledgment of interrupts occurred in one operation. More specifically, at the end of the interrupt sequence the device that posted the interrupt would see an acknowledgment fromthe receiving device. In the C3800 Series, the posting and acknowledgment are three (or four) separate operations.

11.4.1 CPU-to-CPU interrupts

CPU-to-CPU interrupts are handled in a three-step process, which is outlined as follows:

1. A CPU posts an interrupt by sending a SND_X transfer to the Trap Control Register (TCR). The TCR is semaphore protected register located on the CPU Utilities (CU) board.
2. Once posted, the CU forwards the interrupt to the complex's Global Pending Register and clears the TCR semaphore bit.

3. The CU board's Trap Dispatch logic transfers the interrupt to the targeted CPU's Local Pending Register by way of a dedicated point-to-point bus. When the targeted CPU acknowleges the interrupt with TRAP_COMP, the Global Pending Register is cleared and made available to accept another trap or interrupt.

11.4.2 CPU-to-CCU Interrupts

CPU-to-CCU interrupts are handled in a three-step process, which is outlined as follows:

1. A CPU posts an interrupt by sending a SND_X transfer to the Trap Control Register (TCR). The TCR is a semaphore protected register located on the CPU Utilities (CU) board.
2. Once posted, the CU forwards the interrupt to all the Interface Adapters in the system by way of dedicated point-to-point trap buses. After all IAs acknowledge the interrupt with TRAP_COMP, the CU clears the TrapControl Register. Each IA acknowledges receipt of the interrupt on behalf of the targeted CCU and does so before the targeted CCU actually gets the interrupt. All IAs in the system accept the interrupt without knowing whether it is intended for one of their local CCUs.
3. Each IA in the system forwards the interrupt to their respective CCUs. At this point the IA still does not know which CCU is the target. If the interrupt is not acknowledged by one of its local CCUs, then the IA assumes that the interrupt was intended for a remote CCU attached to another IA.

11.4.3 CCU-to-CPU Interrupts

CPU-to-CCU interrupts are handled in a four-step process, which is outlined as follows:

1. The CCU requests an interrupt cycle. Once granted, the CCU drives its interrupt request on the interrupt bus.
2. The Interface Adapter acknowleges the CCU interrupt on behalf of the targeted CPU and then posts the interrupt to the CU's Trap Control Register.
3. The CU forwards the interrupt to the Global Pending Register and clears the Trap Control Register semaphore.
4. The interrupt is transferred to the targeted CPU's Local Pending Register by way of the trap bus

11.5 PBUS

The C3800 Series I/O subsystem may have one or more Interface Adapters (IAs). Each IA supports up to four PBUSes, and each PBUS can have two Channel Control Units (CCUs) for a total of up to eight CCUs per IA.

The PBUS is a synchronous, 10MHZ multi-master bus that uses a block transfer bus protocol. The PBUS serves as a key component in the interface between the I/O subsystem and memory.

Each CCU on the PBUS is capable of being the bus master on any given bus cycle. All PBUS transfers are either to or from the Interface Adapter with bus arbitration performed by the IA. Transfers between CCUs are not supported.

11.5.1 PBUS transfers

All transfers begin with a CCU request to be the next PBUS master. When the IA grants the bus to the requesting CCU, that CCU will drive a header transfer on the PBUS. All other CCUs on the bus should have their PBUS interfaces tristate until they ask for and are granted the bus. The bus cycle immediately following the header transfer is unused.

The header provides information about the specific transfer type being requested, the transfer byte count, and the starting address of the data to be sent or received. Figure 11- 3 illustrates the header format, and Table 11-1 lists the transfer types and their associated transfer codes. The transfer code appears in bits 42-40 of the header.

