AQUARIUS/ARIDUS
Preliminary Information Package

This document is classified as:

DIGITAL RESTRICTED DISTRIBUTION

As such, it must be controlled and protected as a vital business resource. Handling, storage, and reproduction of this document must conform to Proprietary Information Protection Standard 10.1 of the Corporate Security Policies and Standards.

Do not reproduce this document. If additional copies are required, send a request to:

- Ed McFaden
- SUCCES::MCFADEN
- DTN: 297-6955

DOCUMENT NUMBER:

ASSIGNED TO:

digital equipment corporation
maynard, massachusetts

5/26/88

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

The following are trademarks of Digital Equipment Corporation:

DEC
DEC/CMS
DEC/MMS
DECnet
DECsystem-10
DECSYSTEM-20
DECUS
DECwriter
DIBOL
EduSystem
IAS
MASSBUS
PDP
PDT
RSTS
RSX
UNIBUS
VAX
VAXcluster
VMS
VT

Contents

2 Technology and Packaging Descriptions

3 CPU Subsystem Overview

4 SCU and Main Memory Functional Overview

5 I/O Subsystem Description

6 Power and Control Subsystem Overview

7 SPU and Scan Subsystem Overviews

8 System Initialization

9 System Interrupts and Exceptions

10 Diagnostic System Overview

Glossary

Examples

Figures

Tables

About This Manual

The major goal of the PIP is to provide a system resource package early in the Program to familiarize the reader with the basic functional structure, new technologies, and hardware implementations incorporated into the AQUARIUS system family.

It is NOT the intent of the PIP to be completely accurate when compared against the current design. To have included all engineering changes would have prohibited an early PIP distribution.

The PIP was developed using all Logic Design, Technology, Diagnostic, and CSSE plans and specifications as resource material. Wherever possible and appropriate, specification content was abstracted or lifted. The content of the Engineering technical exchange video tapes, and preliminary technical description and instructor guide drafts were also used as resource material.

The organizational structure of the PIP is based on chapters describing major AQUARIUS/ARIDUS subsystems (e.g., CPU Subsystem, Service Processor Subsystem, etc.). Included also is a glossary of system-specific terms. The PIP will support the following groups:

- prototype build and support Field Service engineers
- development and support services engineers
- newly assigned technical writers and course developers
- other Digital personnel with a need to know.

In addition, the PIP will form the basis for the System Description Reference Manual, and outline material for the other reference manuals.

A Vector Accelerator (VBox) description has not been included in this revision of the PIP. The VBox description will be distributed as a supplement prior to the prototype power on milestone. The supplement will also include vector processing concepts.

1
System Introduction and Overview

1.1 Chapter Objective

This chapter provides an introduction and high-level overview of the basic functions of each major subsystem of the AQUARIUS and ARIDUS systems. The chapter content is based on the AQUARIUS Family Description and introductory material from the Engineering specifications.

NOTE
Unless otherwise specified, the term SYSTEM will refer to both the AQUARIUS and ARIDUS systems.

1.2 System Introduction

The AQUARIUS and ARIDUS systems provide a family of high performance VAX systems. The systems provides a range of CPU performance from 21 VUPs to greater than 100 VUPs.

NOTE
A VUP (VAX Unit of Processing) is equivalent to the performance of one VAX-11/780. Thus, 21 VUPs is the equivalent performance of 21 VAX-11/780s.

The family has two members, with each member having several system configurations:

o AQUARIUS is the top-of-the-line system with emphasis on performance, availability, and reliability. It is liquid cooled and provides expansion capabilities up to a quad-CPU configuration. The smallest configuration is a scalar processor with a performance of 30 MIPS. The quad-CPU configuration provides performance in excess of 100 MIPS.

o ARIDUS is the entry-level system in terms of performance, cost, and footprint. It is air cooled and provides expansion to a dual-CPU configuration. The uniprocessor configuration provides performance of 21 MIPS, with the dual-CPU configuration providing 38 MIPS.

All systems use a System Control Unit (SCU) to interconnect single or multiple processors, memory, and I/O subsystems. The I/O subsystem uses the XMI bus to support the corporate interconnect architectures: BI, CI, and NI. Figure 1-1 presents the basic system layout. Figures 1-2 thru 1-4 describe the component complement for both systems.

Both systems support VMS, which includes symmetric multiprocessing functionality (SMP), as their primary operating system. Also supported are the ULTRIX Operating System ,and the VAXELN Operating System on the Service Processor (console) Subsystem.

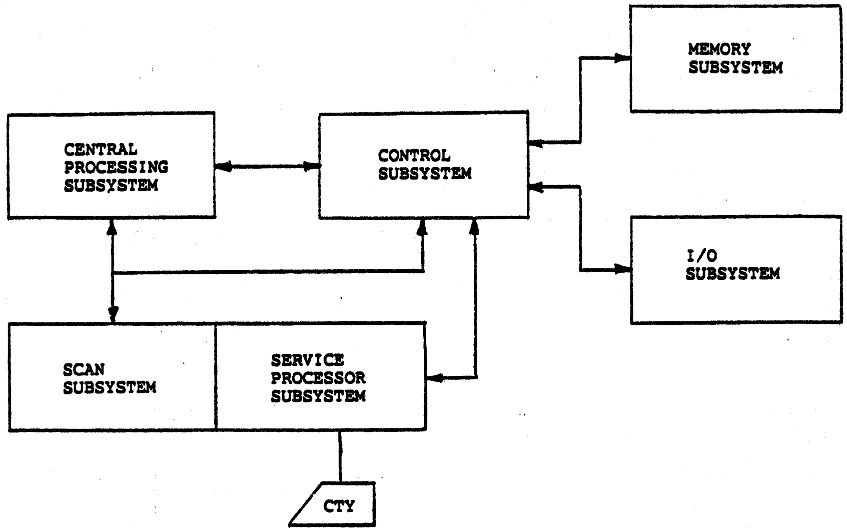


Figure 1-1 Basic System Layout

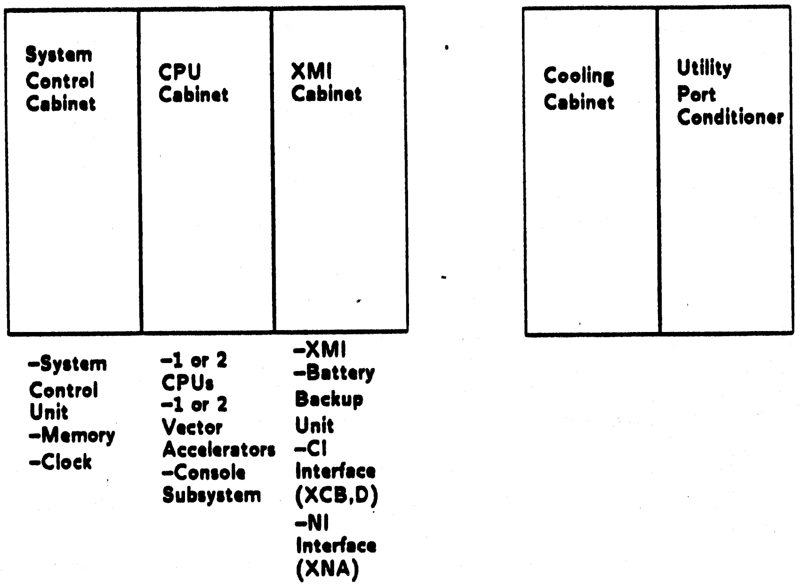


Figure 1-2 Minimum AQUARIUS System Complement

1.2.1 Common System Technologies

All systems use the same basic technology for CPU hardware implementation:

- o Third generation ECL gate arrays - Macro Cell Arrays (MCA-III), which provide high-speed logic gates in dense packaging to minimize interconnect delays
- o Self-timed Random Access Memory devices (STRAMs), which reduce RAM access time and the impact on clock skew
- o Advanced multichip packaging (Multichip Unit - MCU), which house MCA-III gate arrays and/or STRAMs

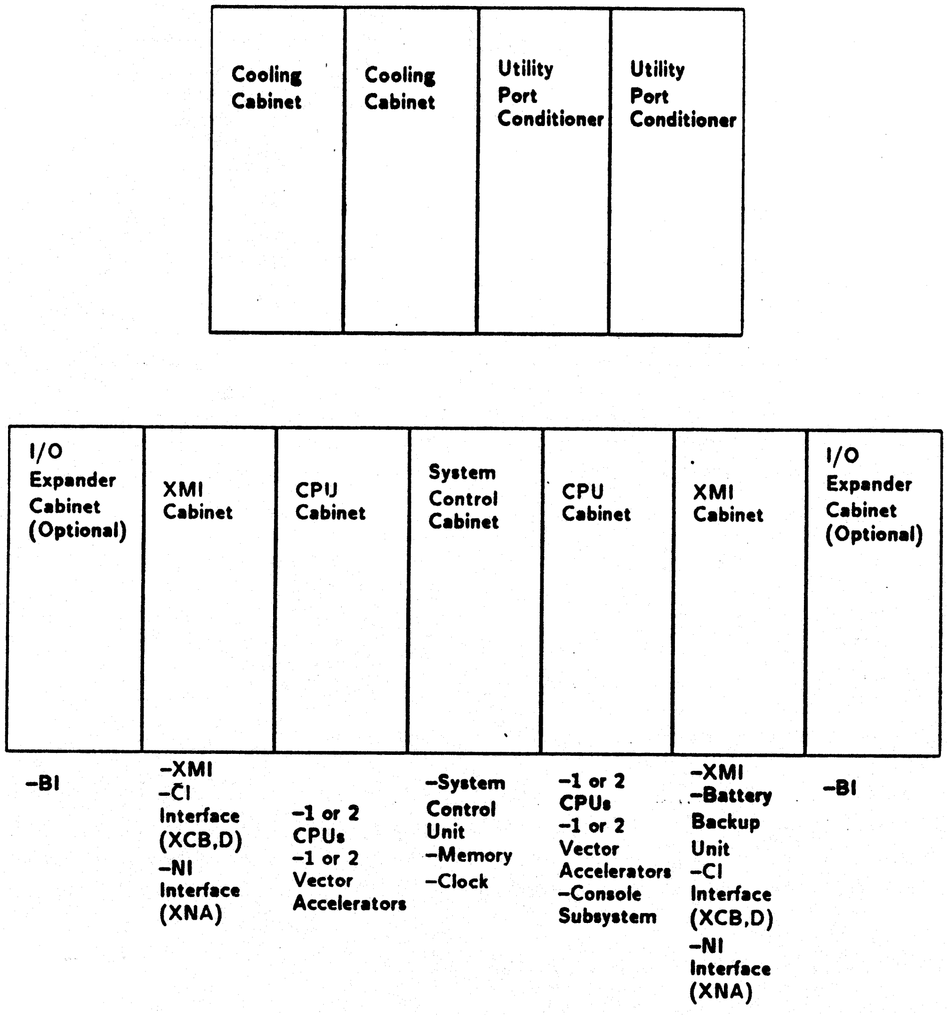


Figure 1-3 Maximum AQUARIUS System Complement

- o High Density Signal Carrier (HDSC), provides the MCU with a 16-layer (signal, ground, and power planes) integrated circuit interconnect
- o Advanced Printed Wire Board (PWB), a vertically mounted 26-layer (signal, ground, and power planes) planar module

The logic is implemented in ECL gate arrays (MCA-III) which provides approximately six to eight times the density of MCA-I devices (as used on the VAX 86XX). Memories are constructed from high performance silicon bipolar RAMs that incorporate a Digital-designed, self-timing mechanism.

The minimum configuration for each system requires a single scalar processor. Each processor is implemented on a single 24-inch by 24-inch planar module. The planar module carries up to 16 MCUs. The scalar processor occupies 13 MCUs.

A high-performance vector processing accelerator option is offered for each system. The accelerator hardware can be added to the scalar CPU planar module using the three remaining (open) MCU postions.

The MCU incorporates an HDSC fabricated from polyimide-copper and provides very short, fast interconnects. The HDSC provides interconnects for up to eight MCA-IIIs, 48 STRAMs, or a combination of gate arrays and STRAMs.

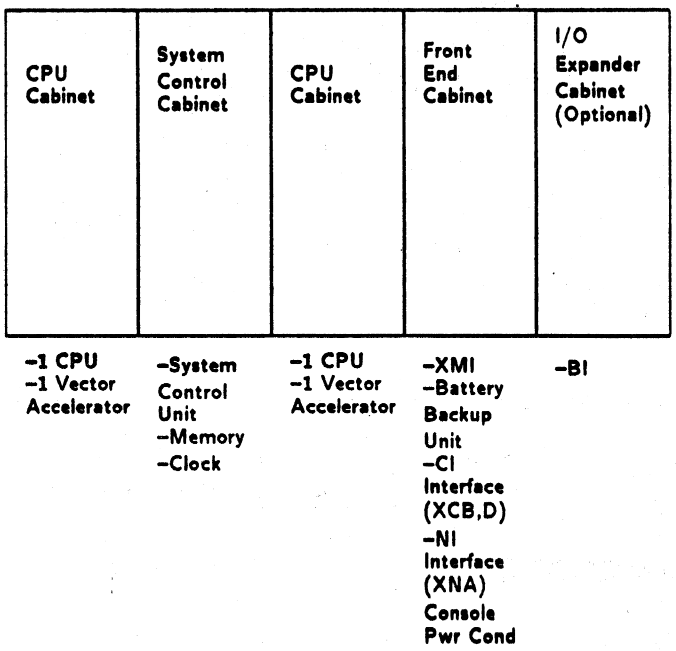


Figure 1–4 Maximum ARIDUS System Complement

The System Control Unit (SCU) is implemented on a single 24-inch by 18-inch planar module containing four or six MCUs, depending on system configuration. As with the processor module, the MCUs can contain MCA-IIIs, STRAMs, or a combination of both. The main memory arrays are integrated into the front of the SCU planar module.

1.3 Performance Characteristics

High performance is accomplished in the system by using several levels of parallelism. At the system level there is tightly coupled multiprocessing. Through problem decomposition, up to four processors can work on a task and communicate efficiently through shared memory.

Multiple I/O busses provide parallel paths to mass storage and other devices, resulting in high speed, extensive connectivity, and redundancy. Performance is further enhanced by using quadword (8 byte) wide data interfaces at critical points throughout the system.

Parallelism is accomplished in the processor by pipelining, that is, separating the execution of a VAX macro instruction into small individual operations. All VAX systems cycle through a similar set of operations during macro instruction execution. The execution operations include instruction fetch, instruction decode, operand fetch, instruction execution, and result store.

These individual operations are then performed by dedicated and independent functional units that have been optimized for that particular operation. The objective is to maintain the computing resources as busy as possible, with minimal wait time while information is being retrieved from memory or the I/O subsystem.

As a result, the execution operations can be overlapped, increasing total instruction throughput. Figure 1-5 compares the relative instruction time of a non-pipelined processor and a pipelined processor.

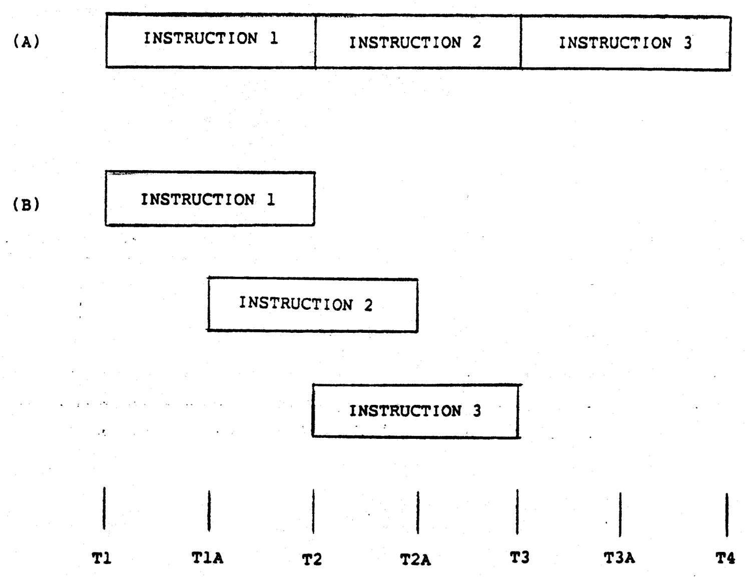


Figure 1-5 Sequential/Pipelined Instruction Execution

The system has hardware to detect and resolve most pipeline hazards (i.e., those conditions that interrupt normal pipeline operation). The pipelines are relatively short, helping to reduce conflicts (stalls) such as when:

- the execution unit (EBox) ALUs require multiple cycles for executing floating point instructions, integer multiplication, and division. Thus, instructions requiring the results of those type instructions cannot be issued until the previous one is complete.
- if one instruction writes to a memory location or general purpose register (GPR) and a following instruction attempts to read that location, the read is stalled until the write is complete.
- if the result of one instruction is to be used to form an address for a subsequent instruction, the address cannot be formed until that result is available.

For reasons of speed and signal integrity, the majority of connections in the systems have single sources, and all destinations are on a single substrate. Unlike other VAXes, there are no internal busses in the systems.

1.4 Basic System Configuration

The systems are organized into several major subsystems:

- CPU Subsystem
- Clock Subsystem
- Service Processor and SCAN Subsystems
- System Control Subsystem
- Main Memory Subsystem
- I/O Subsystem
- Power Subsystem

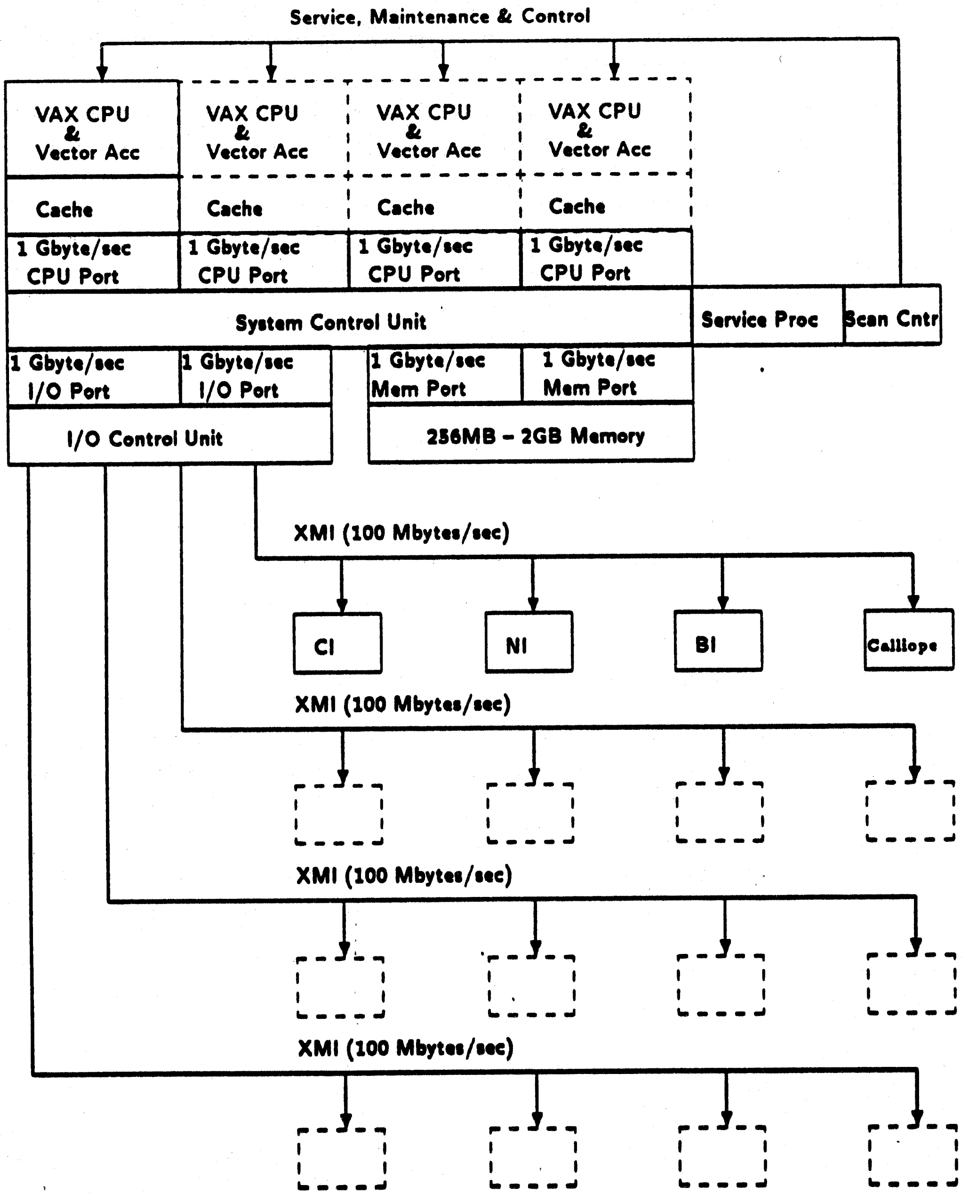


Figure 1–6 AQUARIUS Logical Layout

Figures 1-6 and 1-7 describe the logical layouts of each system.

1.4.1 CPU Subsystem

The CPU subsystem is partitioned into four functional units: MBox, IBox, EBox, and VBox. Much of the power of the CPU comes from the ability of the functional units to operate in parallel. Figure 1-8 illustrates the independent functional units of the CPU, as well as the SCU.

- IBox - prefetches instructions, decodes opcodes and operand specifiers, fetches operands, and updates the program counter. With additional features included in the IBox, more parallel processing is achieved together with a lower number of cycles per instruction (CPI).

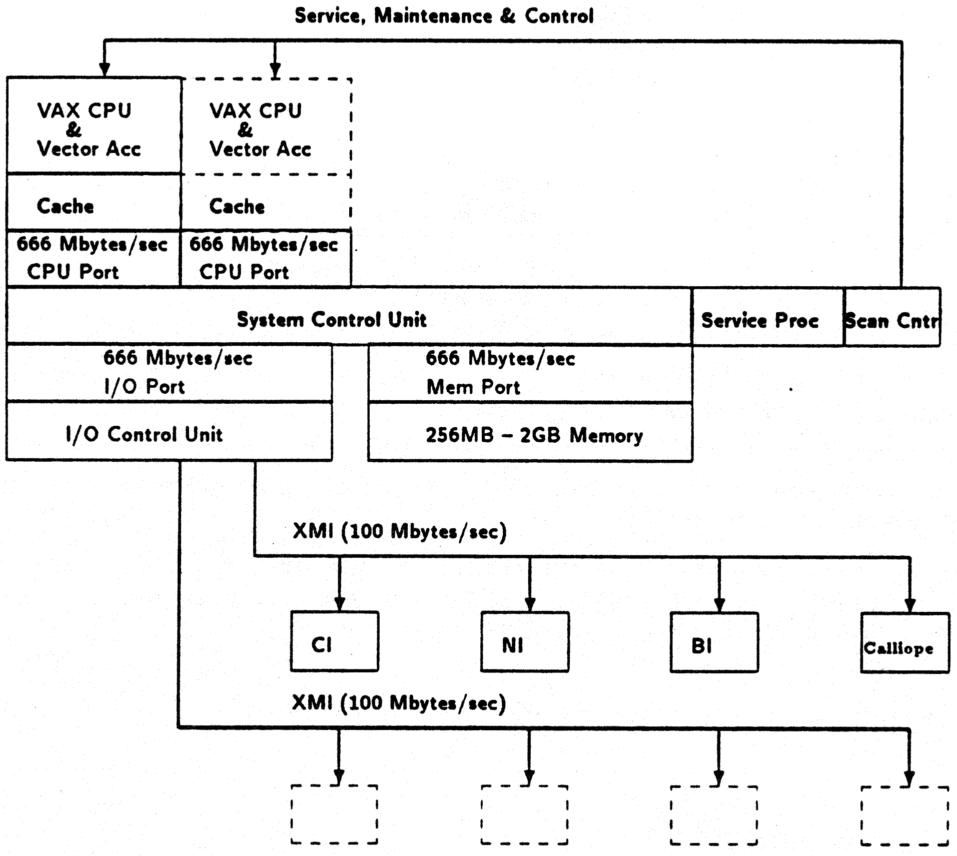


Figure 1–7 ARIDUS Logical Layout

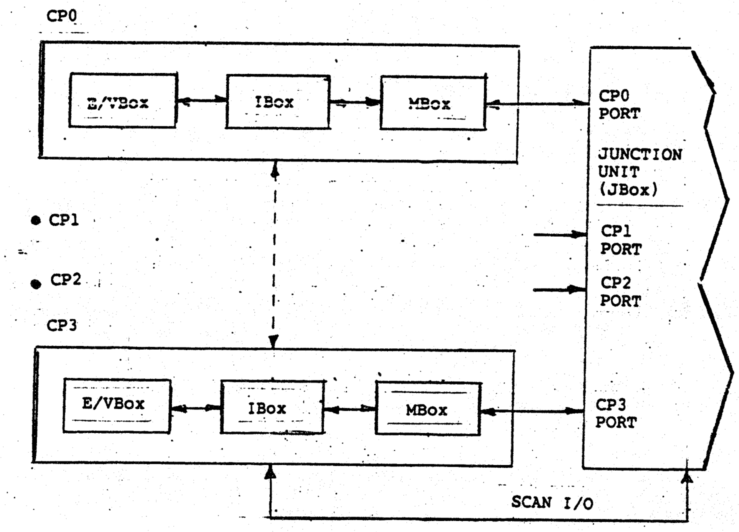


Figure 1–8 Basic CPU Subsystem Block Diagram

The IBox is equipped with several new pipelined functions, for example virtual instruction cache, branch prediction, and multiple operand specifier decoding. A number of complex functions normally found in other VAX EBoxes have been removed and implemented in the IBox.

- MBox - provides the CPU interface to the SCU and subsequently to main memory, I/O and other processors (in a multiprocessor configuration). It is equipped with a Translation Buffer (TB) and a large data cache. In addition, a new feature - TB Fix-up Unit - has been implemented. Its major function is to resolve TB misses early in the pipeline.

  The MBox accepts memory references - usually virtual - translates the addresses to physical, and accesses the memory data either in main memory or in its data cache.

- EBox - serves as the CPU execution unit, accepts instruction source and destination data from the I and M boxes and provides integer, floating point, multiply, and divide operations. The EBox contains four processing units capable of parallel operation.

  Result data is passed to an internal GPR, or to the MBox for subsequent transfer to the data cache, memory, or I/O.

- VBox - provides an optional vector accelerator contained in three MCUs of the CPU planar module. The unit accepts instruction source and destination data from the IBox through the 32-bit load path of the EBox.

  The VBox can produce a double precision result on every clock cycle. Result data is passed to the MBox through the 64-bit EBox result data path.

1.4.2 Clock Subsystem

The Clock Subsystem includes the generation and distribution of all system and control clock signals throughout the CPU, memory, and I/O subsystems. Except for the difference in clock frequencies, both AQUARIUS and ARIDUS use the same clock components. Additional copies of the clock signals are required for a quad processor configuration.

The clock signals originate on the Master Clock Module (MCM) located in the SCU cabinet. The basis of the MCM is a phase-locked, voltage controlled oscillator (VCO). The output of the VCO is fed back to its input through its control logic to automatically regulate its frequency using a phase-locked loop technique.

The VCO provides highly stable frequency outputs and is capable of changing its frequency up to 4 MHz steps over the following ranges:

- AQUARIUS: 352 to 600 MHz, with an operating frequency of 500 MHz
- ARIDUS: 240 to 424 MHZ, with an operating frequency of 352 MHz

The clock subsystem is comprised of the following major components:

- VCO oscillator: basic clock oscillator
- Power amplifiers: amplifies and shapes all clocks (i.e., main reference, and control clocks)
- MCM Power divider: act as power splitter and provides the clock signal fanouts for distribution to the CPU and SCU power dividers
- Semiflexible coax cable: provides the clock transmission lines between the SCU and CPU cabinets

- o CPU and SCU power dividers: provides the same basic functions as the MCM power dividers, however, these dividers provide clock signal fanout to the individual MCU rows on the CPU and SCU planar modules.

The system requires eight clock edges to define one machine cycle. The eight clock edges are derived from the rising edges of the VCO differential sine wave output. In addition, a clock reference signal is generated at a frequency equal to 1/8 of the clock frequency. The reference clock defines the beginning of a machine cycle, and provides overall synchronization.

A clock control signal - which can be controlled by the SPU - is generated from the reference clock also at a frequency 1/8 of the clock frequency. The clock control signal is distributed to the Clock Distribution Chips (CDC) located on the CPU and SCU MCUs. Its major function is to determine if clock signals are generated for the current machine cycle.

1.4.3 Service Processor and SCAN Subsystems

The Service Processor Unit (SPU) provides two major functions:

- o system initialization controller and operator interface
- o service and maintenance processor

The SPU provides the traditional console functions. As a service processor, the SPU provides error processing, including data collection, error correlation, and fault recovery. A fault manager is the focal point for this activity, providing automatic system configuration, security, and automatic call logging.

All remote service features are supported, including remote diagnosis, hardware monitoring, performance analysis, and patch distribution.

The SPU uses the BI form factor. It contains 16 Mbytes of memory, and is installed in a BI backplane located in the SCU cabinet. As shown in Figure 1-9 the SPU consists of:

- MicroVAX-driven Servcie Processor Module (SPM)
- MicroVAX-driven SCAN controller module
- KFBTA disk controller supporting an RD54 disk drive
- DEBNT tape controller supporting one TK50 cartridge tape drive

In addition, the Power and Environmental Monitoring module (PEM) interfaces to the SPM. The PEM provides the communications link to the power and environmental subsystems.

The SPU has access to all SCAN storage elements in the CPUs, SCU, and MCM through the SCAN Subsystem. The subsystem interfaces to the SPU through the SCAN Controller Module (SCM) The SCAN subsystem data paths are used for testing, system initialization, reading error status, and error correction.

A single SPU has the capability to handle up to four CPUs. The CPUs communicate with the SPU by reading and writing SCU registers. There is a single copy of the standard RXCS, TXCS, RXDB, and TXDB, and TOY registers. They are accessible to all CPUs. In addition, there are four DMA registers for directly accessing main memory.

The SPU will run under the VAXELN Operating System. VAXELN is a real time operating system and forms a memory-resident kernel. It allows the inclusion of SPU functions in the operating system rather than executing those functions as an application program. This feature provides a consistent functionality across the various console modes.

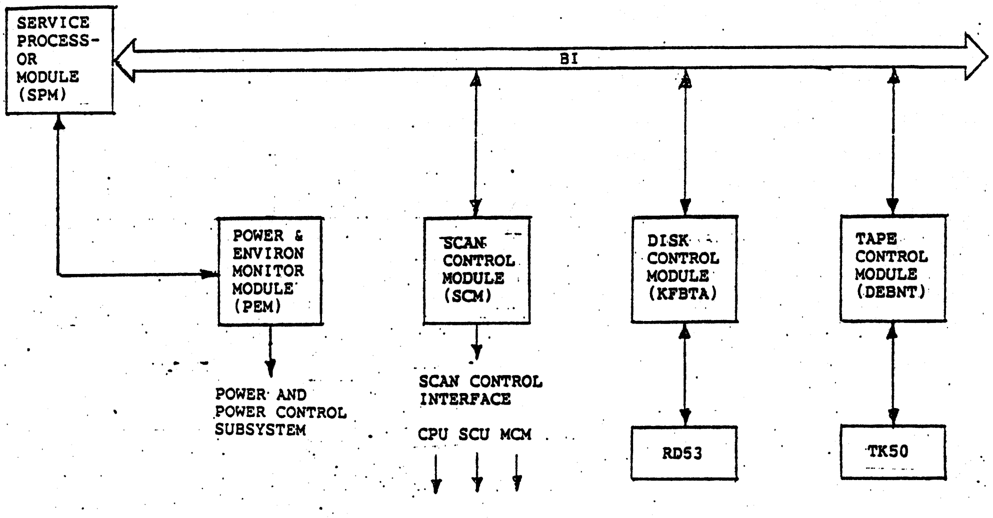

SERVICE PROCESS-OR MODULE (SPM)
BI
POWER & ENVIRON MONITOR MODULE (PEM)
SCAN CONTROL MODULE (SCM)
DISK CONTROL MODULE (KFBTA)
TAPE CONTROL MODULE (DEBNT)
POWER AND POWER CONTROL SUBSYSTEM
SCAN CONTROL INTERFACE
CPU SCU MCM
RD53
TK50

Figure 1–9 Basic SPU Subsystem Block Diagram

VAXELN provides memory management (i.e., P0/P1 space mapping), with no paging or swapping. The lack of page and swap files increases subsystem reliability and performance by removing the dependency on disk drives. In addition, it also supports a VMS-compatible file system, and DECNET end node network service.