Figure 11-3 PBUS header format

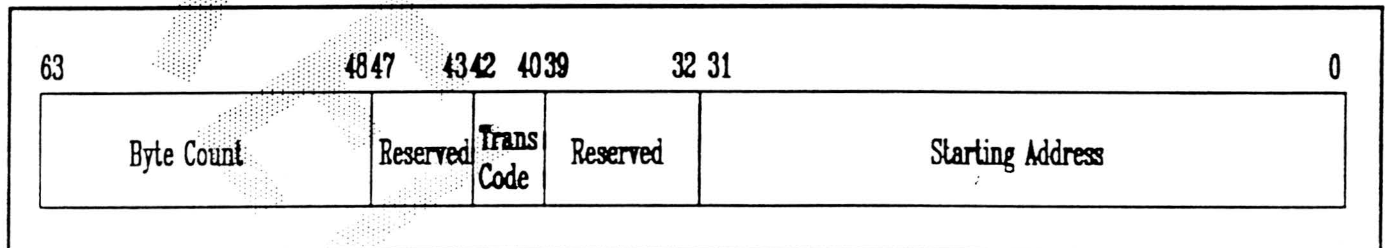


Table 11-1 PBUS transfer types

Transfer code	Transfer type
0	Memory Write
1	Memory Read
2	Test and Set
3	Test and Clear
4	Illegal Code
5	Illegal Code
6	I/O Read
7	No Operation

If the CCU requests a write transfer, it may send data as long as the IA asserts the memory buffer available signal.

After a read request header, the IA automatically sequences read requests to memory. When the requested data begins to return from memory, the IA will check the channel buffer available signal to determine whether the CCU is able to accept the data. If the signal is detected, the IA will drive the data longword on the PBUS. After the last data longword is transferred, the IA will remove that CCU's grant signal and the transfer is complete.

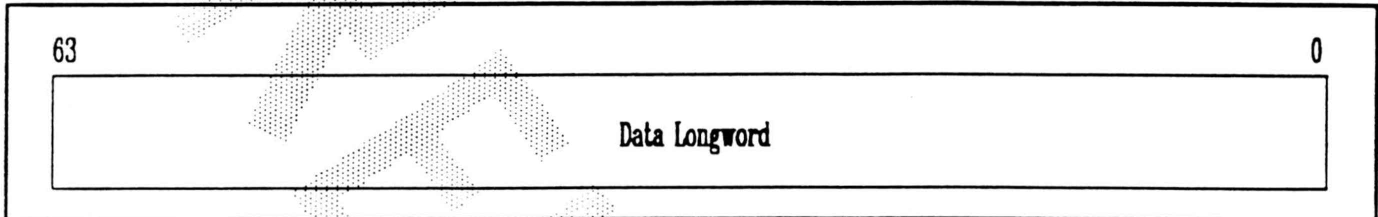
11.5.2 PBUS data path

The PBUS data path consists of those lines used by the CCUs to send header transfers, send write data, and receive memory read data.

When the IA selects a new CCU as the bus master, the first transfer from the CCU is expected to be a header transfer. Header transfers are used to initiate the transfer type specified in the transfer code.

After the initial header transfer, all subsequent PBUS transfers are data transfers. Data transfers always consist of 64 bits of data and they are either data write transfers or data read transfers. Data write transfers originate from a CCU and are destined for memory. Data read transfers originate from memory and are distined for the CCU that requested the data. Figure 11-4 shows the format for 64-bit data longwords sent across the PBUS.

Figure 11-4 PBUS data format



The 64-bit data bus uses standard CONVEX byte ordering. No rotation or alignment of data is done by the IA; this is solely the responsibility of the CCU.

Data signals are labeled Px_DATA<63..0>. Since the IA supports four PBUS interfaces, Px represents any one of the four buses.

The PBUS data path has eight parity bits. Each bit represents the parity of one data byte. Parity signals are labeled Px_PAR<7..0>. The number, 7..0, specifies which data byte the parity bit is associated with. Whenever the data bus is being used to transfer information, the parity on all bytes of the bus must be correct. Parity on the bus should not be checked when information transfers are not taking place.

Parity is correct when it is even; that is, the sum of the logical ones in a data byte and its parity bit must be even. This convention is used so that a floating TTL bus will have bad parity.

11.5.3 PBUS control path

The PBUS control path consists of those line used by PBUS devices for PBUS arbitration, data path handshake, and PBUS error notification. PBUS control path signals are identifed in Table 11-2 and described in the text that follows the table.

Table 11-2 PBUS control signals

Signal	Name
Px_CCR<n..0>*	PBUS request
Px_CCG<n..0>*	PBUS grant
Px_BUSLOCK*	Bus lock
Px_HDR*	Header
Px_DVAL*	Data valid
Px_MBAV*	Memory buffer available
Px_CBAV*	Channel buffer available
Px_BUSERR*	Bus error
Px_MEMERR*	Memory error

11.5.3.1 PBUS request

Each CCU has its own discrete request line that it asserts to ask the bus arbiter for permission to use the PBUS. Initially, a CCU may assert a PBUS request only when the PBUS grant signal is not active. After assertion of a PBUS request, the grant signal will become active; the grant signal will remain active for the duration of the transfer. Likewise, the request signal must be active for the entire duration of a PBUS transfer; if it becomes inactive, any transfer in progress will be immediately terminated.

To do two back-to-back transfers, the CCU must:

1. Deactivate the PBUS request signal at the end of the first transfer.
2. Wait for the grant signal to become inactive.
3. Reassert its PBUS request signal.

11.5.3.2 PBUS grant

Each CCU receives a discrete grant signal from the PBUS arbiter. Only one grant can be active at any given time. A round robin arbitration scheme is used to determine which CCU becomes the bus master whenever there are multiple requests.

The assertion of a grant signal begins the transfer cycle, and it ends when the grant signal is no longer active. The bus aribiter may deactivate the grant signal at any time.

11.5.3.3 Bus lock

This function is not supported by the C3800 Series Interface Adapter. The signal is terminated in the CCU backplane.

11.5.3.4 Header

The header signal is asserted by the bus arbiter during the bus clock in which it expects the current bus master to drive its transfer header onto the data bus. This signal immediately follows the clock cycle in which the grant signal was asserted. Normally, this signal is not used by the CCUs; it is primarily for use during system debug.

11.5.3.5 Data valid
This signal is asserted by CCUs during memory write transfers and by the IA during memory read transfers. It indicates that valid data has been placed on the PBUS. This signal is valid only when both the request and the grant signals are currently active. This signal may be modulated during a transfer sequence if the data source cannot sustain transfers at the full PBUS rate.

11.5.3.6 Memory buffer available
The IA asserts this signal during a memory write transfer sequence to indicate when it is ready to accept data. The simultaneous assertion of request, grant, Px_DVAL, and Px_MBAV constitutes a memory write information transfer cycle. This signal may be modulated during a memory write transfer sequence if the memory subsystem cannot accept data at the full PBUS rate.

11.5.3.7 Channel buffer available
The CCU asserts this signal during memory read transfers to indicate when it can accept data from the IA. The simultaneous assertion of request, grant, Px_DVAL, and Px_CBAV constitutes a memory read information transfer cycle. If the CCU cannot accept data at the full PBUS rate, the Px_CBAV signal may be modulated by the CCU during a memory read transfer sequence.

11.5.3.8 Bus error
The IA uses the bus error signal to notify a CCU that a transfer error has been detected. The details of the error are kept in the IA and are not returned to the CCU. Bus errors may be caused by: bad parity in a header transfer froma CCU, an attempt to access non-existent memory or I/O space, and headers that specify illegal operations. A bus error causes any transfer currently in progress to terminate.

11.5.3.9 Memory error
In the C3800 Series, memory errors are not reported to the PBUS. Therefore this signal should never be active.

11.6 Expansion port

The Expansion Port (XP) is a high speed input/output bus that provides a higher performance alternative to the PBUS. It is for applications that require the use of high speed channel devices or other high performance data links.

The XP may be used with the High Performance Peripheral Interface (HiPPI). The HiPPI must be physically located in an adjacent board slot in the same backplane as the IA.