1.4.4 System Control Subsystem

The system control subsystem consists of the SCU which interconnects the CPU, I/O subsystem, main memory, and SPU. The unit initially handles all I/O device, SPU, and inter-CPU interrupt and exception conditions.

As shown in Figure 1-10, the SCU is partitioned into three major functional units:

- o Junction Box (JBox): provides up to four ports which interface up to four CPU subsystems
- o I/O Control Unit (ICU): provides up to four ports which interface to the I/O system bus and bus adapters
- o Array Control Unit (ACU): provides up to two ports which interface to the main memory arrays.

The SCU has the capability of maintaining all 10 functional unit ports active. However, the port activity requires extensive communication, validity checking, and parallel operations. The SCU contains two, unidirectional, crossbar interconnects between each port (i.e., one interconnect/direction). The crossbar allows simultaneous transactions between ports - if there are no conflicts.

Another major SCU function is to manage memory access. Since memory data is distributed in the CPU caches, the data in main memory may not be valid. The SCU tracks the location of valid data through a cache consistency unit, and insures that a memory port request results in a valid read or write operation.

Interrupts from the I/O devices, SPU, and between CPUs are distributed by the SCU. Except for inter-CPU interrupts, they are distributed in either round-robin fashion, or directed to a single CPU.

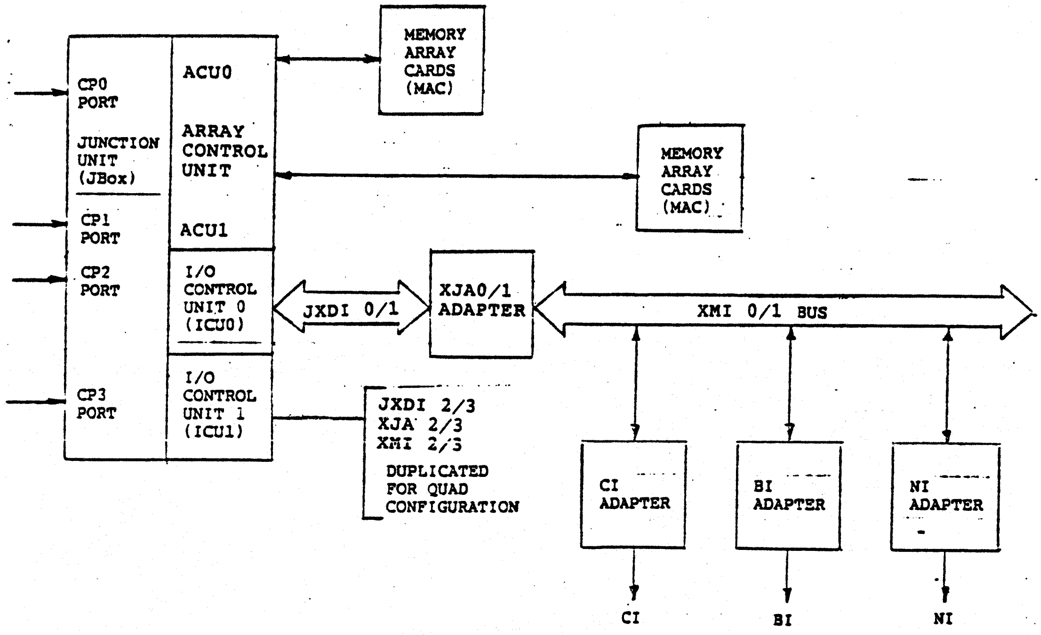


Figure 1-10 Basic SCU Subsystem Block Diagram

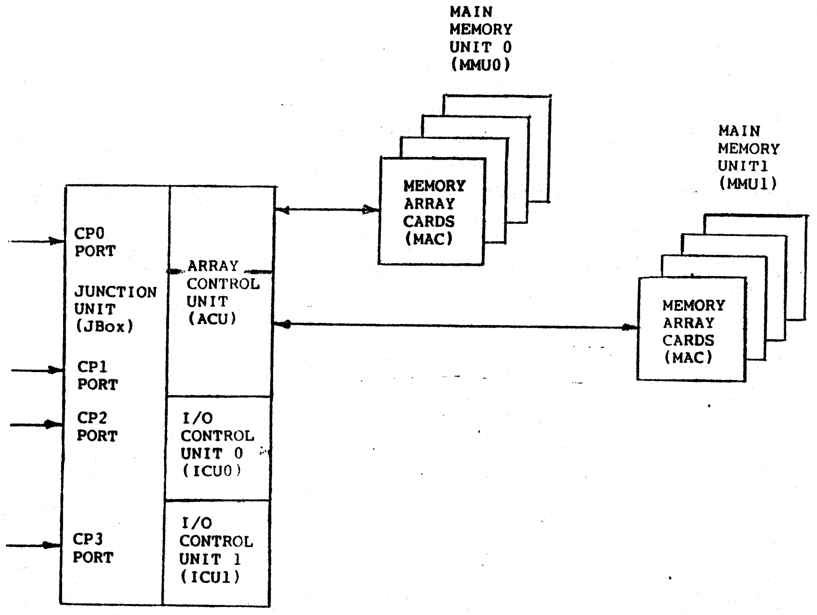


Figure 1-11 Basic Main Memory Block Diagram

The SCU connects the SPU to the remainder of the system. Communication is provided through SCU registers which are assigned I/O space addresses. SCU registers are also used to configure memory and I/O, and for interrupt and exception status.

1.4.5 Main Memory Subsystem

As shown in Figure 1-11, the main memory subsystem is driven through the two ports of the ACU. Each ACU port supports one MMU, which consists of four Memory Array Cards (MACs). A maximum of eight MACs can be installed in the front of the SCU planar module.

Each MAC contains 32 Mbytes of memory using 1 Mbit dynamic RAMs (DRAMS). The DRAMS are surface mounted on conventional extended hex modules. In addition, two, 16 Mbyte Daughter Array Cards (DACs) can be installed on each MAC, for a total array card capacity of 64 Mbytes.

The memory structure consists of a data path, DRAMs, and address and control input. There are two, 8-byte (quadword) wide data paths for each MMU, where one data path handles read data, the other handles write data. Included in each data path are the ECC bits (i.e., 64 data bits and 14 ECC bits).

Memory data is stored with ECC on a longword basis. The ECC bits provide double error detection and single bit correction. Write operations of less than a longword require a read-modify-write operation. Because of the MBox write-back cache, this occurs only on byte and word write operations from I/O or a CPU. ECC is generated and checked in the SCU.

In addition, the memory subsystem supports 2-way, and 4-way interleaving. The current maximum main memory capacity for AQUARIUS and ARIDUS is 512 Mbytes.

1.4.6 I/O Subsystem Introduction

As shown in Figure 1-12, the I/O subsystem consists of five functional units:

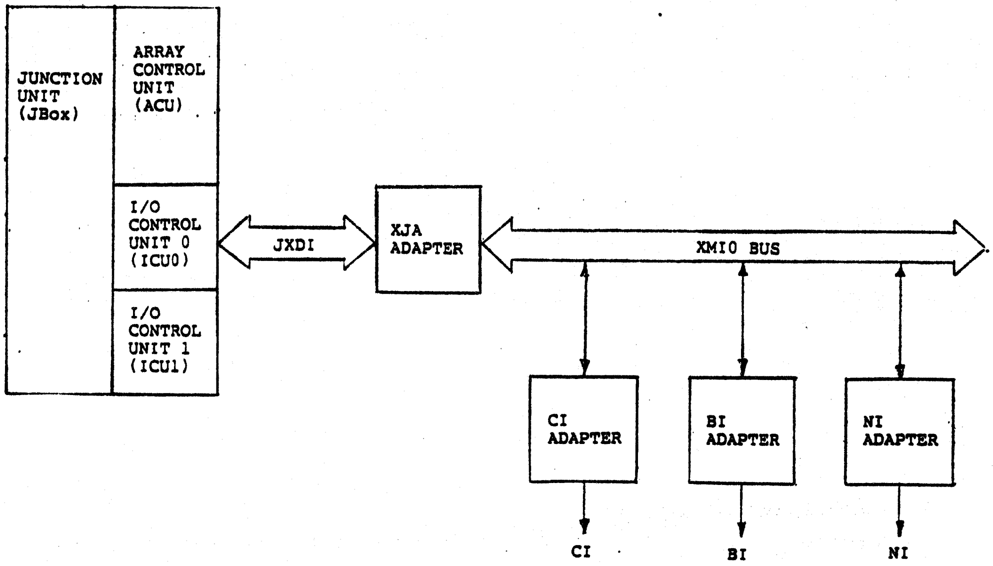


Figure 1–12 Basic I/O Subsystem Block Diagram

ICU - located in the SCU cabinet and provides the two, 16-bit data ports to the I/O subsystem. It multiplexes a maximum of four XJAs - through the associated JXDI interfaces - to the XMIs.

JXDI - provides the data interconnect between the ICU and XJA. Physically it is a 12-foot cable between the SCU cabinet (ICU) and the XMI cabinet (XJA). All signal lines are unidirectional and differential. Data transfers are asynchronous with clocks supplied by the transmitter at a cycle time equal to the CPU clock cycle. The JXDI is symmetrical in that the same data and handshake signals are sent and received by both the ICU and XJA.

XJA - ICU to XMI bus adapter provides the interface between the ICU - through the JXDI interface - to the XMI bus. The XJA module resides in the XMI card cage located in the XMI cabinet.

XMI - the XMI is the I/O subsystem bus, and is a limited length, pended, synchronous bus with centralized arbitration. It provides a 64-bit data path, with a 64 nsec bus cycle time. Several transactions can be in progress at a given time. Arbitration and data transfers occur simultaneously using multiplexed address and data lines.

The remaining functional units consist of various XMI and VAXBI adapters.

The systems support first- and second-generation I/O adapters and devices. This provides compatibility with existing VAXcluster and Ethernet based systems while providing connectability to XI-based systems. The differences between the I/O capabilities of each system is specified in Table 1-1.

Table 1–1 System I/O Differences

PARAMETER	AQUARIUS	ARIDUS
I/O Port	2	1
I/O Port Bandwidth	1 Gbytes/sec	666 Mbytes/sec
XMI Interfaces Supported	4	2
Maximum BIs Supported	14	8
Maximum CIs Supported	16	16

Table 1-2 lists the supported interconnect adapters.

Table 1–2 Supported Interconnect Adapters

INTERCONNECT	ADAPTER DESCRIPTION
CI	XCB/XCD (XMI - CI Adapter)
NI	XNA (XMI - NI Adapter)
BI	XBI (XMI - BI Adapter)
SI	HSX (XMI - SI Adapter)

Table 1-3 lists the supported BI devices.

Table 1–3 Supported BI Devices

DEVICE	DESCRIPTION
KDB50	Disk adapter (4 disks)
TU81E	Directly connected tape subsystem
DMB32	8-line asynchronous multiplexer printer port synchronous port
DRB32	Medium speed 32-bit parallel
DEBNA	Ethernet adapter
BCA	CI adapter
DHB32	16 Asynchronous lines
DSB32	2 Synchronous lines

Table 1-4 lists the primary supported storage devices.

Table 1–4 Supported Storage Devices

DEVICE	DESCRIPTION
RA90	1.2 Gbyte Winchester disk
RA70	280 Mbyte Winchester
TA90	IBM 3480-compatble cartridge magtape backup subsystem

1.4.7 Power Subsystems

The AQUARIUS and ARIDUS systems share the majority of the same power subsystem components. The differences are highlighted in the following paragraphs.

The 3-phase ac facility power is individually supplied to the: XMI cabinet, BI expansion cabinets, and the Water Cooling Unit. As shown in Figure 1-13, the power subsystem consists of the following elements:

- Utility Port Conditioner (UPC) - AQUARIUS
- Power Control Unit (PCU) - ARIDUS
- Ac-to-Dc Converter - ARIDUS
- Dc-to-Dc converters
- Power Control Subsystem (PCS)
- Regulator Intelligence Cards (RICs)
- Power and Environmental Monitoring module (PEM)

- Battery backup units (BBU)

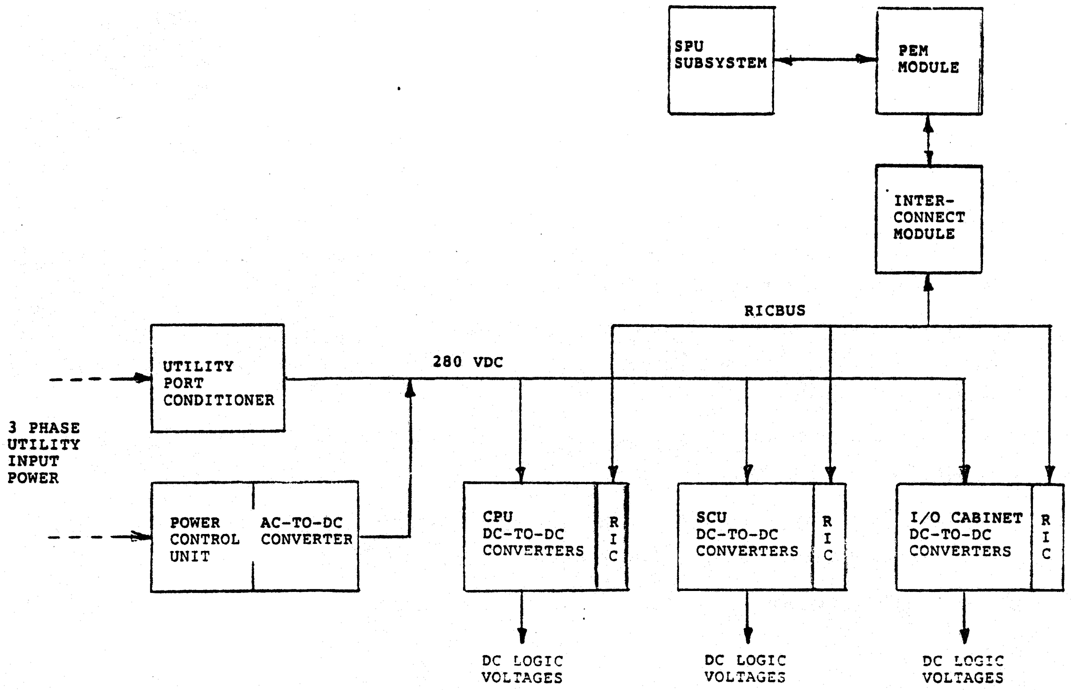


Figure 1-13 Basic Power Subsystem Block Diagram

The optional UPC is a free-standing device used to convert the 3-phase ac utility input to an isolated, regulated 280 vdc output. The UPC will maintain a 90% power factor, and harmonic current control. It will also provide inrush current limiting. The UPC supplies 280 vdc to two sets of dc-to-dc converters in the CPU and SCU cabinets. The power control unit within the UPC provides dc power to CPU and SCU air movers. Note that the air movers have an integral dc-to-ac converter.

Since the UPC is optional for both systems, a Power Control Unit (PCU) and an Ac-to-Dc Converter are required in place of the UPC. The PCU distributes ac power to the ac-to-dc converter which in turn supplies 280 vdc to the dc-to-dc converters and air movers in the CPU and SCU cabinets (similar to the UPC).

The CPU logic has ten, 240-amp converter modules. Five converter modules supply -5.2 vdc, while the remaining five modules supply -3.4 vdc. Each set constitutes a regulator group. However, only four converters in each group are required to supply the entire load for each group. The fifth converter in each group provides N+1 redundancy for added reliability.

Dc power for the SCU and main memory is supplied through a set of 240-amp multiple output dc converters. This group also has N+1 converter redundancy. Power for the XMI cabinet is provided by a standard XMI power supply.

Each converter group is controlled by a Regulator Intelligence Card (RIC). The RIC forms part of the Power Control Subsystem (PCS).

The PCS is a distributed intelligence, data acquisition subsystem, and is responsible for:

- monitoring and controlling dc voltage and current levels
- monitoring the system environmental parameters

o communicating power and environmental status to the SPU.

In addition, the PCS acts as an intelligent peripheral to the SPU. During normal system operation, the PCS receives commands from, and reports status changes to the SPU.

The PCS consists of the following elements:

o Regulator Intelligence Cards (RIC): provide the interface to the power and environmental subsystems.

o Regulator Intelligence Card Bus (RICBUS): provides bidirectional serial control and data communications between the Power Environmental Monitor (PEM) - through the Interconnect Module - and the RICs.

o Interconnect Module: a passive module which serves as a cable connection point between the PEM and the RICBUS. (The module is also referred to as the Signal Interconnect Panel - SIP.)

  The Interconnect Module also provides the PEM interface to the Bias Supply, Operator Control Panel, and BBUs.

o PEM: provides the communications link between the SPU and the power and environmental subsystems. The PEM is also the RICBUS controller.

o Dc-to-dc power converters

o Power regulators

o Operator Control Panel (OCP): provides operator indicators and controls for system status, emergency power off, boot, restart, etc.

1.5 Cooling Subsystems

1.5.1 AQUARIUS Cooling Subsystem

The AQUARIUS system is cooled using a combination of air movers and a water-cooling system. Water cooling is provided by a self-contained Water Cooling Unit (WCU). Water is piped from the WCU to the CPU and SCU planar modules to cool the high-power MCA III gate arrays. Two WCUs are required for quad-CPU, and redundant dual-CPU configurations.

The WCU uses circulation pumps for water flow, and a liquid-to-air heat exchanger. Water is pumped from the WCU through the cold plates bolted to the MCUs. The dissipated MCU heat is transferred to the water and returned to the WCU heat exchanger, where it is then exhausted into the air. Remote sensors are incorporated into the WCU to allow the SPU to monitor the flow, temperature, pressure, and air flow.

The WCU is equipped with redundant (dual) circulation pumps. Since a single pump has enough capacity to provide the required water flow, only one pump is in operation at any one time. To extend pump life, the pumps are switched every 400 hours. In addition, should one pump fail, the remaining pump is automatically switched on.

The memory arrays, SPU, and the power subsystem are cooled by air movers and associated ductwork. The air movers are located directly above the component they cool. Ambient temperature air is taken in through the bottom of the cabinet and directed through the ductwork. It is then passed over the components, and exhausted through the rear of the cabinet .

1.5.2 ARIDUS Cooling Subsystem

The ARIDUS cooling system uses several air movers and associated ductwork. Since the CPU and SCU use low-power MCA III gate arrays, MCU cooling is provided through air-cooled, cold plates bolted directly to the MCUs. The air movers are located above the component they are intended to cool, and provide the following component cooling:

- all MCUs for two CPU planar modules
- SCU MCUs and memory arrays
- dc-to-dc converters in the CPU cabinet
- dc-to-dc converters in the I/O cabinet
- all remaining logic and power in the I/O cabinet.

Ambient temperature air is supplied through the bottom of the CPU and SCU cabinets, directed through the ductwork, and across the components and cold plates. The heated air is then exhausted through the rear of the cabinet. Air flow through the I/O and expansion cabinets follows the same basic path.

1.6 Diagnostic Test Strategies

The diagnostic testing and maintenance strategies are directed at isolating hardware failures using two diagnostic strategies: Test Directed Diagnosis (TDD), and Symptom Directed Diagnosis (SDD).

The system is equipped with a variety of testability features included in its design. These features include SCAN latches in the CPU and SCU, built-in self tests (BIST) in the memory, XMI and BI controllers, power subsystem, and bus loopback. The diagnostics will use these features to provide a high level of fault detection and isolation to a Field Replaceable Unit (FRU).

The system will use TDD, including SCAN pattern generated diagnostics, and BISTs to detect and isolate "stuck at" fault conditions. Intermittent faults will be detected and isolated using SDD. Dynamic faults in the system will be detected by macro-level functional diagnostics and SDD.

1.6.1 Test Directed Diagnosis

TDD represents the traditional diagnostic approach to fault isolation. TDD uses a number of micro and macro test routines at various levels of complexity to exercise the hardware to reproduce a fault condition. Once the fault has been reproduced and recognized by the test routine, an FRU or function callout can be produced.

Generally, a bottom up and building block approach is used in TDD testing. This approach allows previously tested logic to be used as an aid in testing and isolating logic faults at a higher level.

The TDD diagnostic programs and routines are used to verify proper machine operation starting with the SPU and SCAN path, progressing through the CPU and SCU logic, macro instruction execution, and I/O subsystem interaction. TDD also includes hardware-based self test routines which execute during system power-up and initialization. Typically, these diagnostics would be used for acceptance testing during initial installation or system upgrade.

TDD routines will also be selectively called to provide repair verification testing after FRU replacement. In addition, in those cases where SDD cannot provide FRU callout the TTD diagnostics will be used for fault isolation.

1.6.2 Symptom Directed Diagnosis

SDD uses routines to analyze fault symptom data collected at the time of the error event. The routines analyze the collected symptom data and produce a particular fault syndrome. The syndrome is then correlated to a particular FRU or component.

Given the high gate densities of the CPU and SCU logic, the predominant failure mode will be intermittent in nature. Thus, the Field Service fault isolation strategy will focus on SDD techniques. In addition to using the dynamic fault detection circuits strategically placed throughout the CPU and SCU, SDD will use the capabilities of the SPU to:

- capture and record the state of the CPU and SCU hardware at the time of error
- execute SDD routines capable of providing an error syndrome based on analysis of the error symptom data collected at the time of the fault
- select and execute TDD routines based on symptom and error syndrome code for purposes of repair verification.

1.6.3 Diagnostic Test Techniques

The system will use several distinct diagnostic techniques for detection and isolation. The general diagnostic techniques and fault coverage goals are:

- Built-in Self Test: used for quick verification of devices when power is applied and used to isolate faulty devices.

  Fault coverage of 95% - 99% depending on the unit under test, with an average isolation to 1.5 FRUs.

- SCAN Patterns: used for quick verification when power is applied and to isolate hard logic failures.

  Greater than 98% coverage depending on the unit under test, with an average isolation to 1.1 MCUs and two components within an MCU.

- Standalone Macro-coded Diagnostics (Level 4): used for quick verification of the XJA and XBI devices when power is applied. Also used for VAX Macrohardcore instruction test prior to executing the VAX Diagnostic Supervisor (VDS).

  Fault coverage and isolation to be supplied.

- Diagnostics that execute under VDS in a standalone environment (Level 3): used for functional verification of device operation when the system is installed or a when a device is replaced.

  Fault coverage of 95% depending on the unit under test, with an average isolation to 1.5 MCUs.

- Diagnostics that execute under VDS in a VMS user environment (Levels 2 and 2R): used for functional verification of device operation in the VMS environment.

  Fault coverage and isolation to be supplied.

- Diagnostics that run in a VMS user environment without the support of the VDS (Level 1): used to simulate the operation of the system in a user application environment, and to isolate XJA and XBI failures in a VMS environment.

  Fault coverage and isolation to be supplied.

- Symptom Directed Diagnosis (SDD): used to isolate hardware detected faults that occur during normal system operation.

  Fault coverage of 85% - 90% depending on the unit under test, with an average isolation to 1.2 MCUs.

1.7 System Configurations

1.7.1 AQUARIUS Kernel Configuration

The kernel configuration is specified in Table 1-5.

Table 1–5 AQUARIUS Kernel Configuration

SYSTEM CABINETS	CABINET CONTENT
CPU	CPU planar module, including space for VBox; power regulators; SPU including RD54, TK50, and SCAN controller
SCU	SCU planar module, including I/O and memory control; memory arrays; clock subsystem; power regulators
XMI	XMI card cages; power subsystem; Battery Backup Units; Ac-to-Dc Converter and Power Control Unit - if UPC is not installed; CI and NI adapters

The options available are:

- a second CPU for expansion to a dual processor configuration
- a second CPU cabinet for expansion up to a quad processor configuration
- vector accelerator (VBox)
- UPC
- transformer for non-North American utility power

1.7.2 AQUARIUS Configuration Limits

The AQUARIUS system has the following configuration limits:

- maximum of four CPUs
- one VBox per CPU
- 256 Mbytes to 512 Mbytes main memory
- maximum of four XMIs
- maximum of 14 BIs
- maximum of 16 CI interfaces (XCB/XCD)

1.7.3 ARIDUS Kernel Configuration

The ARIDUS kernel configuration is specified in Table 1-6.

Table 1–6 ARIDUS Kernel Configuration

SYSTEM CABINETS	CABINET CONTENT
CPU	CPU planar module, including space for VBox; power regulators; SPU including RD54, TK50, and SCAN controller
SCU	SCU planar module, including I/O and memory control; memory arrays; clock subsystem; power regulators
XMI	one XMI card cage; power subsystem; BBUs, Ac to Dc Converter - if no UPC installed; CI and NI adapters

The options available are:

- o a second CPU, with cabinet, for expansion to a dual processor configuration
- o vector accelerator
- o Utility Port Conditioner

1.7.4 ARIDUS Configuration Limits

The ARIDUS system has the following configuration limits:

- o maximum of two CPUs, with one CPU per cabinet
- o one VBox per CPU
- o 256 Mbytes to 512 Mbytes of memory
- o maximum of two XMIs
- o maximum of eight BIs
- o maximum of 16 CI interfaces (XCA)

2
Technology and Packaging Descriptions

2.1 Chapter Objective

This chapter will introduce and describe the new system technologies, including system packaging and cooling. The system and I\O cabinet configurations will also be described including cabinet contents and major component locations.

2.2 Technology Overview

The systems use the same basic technology for the CPU, VPU, and SCU implementations:

- third-generation ECL MCA-IIIs
- high density signal carrier (HDSC) integrated circuit interconnect
- advanced Multichip (MCU) packaging
- advanced printed wire board planar module
- self-timed RAMs (STRAMs)
- self-timed registers (STREGs)

2.2.1 Macro Cell Array (MCA-III)

The MCA-III (MCA) is the basic logic building block for the CPU, VBox, and SCU. It is implemented in third generation ECL gate arrays, with approximately eight times the density of MCA-I devices.

2.2.1.1 Physical and Functional Structure

Each MCA has a 360-pin array, with 200 output cells, 224 input cells, two clock driver cells, and 256 I/O ports. The MCA occupies a die size of 385 x 385 mils, and has a floor plan as illustrated in Figure 2–1.

Each MCA has 414 major cells, where each cell has 76 transistors and 76 resistors. Major cells can be subdivided into quarter cells, producing a total of 1656 quarter cells. A detail MCA layout is illustrated in Figure 2–2

Equivalent gate count depends on the type of logic implemented. For example:

- over 10,000 equivalent gates are achieved if full adders, output latches, and input ORs are implemented in all cells,

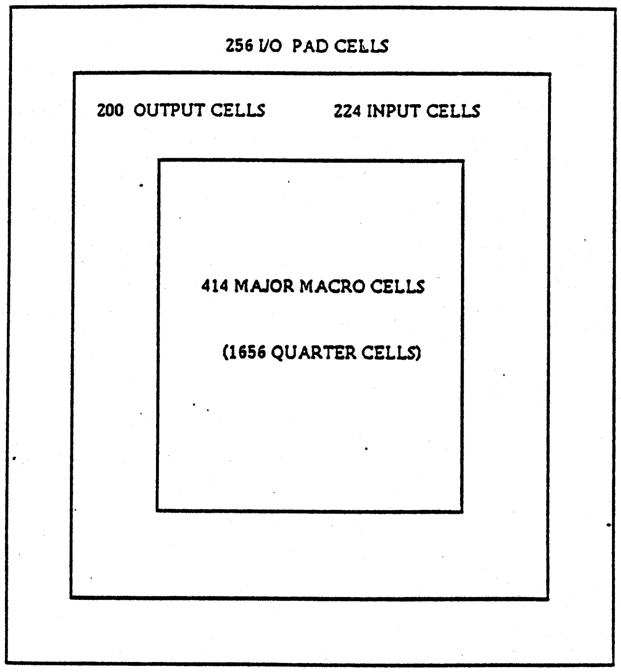


Figure 2–1 MCA-III Floor Plan

- over 7000 equivalent gates are achieved if flip-flops, output latches, and input ORs are used in all cells.

Depending on the clock speed, an MCA will dissipate approximately 15 or 30 watts.

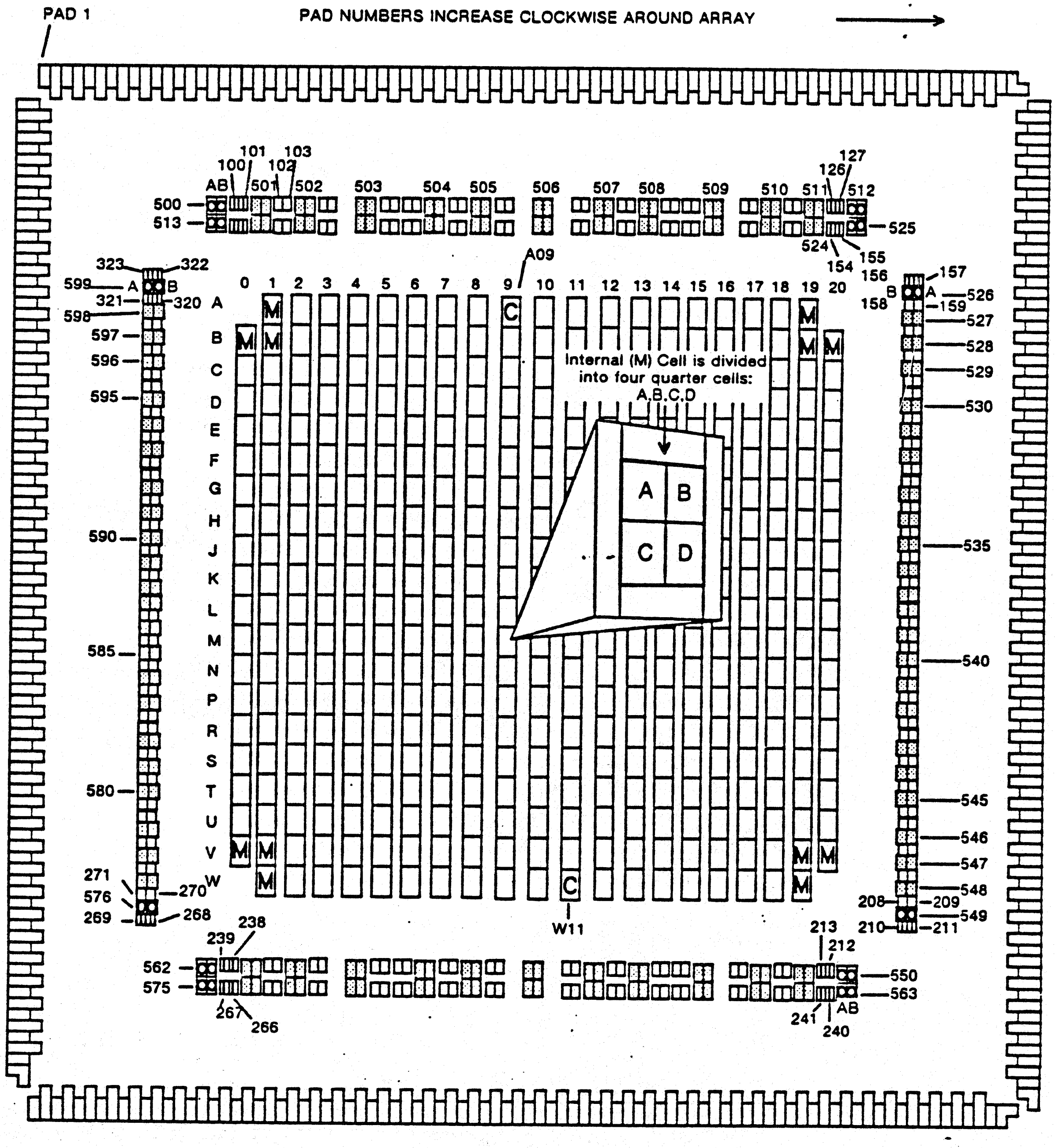
2.2.2 Multichip Unit

The MCAs and STRAMs are assembled in a Multichip Unit (MCU), together with one Clock Distribution Chip (CDC) per MCU. The MCU is the CPU and SCU field replaceable unit (FRU). The MCU is four inches square and can accommodate eight MCAs, 48 STRAMs, or a combination of both.

As shown in Figure 2-3, the MCU is a complicated and finely machined assembly. The high density signal carrier (HDSC) consists of nine layers and provides the I/O, signal, and power interconnects for all MCU components. It consists of three components: base plate, and power and signal cores.