11.6.1 XP and PBUS differences
The primary difference between the PBUS and the XP is throughput. The XP bandwidth is higher than the PBUS due to a faster cycle time (32ns). One other difference is the fact that the PBUS uses even parity and the XP uses odd parity.

11.6.2 XP transfer types
The XP supports the following transfer types:

- Memory Read
- Memory Write
- Test and Set
- Test and Clear
- I/O Read
- Memory Scrub

11.6.3 XP header transfers

Memory requests are initiated by the XP device using a header transfer. They contain a 32-bit starting address, a two-bit encoded transfer type, and a 16-bit byte count. The format of the XP header is identical to the format used for PBUS headers.

11.6.4 Expansion port interface signals

The expansion port interface signals are divided into four groups: data bus, data handshake, error status, and interrupt bus. These signals are listed in Table 11-3 and defined in the text that follows the table.

Table 11-3 XP bus interface signals

Bus signal group	Signal mnemonic	Signal name
XP data bus signals	NXP.DATA<63..0>	XP data bits
	NXP.PAR<7..0>	XP parity bits
XP data handshake signals	XIOP-NIA.CBVF_AVL	XP channel buffer available
	XIOP-NIA.WRT_VAL	XP device write data valid
	NIA-XIOP.MBVF_AVL	IA memory buffer available
	NIA-XIOP.RD_VAL	IA read data valid
	XIOPx_IA.BUS_REQ	XP device bus request
	IA_XIOPx.BUS_GNT	IA bus grant
XP error status signal	NIA-XIOP.BUS_ERR	IA bus error
XP interrupt bus signals	NXP.INT_VEC<7..0>	XP interrupt vector bus
	XIOPx-NIA.IBREQ	XP device x interrupt request
	NIA-XIOPx.IBGRNT	IA interrupt grant to device x
	NIA-XIOP.IBINT	IA interrupt valid

11.6.4.1 XP data bus

The XP data bus consists of 64 data bus lines and their associated byte parity. The bidirectional data bus provides a path for transferring write data, read data, and control information between an XP device and the IA.

There are eight parity bits, one for each byte of write or read data. Correct parity is odd; that is, the sum of all logical ones in a data byte and its associated parity bit should equal an odd value. Parity checking occurs for all data bus read or write transfers (both data and header transfers). Parity checking does not occur when the data bus is idle.

11.6.4.2 XP bus data handshake signals

There are six XP bus data handshake signals; they are defined in the text that follows.

XP channel buffer available

The XP channel device asserts this signal during memory read operations to indicate when it can accept data from the IA.

XP device write data valid
XP devices assert the write data valid signal during write transfers to indicate that valid data is being driven on the XP data bus.

IA memory buffer available
The IA asserts this signal to indicate when it can accept data from the XP.

IA read data valid
The IA asserts the read data valid signal whenever it drives valid read return data on the XP data bus.

XP device bus request
The XP device asserts this signal to request permission from the bus arbiter to use the XP bus.

IA bus grant
The IA issues this signal to give an XP device permission to use the XP bus.

11.6.4.3 IA bus error

The IA uses this signal to tell an XP device that a transfer error has been detected.

11.6.4.4 XP interrupt bus

There are four interrupt vector bus signals; they are defined in the text that follows.

Interrupt vector bus
The interrupt vector bus is a bidirectional bus that is used to transfer interrupt vectors between XP bus devices and the IA.

XP device interrupt request
The XP device asserts this signal to request interrupt bus access for the purpose of sending an interrupt vector to the IA.

IA interrupt grant
This signal is issued by the IA to grant use of the interrupt bus to a requesting XP device. The IA asserts the signal after the XP device requests the bus and when the IA is prepared to accept an interrupt from the requesting XP device.

IA interrupt valid
This signal will be active during the same bus cycle that the IA sends a valid interrupt vector to the XP device.