The base plate is chrome copper and is fastened to the cold plate to dissipate MCU component heat into the cooling subsystem. As shown in Figure 2-4 the signal and power cores consist of the following layers:

- one footprint
- two reference
- two controlled impedance signal lines
- four power distribution planes



M - M CELL, DIVISIBLE INTO 1/4 CELLS

O - OUTPUT CELL (All shaded areas are O CELLS)

All unmarked internal cells are M CELLS

I - INPUT INTERFACE CELL (All white peripheral cells are I CELLS)

C - CLOCK PULSE GENERATOR CELL

NOTE: Pad arrangement is for TAB bond. Drawing not to scale.

Figure 2-2 Detail MCA Layout

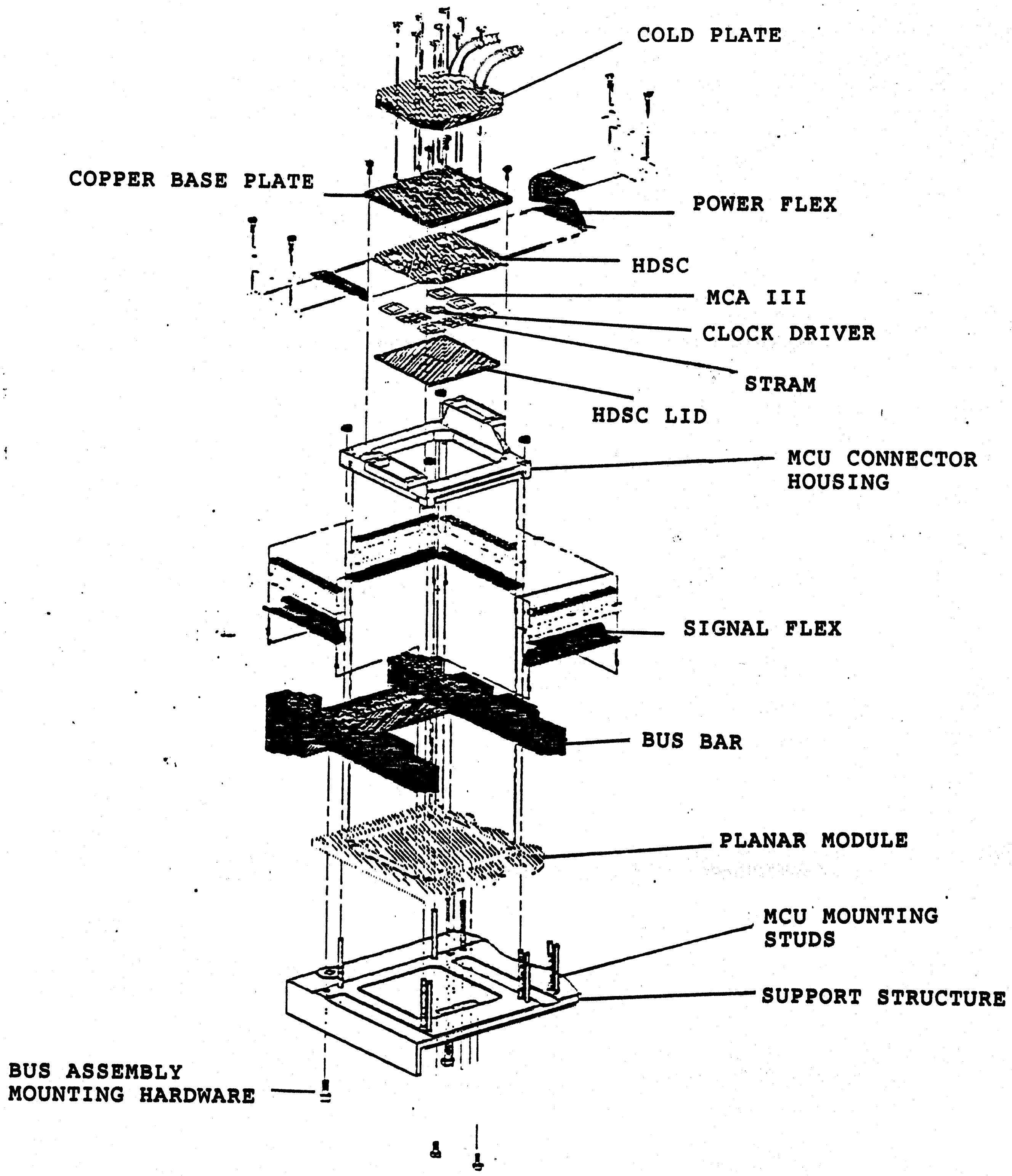


Figure 2–3 MCU Exploded View

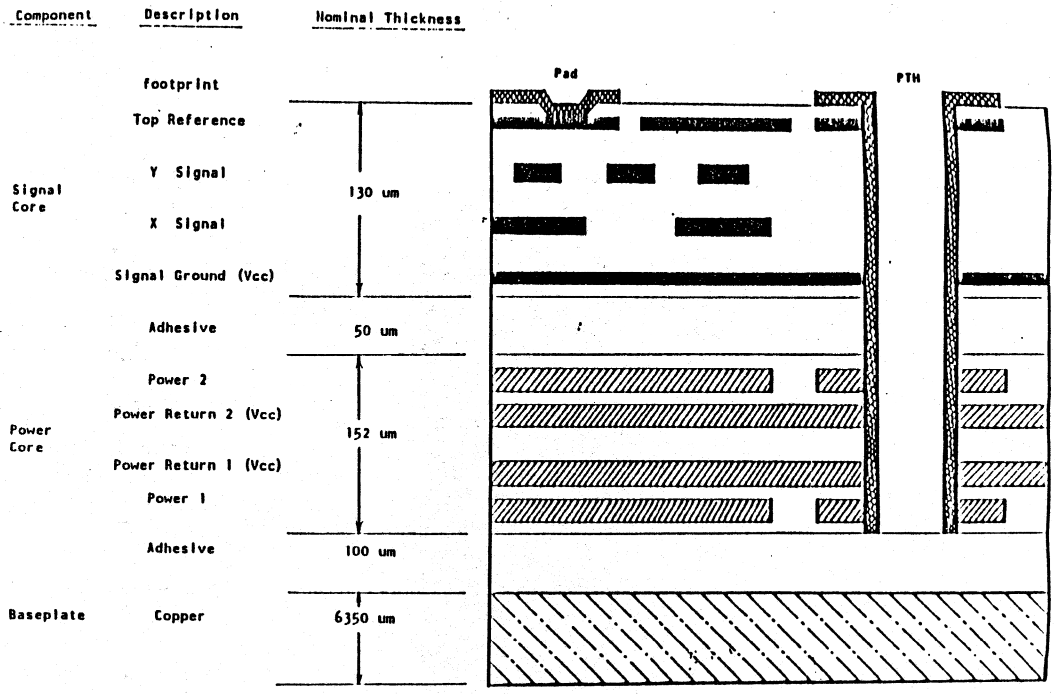


Figure 2-4 HDSC Layers - Side View

Signal interconnects from the MCU are provided by four, 201-pad, signal-flex connectors (identified as Signal Flex in Figure 2-3) and illustrated in Figure 2-5. The power planes are connected through two, power-flex connectors (identified as Power Flex in Figure 2-3) and illustrated in Figure 2-6.

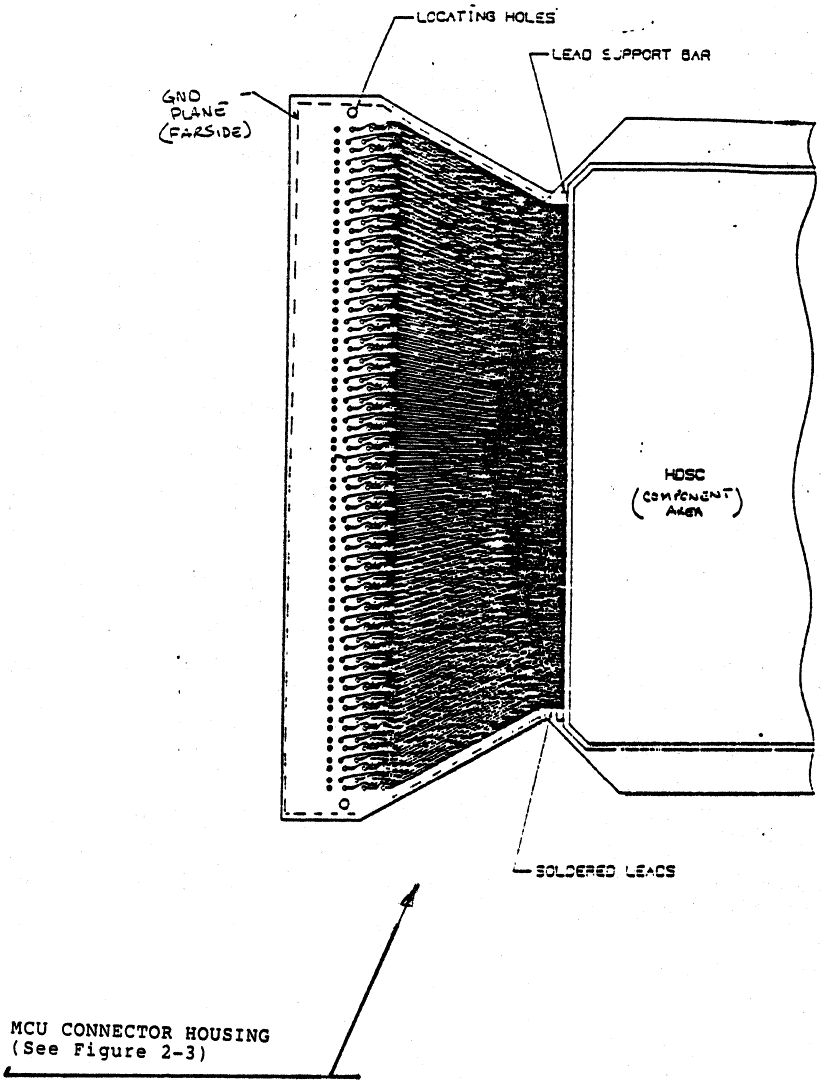


Figure 2-5 MCU Signal Flex

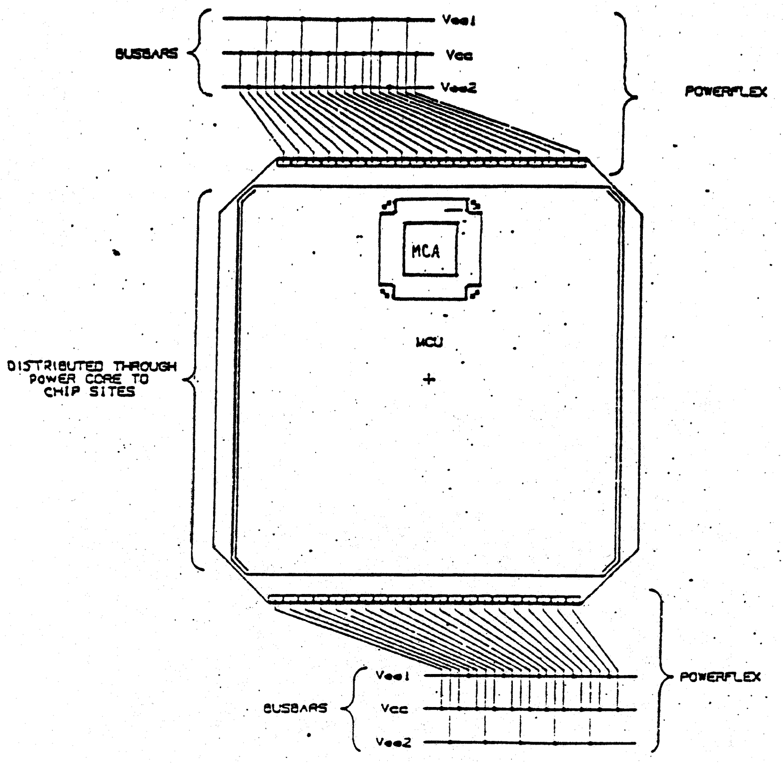


Figure 2-6 MCU Power Flex

A typical MCU layout is illustrated in Figure 2-7. Note that the illustration orientation is opposite that of Figure 2-3.

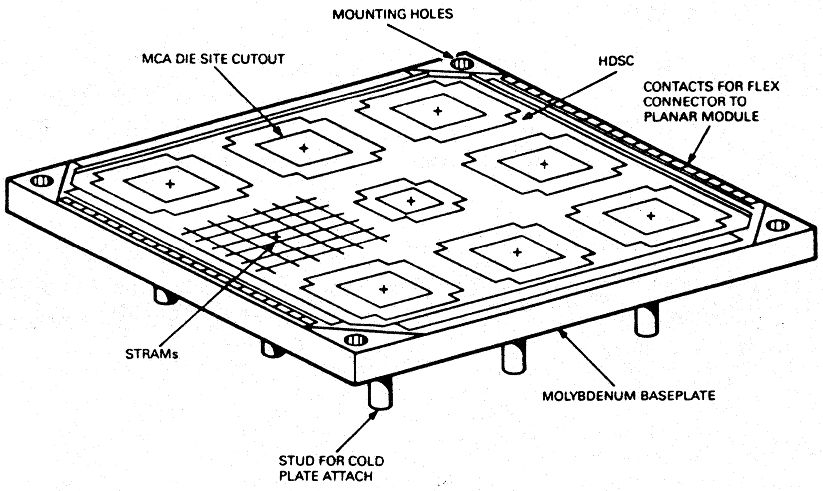

MOUNTING HOLES
MCA DIE SITE CUTOUT
HDSC
CONTACTS FOR FLEX CONNECTOR TO PLANAR MODULE
STRAMs
MOLYBDENUM BASEPLATE
STUD FOR COLD PLATE ATTACH

Figure 2-7 MCU Layout

2.2.3 Planar Modules

The CPU and SCU MCUs are assembled on a planar module. As shown in Figure 2–8, the CPU planar module accommodates 16 MCUs, which provides 13 MCU dies for a scalar processor and 3 dies for a vector accelerator. The CPU planar module is approximately 25 inches square, contains a 24-layer backplane (signal and ground planes), planar module power bus, and 16 cold plates attached to a planar casting. The assembly weighs approximately 180 pounds.

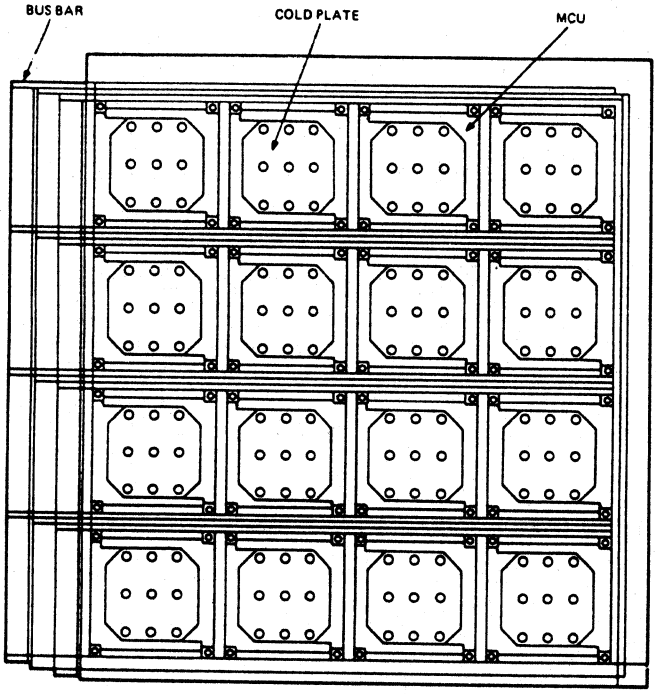


Figure 2–8 CPU Planar Module

The SCU planar module can accommodate a maximum of six MCUs. The minimum SCU configuration requires four MCUs. The planar module is 25 inches high and 19 inches wide, contains 24 layers, and weighs approximately 100 pounds. The module also contains a power bus, and MCU cold plates attached to a planar casting.

2.2.4 Self-Timed Storage Arrays

Two self-timed storage arrays are used in the system: Self Timed RAMs (STRAMs), and Self Timed Register Files (STREGs). The following subsections provide a functional summary of each component.

2.2.4.1 STRAM Functional Overview

The STRAM is a synchronous, self-timed, static RAM device. The STRAM is similar to a traditional RAM except it requires write, differential clock, and reference voltage inputs.

Two different STRAM chips are used in the system: 1K x 4, and 4K x 4. The major differences are the number of address bits required, and chip propagation delays. A STRAM requires the following inputs:

- address: A <11:00> (4k), or A <09:00> (1K)
- data input: DIN <03:00>
- data output: DO <03:00>
- chip select
- write enable
- differential clocks

Figure 2–9 illustrates the logic symbols for the 1K and 4K STRAMs.

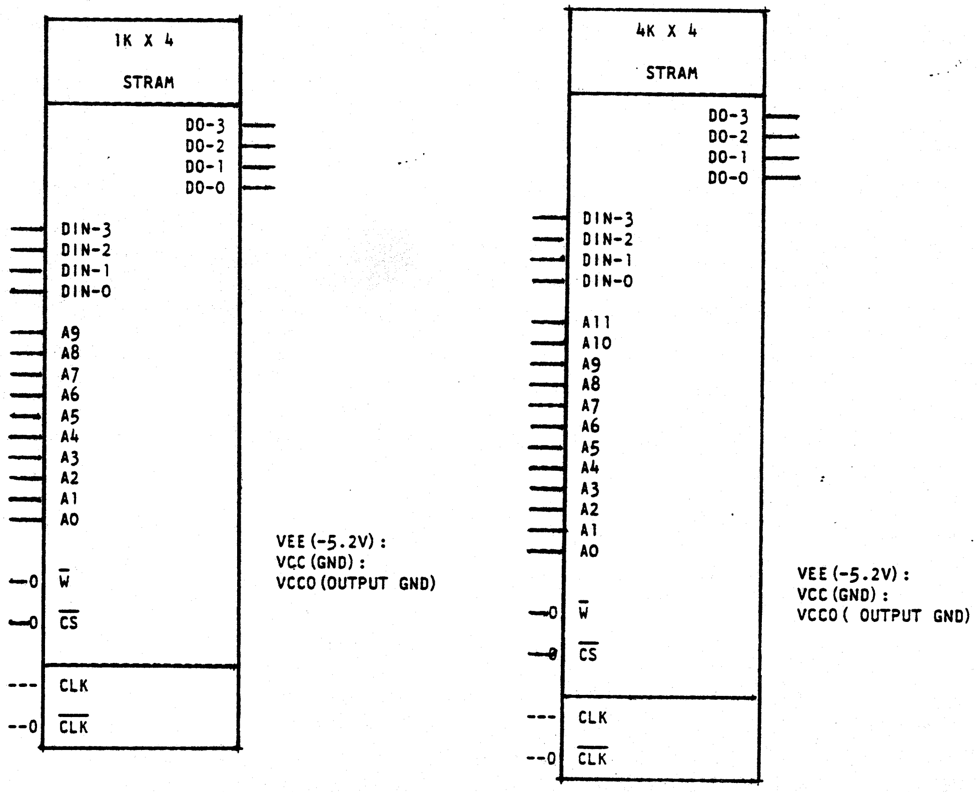


Figure 2–9 1K and 4k STRAM Logic Symbols

The core structure of the STRAM contains a memory cell (i.e., either 4K or 1K), and decode and read/write logic. The core is surrounded by latches which store address, input data, control signals, and output data. During a write operation, an internal write pulse generator (driven by the external differential clocks) causes data stored in the data in latches to be written to the array and to the output latches. During a read operation, the selected array data is stored in the output latches.

As shown in Figure 2-10 address, data, and write and chip select functions are passed through the input latches when the internal clock is a logic low, and stored when the clock is high. The output latch will store data when the clock is low, and will allow data to pass through when the clock is high.

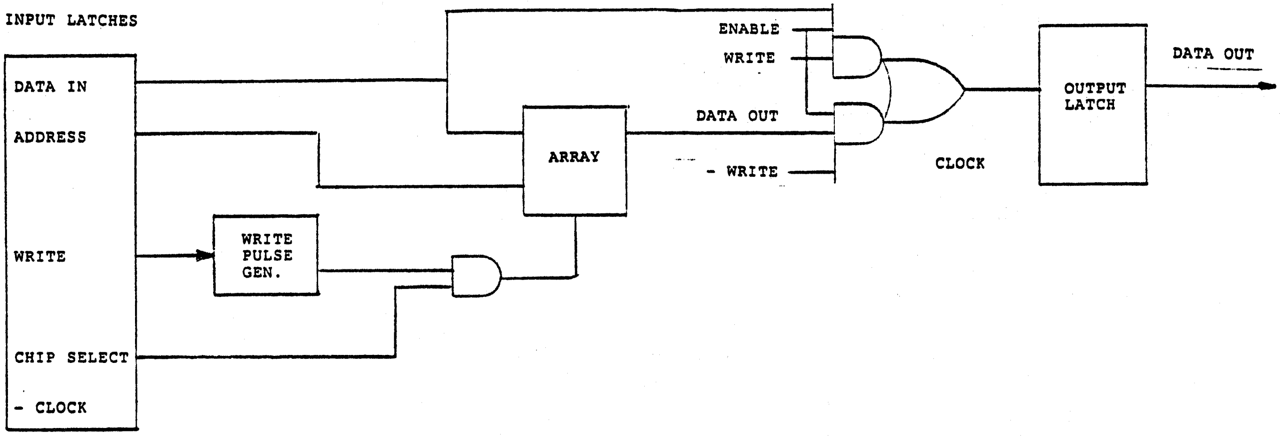


Figure 2–10 STRAM Cell Block Diagram

Read Operation: A read operation starts when the clock goes low. At this time the address, chip select, and write input latches open and allow the signals to propagate to the array. (See Figure 2-10.) Note that at this time the output latch is closed, storing data from the previous cycle. When the clock goes high the new data from the array is latched into the output latch, overwriting the previous data. If a read operation were attempted with the chip select function not asserted, the STRAM would load the output latches with logic lows.

Write Operation: A write operation also is initiated when the clock is low. At this time the address, data, chip select, and write input latches will open and allow the signals to propagate to the array. When the clock goes high, the new data is written into the array and to the output data latch. (See Figure 2-10.) If a write operation were attempted with the chip select function not asserted, the STRAM will prevent new data from being written to the array. However, it will write a logic low into the output latch.

2.2.4.2 STREG Functional Overview

The STREG is a 64 x 18 bit register file containing three write ports and two read ports.(See Figure 2-11). As shown in Figure 2-12 the 64 locations are separated into four, 16-location storage array sections (banks). The two read ports have independent access to all 64 locations. Simultaneous reads to the same location is allowed.

Simultaneous write operations are possible through the three write ports. However, there is no provision to detect writes to the same location from multiple ports. That is, data integrity is not guaranteed for simultaneous writes to the same address.

There are two clock inputs to the STREG: read clock (RCLK) and write clock (WCLK). The clocks control latching of the address, write enable, byte select, and data inputs. However, the array timing is internally generated and controlled.

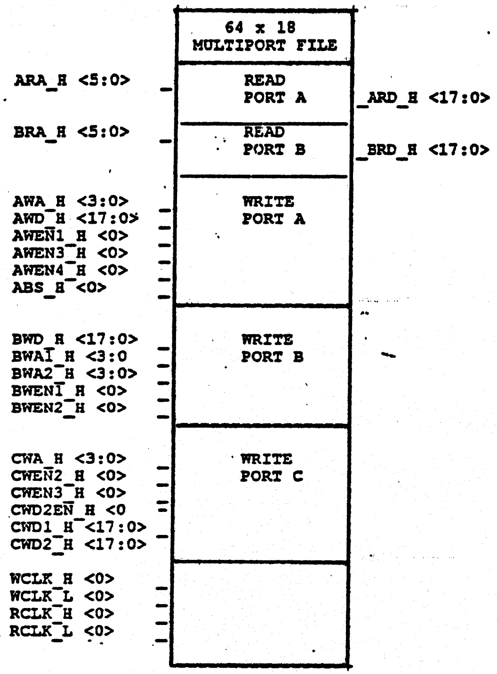


Figure 2–11 STREG Logic Symbol

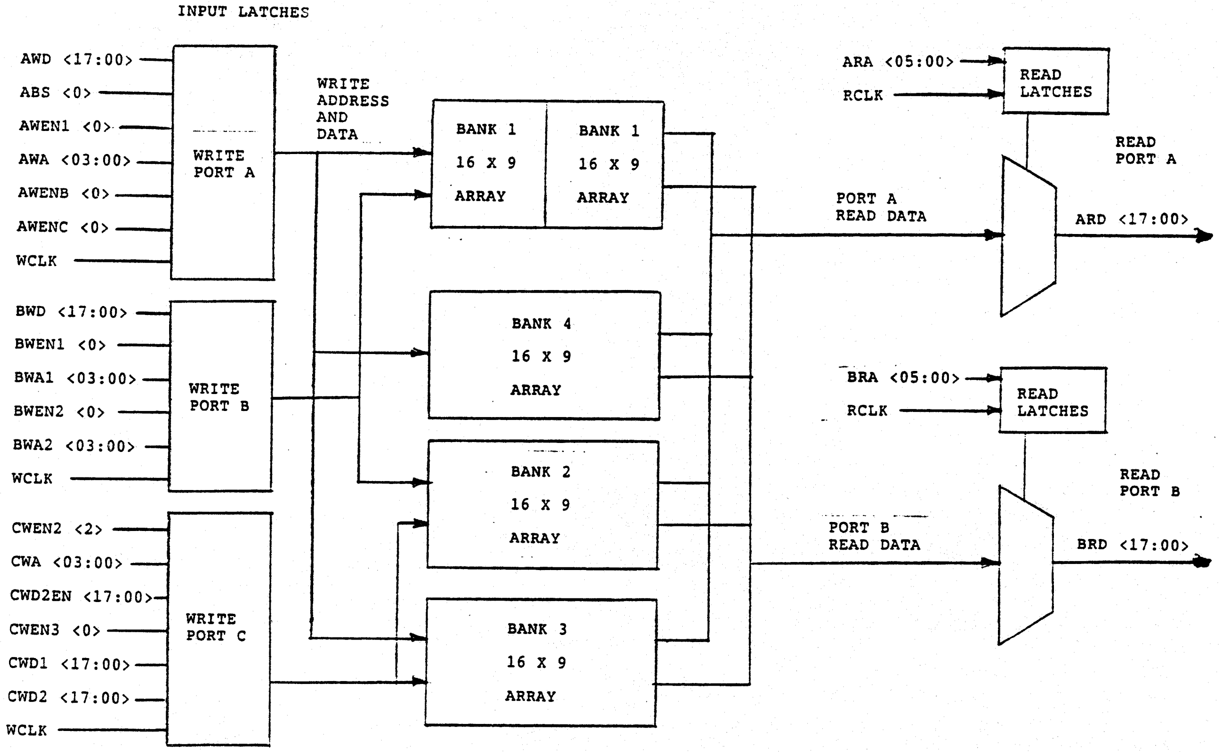


Figure 2–12 Basic STREG Block Diagram

Read Operation: Both read ports have a 6-bit address, and an 18-bit read data field as specified below:

- o Port A Input: ARA <05:00>; Data: ARD <17:00>
- o Port B Input: BRA <05:00>; Data: BRD <17:00>

Read addresses are latched on the failing edge of the read clock (RCLK). Table 2-1 specifies bank and location. (Refer to Figure 2-12.)

Table 2–1 Read Port Addressing

ADDRESS	ARRAY	LOCATION
00 - 0F	Bank 1	0 - 15
10 - 1F	Bank 2	0 - 15
20 - 2F	Bank 3	0 - 15
30 - 3F	Bank 4	0 - 15

Write Operation: Address, data, write enable, and clock inputs are required for a write operation. Each bank has a Write Enable bit (xWENy: where x is the port, and y is the bank). All write port input is latched on the falling edge of the write clock (WCLK). The three ports write to different combinations of banks; no port is allowed to write all 64 locations. In addition, each port has a different set of input requirements, and write capabilities.

Write Port A has one set of address and data lines as input. The address and data lines are applied to banks 1, 3, and 4. In order to write into a bank (or a combination of banks) its write enable bit (AWENy) must be asserted.

Write Port A is the only port with a byte write capability. Byte selection is provided by the byte select input (ABS). (Refer to Figure 2-12.) ABS selects a write into either the low-order byte or the entire word at the location as specified in Table 2-2.

Table 2–2 Write Port A Addressing

AWEN1	ABS	Bits Selected
0	0	none
0	1	none
1	0	08:00
1	1	17:00

Write Port B has two sets of bank address lines, and two corresponding bank write enable bits. That is: BWA1 <03:00> and BWEN1 for Bank 1, and BWA2 <03:00> and BWEN2 for Bank 2. The port has the capability to write to different locations in both banks, provided the corresponding BWENx is set. (Refer Figure 2-12.) As with Write Port A these inputs are latched on the falling edge of the write clock (WCLK).

Write Port C has one set of lines that address Banks 2 and 3 (CWA <03:00>). Each bank has a corresponding write enable input (CWEN2 and CWEN3). In addition, the port has two sets of data lines (CWD1 <17:00> and CWD2 <17:00>); CWD2 has a path enable bit (CWD2EN). Both data line sets can be used to write two locations simultaneously. That is, CWD1 data can be written to banks 2, or 3, or both 2 and 3. CWD2 data can only be written if CWD2EN is asserted, and only to location CWA + 1. The input data is latched on the falling edge of WCLK.

The following example describes port operation. (Refer to Figure 2–12.)

Assume the following input conditions:

- Address 1 (CWA <0001>)
- Bank 2 disabled (CWEN2_L)
- Bank 3 enabled (CWEN3_H)
- Data Path 1 122 (CWD1 <112>)
- Data Path 2 3F09 (CWD2 <3F09>)
- Data Path 2 enabled (CWD2EN_H)

Bank 2 is not involved in the write operation (CWD2EN_L). CWD1 data (112) is written into location 1 of Bank 3. At the same time, CWD2 data (3F09) is written into location 2 of Bank 3.

If the first set of data had been written into location 15, the second set of data would have been written to location 0 i.e., wrapped around).

2.3 System Cabinet Descriptions

Note that the line drawing of Figure 2–13 does not represent the latest cabinet design. It is included as a cabinet overview, and to help as a form of reference for the remaining illustrations.

Except for Figure 2–13, all other illustrations are screened photographs identifying some of the major components of the system cabinets. The photographs presented are of an AQUARIUS Macro Power Test Vehicle, and NOT of a prototype cabinet. Considerable mechanical redesign has taken place since the photographs were taken.

2.4 Water Cooling Unit Description

Each MCA III gate array in the CPU and SCU dissipates approximately 30 watts from an IC die area approximately the same size as current VLSI gate arrays. Given the logic gate density, water cooling is the most effective method for dealing with the high thermal MCA III densities.

One of the primary factors in IC reliability is junction temperature (Tj). Liquid cooling provides an efficient means of reducing Tj for these devices from the typical range of 80 - 90 degrees C of current air cooled devices to 65 degrees C.

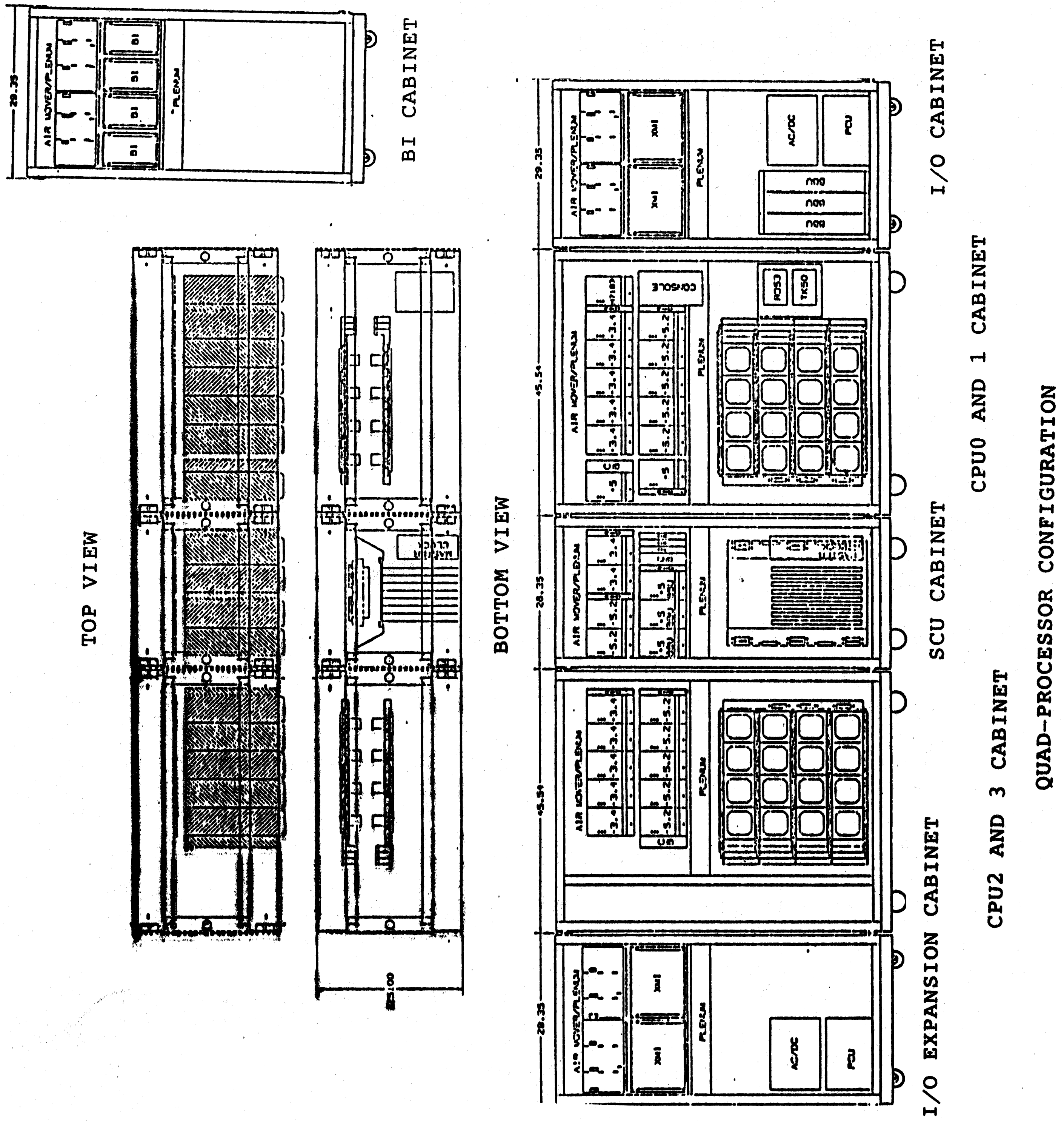


Figure 2–13 Basic System Cabinet Views

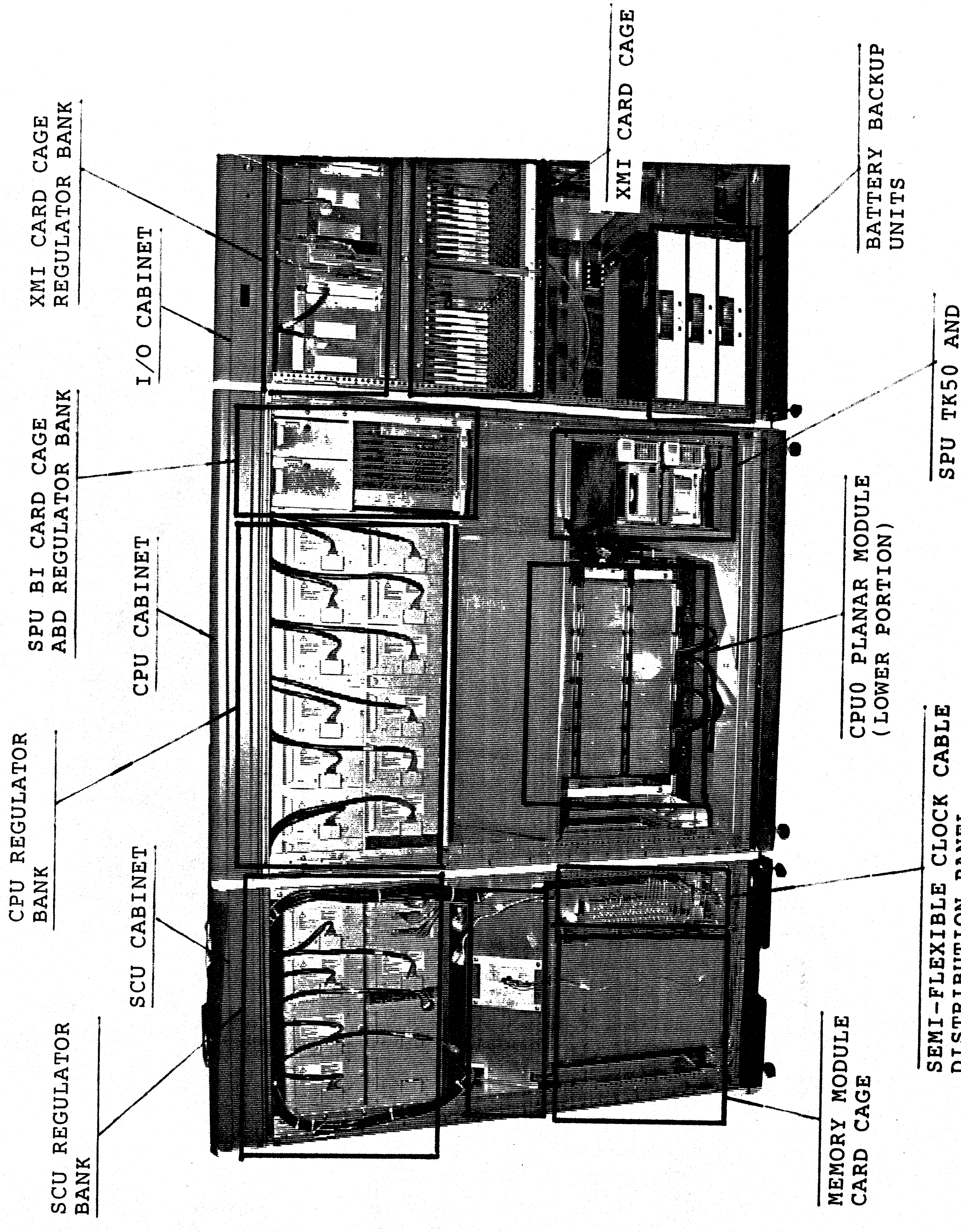


Figure 2-14 System Cabinets - Front View

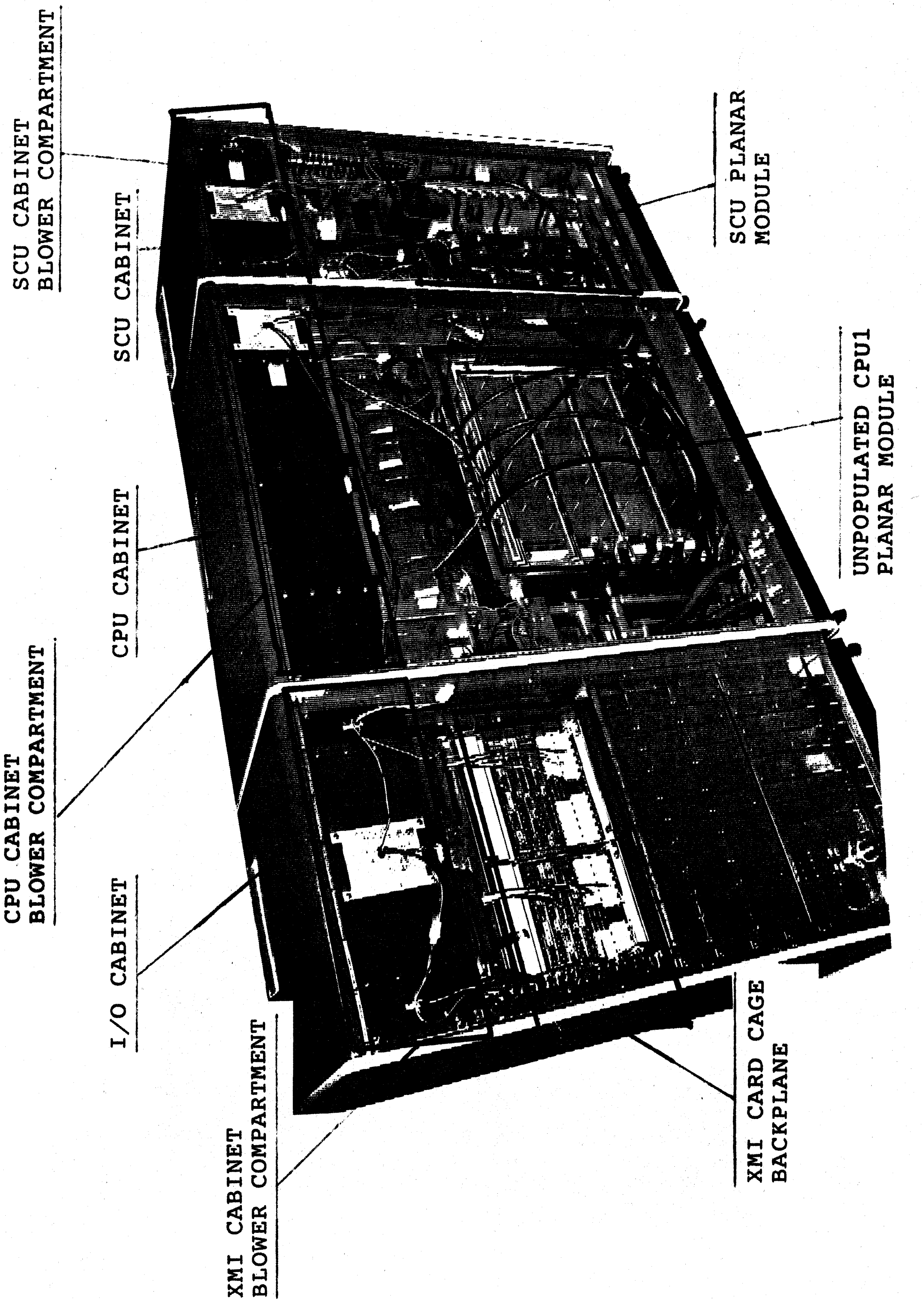

SCU CABINET BLOWER COMPARTMENT
SCU CABINET
CPU CABINET
CPU CABINET BLOWER COMPARTMENT
I/O CABINET
XMI CABINET BLOWER COMPARTMENT
SCU PLANAR MODULE
UNPOPULATED CPU1 PLANAR MODULE
XMI CARD CAGE BACKPLANE

Figure 2–15 System cabinets - Rear View

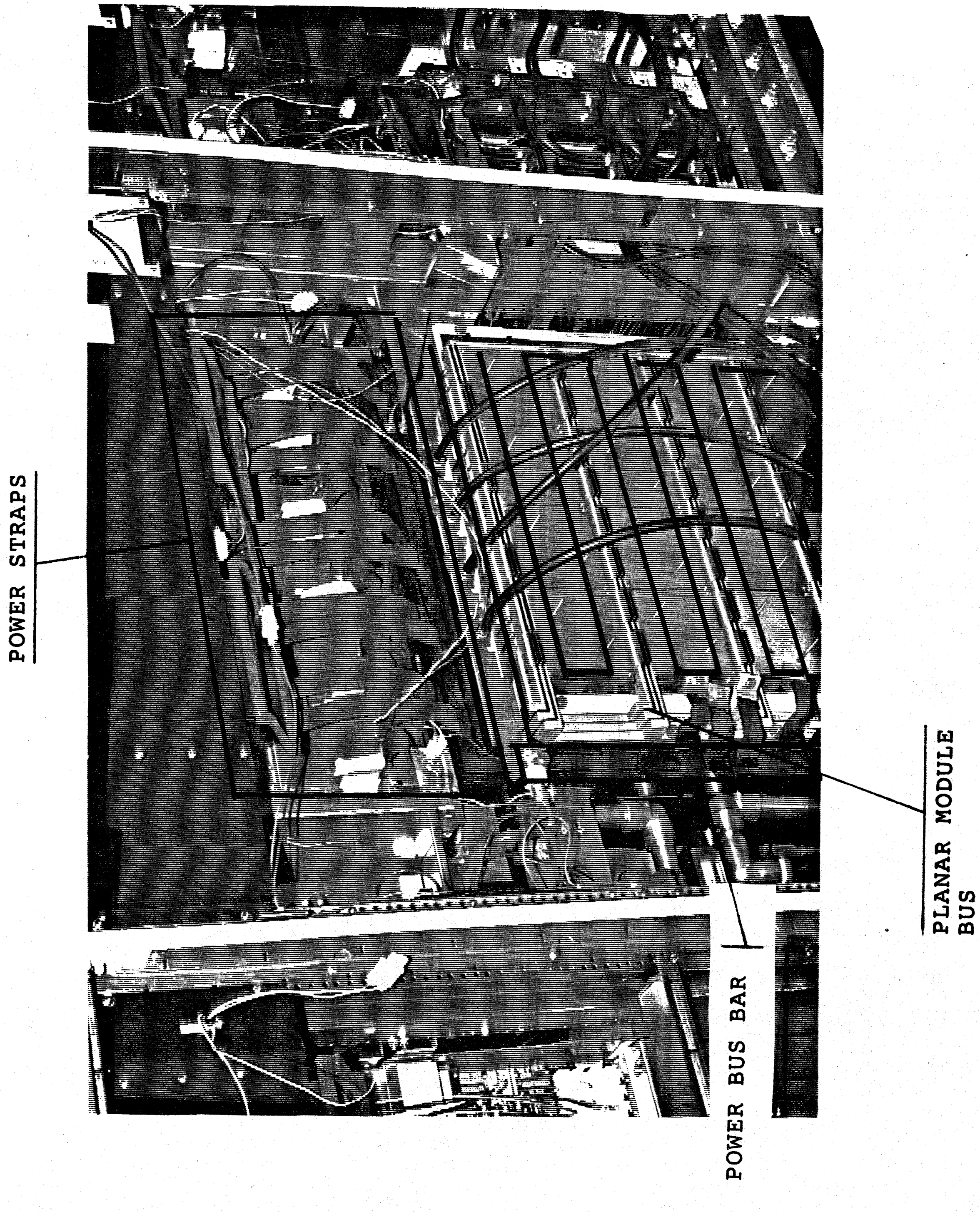


Figure 2–16 CPU Cabinet - Planar Module

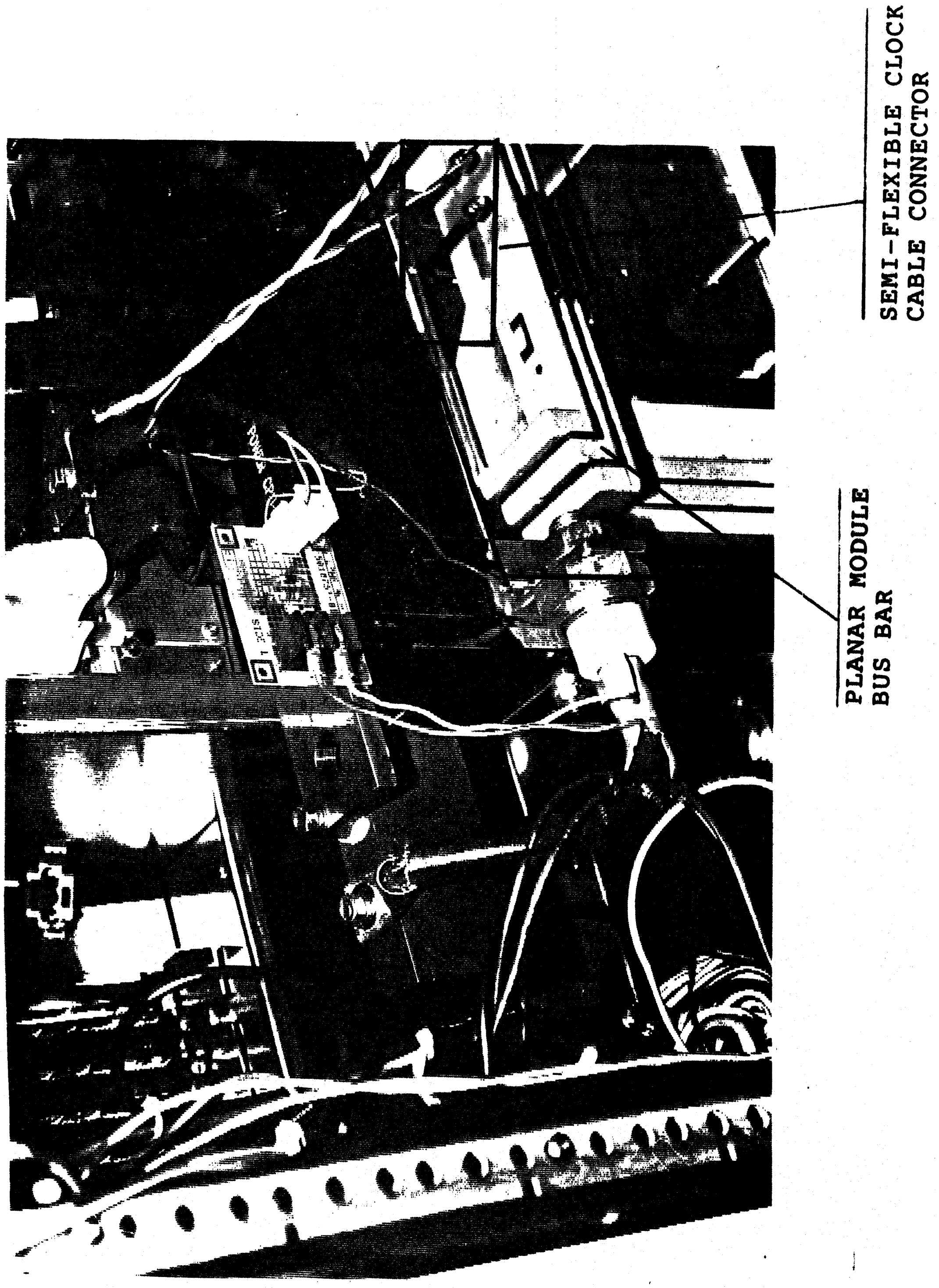


Figure 2–17 CPU Cabinet - Power Bus

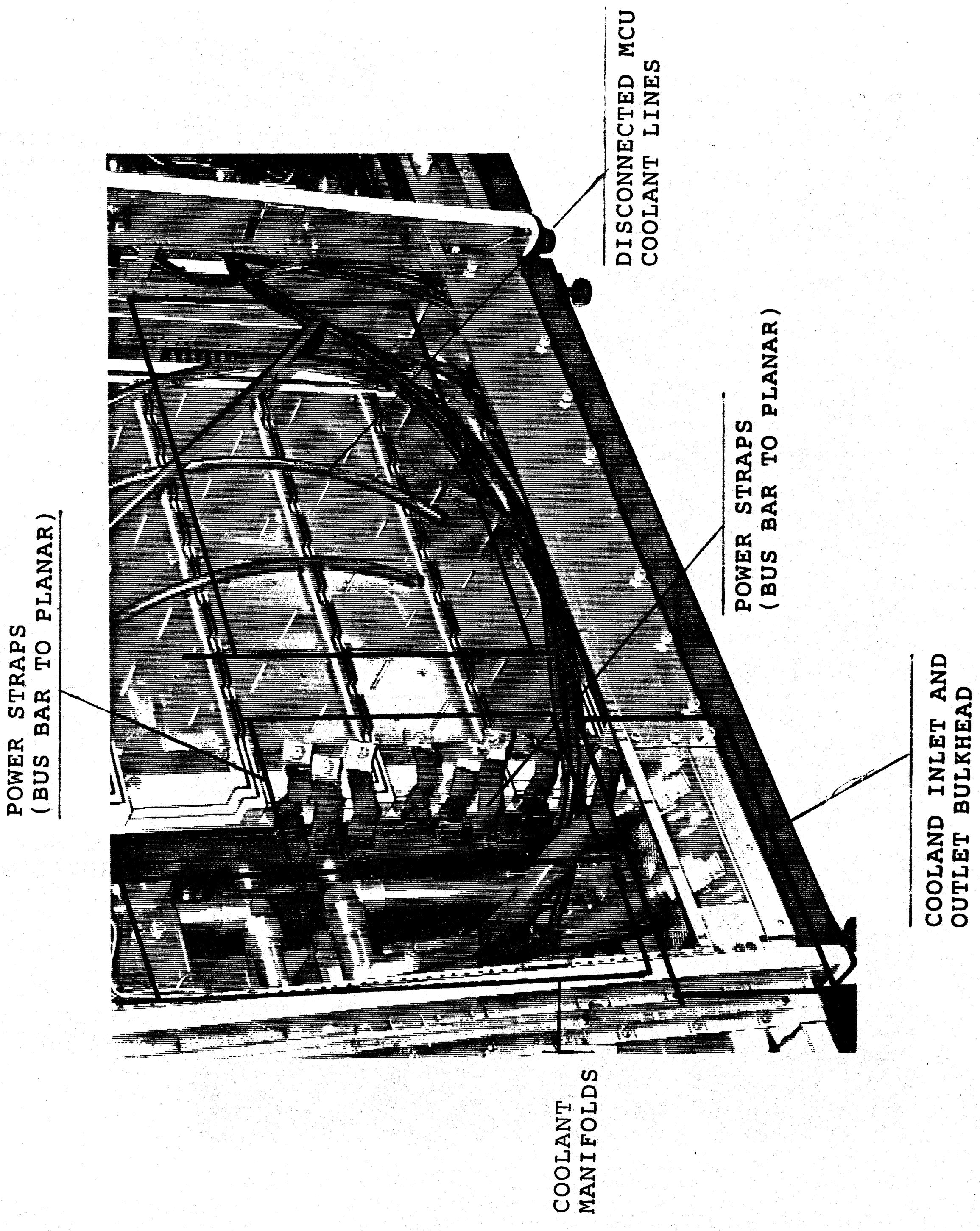


Figure 2–18 CPU Cabinet - Coolant Bulkhead

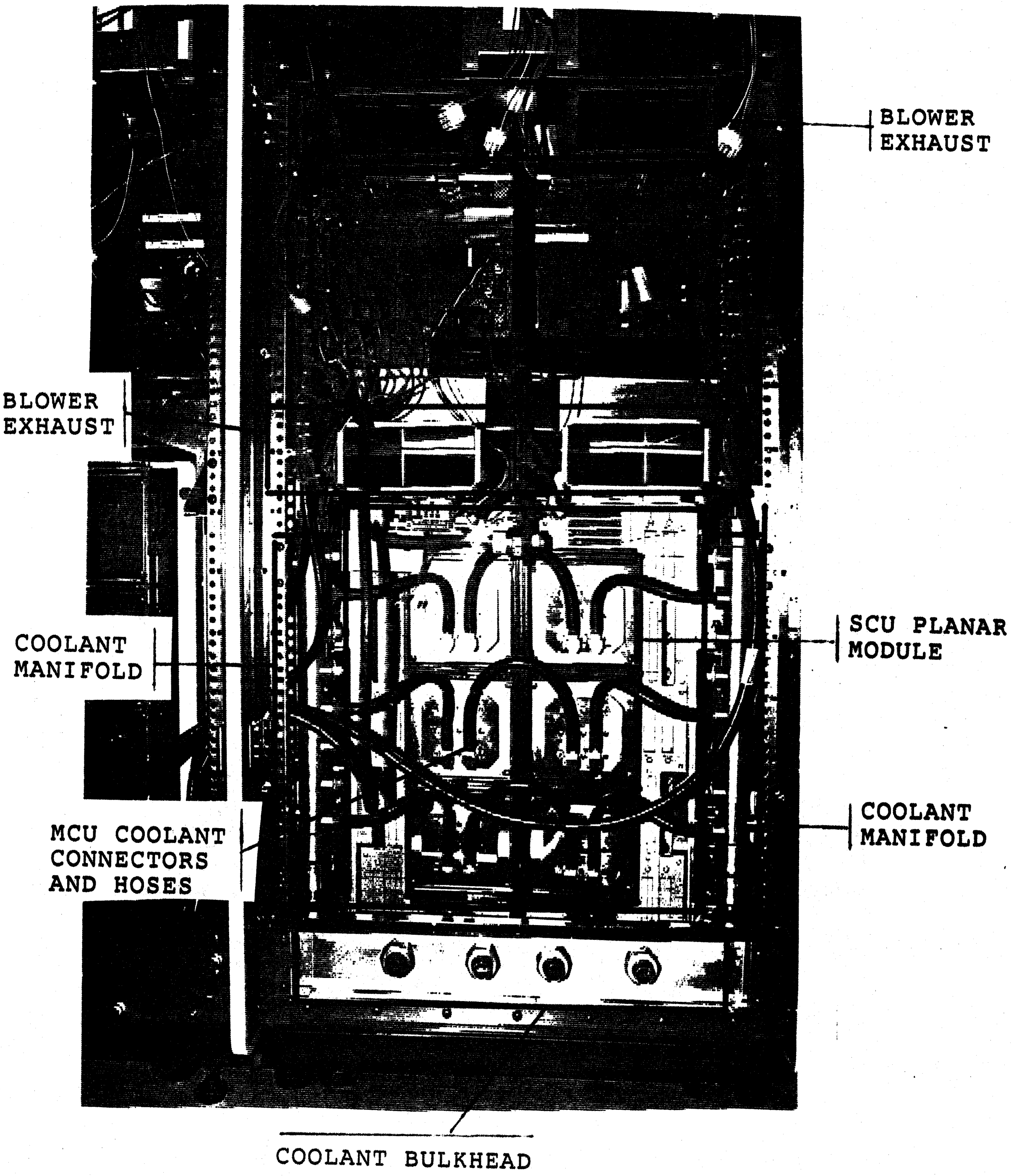


Figure 2–19 SCU Cabinet - Rear View

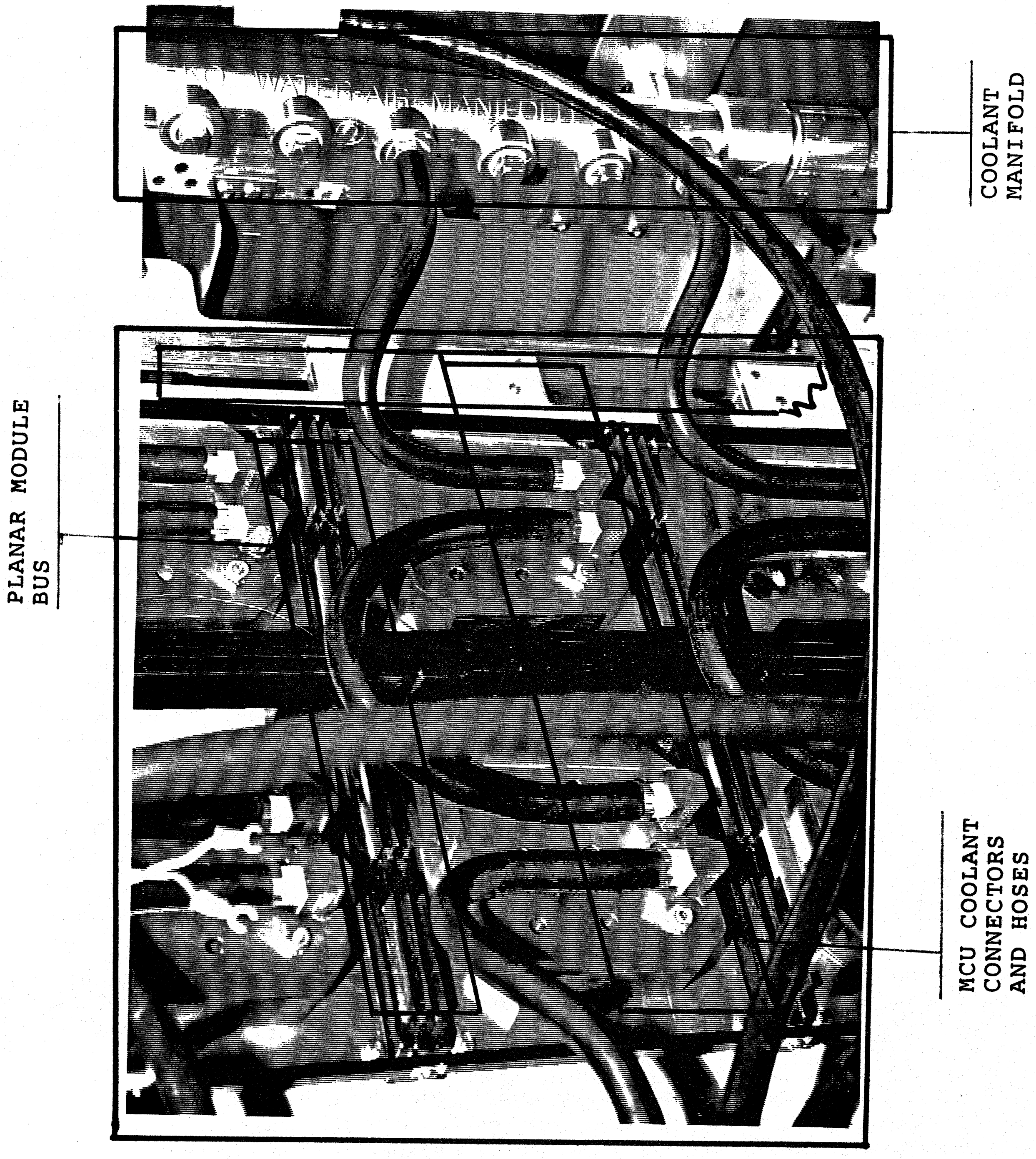


Figure 2–20 SCU Cabinet - Planar Module

Water cooling is far more efficient than air in transferring heat from a plate to a fluid, and will consume less power to cool the same heat load than air. Since water is approximately 1000 times denser than air, it provides heat transfer rates which are approximately 100 times higher than air. In addition, a water cooling system eliminates much of the acoustic noise problems associated with air cooling systems.

2.4.1 Functional Overview

The Water Cooling Unit (WCU) provides cooling for the following system logic components:

- Scalar processor (CPU)
- System Control Unit (SCU)
- Vector Accelerator (VBox)

The remainder of the system is cooled with forced air supplied by dc blowers.

As shown in Figure 2–21 the WCU is housed in a cabinet approximately 60 inches high, 62 inches wide, and 30 inches deep. The WCU can be located up to 50 feet from the system cabinets. The WCU is a buy-out product. It is being designed and manufactured by Liebert Corporation, and will operate in a Class A computer room environment.

The cooling subsystem consists of two sections: the WCU cabinet, and the system thermal load, which consists of the MCU cold plates. These sections are interconnected by pairs of rubber hose. Connections at each end are made through quick disconnect fittings which allow the hose to be connected and disconnected without draining the system.

The WCU acts as a heat exchange and pumping station for the coolant which is delivered to, and returned from, the system cabinets. A WCU is capable of dissipating the heat load of a dual CPU configuration (two CPUs and one SCU) of approximately 8.5 KW while maintaining a Tj of 65 degrees C.

In quad and redundant dual processor configurations, a second WCU is required. The WCUs are cross coupled to enable cooling of the configuration in cases where one WCU has failed. In such cases, the water cooled components will experience a gradual rise in junction temperature of approximately 7 degrees C until the failed WCU is returned to service.

As shown in Figure 2–22 the WCU uses circulation pumps for coolant flow. Coolant (i.e., treated water) at ambient room temperature is pumped from the WCU through a set of supply hoses to the System and SCU cabinets. The coolant flows through two supply manifolds to the MCU cold plates containing the logic devices which are the primary heat load.

The heated coolant is then returned through two return manifolds in the cabinets to the WCU through a set of return hoses. The heat is ejected into the room environment by way of a liquid-to-air heat exchanger (i.e., radiator). The heat is ejected from the exchanger by a pair of blowers.

Since the system uses ambient room air as the secondary coolant, temperature or humidity controls are not required. Coolant temperature tracks room air temperature. This also eliminates the need to contend with condensation since no part of the WCU system is cooler than the room air.