PRELIMINARY

Power Subsystem

12

12.1 Overview

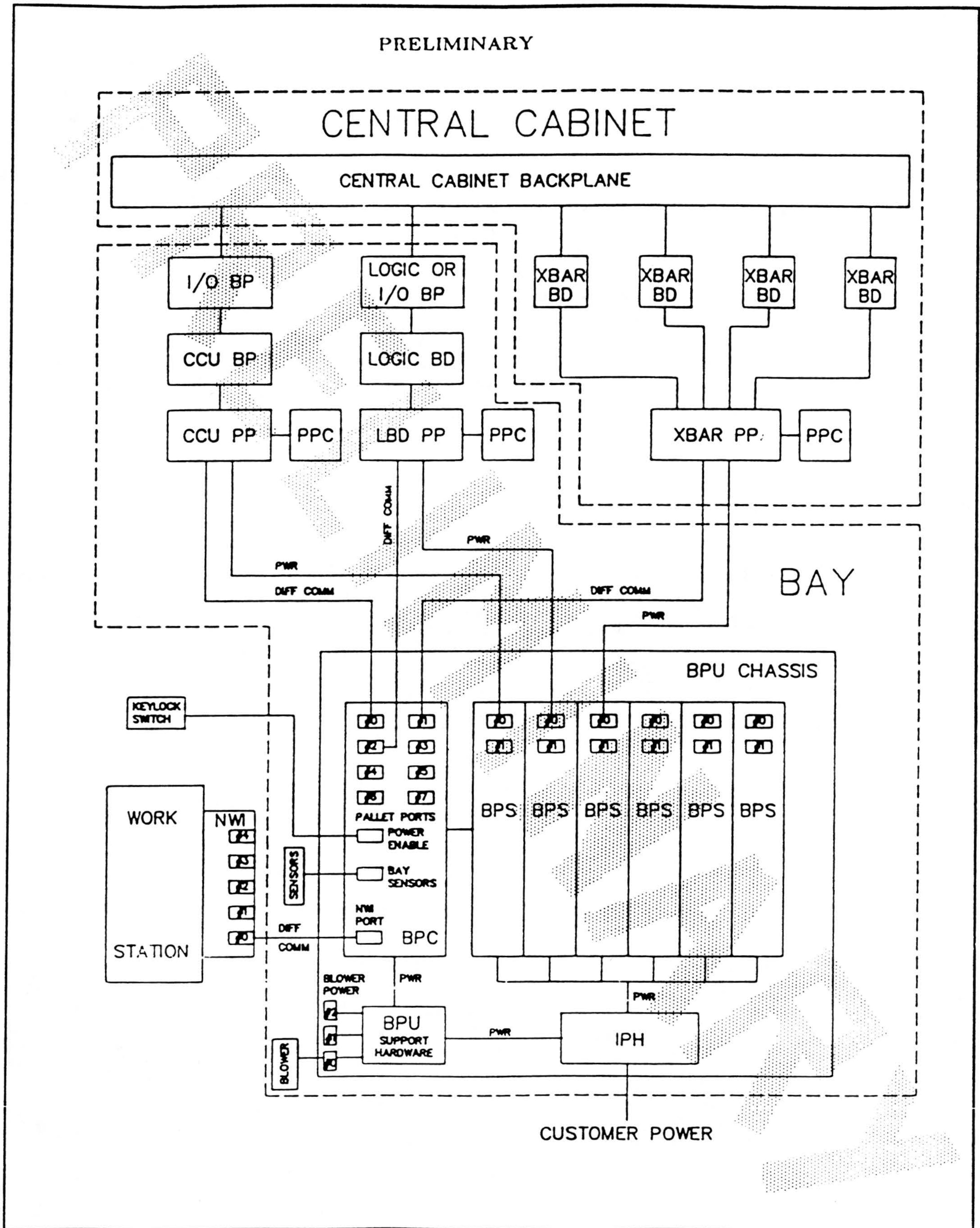
The C3800 Series power subsystem uses a distributed power architecture. Input AC power is rectified into unregulated DC and then distributed to DC-to-DC converters that are physically located close to the load.

The control and monitoring of power and environmental functions is also distributed. Rather than cabling all control and sense lines to a central processor, each load supply contains its own local processor. In addition, each bay contains a processor to concentrate communications to all loads within the bay and to control and monitor bay-level functions.