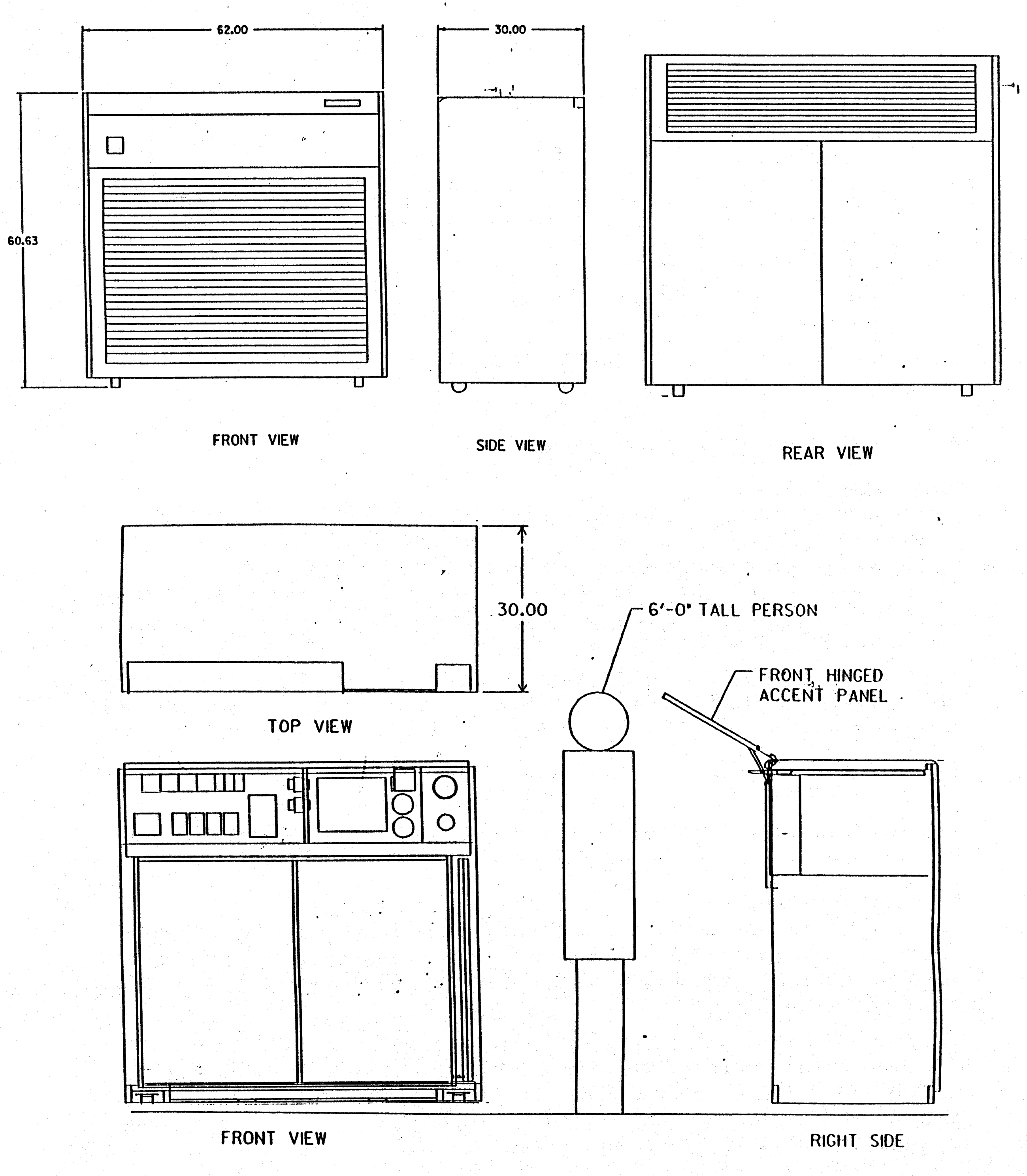


Figure 2–21 Basic WCU Layout

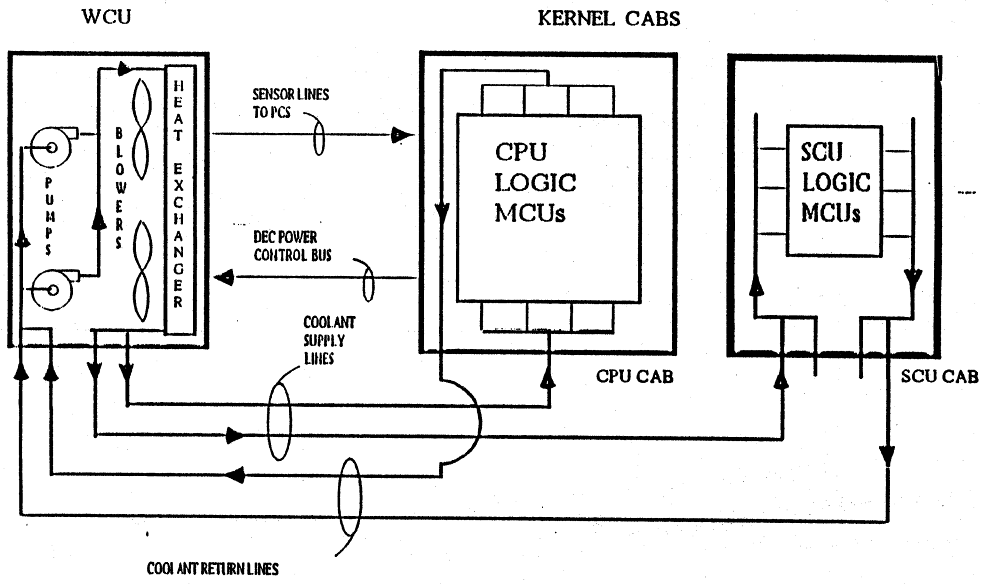


Figure 2–22 Basic WCU System Diagram

2.4.1.1 Coolant Flow

Within the CPU and SCU cabinets coolant flow is in parallel (see Figure 2–23). The coolant is received and returned at the cabinet quick disconnect bulkheads. The CPU cabinet bulkhead accommodates one, 1-inch supply and one, 1-inch return disconnect. The SCU cabinet bulkhead accommodates two, 3/4-inch disconnects and two, 3/4-inch disconnects. Coolant flows through the inlet bulkheads of each cabinet, through rubber hose to the PVC distribution manifolds.

The PVC supply manifold in the CPU cabinet is positioned across the bottom of the CPU planar module. Coolant flows in parallel through four sets of four, series-connected, CPU cold plates (uniprocessor configuration). (See Figure 2–24).

The PVC supply manifold in the SCU is positioned vertically and to the right of the SCU planar. The coolant flows through three sets of two SCU cold plates.

The coolant then flows from the cold plates in each cabinet to the PVC return manifolds, through the return hoses to the quick disconnect bulkheads. (See Figure 2–23.)

2.4.1.2 MCU Cold Plates

The MCA IIIs, STRAMs, and Clock Distributions Chips form the majority of the WCU heat load. These components are bonded directly to the copper base plate of the MCU. Heat is transferred through the base plate, across a dry joint to a sealed copper cold plate bolted to the MCU base plate.

There are 16 cold plates for each CPU planar module, and six plates for the SCU. The cold plates are interconnected through rubber hoses with slip-on fittings. Each cold plate is approximately four inches square The mating surfaces between the base plate and cold plate are machined to a fine tolerance and finish to insure proper heat transfer. Contact pressure at this interface is maintained by nine cap screws.

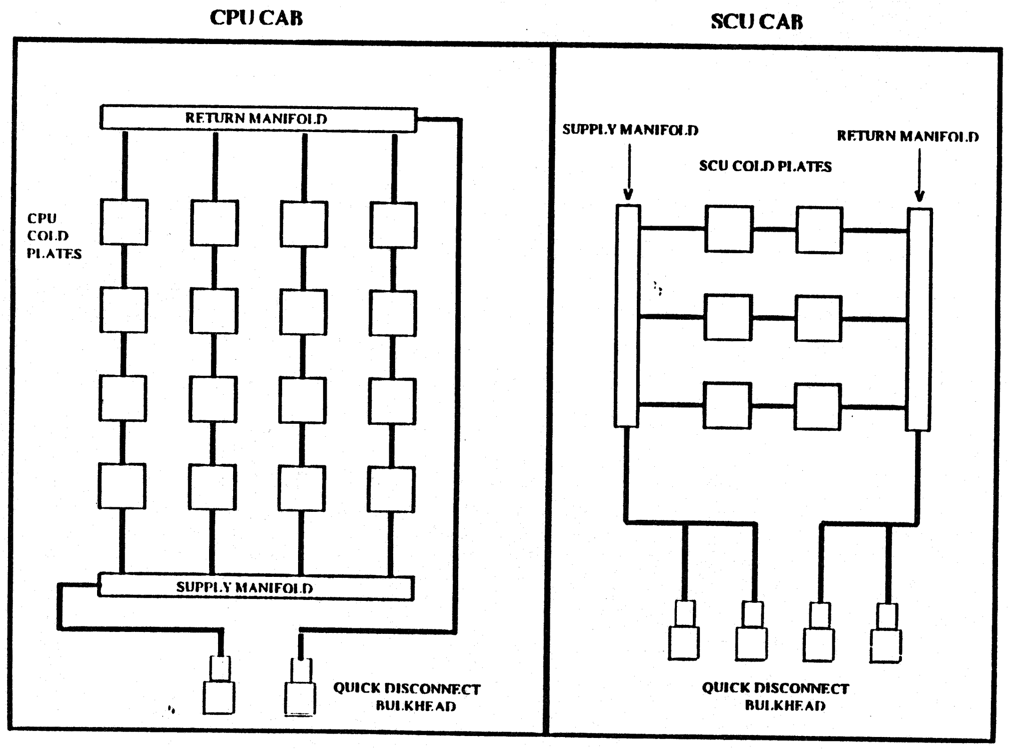


Figure 2–23 CPU Cabinet Coolant Flow

2.4.2 Physical Layout

Figures Figure 2-25, Figure 2-26,and Figure 2-27 describe the component layout as well as basic electrical and coolant connections.

2.4.3 WCU Coolant

The coolant is distilled water with sodium borate and Benzotriazole added to control pH and corrosion. A dye will also be added to discriminate AQUARIUS coolant from tap water or other liquids.

2.4.3.1 Coolant Testing

Long term reliability and performance of the cooling system is dependent on the chemical balance of the coolant. As such, yearly on-site analysis of the coolant during PM has been recommended. The testing is designed to insure that the following key parameters of coolant chemistry are within proper limits

o pH

o conductivity

o Benzotriazole concentration.

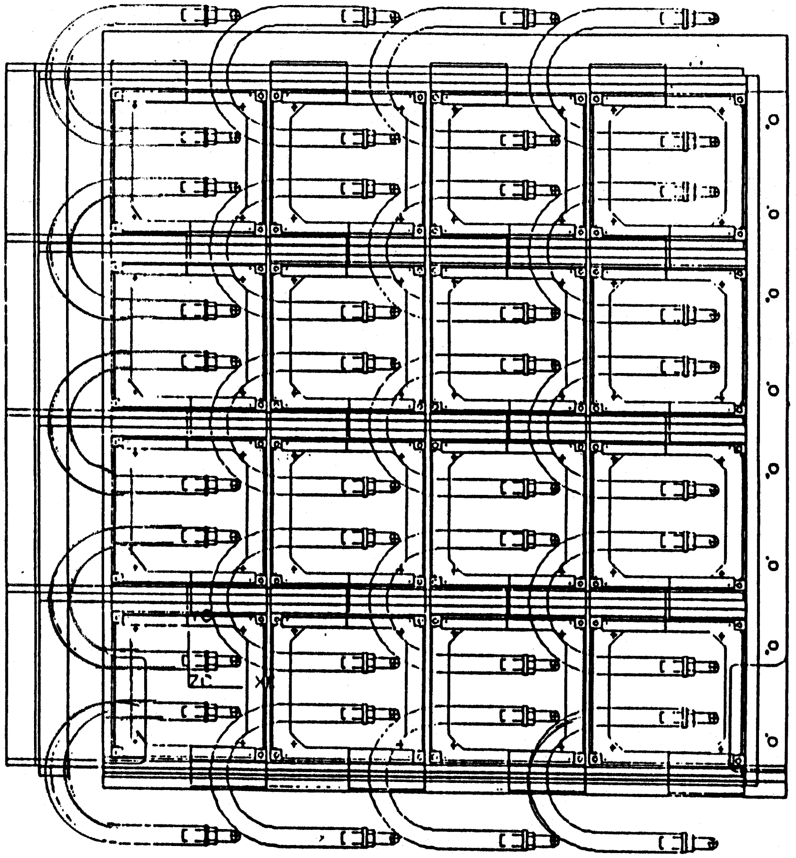


Figure 2–24 CPU Planar Module Flow

These parameters are checked to insure that excessive corrosion is not occurring in the system. Portable and hand-held instrumentation will be identified to allow field personnel to conduct these tests. Charts and procedures will describe allowable variance from nominal readings, and describe how to make adjustments with additional additives.

The data acquired from the coolant analysis, as well as the amount of additives that may have been used for adjustment, will be entered on the Coolant Analysis and Tracking System (CATS). The data cards will be returned to CSSE/MIS in Stow, MA and entered into the CATS data base to allow trend analysis and cooling system maintenance activity tracking. In cases where readings show an unacceptable variance, off-site laboratory analysis may be used to identify the problem and suggest corrective action.

The packaging of these additives (as well as the coolant), and potential health hazards to field personnel are currently under investigation by CSSE and the Field Service Environmental Health and Safety Group.

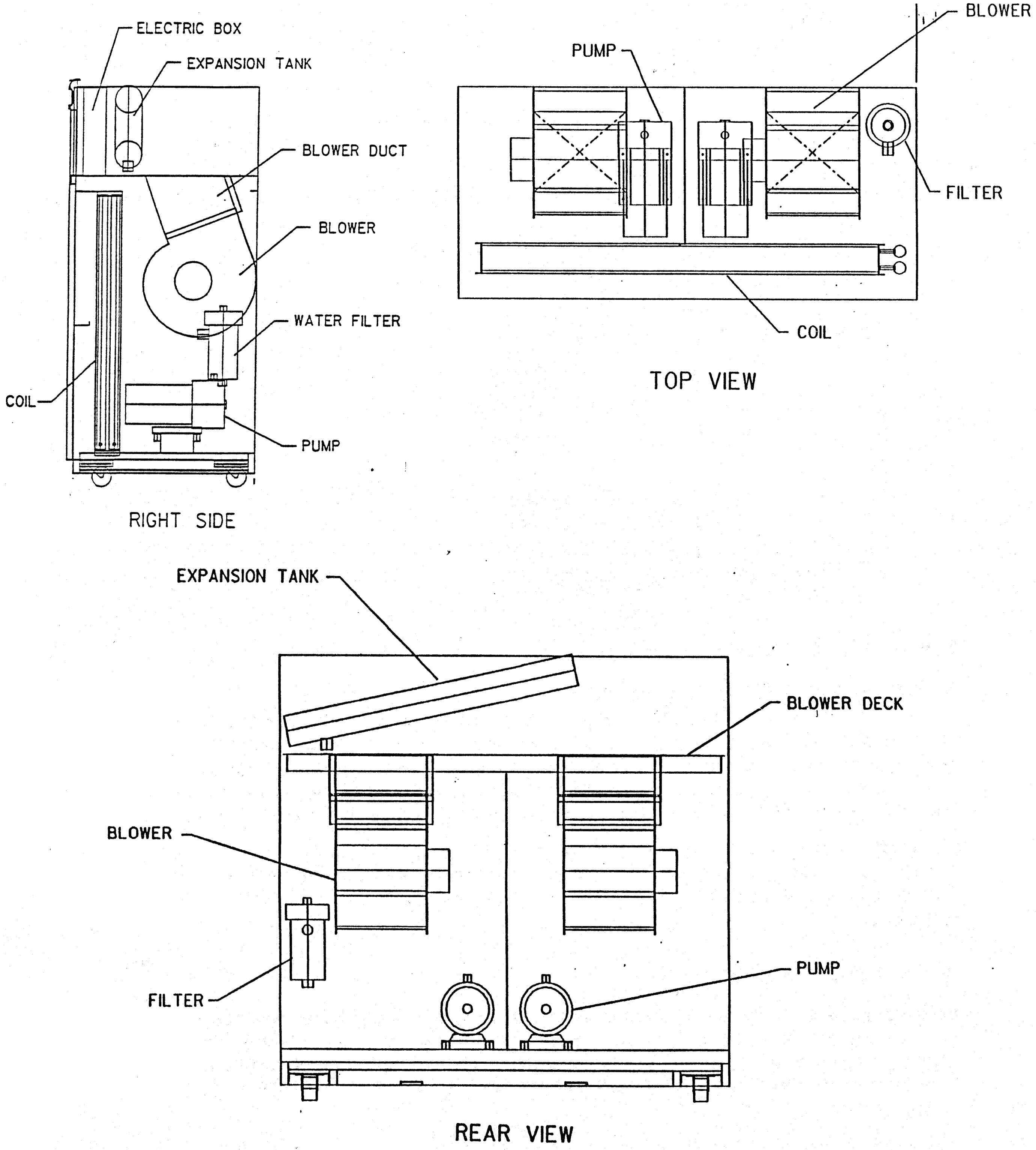


Figure 2–25 WCU Component Layout

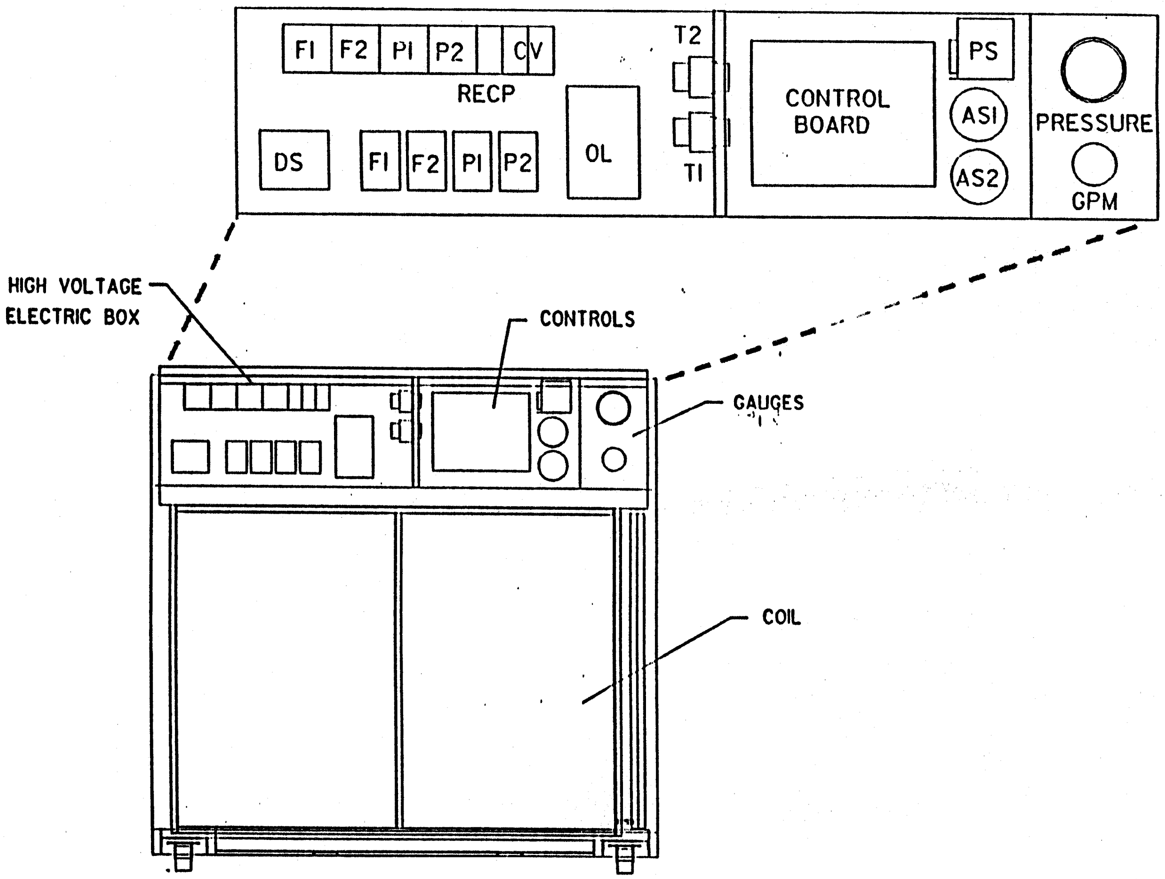


Figure 2-26 Electrical Box Layout

2.4.3.2 Coolant Shipment and Handling

The WCU and kernel will be shipped dry to the installation site. The coolant will be shipped pre-mixed in special TBD gallon containers for system installation. Spare quantities of coolant will also be available in a one gallon version of this container to allow adding top-off quantities of coolant to the system after FRU replacement. Figure Figure 2-28 illustrates a coolant shipping container connected to a system fill pump.

Handling and disposal of any coolant removed from the system may be complicated by environmental regulations and ordinances at the Federal, State, and municipal levels. Quantities of coolant will be removed from the site and consolidated at a DEC facility for disposition.

2.4.4 Fault Isolation

Sensors in the WCU monitor various operational parameters such as coolant temperature, flow, pressure, and air flow. All sensor data is returned to the Power Control Subsystem (PCS), and SPU and system error logs. Fault isolation rules will be included in the SDD tools to provide FRU identification callout both in auto-call and remote diagnosis service activity.

The WCU is capable of shutting itself down based on sensor status, and independent of the PCS. A second set of "backup" sensors in the Clock Distribution Chip on each MCU are available to trigger a system shut down in the event that the WCU sensors fail to properly function.

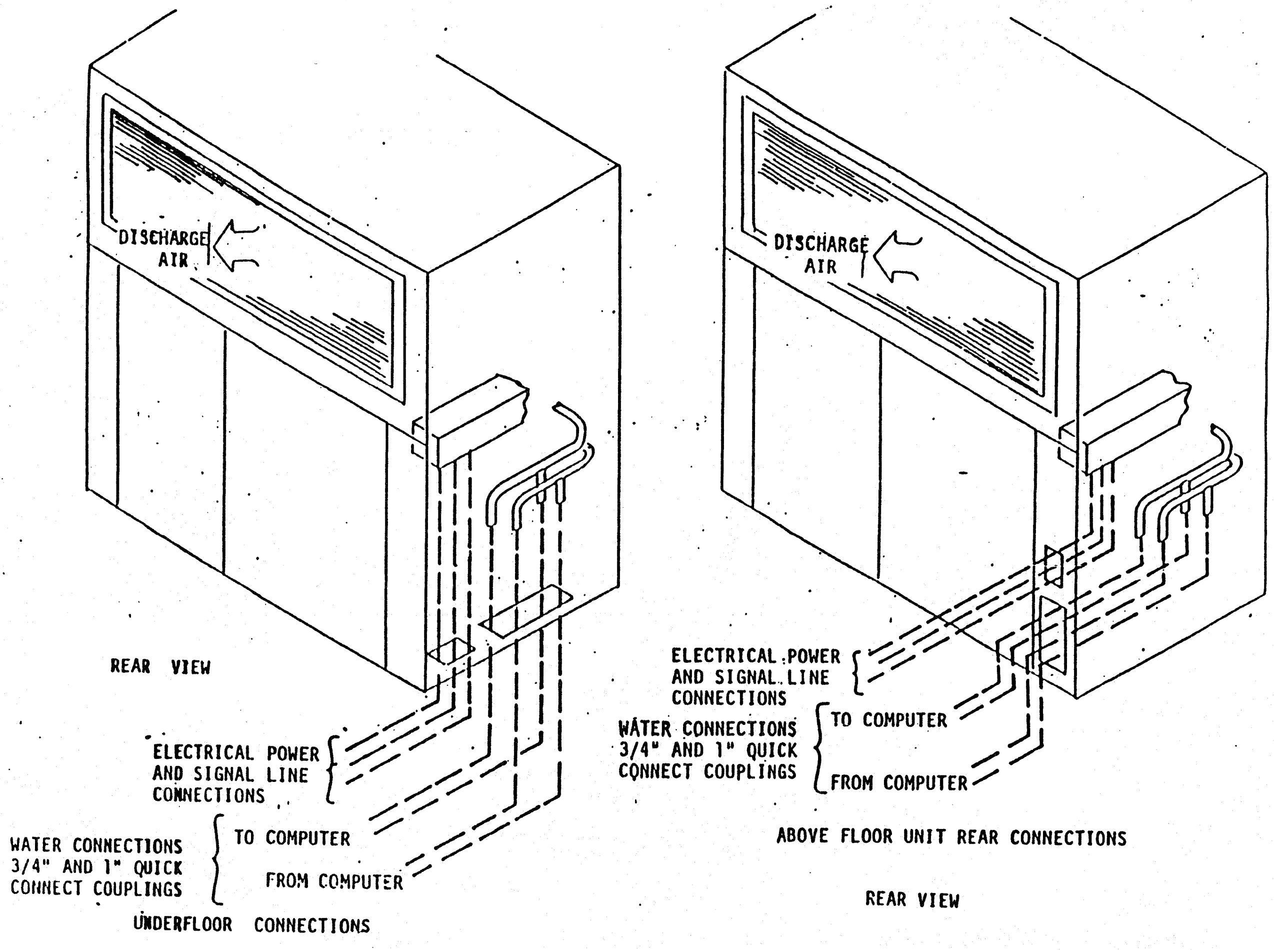

DISCHARGE AIR
REAR VIEW
ELECTRICAL POWER AND SIGNAL LINE CONNECTIONS
WATER CONNECTIONS 3/4" AND 1" QUICK CONNECT COUPLINGS
TO COMPUTER
FROM COMPUTER
UNDERFLOOR CONNECTIONS
DISCHARGE AIR
ELECTRICAL POWER AND SIGNAL LINE CONNECTIONS
WATER CONNECTIONS 3/4" AND 1" QUICK CONNECT COUPLINGS
TO COMPUTER
FROM COMPUTER
ABOVE FLOOR UNIT REAR CONNECTIONS
REAR VIEW

Figure 2–27 Basic Electrical and Coolant Connections

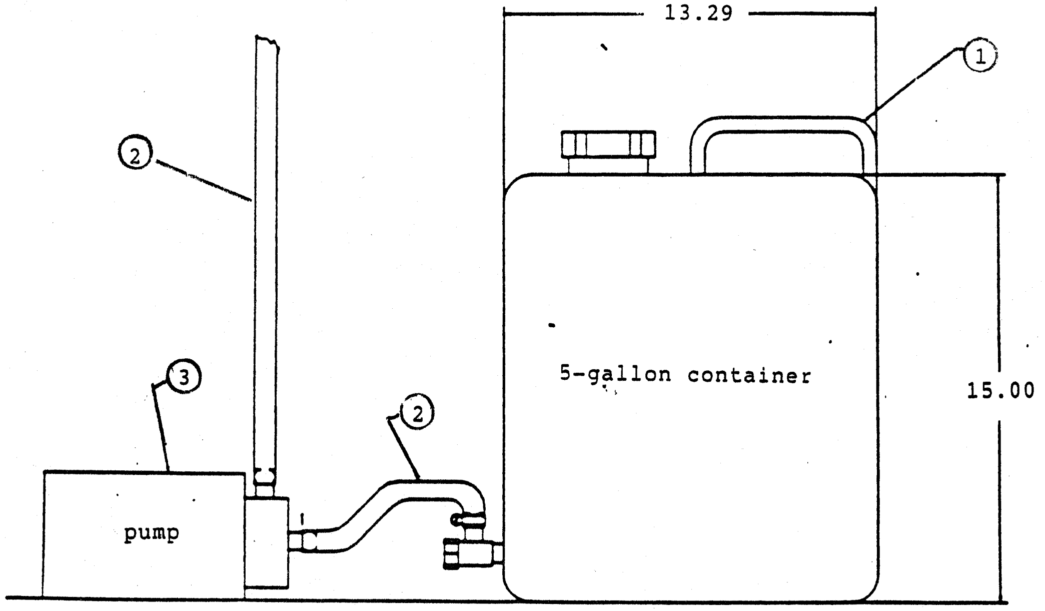
Additional visual fault indicators are provided in the WCU. In cases where problems are less obvious, such as a partial blockage of a cold plate, hand held temperature probes can be used for isolation.

2.4.4.1 Sensor Fault Logic

The sensor fault logic is designed such that corroborating information from at least two sensors is required to indicate a fault condition in the cooling system which may require a shutdown. Since most fault conditions will effect at least two parameters this provides a means to discriminate between sensor failures and actual system faults.

The exception to the 2-fault shut down requirement involves the Low Coolant Level Sensor. Its single fault signal will trigger an automatic shut down (ASD) of the system.

All sensors, with the exception of the drip pan moisture detector, are connected to the WCU logic module and PCS in a normally closed state. Thus, a disconnected sensor or harness will indicate a fault. All sensors in the system are considered FRUs and can be replaced using hand tools.



1 5-gallon, rectangular, seamless container with spigot.
spigot /

2 1/2-inch ID plastic tubing, fits container spigot, and WCU inlet and outlet connectors.

3 115 vac magnetic drive pump with thermal overload protection.

Figure 2–28 Shipping Container and Fill Pump

2.4.4.2 Sensor, Switch, and Indicator Summary

The WCU logic module provide indications and switches for:

- o Pump A or Pump B Running
- o Blowers Running
- o Pump A or Pump B Fault
- o Moisture Detected in WCU drip pan
- o Clear Fault Switch
- o Manual Pump Switchover
- o Pump Selection Override

Additional sensors monitor the following parameters;

- o Inlet and outlet coolant temperature (A)
- o Inlet and outlet air temperature (A)
- o Coolant Flow (in & out)
- o Cooling System Pressure
- o Coolant Level (High and Low)

- Pan Moisture Detector
- AC Input Power Phase Reversal
- Blower Air Flow (A & B)

(A) = Actual analog measurement. Remaining sensors act as limit switches.

The water and air temperature, blower air flow, and pan moisture sensors may be replaced while the system is operating by disconnecting the wire harness and removing the unit.

The following sensors are in-line with the piping and require the unit to be shut down for replacement. In each case isolation valves are provided around the unit, which must be closed prior to removing:

- coolant flow switches
- cooling system pressure switch
- coolant level switches
- pressure gauge.
- flow gauge

2.4.4.3 A/C Phase Rotation Sensor

Since the WCU uses 3-phase pump motors and blowers, proper operation requires A/C power be in correct phase. The sensor will detect an out of phase condition and send a warning/fault message to the PCS and SPU and then written to the system error log.

System shutdown would probably occur due to insufficient pressure, coolant flow, air flow, etc. If these associated conditions did not occur, and a manual check of input power found proper power phasing, replacement of this unit would be indicated.

2.4.4.4 Coolant Pumps

If a pump failure is detected, the logic module will automatically switch to the standby pump. Switches are also provided to force a manual switch over to either pump. This feature can be used to test newly installed pump prior to leaving the site.

A timer circuit is included on the logic board to initiate a pump switch-over every 400 hours. The switch-over equalizes the run time on both pumps, and insures long-term reliability of both units. Pump switchovers are logged through the PCS to the SPU and system error log. This information can be tracked by system-based SDD to insure proper operation of the timer circuit.

2.4.4.5 Blowers

Dual 3-phase blowers are used to eject the heat load dissipated in the heat exchanger coil. Each blower handles approximately 50% of this load. Air flow switches are provided to monitor blower operation. Air temperature across the exchanger coil is also measured. In case of blower failure, a signal is sent to the PCS which in turn will be logged in the system error log and ultimately generate a service call.

The system will continue to operate on the remaining blower. However, a gradual rise in IC junction temperature of approximately 5 - 7 degrees C will occur. The defective blower can be replaced, while the system is operating, by removing A/C power and the power connector from the blower, and removing the unit from the cabinet.

3
CPU Subsystem Overview

3.1 Chapter Objective

The chapter objective is to provide a functional and operational overview of each scalar processor functional unit: IBox, EBox, and MBox. The major features and functional units of each box will be introduced together with an operational summary of each unit. .s Note that the VBox is not presented in this chapter. The VBox and vector processing concepts will be distributed as a supplement prior to the prototype power on milestone.

3.2 IBox Introduction

The IBox is an independent functional unit that fetches and decodes instructions and their specifiers from the MBox and passes them to the EBox for execution. The EBox is provided with data and control functions such that it can execute many instructions in one cycle.

More instruction handling functions have been assigned to the IBox in this system than in any previous VAX system. The system objective is to maintain the IBox ahead of the EBox and allow all EBox processing resources to operate at capacity. The overall result is more parallel processing, and a lower cycles per instruction (CPI).

The major functions of the IBox are to:

- prefetch instructions by retrieving instructions from the MBox cache
- decode opcodes to determine the instruction operation code
- decode operand specifiers to determine the instruction access and data types
- handle operands by sign extending immediate mode operands or zero extending short literals, as well as fetch operands
- update the program counter (PC)
- source operand queuing by passing source operands and associated pointers to the EBox source queues
- destination address queuing by passing memory destination addresses to the MBox
- destination pointer queuing by passing GPR and memory destination pointers to the EBox.

The IBox provides all required instruction data to the EBox (i.e., source, destination, PC, fork address as well as pointers to the source data and destination). The instruction data is stored in the EBox source and destination queues or GPRs. In the case of a memory source operand, the IBox generates the operand address and passes it to the MBox requesting a memory read operation. The MBox prefetches the data and then writes it into the EBox source queue.

Using the pointers the EBox accesses its source list and executes the instruction. Depending on the destination, the EBox writes the results to a GPR or transfers the results to the MBox to be subsequently written to memory. In effect, the EBox only deals with data (i.e., no op codes or operand specifiers).

In addition, the IBox has the capability to decode and store several instructions ahead of the EBox. However, given the capabilities of the EBox it is difficult for the IBox to remain several instructions ahead.

3.2.1 Hardware Implementation

The following subsections describe the major new hardware implementations incorporated into the IBox. (See Figure 3–1.)

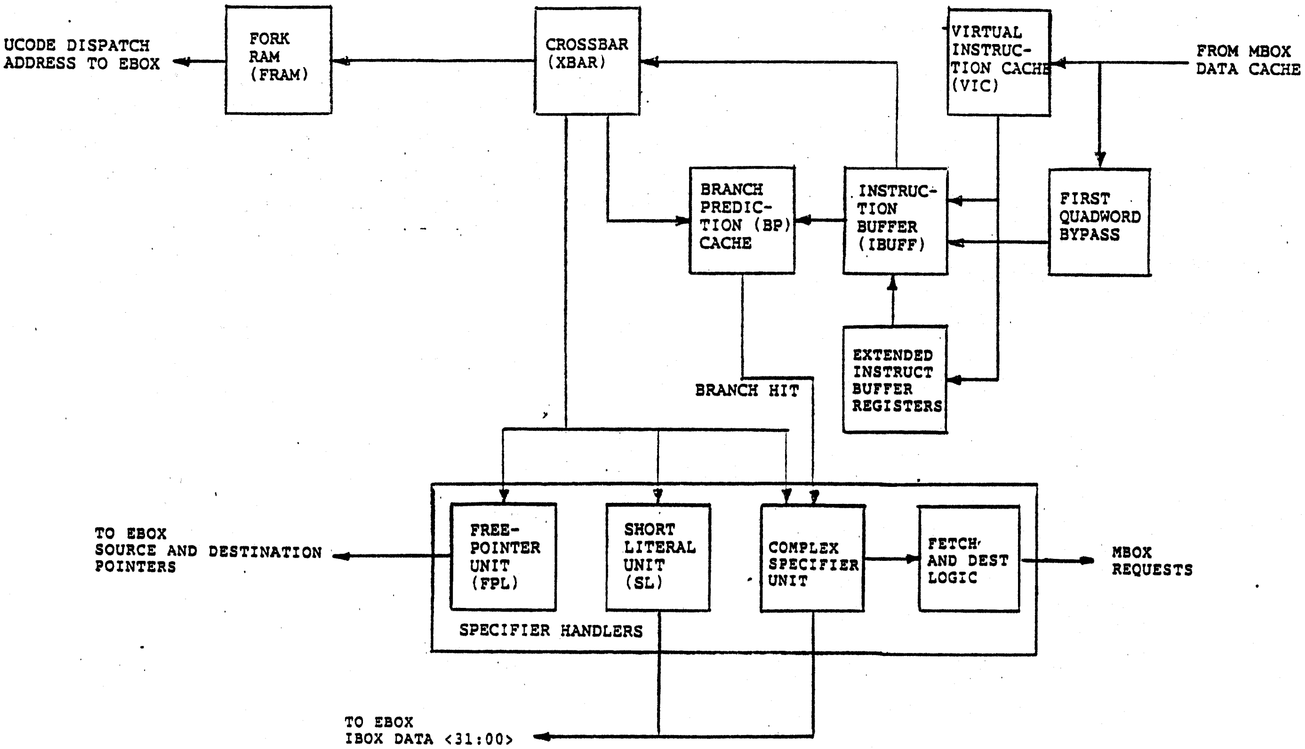

UCODE DISPATCH ADDRESS TO EBOX
FORK RAM (FRAM)
CROSSBAR (XBAR)
VIRTUAL INSTRUCTION CACHE (VIC)
FROM MBOX DATA CACHE
BRANCH PREDICTION (BP) CACHE
INSTRUCTION BUFFER (IBUFF)
FIRST QUADWORD BYPASS
EXTENDED INSTRUCT BUFFER REGISTERS
BRANCH HIT
TO EBOX SOURCE AND DESTINATION POINTERS
FREE-POINTER UNIT (FPL)
SHORT LITERAL UNIT (SL)
COMPLEX SPECIFIER UNIT
FETCH AND DEST LOGIC
MBOX REQUESTS
SPECIFIER HANDLERS
TO EBOX IBOX DATA <31:00>

Figure 3–1 Basic IBox Block Diagram

3.2.1.1 Virtual Instruction Cache

The Virtual Instruction Cache (VIC) is a virtually addressed, 8K byte, direct mapped, one-way associative cache, which reduces the number of Istream requests issued to the MBox. By flushing the VIC on every REI, the IBox can ignore writes to memory. Having a virtually addressed instruction cache eliminates the need for address translation and translation logic. Since the VIC has its own MBox request port, it is refilled from the MBox data cache, rather than memory.

3.2.1.2 Branch Prediction Unit

The Branch Prediction Unit (BPU) detects branch instructions, predicts branch direction, and redirects the Instruction Buffer to fetch instructions from the new instruction stream (Istream). That is, if it predicts that the branch is to be taken. The BPU incorporates a history cache of previous branch predictions and target PCs.

3.2.1.3 Branch Prediction Cache

To minimize the idle time spent flushing and refilling the pipeline after every branch instruction, the Instruction Fetch stage includes a Branch Prediction Cache (BPC). This 1k virtual cache increases performance by storing information about the branch validity, and the target address. As a branch instruction is being decoded it is referenced, in parallel, in the BPC and a prediction is made whether to take it or not. The IBox uses the cached target address to redirect the instruction fetch stage to the new Istream if the branch is taken. For performance reasons the BPC is not flushed.

3.2.1.4 Extended Instruction Buffer

The IBox incorporates a 25-byte Instruction Buffer (IBUFFER) which decodes the Istream. The IBUFFER is partitioned into a 9-byte Instruction Buffer (IBUF), 8-byte Extended Instruction Buffer (IBEX), and an additional 8-byte Extended Instruction Buffer (IBEX2).

The nine IBUF bytes satisfy all VAX instructions and their addressing modes. The content of the nine IBUF bytes are latched, decoded, and shifted. The remaining 16 bytes of IBEX and IBEX2 contain additional prefetched Istream from the VIC which is passed to the IBUF as required.

NOTE

The term IBUFFER refers to the entire Instruction Buffer which includes the IBUF and both IBEXs.

3.2.1.5 Multiple Specifier Decode Unit

The Multiple Specifier Decode Unit (Crossbar - XBAR) is implemented as a set of multiplexers which provide the capability of simultaneously decoding up to three operand specifiers. The XBAR consumes the Istream data from the IBUF. The Istream is presented to the XBAR nine bytes at a time.

The actual number of specifiers decoded depends on the specifier type. The XBAR can decode up to three specifiers (e.g., two simple and one complex, three simple, etc.). Simple specifiers are considered register mode or short literal, while all other specifiers are considered complex.

3.2.1.6 Specifier Handlers

The Specifier Handlers are contained in the Operand Processing Unit (OPU). The logic is comprised of three dedicated specifier handling units: Complex Specifier Unit, operand fetch and destination logic, Short Literal Unit (SL), and the Free Pointer Logic (FPL).

In addition, the OPU maintains a set of GPRs. The GPRs are implemented in Self Timed Register Files (STREGs) which provide multi-ported, read/write access capabilities. .s The OPU has its own virtually addressed MBox request port.

3.2.2 Physical Structure

As shown in Figure 3–2, the IBox logic is physically contained in three MCUs. The following subsections introduce each MCU and its related MCAs. Figure 3–3 correlates major MCA functions with the related MCUs.

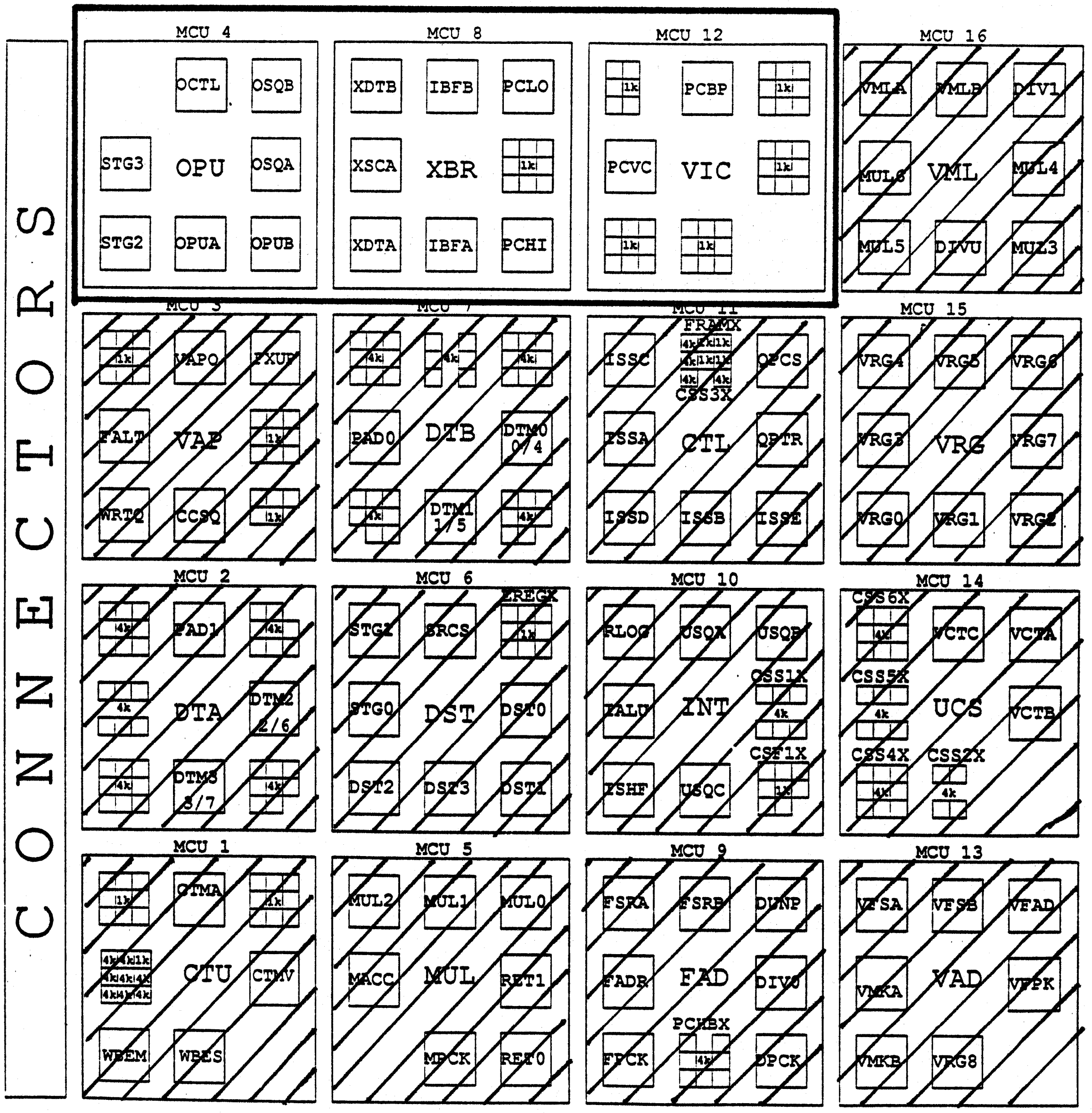


Figure 3–2 IBox MCU/MCA/RAM Placement

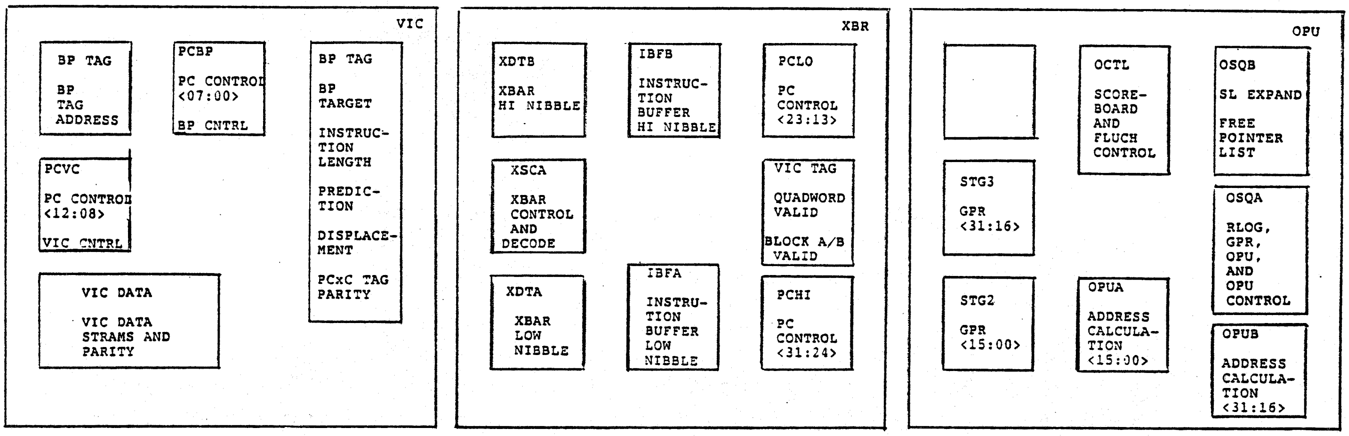


Figure 3–3 MCU/MCA Placement

3.2.2.1 VIC MCU

The VIC MCU is comprised of two MCAs and 42, 1K by 4 STRAMS. Two of the bit slices of the Program Counter (PC) and the data STRAMS for branch prediction and VIC are resident in this MCU. The following list introduces the MCA and STRAM functions:

- PCBP MCA - Program Counter/Branch Prediction Control MCA contains bits <12:06> of the PC.
- PCVC MCA - Program Counter/VIC Control MCA contains bits <05:00> of the PC.
- VIC Data STRAMS - these 18, 1K by 4 bit STRAMS are dedicated to the VIC data and associated byte parity.
- Primary Branch Prediction STRAMS - these 24, 1K by 4 bit STRAMS are dedicated to the branch prediction function. The branch PC tag, prediction PC, branch instruction length, and the prediction bit are stored in these STRAMS.

3.2.2.2 XBAR MCU

The XBAR MCU contains the IBUFF, XBAR, two of the four PC slices, and STRAMS that store the tag field for the VIC. The following list introduces each MCA and STRAM function:

- PCLO MCA - Program Pointer Low MCA contains bits <23:13> of the PC
- PCHI MCA - Program Counter High MCA contains bits <31:24> of the PC
- IBFA MCA - Instruction Buffer A contains the low-order nibble of the IBUFF
- IBFB MCA - Instruction Buffer B contains the high-order nibble of the IBUFF. IBUFF parity checking is performed in this MCA by multiplexing partial parity from IBFA.
- XDTA MCA - Crossbar Data A contains the low-order nible of the XBAR. The major MCA outputs are displacements for the Operand Processing Unit (OPU).
- XDTB MCA - Crossbar Data B contains the high-order nibble of the XBAR data path.
- XSCA MCA - Crossbar Control MCA is the XBAR control unit. It receives Istream data from the IBUFF and performs some simple instruction decoding. The IBUFF shift control is generated from the number of specifiers decoded and the number of specifiers the instruction contains.

- VIC STRAMS - Five of the nine, 1K by 4 bit STRAMS contain bit <31:13> of the VIC tag. Two of the STRAMS provide bits <03:00> and parity for the VIC quadword valid bits. The remaining two provide the VIC block valid STRAMS.

3.2.2.3 OPU MCU

The OPU MCU contains the logic responsible for the specifier decode process. The operand port interface to the MBox also resides in this MCU. The MCU also contains a pair of Self Timed RAMs (STREGS) which provide the IBox GPRs. The following list introduces each MCA and STRAM function:

- OPUx MCA - OPUA and OPUB MCAs provide the OPU data path (OPU A provides the low-order word; OPU B provides the high-order word). The OPU receives up to 32 bits of displacement from the XBAR. The OPU output can be directed to the MBox, EBox, or looped back to the OPU.
- OSQA MCA - This MCA provides control for the GPR STREGs. The OPU A and B multiplexers (i.e., AMUX; BMUX) are also provided by this MCA.
- OSQB MCA - This MCA receives short literal (SL) data from the XBAR, expands it into the correct context and passes it through the OPU to the EBox.
- OCTL MCA - The Operand Control MCA is responsible for control of the read and write masks generated by the XBAR.

3.2.3 Pipeline Stages

The Ibex consists of three main pipeline stages which closely relate to the MCU physical Structure. Each pipe stage can generate an error indicator to aid in isolating to an MCU FRU. The three pipeline stages are:

1. Instruction Fetch Stage consists of the logic involved in fetching the Istream before it is latched in the IBUFFER, and includes the:
   - VIC
   - Branch prediction logic
   - Program Counter logic (PC Unit)
   - IBEXs and a portion of the IBUF
   - IBUFF to MBox interface
2. Instruction Decode and Branch Prediction Stage consists of the logic involved in decoding the instruction in the IBUF, and includes the:
   - Crossbar (XBAR)
   - Branch Prediction Unit (BPU)
3. Specifier Decode Stage consists of the Operand Processing Unit (OPU) logic which is involved in the evaluation of operand specifiers, and includes the:
   - Operand fetch and destination logic
   - Free Pointer Logic (FPL)
   - Short Literal Unit (SL)
   - Operand Control Unit (OCTL)

- OPU to MBox interface
- OPU to EBox interface

Since the IBox is pipelined, when a stage cannot complete its operation, previous stages must be stalled. That is, previous stage operations are halted.

3.2.4 IBox Interfaces

The IBox interfaces to each of the other CPU functional units through dedicated interfaces (or ports). Each interface is described in the following subsections.

3.2.4.1 MBox Interface

The IBox interfaces to the MBox through two ports located in the MBox:

- Instruction Buffer Port (IBUF Port) - requests data from the MBox when a miss is encountered in the VIC
- Operand Processing Unit Port (OP Port)- used to request memory related operands from the MBox, and passes virtual addresses of operands which specify memory destinations to the MBox.

IBUF Port The IBUF Port is a read-only port to the MBox. The IBox uses the IBUF Port to issue requests to the MBox for Istream data - 64 bits at a time with byte parity. The Istream is retrieved from the MBox Cache. In the case of a cache miss, the request is forwarded to memory through the SCU. Typically a request is for four (aligned) quadwords to fill the VIC. Requests are initiated by the IBUF on a VIC miss.

OP Port The OP Port is a read/write operand access port to the MBox. The port has a 32-bit wide data path with byte parity. Any rotation of the data (i.e., justification), is performed by the MBox.

The IBox uses the Op Port for three reasons:

- operand prefetch from cache or memory on behalf of the EBox/VBox
- queuing of addresses for operands destined to cache/memory
- prefetching operands that are address deferred (indirect) from cache or memory

The IBox issues requests on behalf of the EBox/VBox for operands that come from the MBox cache or memory. The operands are passed directly from the MBox to the EBox Source List Queue.

The IBox sends the destination address to the MBox. In turn the MBox performs a TB lookup, and stores the physical address in the Write Queue, and waits for the result data from the EBox/VBox.

For deferred addressing the MBox returns the address of the operand to the IBox for a successive fetch for the data (operand). The data operand is returned to the EBox/VBox.

3.2.4.2 EBox Interface

The IBox interfaces to the EBox through the Queue Functional Unit port located in the EBox. This unit contains a set of FIFO buffers (queues) which accept instruction control information and operands from the IBox.

IBox to EBox Interface This interface is used to send operands to the EBox 32 bits at a time with byte parity and their respective pointers. These are operands that are handled by the Operand Processing Unit (OPU) within the IBox (i.e., sign or zero extended data, integer and floating short literals, immediate mode data, etc. The source and destination pointers are maintained in queues within the EBox, and allow the EBox to access the proper data.

In addition, control information is passed to the EBox for microcode control, RLog information, program count, and errors.

EBox to IBox Interface This interface is used to transfer:

o result data

o a starting or flushing PC

o RLog unwind data

o control information: Branch Valid, Queue Full, Keep Masks

This interface transfers 32-bit EBox result data (including byte parity) to the IBox. The result data is generally an operand that has a destination of a GPR. The data, including the byte parity, is written to the IBox GPR set.

In addition to the various control signals, the EBox result data may also be of a controlling function. For example, an address passed to the IBox to initiate instruction fetch.

3.2.4.3 VBox Interface

The VBox issues requests for operands through the IBox. It sends the address (32 bits) together with byte parity and control data. Control data is the reference data size, type of reference (read or write), and, whether it is a block read or not.

3.2.4.4 Scan Interface

Scan latches are strategically located throughout the IBox data and control paths, and error logic for error detection and recovery,. The SPU can retrieve and store the state of the IBox scan logic for error reporting and possible recovery. Field Service can then perform analysis on the stored error symptom data for identification of the failing FRU.

The scan paths can implement symptom directed diagnosis (SDD) as well as test directed diagnosis (TDD). The scan latches have scan data/clock inputs and the normal system data/clock inputs. The scan inputs can be controlled by the SPU and diagnostics to shift test patterns in, and test for the correct pattern that is subsequently retrieved.

3.2.5 Operational Summary

The following subsections provide a hardware overview and operational summary of each major functional unit. Refer to Figure 3-4.

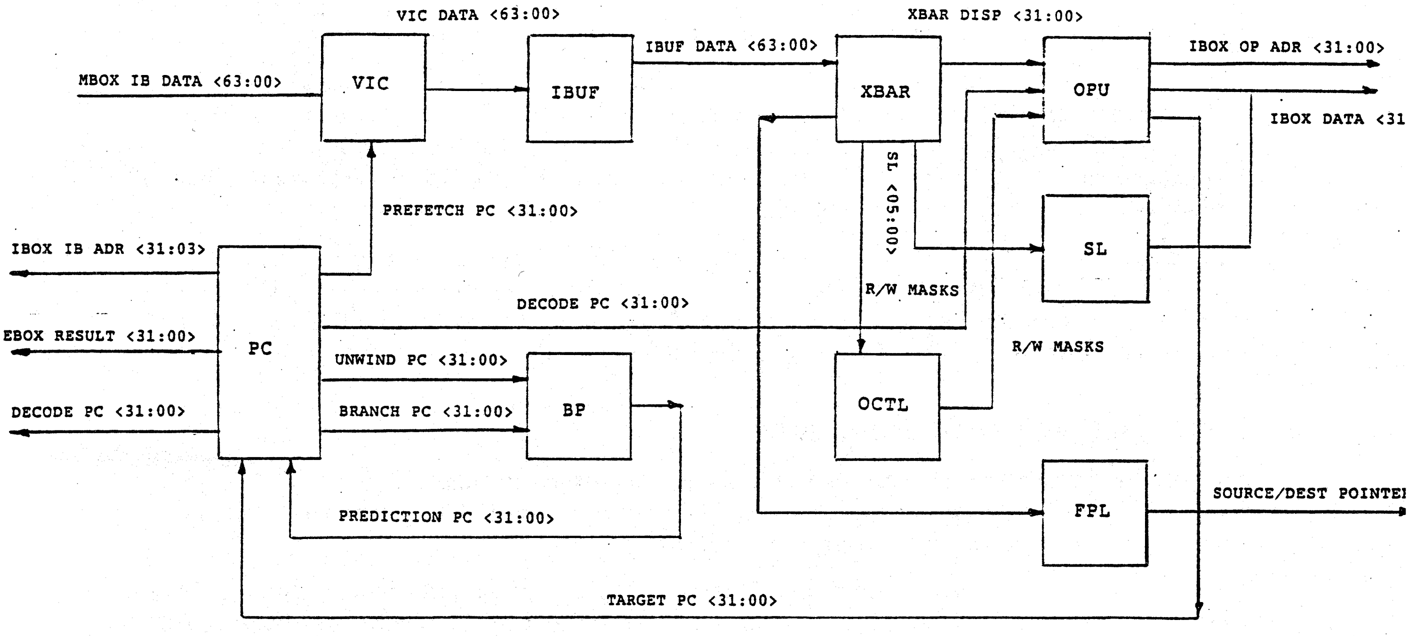


Figure 3-4 Second-Level Block Diagram

3.2.6 PC Unit

The PC Unit (PCU) is responsible for directing the Istream that the IBox will process (i.e., fetch, decode, branch, unwind (flush), etc.). The PCU consists of four MCAs':

- o PCBP: Program Control and Branch Prediction Control (VIC MCU)
- o PCVC: Program Control and Virtual Instruction Cache Control (VIC MCU)
- o PCLO: Program Control and PC Low Slice (XBR)
- o PCHI: Program Control and PC High Slice (XBR)

As indicated by the content of the MCAs, the PCU contains the control logic for the two IBox caches: VIC, and Branch Prediction Cache (BPC). The PCU also controls the Secondary Branch Prediction mechanism. (Refer to Figure 3-3.)

The PCU generates the address or PC used in the E and M boxes. The IBox operates on four PCs: Prefetch, Decode, Branch, and Unwind.

Prefetch PC: The Prefetch PC is used during instruction fetch to address the VIC early in the cycle. If there is a VIC miss, a request is issued to the MBox for the Istream using the Prefetch PC sent out late in the cycle. The Prefetch PC is always an address pointing to the next byte following the last valid byte in the IBUFF.

Decode PC: The Decode PC is used by the EBox and the OPU. It is the address or PC of the instruction whose Op Code is currently under evaluation in the IBUFF. The EBox stores the Decode PC in a queue until it executes the instruction. At that point, it is loaded into the PC History Buffer. (The PC History Buffer will aid Field Service track the macro PCs executed for error analysis.) The Decode PC is used by the OPU to handle relative addressed operands, branch displacements, and implied specifiers; (e.g., the return PC for the BSBB - Branch to Subroutine with Byte Displacement instruction).

Branch PC: The Branch PC is used by the PCU to address the BP. It is the virtual address of the branch instruction the PCU is processing. When the IBox encounters a branch instruction in the IBUF, the Decode PC is latched and used as the Branch PC. The PCU can process a second branch instruction concurrently as in the case of nested branches. It saves the PC as the Second Branch PC.

Unwind PC: The Unwind PC is used when the IBox generates an incorrect branch prediction to restore the correct Istream. When the IBox predicts a conditional branch to be taken or not, it stores up to two Unwind PC(s). The EBox will notify the IBox of an incorrect prediction and the IBox will recover (or unwind) using the Unwind PC.

3.2.7 Virtual Instruction Cache

The VIC is direct mapped and 8K bytes deep. As shown in Figure 3–5 the VIC block size is 32 bytes (4 quadwords) with a valid bit per quadword. VIC access is 8 bytes (quadword), whether for read or refill operations.

By providing a quadword in the first VIC fetch, all required Istream data is often retrieved in the first cycle. (The average VAX instruction length is 4 bytes.) With the VIC bypass enabled on refill operations, the first quadword is sent directly from the cache to the IBUF.

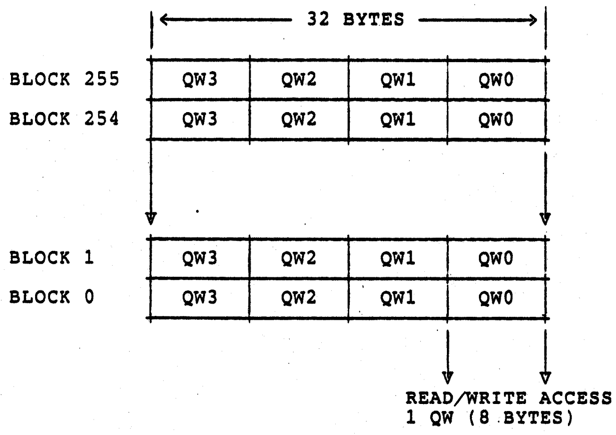


Figure 3–5 Basic VIC Structure

The IBUF Port has a 64-bit (quadword) wide data path, with byte parity. IBUF requests are issued when the it is empty and there is a VIC miss. The requests are generally for four quadwords (i.e., VIC block size). VIC data output (64 bits + byte parity) is passed to IBEX2.

The VIC Tag STRAMS, located on the XBAR MCU, contain the virtual address tag, tag parity, block valid bits, quadword valid bits, and quadword valid parity. The high-order address (Prefetch PC 31:13) from the PCHI and PCLO MCAs is matched against the VIC Tag (31:13) together with the block and quadword valid bits to determine if there is a VIC hit. In the case of a miss, an IBUF request is issued to the MBox for the required Istream. (See Figure 3-6.)

VA TAG	TAG PARITY	QUADWORD VALID BITS				QUADWORD PARITY	BLOCK VALID BITS	
<31:13>	<03:02>	QW3	QW2	QW1	QW0	PARITY	VICA	VICB

Figure 3-6 Tag Store Format

Since the VIC is flushed on each context switch, two sets of block valid bits are maintained in the Tag STRAMS. A block is comprised of four quadwords where the associated valid bit indicates that particular Istream block is valid. Since one set is usually clear, the VIC can be switched to the cleared set without waiting for the invalid set to be cleared. This switching action speeds the flush operation.

Parity is checked on the VIC output data at the input to the IBUF logic at the output of the IBEX logic. Partial parity is sent from IBFA MCA to the IBFB MCA. IBFB generates the IBEX PARITY ERROR signal, and IBFB Fetch Error. VIC Tag Parity, Quadword Valid Parity, and Block Valid Parity are checked by the IBFA MCA. All generate IBFA VIC Error sent to IBFB which generates IBFB Fetch Error.

3.2.8 Instruction Buffer

The IBUF is a 25-byte FIFO buffer and is partitioned between two MCAs: IBFA and IBFB. The MCAs control the low and high nibble respectively, and are located on the XBAR MCU.

The IBUF contains the nine bytes of Istream that will be presented to the decode units of the IBox. Byte 0 of the the IBUF contains the opcode being decoded. As the specifiers are decoded they are shifted out of the IBUF. The remaining bytes are shifted down and the empty byte positions are replenished from the VIC or the extension registers (IBEX and IBEX2). When all specifiers have been decoded, the opcode is then shifted out. (See Figure 3-7.)

IBEX and IBEX2 are quadword buffer registers between the VIC and the IBUF. The IBUF will attempt to replenish any invalid bytes in the IBUF with valid data from the extension registers.

The IBUF contains a shifter. The shifter operates in only one direction, shifting high-order bytes into low-order byte positions. In addition, the shifter holds byte 0 (opcode position) while shifting the other bytes.

As the Istream is decoded, the Crossbar (XBAR) determines the number of specifiers, and the number of specifiers decoded in each cycle. A shift count and a determination of when to shift out the opcode is derived from this information.