The major units of the C3800 Series power system are illustrated in Figure 12-1 and described in the text that follows.

PRELIMINARY

Figure 12-1 C3800 Series Power Subsystem



12.2 Input AC Power

Each bay has an independent power cord for power input from the customer's AC power source. The only exception is the central cabinet; it receives 300 Volts DC directly from the I/O Bay and therefore has neither a power cord nor a Bay Power Unit.

12.3 Bay Power Unit (BPU)

The BPU chassis is physically located at the bottom of the bay and contains all the front end power filtering, rectification, and bay-level control and communication hardware. Specifically, the BPU includes the following:

- Input Power Harmonizer (IPH)
- Bay Power Controller (BPC)
- From one-to-six 2.5KW Bay Power Supplies (BPS)

12.3.1 Input Power Harmonizer

The Input Power Harmonizer brings AC input power into the BPU and distributes it to the Bay Power Supplies, the Bay Power Controller and the blower motor. An important feature of the IPH is the fact that it contains all the components required to configure the power system for either domestic or international AC power.

12.3.2 Bay Power Supply (BPS)

There are from one-to-six Bay Power Supplies. The number of BPS used is configuration dependent; that is, dependent on the number and size of the loads in the bay. Each BPS generates 2.5KW of unregulated 300 VDC that is distributed to a maximum of two Power Pallet loads.

12.3.3 Bay Power Controller (BPC)

The BPC is responsible for controlling the Bay Power Supplies, monitoring the bay-level environment, and handling communications between the Workstation Interface and the attached Power Pallet Controllers (PPCs).

12.3.4 Power Pallets

Figure 12-1 shows the three types of power pallet loads. Each power pallet provides the mounting and peripheral circuitry needed to support a number of DC-to-DC converters, which are called Power Bricks. The power bricks accept unregulated 300 VDC from the BPSs and regulate down to the required low voltage power buses.

12.3.4.1 Channel Control Unit Power Pallet (CCUPP)

The first type of power pallet load (see Figure 12-1) is the CCUPP. It contains up to 14 DC-to-DC converters (power bricks) that provide DC voltages for up to four CONVEX Channel Control Units. The CCUPP plugs into the CCU backplane and provides power through copper planes in the CCU backplane to the CCU connectors.

12.3.4.2 Logic Board Power Pallet (LBDPP)

The second type of power pallet load is the LBDPP. It contains up to 14 power bricks that provide DC voltages to the logic board attached to the power pallet. In addition to DC power there are sense lines, EEPROM interface lines, temperature sensors, and ID address information that interface to the logic board.

12.3.4.3 Crossbar Power Pallet (XbarPP)

The third and last type of power pallet load is the crossbar power pallet. It holds the power bricks that provide DC power to the crossbar boards. There are two identical crossbar power pallets in the system. One provides power to three crossbar boards, the other to the remaining four crossbar boards. Foreplane jumper boards provide interfacing between the pallet and the crossbar boards for DC power, EEPROM interfaces, and ID address information.

12.3.5 Power Pallet Controller

All power pallets are controlled by a PPC. The PPC is a common controller that plugs into any of the power pallet load types. It has an on-board processor that communicates by way of a serial link to the Bay Power Controller of the bay where the pallet resides. The main functions of the PPC are:

- Power-up configuration checking
- Power-up sequencing
- Power monitoring and margining
- Power system failure detection and shutdown
- Workstation Interface (WI)

All bays communicate to the system console through separate serial links that connect each bay's BPC to the system console interface (sometimes called the workstation interface). The system console is a free-standing workstation that provides the system bring-up function, system monitoring and error logging, and system diagnostics interface to the system. The system console receives status and issues commands to each bay; it is the main control point for all system power.

12.3.6 Power Enable Keylock

A central power enable keylock is provided on the I/O bay. This keylock interfaces to each BPU and provides a central point for emergency shutdown of the entire system. Other keylock functions for remote and diagnostic enable are provided at the system console.

PRELIMINARY