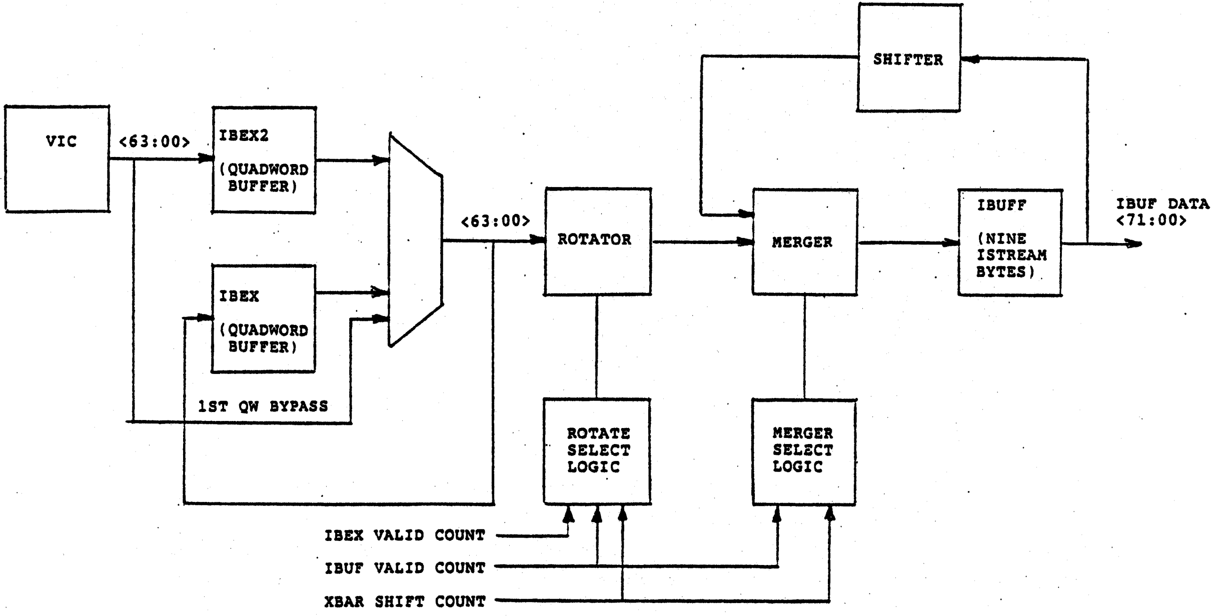


Figure 3–7 Basic IBUF Structure

The rotator of the IBUF aligns the data of the VIC, IBEX, and IBEX2 so that it will be correctly aligned in the IBUF. The rotator uses the IBEX and IBEX2 valid counts, IBUF valid count, and the shift count to determine the correct rotate select function. The rotator selects the data source and passes it to the Merger.

The merger receives data from the rotator and shifter and passes the data to the IBUF. The merger select logic determines where each byte is loaded in the IBUF. If, at the completion of a cycle:

the IBUFF contained invalid data in bytes 1 and 2, and
the IBEX contained valid data, then

the merger select would be for both shifter and rotator data. The shifter would align the remaining IBUF valid bytes (8 - 3) into byte positions 6 - 1, and the rotator would rotate two valid bytes into positions 8 and 7.

The first nine bytes of the IBUF can be shifted as bytes are processed. The remaining 16 bytes, which are effectively prefetched data, can only be shifted into the low-order eight bytes, and not shifted internally. The low-order byte always contains the opcode, when valid.

The IBUFFER also request blocks of Istream on behalf of the VIC.

The IBUF output is parity checked at the output of the IBUF logic. Parity is also checked at the XBAR input.

3.2.9 Instruction Decode

The IBUFFER directs Istream data to the opcode and specifier parsing unit (Crossbar - XBAR). The XBAR attempts to decode an opcode and up to:

- three simple specifiers, where simple is defined as register or short literal, or
- two simple and one complex specifier, where complex is defined as all specifiers other than register and short literals.

The XBAR will only process specifiers of a single instruction concurrently. It passes the decoded specifiers to the specifier handlers in the OPU. Register operands are passed to the Freepointer Unit (FPL), short literals are passed to the Short Literal Unit (SL), and immediate operands and memory specifiers are passed to the operand fetch and destination logic.

A control unit (Operand Control Unit - OCTL in the OPU MCA) produces miscellaneous control functions and various flush signals and maintains the scoreboard masks for the OPU. In addition, branch instructions are decoded in the XBAR. Opcode and displacement data are passed to the OPU for calculating the new PC. <Refer To Figure 3–4.)

The XBAR is physically located on three MCAs on the XBR MCU. The XDTA and XDTB MCUs provide the bit slices of the data path, while the XSCA MCA provides control for the XBAR. As shown in Figure 3-x, the primary input to the XBAR is Istream from the IBUF, with outputs to most of the other IBox functional units.

When the XBAR receives Istream from the IBUF, it decodes the instruction to determine the addressing mode and the number of specifiers it contains. Internally, a specifier count and the number of specifiers are tracked. These signals are used to supply shift count and shift opcode signals for the IBUF. The opcode and extended bit are passed to the EBox to address the Fork RAM (FRAM). The FRAM contains the EBox microcode dispatch addresses (fork addresses).

To maintain the integrity to the GPRs, the XBAR directs read and write masks to the OCTL unit. The OCTL maintains the masks for detecting intra-instruction conflicts. The XBAR passes a register field, register valid bit, and a valid bit to the FPL for generating the EBox source and destination pointers.

The XBAR determines the addressing mode to be used. For example, if the addressing mode is indexed, short literal, or register, the XBAR will pass the data to the appropriate specifier handler.

For GPR addressing, the XBAR decodes the register, and generates the appropriate pointers and masks to the OPU. Pointers are the Source and Destination register addresses. The masks specify the registers being read or written by the EBox for that instruction.

If memory addresses are used, conflicting write/read checks are performed by the MBox. In this case, the pointer indicates that the operand is coming from the MBox and will not be found in a GPR. The pointers are sent to the EBox from the OPU and identify the operand locations.

Masks are used to detect a conflict between the IBox and EBox. These masks are used by the OPU to prevent the EBox from using the wrong data, or the IBox from using stale data. Two conflict cases are described below:

o the IBox is about to autoincrement a GPR just before the EBox reads that GPR for an operand it requires,

- o the IBox is about to read a GPR to calculate an address, however the EBox is just about to write the results of a previous instruction to the same GPR.

The XBAR also generates control information, such as; IBUF shift counts, counts to be used to generate the Decode PC and PC of the specifier being processed, and Specifier PC.

3.2.10 Branch Prediction

While the IBUFFER presents an instruction to the XBAR it also sends it to the Branch Prediction Unit (BPU). The BPU detects branch instructions, predicts the branch direction, and redirects the IBUFFER to fetch instructions from the new Istream - if it predicts the branch is to be taken. (Refer to Figure 3–4.)

Branch prediction is performed in three ways:

1. Through the Branch Prediction Cache (BPC) which stores the target addresses of up to 1K recently executed branch instructions. The current branch instruction PC is used to address the cache. If there is a hit, the history bit is checked to determine the previous branch direction, and the cached target PC is used for the new PC.
2. If there is a BPC miss, an opcode-dependent bias is used to determine branch direction. The new value is then written into the cache.
3. Unconditional jump and loop control instructions are always predicted to be taken.

The Ibex will continue processing the predicted Istream until it:

- o encounters another branch instruction
- o encounters an autoincrement or autodecrement specifier
- o is informed by the EBox that the prediction was wrong.

In the last case, processing performed on the incorrect Istream is flushed from the system.

3.2.10.1 Branch Prediction Cache

The BPC is a 1k deep cache designed to minimize the idle time spent to refill the pipeline when a branch instruction is decoded. The BPC target address content allows a branch to be predicted early in the cycle by maintaining a validity history of previous branch instructions. Thus, the BPC provides a significant performance improvement by having the branch target address calculated and cached for quick recovery, coupled with the fact that the BPC is never flushed.

The BPC is addressed by the Branch PC. This is the address of the current branch instruction under decode by the XBAR (i.e., Decode PC). The low-order 10 bits address the BPC. The remainder of the address is compared against a prediction tag to determine a hit. If there is a hit, the output (prediction PC) is passed to the PCU to address the VIC for the new Istream. In the case of a VIC miss, the Prediction PC is passed to the MBox together with an IBUF request.

The BPC stores the Prediction Tag bits <31:10> which are compared against the Branch PC to determine a hit. The BPC also stores the Prediction PC bits <31:00> (branch target address). The Prediction PC is used to address the VIC as the new instruction stream address.

The branch displacement (Tag Displacement bits <15:00>), and an instruction length (Tag Instruction Length <05:00>)are also stored. The displacement and length tags are compared against the current branch displacement and length to determine further if the information is valid for the current branch instruction.

Table 3-1 summarizes the BPC fields:

Table 3–1 BP Cache Fields

Cache Field	Description
Prediction Tag	22-bit address matched against the current branch address. Parity (3 bits) protects the Tag.
Prediction PC	32-bit branch target address. Parity (4 bits) protects the Prediction PC
Prediction A,B,C,D	4-bit copy of History bit, determines whether to take branch or not. A 1= prediction is to take
Tag Displacement	16-bit branch offset value. Compared against current branch instruction's displacement. Determines Demote. Parity (2 bits) protects the Tag Displacement
Tag Instruction Length	6-bit count of branch instruction length. Compared against current branch instruction's length. Determines Demote. Parity (1 bit) protects the Tag Instruction Length.

3.2.10.2 Branch Prediction Modes

There are two modes of operation for the branch prediction mechanism: primary and secondary. Primary mode is used when there is a hit in the BPC and it's prediction bit determines the branch prediction. Secondary mode is entered when there is a miss in the BPC, or a demote situation exists.

A demote situation exists when the displacement and instruction length of the current instruction do not match the cached displacement and length tags. Although the virtual PCs of the branch instructions match, they are not the same instructions (because of the tag mismatches) and therefore the information is invalid.

Secondary mode uses a fixed bias, with the bias based on the opcode or type of branch instruction. That is, each predicable branch instruction has a fixed bias. For example, a BEQL instruction usually tests two conditions for equality. In mathematical situations equality is a rare occurrence, implying that the BEQL would not be taken. However, the reverse applies to the BNEQ instruction. Usually the condition is more not equal, therefore it is predicted that the BNEQ will be taken.

The previous analysis is applied to all branch instruction opcodes. Each analysis result (or bias) is loaded into the Bias RAM (BRAM). The BRAM is located on the XBAR MCU and is addressed by the opcode from the IBFA and IBFB MCAs. The Target PC is provided by the OPU.

3.2.11 Operand Processing Unit

The OPU receives control information from the XBAR, and generates virtual addresses for memory sources and destinations. The OPU can process most memory operands in a single cycle. In addition, the OPU maintains a copy of the GPRs for address calculation. (Refer to Figure 3–4.)

The OPU performs the autoincrement and autodecrement functions. These changes are written into the Register Log (LOG). The LOG provides the data to allow the GPR content to be reconstructed when the EBox is interrupted to handle interrupts and exceptions.

The OPU processes immediate operands, short literals, and register operands. That is:

- register operands and control functions are passed to the Free Pointer Logic (FPL), which produces the source and destination pointers that are subsequently passed to the EBox.
- short literals are passed to the Short Literal Unit (SL), which expands them for entry into the EBox Source List. The expansion performed depends on the specific data type.
- immediate operands, displacements, and memory specifiers are passed to the operand fetch and destination logic. This logic generates the virtual addresses for the memory source and destinations, and can process most operands in a single cycle.

Register and memory destination specifiers are also processed by the OPU. The OPU sends an entry to the Destination Pointer Queue in the EBox for instruction retiring for both destination types.

For read type specifiers, the OPU sends the read address to the MBox which returns the read data to the EBox through its EBox port. For memory write specifiers, the OPU generates the write address and sends it to the MBox. There it is translated to a physical address, and stored in a FIFO buffer (Write Queue). When the result data is received from the EBox, it is paired with the address at the top of the write queue, and written to memory.

Branch instruction target addresses are also calculated by the OPU.

3.2.11.1 Logic Overview

The OPU logic maintains a copy of the GPRs for address calculation. It contains a simple adder for branch target address calculations, auto-increment/decrement operations, and calculating the operand address for displacement mode. A context shifter (multiplier) drives the adder for index mode. An additional adder is used to calculate the PC for the current specifier that is used for relative addressing.

The OPU control (OCTL MCA) is responsible for generating composite masks. It receives mask inputs from the XBAR as it decodes the specifier. The masks indicate which GPR is being written or read by the EBox when it executes an instruction.

Since an instruction may contain up to six specifiers, the OCTL generates the corresponding mask for each specifier and passes them to the OPU control. The control MCA (OSQA) uses the masks to detect GPR conflicts.

3.2.11.2 Input/Output Summary

The following paragraphs describe the main data inputs and outputs of the OPU.

Input- XBAR Displacement <31:00>: The OPU receives the main data input from the XBAR Displacement signal lines. The XBAR passes the OPU the information 32 bits at a time with byte parity, when it decodes the specifier as complex. The OPU checks the data for the correct parity, then performs the necessary operation to provide the EBox with the operand. This is a simple OPU operation for immediate mode: passing the operand to the EBox by way of the IBox Data lines in a single cycle. It would take the OPU a number of cycles, if for the mode were byte displacement deferred indexed.

Input- STG2/3 X/YGPR <31:00> + Byte Parity: The GPRs are another source of data input for the OPU. The GPRs are STREGs located in the STG2 and STG3 MCAs. The OPU receives GPR content on the YGPR <31:00> data lines, together with byte parity (unless it is an indexed operation). When indexed mode is used the IBox receives the contents of the index register through the XGPR <31:00> with the byte parity.

Input- MBox Op Data <31:00> + Byte Parity: The MBox Op Port passes data back to the OPU only when the data is to be used as an address; that is, when the mode is deferred (or indirect). Otherwise the operands are passed from the MBox to the EBox.

Input/Output- OPUA/OPUB Result <31:00> + Byte Parity: The OPU can use the result of one operation as the input for the next operation. This is performed through the Result <31:00> output of the adder circuit looping back through the A and B multiplexers.

Byte parity is generated on the Adder <31:00> Result data output. The Result and parity is sent to the IBox GPRs to be written for any GPR modifications the IBox makes (i.e., autoincrement, autodecrement, autoincrement deferred).

Output- IBox Data <31:00> + Byte Parity: The IBox Data lines are the output of the adder (Result Data) passed through a different set of multiplexers and scan latches. IBox data lines carry Immediate type operands to the EBox. Byte parity is also generated and sent to the EBox.

Output- IBox Op Address <31:00> + Byte Parity: The IBox Op Port to the MBox receives the address (and parity) from the Result output of the adder. The Op Address is used for any complex specifier that requires an address calculation and a fetch from cache or memory for the operand, or the destination operand is for cache or memory.

The IBox prefetches operands on behalf of the E and V boxes through the Op Port and returns the data to them. The IBox also queues destination address to the MBox write queue through the Op port. The result data is tagged and sent from the E and V boxes, to be matched with the queued address. The only time the Op Address will carry a request for the IBox itself is when the Deferred (indirect) address mode is used.

Output- OPU Target PC <31:00> + Byte Parity: Another output of the adder result is the Target PC <31:00>, together with the byte parity. All branch destination addresses are calculated by the OPU. These are passed to the PCU to fetch the new Istream from the VIC and also written into the BPC.

Current PC <31:00>: A separate adder produces the Current PC from the Decode PC and the Specifier Decode Delta for relative addressing mode. That is, the current instruction PC in the IBUF under evaluation by the XBAR, added to the number of specifiers the XBAR has evaluated (delta or offset), equals the current PC to be used to calculate the PC Relative address of the operand. This result is looped back to the adder inputs and added with the byte, word, or longword offset to obtain the address of the operand.

3.2.11.3 Short Literal Operand Handler

The SL operand handler is responsible for zero expanding the 6-bit short literal (SL <05:00>) for an unsigned integer value of 0-63 coming from the XBAR. It is zero-expanded into the proper data context: byte, word, longword, quadword, or octaword, and passed to the EBox. The SL parity checks the short literal coming from the XBAR with parity that is combined with other control signals.

The SL also expands short literals for the floating point data types: F, D, G, and H ranging in value from 1/2 to 120. The literals are expanded into the proper floating data types (longword, quadword, quadword, and octaword respectively) as well as aligning fractions and exponents.

3.2.11.4 Free Pointer Logic

The FPL tracks the available (free) EBox operand queue (Source List) addresses, and associated pointers to those operands (Source Queue). The FPL establishes the correct SOURCE1 and SOURCE2 pointers for the operands that the EBox will use to execute the instruction. It also generates the proper destination pointer for the EBox to use to write the instruction's result.

The EBox stores the pointer information in queues. The EBox then extracts the pointers from the queues when it requires the operands instruction execution, or store the result.

3.2.12 Read/Write Scoreboards

The IBox processes specifiers while the EBox is executing previously decoded instructions. The IBox must be prevented from performing an address calculation which will depend on the result of a currently executing instruction.

This is accomplished by recording the register number to be written by the EBox, and matching that number against any register selected for use in an operand specifier. When a match occurs, the IBox will stall and wait for the result to be written before calculating the operand address.

The XBAR generates the read and write masks for conflict checking in the OPU. The masks represent General Purpose Register (GPR) 0 thru GPR 14. The masks prevent reading or writing a GPR that is scheduled to be modified by the EBox.

The XBAR maintains the read and write register scoreboards for up to six instructions in the pipeline. Both scoreboards contain 15-bit registers, representing GPRs 0 thru 14.

The read scoreboard tracks the GPRs designated to be read by an instruction. These GPRs cannot be written by the OPU for autoincrement and autodecrement until they are read by the EBox.

The write scoreboard tracks GPRs designated as destinations by instructions in the pipeline. These GPRs cannot be used by the OPU for address calculations until the instruction is retired by the EBox.

3.3 EBox Introduction

The instruction fetching and decoding operations performed by the I and M boxes allows the EBox to be dedicated to the execution stage of an instruction. The EBox has the capability of executing several instructions simultaneously. While most VAX processing units have dedicated hardware for integer, floating point, and multiply operations, this system is the first to have the capability to operate all processing units in parallel.

3.3.1 Pipeline Stages

The microcoded EBox has a three-stage pipeline:

1. the first is the Issue stage and is a cycle used to access source data and read the control store
2. the second is the Execute stage and consists of one or more cycles used by the functional execution units (FEUs) to calculate results and condition codes
3. the third is the Retire stage and is a cycle used to write the results to the specified destination, and update the condition codes.

All pipe stages can be active simultaneously. In all cases, instructions are issued and retired in the order received.

3.3.2 Hardware Overview

As shown in Figure 4-2, the EBox is organized into several independent functional units. The major functional units are introduced in the following subsections.

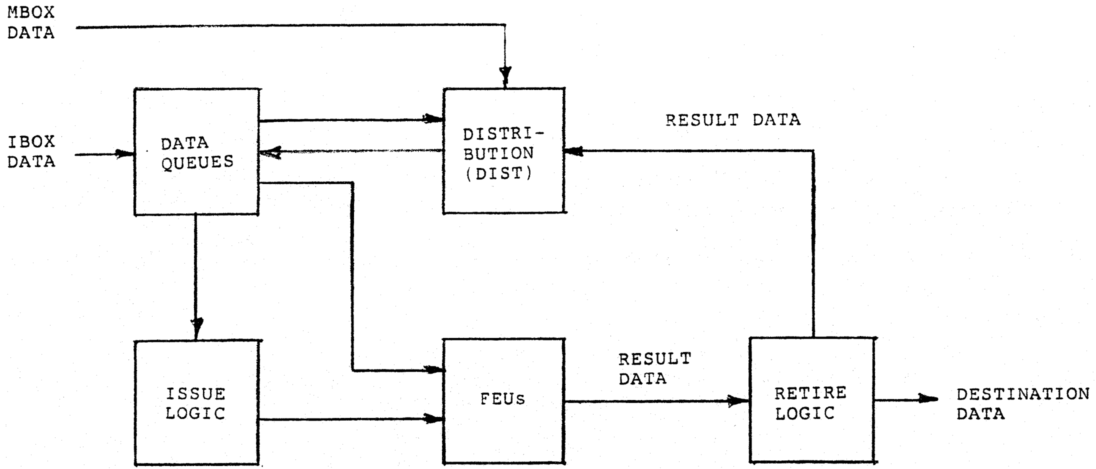


Figure 3-2 Basic EBox Functional Block Diagram

3.3.2.1 Instruction Data Queues

Control information and operands are transferred from the IBox (and in some cases the MBox) to a set of FIFO buffers (queues) in the EBox. The queues decouple (buffer) the EBox from the IBox. With each instruction, the IBox supplies:

- a microcode dispatch address (fork address) which is stored in the Fork Queue,
- a set of source operand pointers stored in the Source Pointer Queue, and
- a set of result store pointers, stored in the Destination Pointer Queue.

Four queues are maintained in the queue logic: fork, source, destination, and PC. The queues are loaded at the location designated by the associated queue load pointer. Since the source queue can accept two entries the load pointer and the load pointer +1 locations are loaded.

Fork Queue: The fork queue is a 17-bit wide by 8-location queue. The queue locations contain a 16-bit fork address field, and an IBox prediction bit. The bit fields are separated and distributed to the appropriate EBox logic.

Source Queue: The source queue is a 5-bit wide (field), by 16-location queue. The high-order field bit specifies a GPR location if set, and a source list location if clear. The low-order four bits specify the GPR or source list location.

Destination Queue: The destination queue is a 5-bit wide (field), by 8-bit location queue. The high-order field bit specifies a destination GPR or memory location. The low-order four bits specifies the GPR or memory location.

PC Queue: The PC queue is a 32-bit wide, by 8-location queue which stores the macro PCs. The microcode can access either the current PC or the backup PC.

3.3.2.2 Source List

The Source List is a Multiport Self-timed Register File (STREG). As shown in Figure 3–9 there are four sets, of 16 registers in the register file (STREG). The contents of the register file are:

- general purpose registers (GPRs) - including the three pointer registers (argument, stack, frame), and the PC work registers
- temporary registers (temps)- restricted to EBox microcode use. The content of the temps is always valid since the EBox has total control over them.
- source list - used by the IBox and MBox to transfer instruction data. The pointers into the source list are controlled by the IBox.
- memory access registers - contains memory data from the MBox on EBox-initiated read operations

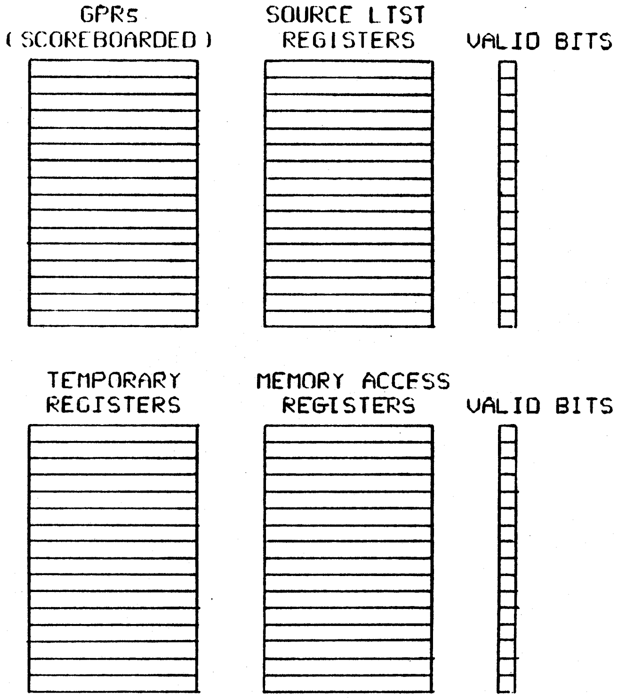


Figure 3–9 Simplified STREG Block Diagram

The source list and memory access register sets are somewhat different. The data in the source list is tracked closely. The IBox controls which location it will write. The IBox also informs the MBox which location to write. When either box supplies the data, it is marked with a valid bit (i.e., verifying that the data was written). However, the source list data can be used only once. After the EBox accesses a particular location, it informs the IBox that the data has been used. The IBox can then reallocate that location for another write.

In the case of the memory access registers, a location can be read many times. Its valid bit is cleared when the EBox requests that the MBox write a register location. When the MBox data arrives, the valid bit is set.

3.3.3 Issue and Retire Control

The Issue Functional Unit (ISSUE) performs overall control of the EBox pipeline. That is, it controls the issuing of an instruction to one of the FEUs, and the completion of the instruction through the Retire stage. Control is maintained in conjunction with the microsequencer and microcode which direct the specific execution functions

Instructions are issued when the source operands are valid and available from the STREG, and the required FEU is available. Register operands are checked for pending writes in the Destination Queue before the instruction is issued. At any one time, several instructions may be executing in the EBox in various stages of completion.

The Issue Control logic consists of two tightly coupled units:

o Issue - which controls the starting of an instruction execution

o Retire - which controls the completion of an instruction execution.

Both issue and retire can be initiated on every cycle.

3.3.3.1 MBox Memory Interface

The control logic also has a memory interface to the MBox. The interface implements two types of read and write operations from the EBox to the MBox.

The first is an Opwrite request. In this type of request the IBox informs the MBox that the EBox will eventually write to a particular memory location. In addition, the IBox passes the EBox write address to the MBox; the write address is not sent to the EBox. The MBox will queue the requests (in the Write Queue).

When the EBox completes the operation, the Retire Unit sends the write data to the MBox. The MBox pops the address from the Write queue. If there is a match with the EBox write data, it is written into the cache. The advantage of an opwrite is that the MBox will perform the address translation in parallel with the FEU operation. Thus, when the write data is available it can immediately be written into cache.

In the other types of read and write operations, the EBox generates the address. There is an interface that allows the EBox to send addresses to the MBox to perform these read and write operations. However, these are not as efficient as the opwrite. For example, if a read operation is initiated the EBox must wait for the MBox to return the read data.

3.3.3.2 Fault handling

The Issue logic validates the instruction data from the IBox and MBox. The boxes flag the data as good or bad. For example, if a particular location page faulted, that data would be marked with a page fault indicator. When the data is selected it would be determined that it was not valid. At that point, the microsequencer traps to the page fault handler. Reserved addressing faults are handled in the same basic way.

Other faults (e.g.,integer overflow type traps, floating underflow faults) are handled in a slightly different way. The operation result is retired and the FEU informs the microsequencer of the trap or fault condition. The microsequencer then will branch to the appropriate fault or trap handler.

3.3.3.3 Issue Functional Unit

Instruction issue is determined while the microword and the source data are being read. If for some reason the instruction can not be issued, the microword is saved and used in the next cycle, and the source data is read again. A number of factors can inhibit an issue on every cycle. For example:

- The pointers to the source data may not be valid; the IBox is still parsing the Istream
- The IBox has not delivered the source data
- the source data may not be valid; the MBox is reading the data,
- The EBox is executing an instruction that will write a destination GPR, that will be used as a source for the next instruction.
- The functional unit is busy, or its internal pipeline is stalled
- The microcode has specified that nothing should be done until certain operations have completed.
- The memory write path is busy.

When an issue is initiated, the result destination data is written into the Result Queue, together with the condition codes and branch checking.

The INTUNIT can perform simple instructions (e.g., MOVL or ADDL3) in one cycle. During the computation portion of the instruction the decision where to write the result (retire) is made.

Where as instruction requires multiple computation cycles, other operations can be issued. The result queue maintains information for all issued operations that have not been retired, and in the order they must be retired. The retiring control logic continuously attempts to retire the next operation found in the queue, and checks that the specified FEU is about to supply the result. If the result is not available nothing will be written in that cycle, and the retire control logic repeats the retire cycle.

3.3.3.4 Bypass Control

Bypass control is also performed by the issue logic. This is done by comparing where data will be next cycle, with where it will be required in the next cycle. If a data path directly connects the two functional units, bypass enabling functions steer a copy of the data where it is needed. The normal flow of the data through the pipeline continues in parallel. Thus, the required data can be obtained earlier through a bypass than by waiting for it to be written into a register file and then read out.

If the microcode specifies an operation to a particular FEU, the Issue logic assures that all source data and the FEU are available. In order to issue, the source data must be validated. That is, the data has arrived and is available, or that it can be retrieved through a bypass. The FEU and source data are selected and the operation initiated.

Issue Bypass: Issue criteria specifies the requirements for an issue cycle. The first criteria is a valid fork (i.e., the address of the next microword). The fork can be either for a new macroinstruction from the IBox, or the microsequencer will access the next microword based on the NEXT ADDRESS Field. The second criteria specifies that all sources must be bypassed to execute the required function. The third criteria requires a destination from the previous issue. If for some reason the IBox is behind in passing destinations, the EBox can not continue to issue. The last criteria is that the target FEU must be available (e.g., if the DIVUNIT is busy it cannot start another operation).

Register Bypass: The temporary registers (temps) and GPRs have bypasses. The temps have no bypass restrictions; however, the GPRs have a restriction. That restriction involves a situation where an FEU must write a GPR, and the next macroinstruction must use that GPR as a source. The results of the GPR write must be available before the second instruction can use the result as a source. The queue in the retirement logic determines if the data is valid. For example, if there is an entry in the queue that specifies the DIVUNIT is to write its result to R2 and the MULUNIT requires R2 as a source. The EBox checks the queue, finds the R2 reference, and stalls until the DIVUNIT has completed its operation.

When the DIVUNIT has completed, the EBox checks if any bypass tasks are enabled for another FEU. If not, one cycle is used passing the result to DIST, and through the bypass to the MULUNIT. In this case there is no need to read the data from the GRP file. It is bypassed and written into the register file at the same time.

Memory Bypass: The EBox can bypass data from memory directly to an FEU. The EBox is not forced to wait for the source data to be written into the source list (or temps) to be read and used. MBox data is passed into the DIST Unit, and immediately distributed to the specified FEU. This type of bypass requires that the source pointer references the source list or the memory access registers.

An additional memory bypass criteria is that one source be missing. Since the EBox can not receive both sources at the same time, it waits for the first source to arrive. When the second source arrives it enables the bypass and issues the FEU.

Data Bypass: Data bypass criteria specifies bypasses between internal EBox functional units. The first criteria specifies that the source data:

- is about to be written into the register file, or
- that applicable results are available elsewhere in the EBox.

The second criteria specifies that the required bypass data path must exist. For example, there is a path from the MULUNIT result to the FLOATUNIT source, and from the FLOATUNIT result to the MULUNIT source. Note that there is no path between the DIVUNIT and either of these units. Thus, all combinations of FEUs can not be bypassed.

The last criteria specifies that the context of the data being written match the context of the data being read. Although most bypasses are 32-bits wide, some are byte controlled in the INTUNIT. Certain sets of macroinstructions could be written that would cause some of the data to be assembled from several different functional units.

For example, two bytes of data could come from of the register file, another byte from a DIST bypass, and another byte from a Retire bypass. The major function is to assemble bytes together from wherever the most valid occurrence of that data is in the EBox.

3.3.4 Retire Functional Unit

The computational results from the FEUs are assembled in the Retire Functional Unit (Retire). RETIRE then distributes the results back to the STREG, or in some cases to the IBox. The Retire unit can also direct data between the FEUs. The Retire control logic is tightly coupled to the Issue control logic to maintain execution control between instruction issue and retire.

The Retire unit contains the Result Queue. When the instruction is issued, its result write destination is written into the Result Queue. The clocking of condition codes and branch checking are also written into the queue. In addition, the unit provides two data paths to the VBox (i.e., 64-bit input path to the VBox; 32-bit output path from the VBox).

Retirement specifies that the FEU is not required to hold its results. The Retire Unit informs the FEU that the results have been distributed to the appropriate destination (i.e., IBox, MBox, or register file). In order to do a retirement, the microcode must specify the function result destination to the control logic.

In a case where the EBox runs somewhat ahead of the IBox, the Result Queue may not have the destination. If the FEU completes its operation before the destination is available, the EBox will stall until the destination is received from the IBox.

Instructions are retired in the same order they were initially received from the IBox. For example, a divide is issued, and is followed by a multiply operation issue. The multiply may finish before the divide; the multiply is queued but not retired. When the divide completes, it is retired followed by the multiply retirement.

Some macroinstructions require several retire cycles, while other require only one cycle. The Retire Unit always performs at least one retire cycle to flag macro instruction completion. It is important that the EBox is notified of the macro instruction boundaries.

3.3.4.1 Result Queue and Retire Criteria

The Result Queue stores result destination pointers. The result queue provides two major functions. It allows issuing FEUs before a previous FEU operation is completed, and maintains the function retire order. That is, instructions are retired in the order received from the IBox.

In addition to the destination pointer and memory destination, each entry in the queue contains:

- the retire unit, and context
- condition codes
- number of destinations, and last retire

A result destination is required in order to retire an operation. When the FEU has indicated that it has completed its operation, the destination is popped from the top of the result queue, the results written, and the operation retired.

At times it may be necessary to stall and stop retiring results when writing to memory. Stalls may be encountered because a memory write might cause a page fault, or the memory pipe is full due to a number of previous memory write operations.

3.3.4.2 Destination Pointer Selection

The destination pointer specifies the destination of the function results. A destination pointer may be available from one of three functional areas.

From the destination queue (i.e.,the queue between the IBox and EBox). The IBox provides the result destination which is written to a GPR or to memory in the form of an Opwrite. (The IBox does not pass the memory address to the EBox for an opwrite.)

The microword can specify destinations directly or indirectly:

- directly to the GPRs or temps in the STREG or STRAM.
- indirectly by using some function of a previous pointer.

The INTUNIT may also provide a destination pointer in the form of a result.

3.3.4.3 Microsequencer and Control Store

The microsequencer accesses the control store to provide the microcode which controls the EBox functions. The control store is constructed of 1K x 4 and 4K x 4 STRAMs, and is configured to provide 4096 microwords, 140 bits wide. Microsequencer Description

The microsequencer accesses the microcode which controls all complex EBox functions. The microcode directs the Issue unit to select commands from the IBox (through the queues), or the microcode fields. The microcode also directs FEU operations.

The microsequencer selects between:

- the next microaddress in the microflow
- a microbranch
- a microsubroutine call or return
- exceptions or interrupts
- an IBox fork.

Much of the checking and selecting control is part of the microword.

3.3.4.4 Distribution Functional Unit

The Distribution Functional Unit (DIST) is the central distribution point for all instruction data (i.e., source, destination, and result data) within the EBox.

3.3.5 Functional Execution Units

All instruction processing is handled by the four FEUs. The FEUs are designated as the: Integer Unit, Multiply Unit, Floating Point Unit, and Divider Unit.

The FEUs are interconnected through bypass data paths to allow data to be moved from one unit to another as quickly as possible. The DIST unit distributes the source data to the specified FEU. Figure 3–10 illustrates the partial control and basic data path.

They are independent units which receive two, 32 bit operands per cycle from the register file, through DIST. Only one FEU can be initiated per cycle. Only one 32-bit result can be handled in a cycle. The arbitrating logic of the Retire Unit selects which FEU result to write.

3.3.5.1 Integer Unit

The Integer Unit (INTUNIT) performs general integer arithmetic, logic functions, and other specific functions to increase simple instruction execution times. INTUNIT contains a 32-bit ALU, and a 64-bit Barrel Shifter. In addition, it also contains an address generation unit capable of producing a memory address each cycle.

The INTUNIT executes simple instructions (e.g., MOVL, ADDL, etc.) at a rate of one per cycle with little microcode control. Complex instructions (CALLS, MOVC, etc.) are executed by repeated passes through the data path. For those instructions the microcode controls access to EBox resources.

The ALU is made up of three subunits: binary adder, BCD adder, and a Boolean subunit. The ALUs perform 32-bit integer and BCD adds and subtracts. The ALUs also calculate condition codes depending on the format: byte, word, or longword. The Boolean unit performs ANDs, ORs, and XORs the operands or the appropriate complements, depending on the UALU_FUNCTION microcode field encoding.

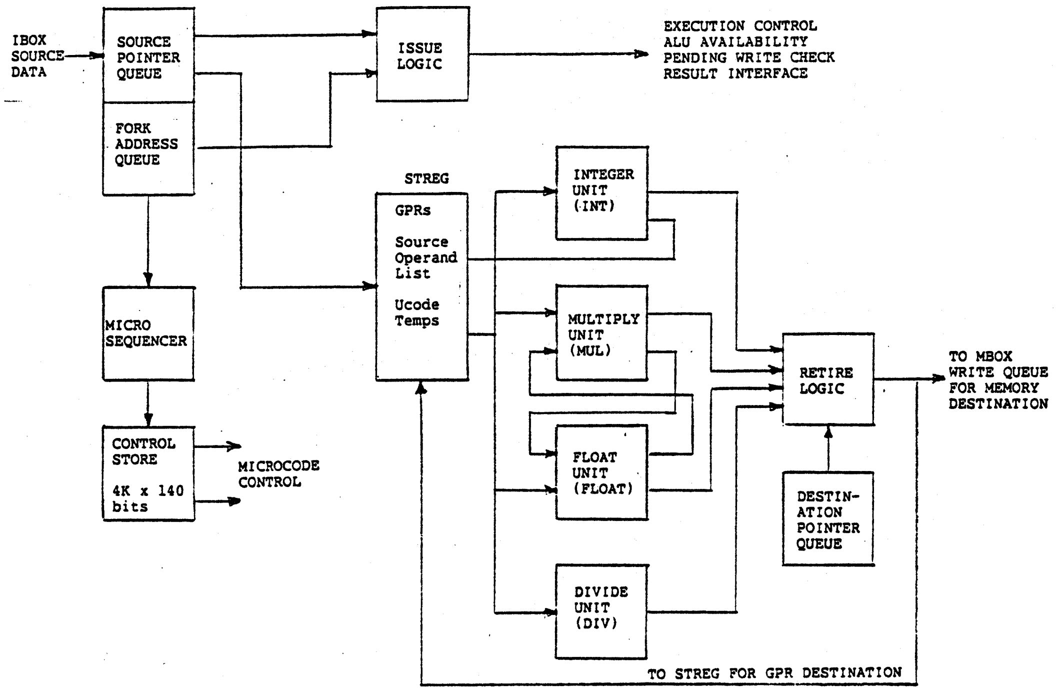


Figure 3–10 Partial Control and Data Path Block Diagram

The Shifter is capable of 64-bit logical shifts, 32-bit arithmetic shifts, byte swaps, nibble swaps and packed to numeric conversions. There is also special hardware for mask instructions, and an 8-bit counter to help with microcode loops.

The INTUNIT has only one operation cycle. It receives a microword every cycle to control the choice of operands and the ALU and Shifter operations. Source data is received from DIST.

The ALU and the Shifter can zero out either input operands; this allows much parallelism. Both units can operate on the same data, or part of the data; this concurrency saves cycles.

After the binary, decimal, and Boolean results have been calculated, condition codes are derived. Binary and Boolean condition codes are also based on format. The results are multiplexed and the appropriate one is sent an latched to DIST and RETIRE. The condition codes are latched to be sent to CC logic.

The Normalizer logic is special hardware used for mask and field instructions. The logic is controlled by by the microcode ULIT field. Its function is to take both the operands and the bit-swapped operands and finds the position of the first trailing one.

3.3.5.2 Multiply Unit

The Multiply Unit (MULUNIT) performs integer and floating point multiplication. It is based on a custom-designed multiplier chip, and is capable of performing a 32-bit by 32-bit multiplication each cycle. The MULUNIT is pipelined, and is capable of accepting instructions on every cycle.

The MULUNIT executes the following instruction: MULB, MULW, MULL EMUL, AN MULF, D, AND G. In addition it performs an unsigned 32- by 32-bit multiply. The MULUNIT performs the following tasks: unpacking, exponent handling, sign handling, multiplication; condition code generation; sub-product accumulation; floating point rounding, packing, control, and error handling.

The source operands are received on the two, 32-bit DIST busses, or on the single, 32-bit Floating Point Unit bypass bus. Context information is supplied form the Microsequencer Unit

The MULUNIT is a variable length pipe unit. If the context is integer, the multiply is performed in one cycle. If the context is floating point, the multiplication requires two or three cycles to perform, depending on format. It requires two cycles if the multiply is single precision (F format). Three cycles are required for double precision multiplies (D and G format), and single precision multiplies with a full pipeline. A 3- cycle multiply will pass through three MULUNIT pipe stages: Unpack, Save, and Accumulate, Round, and Pack.

3.3.5.3 Floating Point Unit

The Floating Point Unit (FLOATUNIT) performs floating point addition, subtraction, and certain data format conversions. It executes floating point operations for the ADD, SUB, CMP, CVT, and MOV instructions in F, G, and D floating point formats.

The FLOATUNIT is pipelined and is capable of accepting instructions on every cycle as the Issue logic issues and retires them. Although it receives 32-bit source operands, it operates on an internal 64-bit data path.

The FLOATUNIT is implemented in hardware, following a basic flow of unpacking the sources, aligning, adding and/or subtracting the fractions, normalizing and rounding the fraction, packing, and transferring the final result.

The FLOATUNIT is implemented with a 64-bit wide data path. The fraction, adder, subtractor 64-bit wide. The logic that performs alignment, adding, normalizing, and rounding accomplish their tasks in one pass through the logic for all possible conditions. For example, the normalizer can shift the fraction through the complete range from 0 bits to 64 bits. For a shift beyond the 64-bit range, the correct value is still generated by the hardware. There is no requirement for special control action.

There are no loops in the data path. In all cases each stage of logic performs its entire function. Generally all data flows through all stages regardless of the instruction to be executed. Depending on the instruction executed, certain stages may execute essentially a no op, or passthrough. All instructions take the same amount of time to be executed, there are no special cases that modify execution time.

The FLOATUNIT receives 64 bits of source data per cycle, and transmit 32 bits of result data per cycle, including condition code and status bits. When double precision data is received, half of each source arrives in the first cycle, and the second half of each source arrives in the second cycle. Since the exponent fields arrive with the first half, part of the exponent processing is performed and the fraction saved, while the second half source is being transmitted. The complete package of fraction and partially processed exponent is then merged, and sent through the remaining FLOATUNIT data path.

The FLOATUNIT produces a complete 64 bit result of which 32 bits can be immediately transmitted, and the other 32 bits are saved to be sent in the following cycle.

The FLOATUNIT is pipelined so that several instructions can be issued to it and have them stream through, or stall during retirement without losing any results. If the EBox issued FLOATUNIT instructions without retiring any, a maximum of two complete instructions, and the first source cycle of a third instruction could be issued before the FLOATUNIT would become unavailable for further issues.

3.3.5.4 Divider Unit

The Divider Unit (DIVUNIT) performs integer and floating point division, and is also based on a custom-designed chip. Since the DIVUNIT is not pipelined, only one Divide instruction can be in progress at any one time.

While instructions may be issued to other FEUs while the Divide unit is busy, they cannot be retired until the division has completed. Divisions can be completed in 12 cycles, including D and G formats.

The DIVUNIT executes the following instruction: DIVB, DVIW, DIVL, EDIV, DIVF, DIVG, DIVD. Source operands are received on the two, 32-bit DIST data busses. Context information is supplied form the Microsequencer Unit. The Issue Unit provides issue and retire control.

The functional tasks performed include: unpacking, exponent handling, sign handling, iterative division, condition code generation, quotient accumulation, floating point rounding, packing, control, and error handling.

3.3.5.5 Condition Codes

The condition code logic performs two functions: set the PSL condition codes, perform macro branch checks to inform the IBox if the branch prediction was correct. The PSL condition codes can be set:

- by being clocked by the microcode (usually at the end of a macro instruction) based on the results of an instruction
- be written directly when returning from an exception or interrupt handling routine
- executing a BSPSW or BSPSL instruction.

To clock the condition codes, the Condition Code (CC) Unit selects the conditions from the currently retiring FEU, and multiplexers them with the previous values according to the microcode UCCK field. Since updating the CCs is part of retirement, UCCK must be piped through the Result Queue (similar to the destination pointer) to insure retirement in order

The condition codes are the four low-order bits of the PSL. With the appropriate logic functions enabled the low-order ALU result is written to the CC bits.

3.3.5.6 Register Log Description

As the IBox processes instructions with autoincrement and autodecrement specifiers, the GPRs are incremented and decremented. When the EBox is interrupted (e.g., interrupt, exception, page fault), the system requires a mechanism to recover the original GPR content and state of the machine (Unwind process). That mechanism is the Register Log (Rlog).

The Rlog is a LIFO buffer (or push-down stack) which stores data relating to GPRs that may change during instruction execution. The Rlog tracks these specifiers on a macroinstruction basis. Since a macro instruction may alter up to six GPRs, the Rlog must be able to store six entries for each instruction. For each macroinstruction the Rlog stores

- o the number of increments and decrements
- o which registers were affected
- o the amount each register was affected

The IBox can be decoding specifiers that are several instructions ahead of the instruction currently being executed. That is:

- o the EBox is executing instruction N
- o the IBox has evaluated specifies for instructions N+1, N+2, and N+3.

Should the EBox handle an interrupt or exception after N, the GPRs must be unwound to the state they would have been in if the IBox had not evaluated specifiers for N+1, N+2, and N+3. Thus, the GPRs must be reconstructed with the content identical to the beginning of instruction N. (See Figure 3–11

As the IBox begins to decode an instruction, it assigns a 2-bit tag to that instruction. That is, instruction N is assigned 0, instruction N+1 is assigned 1, N+2 = 2, N+3 = 3. As the IBox evaluates an autoincrement or autodecrement specifier, it passes to the EBox:

- o the 2-bit tag (0, 1, etc.)
- o a 4-bit context field - coded to specify size (1, 2, 4, 8, or 16 bytes), and whether it was an increment or decrement, and
- o the GPR data.

Each counter entry specifies the number of entries in the queue for that instruction.

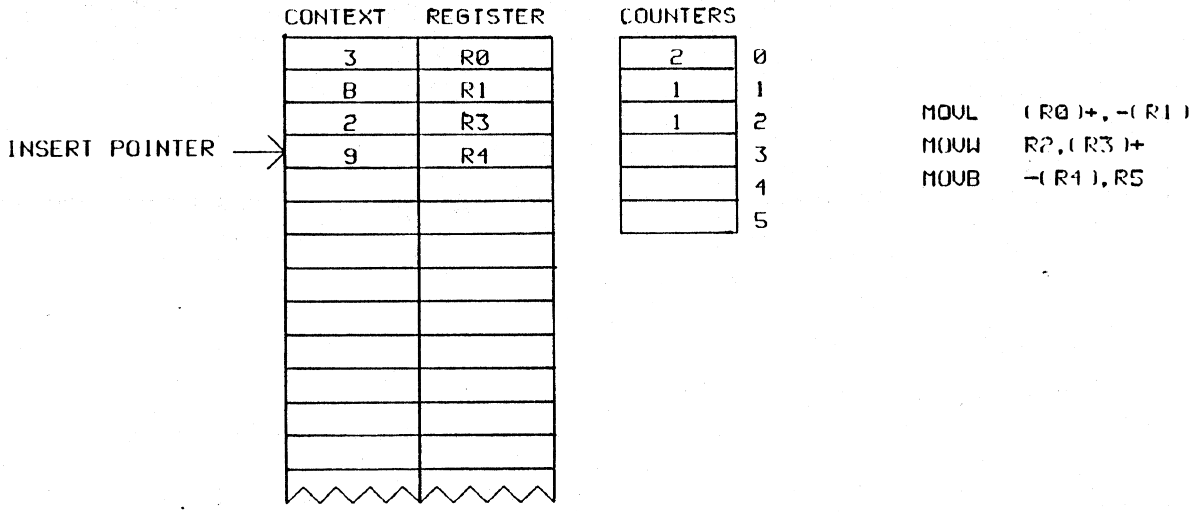
For each IBox GPR write, the EBox stores the context and the GPR numbers in one of four logical stacks according to the tag. It then marks that stack valid. Each stack is invalidated as the EBox completes the instruction.

On an interrupt the EBox sends a flag to the IBox indicating it will unwind. The IBox ceases evaluating specifiers and passes a flag to the EBox indicating it has ceased. The EBox then proceeds with its unwind sequence. If any of the logical stacks are valid, they must be emptied and the E and I box GPRs updated.

The E and I boxes must keep the tags and stacks in step. That is, the EBox must know which stack to invalidate first, and the IBox must use the same one for the first instruction.

Also since the:

- IBox does not evaluate auto-increment or decrement specifiers until a branch is validated, and



CONTEXT	REGISTER
3	R0
8	R1
2	R3
9	R4

COUNTERS	
2	0
1	1
1	2
	3
	4
	5

```
MOVL    (R0)+,-(R1)
MOVW    R2,(R3)+
MOVB    -(R4),R5
```

Figure 3-11 Simplified Rlog Structure

- the EBox flush should only occur in instructions that suspend the IBox, or during an interrupt/exception routine

the RLOG stacks will be empty in these cases.

As shown in the sample Istream of Figure 3-11 the first instruction (MOVL) performed an autoincrement on R0, and an autodecrement on R1. Thus, there are two entries in the queue for that instruction. The register field contents are used to form pointers to the GPRs (R0 and R1). The context fields contains encoding which is used in part to form a constant to restore each register.

In the case of a page fault, the EBox traps to the page fault handler. When the page fault has been resolved, the EBox accesses the Rlog to unwind and restore the GPRs. The EBox removes Rlog entries as instructions complete, based on an instruction done indicator. The EBox determines when an instruction has completed to the point where a page fault could not interfere. The instruction done indicator is sent to the Rlog and the appropriate entry is removed.

In addition, should the EBox handle an interrupt during a long instruction that has started but not completed, it will save instruction information and set First Part Done for the interrupt routine. In these cases, the CPU will not unwind the registers to the beginning of the instruction. The EBox invalidates the RLOG stack for that instruction before it starts the interrupt routine.

3.3.6 Physical Structure

As shown in Figure 3–12, the EBox is physically contained in six MCUs:

- Microsequencer and Control (CTL)
- Control Store (UCS)
- Integer Functional Unit (INT)
- Floating Add and Divide Functional Unit (FAD)
- Floating Multiply Functional Unit (MUL)
- Data Distribution Unit (DST)

The following subsections lists the MCUs, planar module locations, the MCA complements.

3.3.6.1 Distribution MCU Description

The DST MCU is positioned at planar module location MCU 6 and contains the source and destination lists.

The DST MCU is located in the central position of the EBox layout. This position places DST next to the MBox data cache MCUs allowing data to move very quickly between the EBox and cache. In addition, the MUL, INT, and FAD MCUs are also placed around DST to reduce the data transfer distances.

The MCA complement is listed below.

STG0 and STG1

DST0 - 3

ERGX ERGX contains a set of 1K x 4 STRAMs used for microcode temporary storage and constants.

3.3.6.2 Integer MCU Description

The Integer (INT) MCU is positioned at module location MCU 10. It contains the integer ALU and related Shifter which perform integer adds and substracts. Included in the MCU are the microsequencer and two sets of 4K and 1K control store STRAMS. Included also is the GPR Register Log (RLOG). The MCA complement is listed below.

IALU

ISHF

USQA-C

RLOG

CSS1X

CSF1X

3.3.6.3 Control Store MCU Description

The Control Store (USC) MCU is positioned at module location MCU 14. It contains the major portion of the control store. This control store portion is constructed of 4K x 4 STRAMS. The MCA complement is listed below.

CSS2X, 4X - 6X

VCTA - C

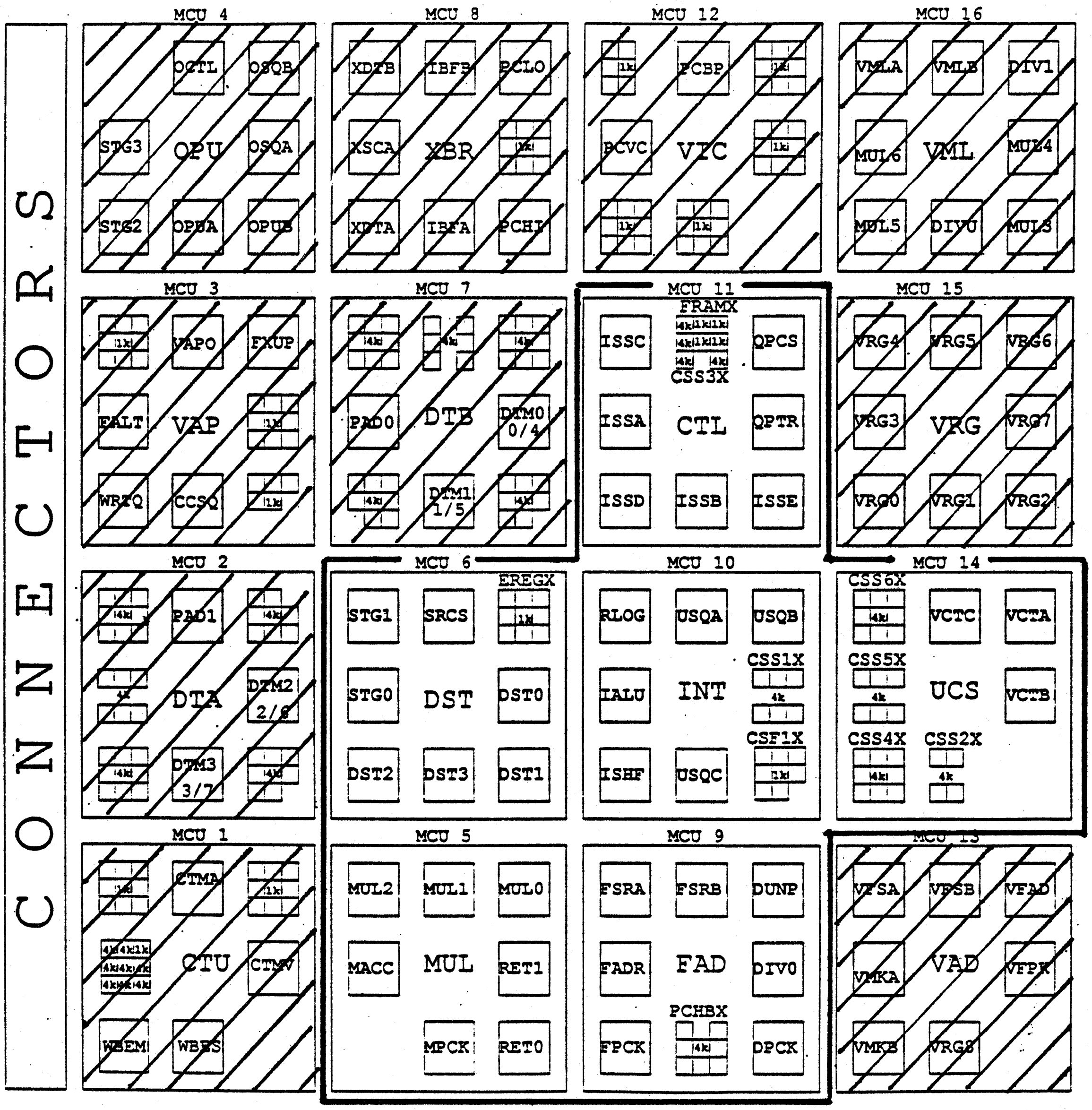


Figure 3-12 EBox MCU Planar Module Placement

3.3.6.4 Multiply MCU Description

The Multiply (MUL) MCU is positioned at module location MCU 5. It contains the floating point multiplication functional unit. Included also is the Retire functional unit. The MCA complement is listed below.

Multiply units: MUL0 - 2

Condition Code unit: MACC

Unpacking: MPCK

Retire unit: RET0 - 1

3.3.6.5 Floating Add and Divide MCU Description

The Floating Add and Divide (FAD) MCU is positioned at module location MCU 9. It contains the floating point addition and divide arithmetic functional units. The MCA complement is listed below.

FSRA and FSRB

FADR

FPCK

DUMP

DIV0

DECK

PCHBX STRAMS

3.3.6.6 Control Unit MCU Description

The Control (CTL) MCU is positioned at module location MCU 11, and contains the following MCA complement.

ICCA - E

QPCS

PTR

FRAMX

3.3.7 Basic Instruction Flow

This subsection describes the ADDL3 instruction execution flow, and is based on the following assumptions:

- GPR sources and destination
- all instruction data is resident in the queues
- execution activity is divided into system clock cycles.

In addition, Figure 3–13 complements the flow and describes some of the major execution parameters.

Instruction Flow

Clock 1, Fork Cycle: The fork address (the first microword address) which is contained in the Fork Queue, is used by the microsequencer to start accessing the control store.

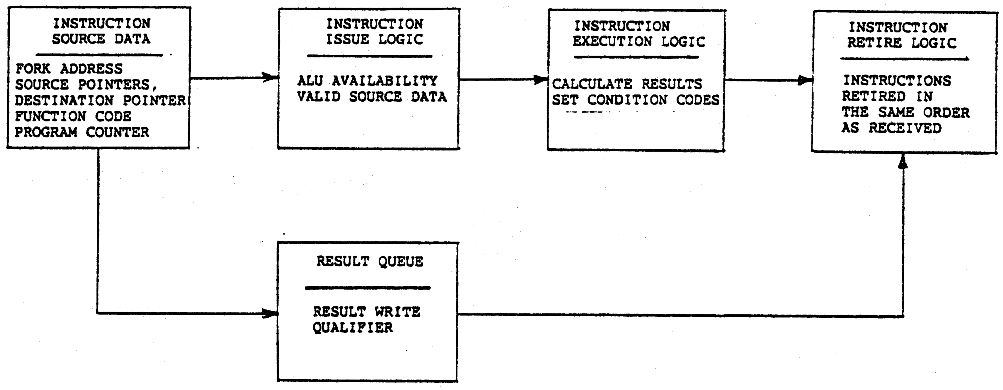


Figure 3-13 Instruction Execution Parameters

Clock 2, Issue Cycle: During this cycle the microword is read from the control store and distributed throughout the EBox. The source data is read out of the GPRs and distributed to the INTUNIT, while the ISSUE Unit verifies that the instruction data is valid and that the INTUNIT is available.

Clock 3, Execute Cycle: The INTUNIT receives the appropriate microword fields and source data and computes the result. Since this is an INTUNIT operation, the result write destination is determined during this cycle.

NOTE

For FEUs requiring multiple computation cycles, the result write destination is determined in the last compute cycle.

Clock 4, Retire Cycle: The result is transferred through the Retire Unit to the destination GPR.

Clock 5, Write Cycle: The result is written into the destination GPR. All instructions follow this basic flow. At times an instruction may stall at a certain point because of a missing or unavailable resource (e.g. instruction data, an FEU, a pointer, etc). In addition, since the MULUNIT, FLOATUNIT, and DIVUNIT. require multiple computation cycles, there is the opportunity to issue other operations.

Regardless of the conditions encountered, instructions always flow through the pipe in the same order as received from the IBox. In addition, instructions are retired in the same order as received from the IBox.

3.3.8 Operational Summaries

Figure 7-18 describes the EBox data path; note that data path control functions are not included. Unless otherwise specified all data lines in the figure are 32-bits wide.

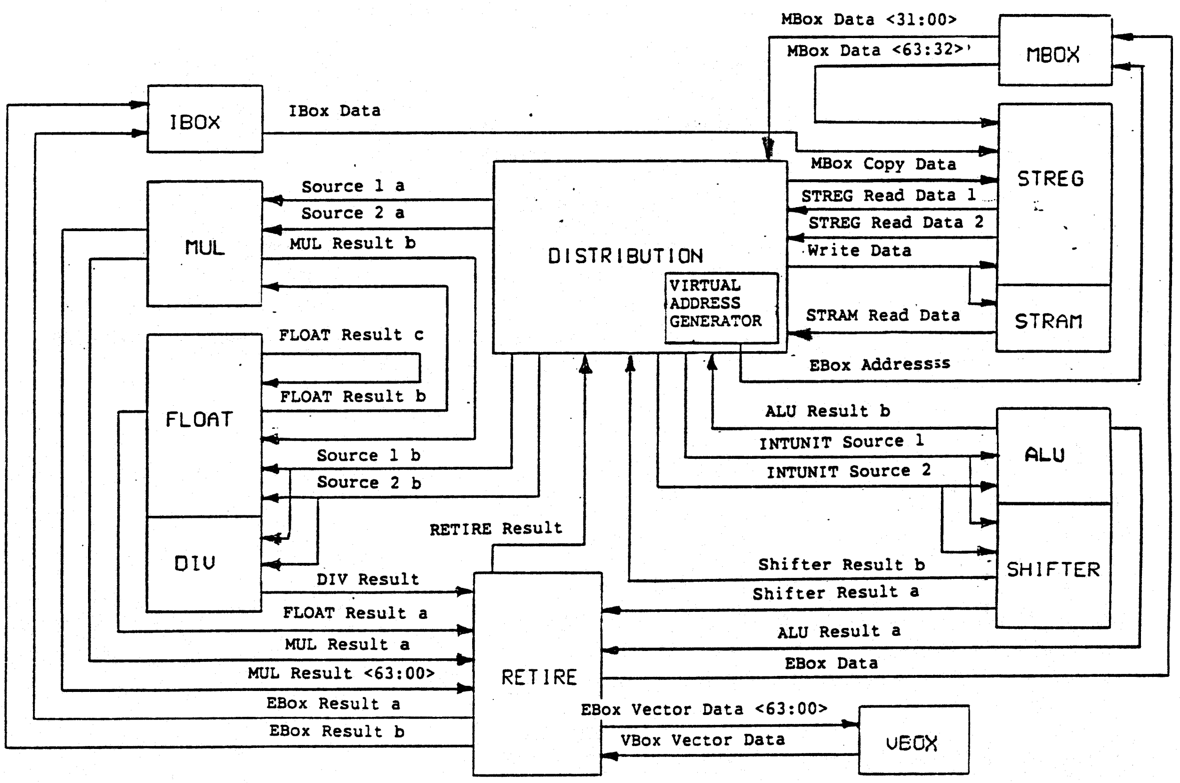


Figure 3–18 EBox Data Path Block Diagram

3.3.8.1 Data Path Overview

As shown in Figure 7–18 instruction data is transferred from the IBox and written directly into the STREG (Source List). Note that MBox input data not only has a path to the STREG (MBox Data 63:32) but also into the Distribution Unit (MBox Data 31:00). The latter input is written into the STREG as MBox Copy Data.

If the MBox source data is not required immediately, the entire QW is available from the STREG. However, if the EBox is waiting for that source data, it would require an extra cycle to write the data into the STREG, and then read it out to the Distribution Unit (DIST).

Although the EBox can read two locations out of the STREG simultaneously (Read Data 1 and Read Data 2), this arrangement allows the low-order LW to be immediately passed to the selected FEU. The EBox would then write the high-order LW into the STREG since it would not be needed until the next cycle.

When source data is retrieved from the STREG (or STRAM), a copy of both sources (Source 1 and Source 2) is passed to each FEU (i.e., INT, MUL, and FAD). Although the source is available to each FEU, only the FEU designated by the Control Unit (CTL) will process the sources.

After an FEU has completed its operation, the result is passed to the Retire Unit which is basically a large multiplexer . Retire queues the results, and then selects results for retirement in the order that they were initially received by the EBox. Retire can then transfer the result data to the IBox, MBox, or GPRs. If the result is to be written to a GPR, the EBox will not only write its own GPR, but the corresponding IBox GPR as well.

Retire has a result path to the MBox. This path is for results that are to be written back to memory. There are effectively two result paths to the IBox. One path is used to update the IBox GPRs. The other path is used to notify the IBox when the EBox requires it to start parsing the Istream at a different point. In this case, the EBox would send a new PC value and direct the IBox to start parsing instructions at that PC.

The DST unit contains registers that are dedicated to the INTUNIT (i.e., only the INTUNIT can use them). The registers are used by the ALU and Shifter on a cycle basis with no stalls between cycles. The ALU and Shifter result data is transferred to DST and written into the registers. DST can then return the register content to the ALU or Shifter as source data on the next cycle. The registers eliminate the need to retrieve this particular source data from the STREG, STRAM, or MBox.

As shown in Figure 7–18, the MUL and FLOAT units have cross-coupled source and result bypasses (data paths). Also, FLOAT has a result bypass to its source input. The bypasses allow the results of one FEU operation to be available as source input for the other FEU in the next cycle. Or in the case of FLOAT, the result is returned as source data for the next cycle.

In each case the EBox selects sources in one cycle and uses them in the next cycle. Sources for an operation are selected one cycle ahead of the operational (execution) cycle. If the EBox is to add two numbers with the sources in GPR0 and 1, one cycle would select the source GPRs, the next cycle would initiate the INTUNIT to add the two numbers together.

Many macro instructions can be executed in one cycle (e.g., ADD, MOV, etc.). In those cases the source selection is supplied from pointers provided by the IBox.

3.3.8.2 EBox Pipelining

Figure Figure 3–15 describes the number of FEU operation cycles required for selected macroinstructions. Issue and retire cycles are shown for each macroinstruction execution. During the issue and retire cycles the FEUs are not operating unless they are overlapped as shown in the D and G format cases.

The INTUNIT (INT in Figure 3–15) requires one cycle to perform any operation (e.g., add, subtract, XOR, etc.). ISS (Issue Unit) is the cycle in which the INTUNIT is selected, preselection of sources is determined, and any required bypasses are enabled. In the second cycle (ALU) the INTUNIT has received the source data and performs the operation. During the retire cycle, the results are multiplexed through the RETIRE Unit and distributed to the destination.

Note that although three cycles were used in the previous example the actual function (operation) required only one cycle. Cycles can be overlapped. That is, another ISS cycle could select the INTUNIT during the ALU cycle of the previous instruction.

All F format floating point operations require two cycles in the FLOAT Unit. As shown in Figure 3–15, the floating point add uses one cycle to unpack, and one cycle to add and pack. As before the issue and retire cycles can be overlapped with other FEU operations. By overlapping control, the FLOAT Unit can operate on D and G formats in three cycles.

The MUL Unit can perform all:

- integer (B, W, LW) operations in one cycle two
- F format operations in two cycles
- D and G format operations in three cycles.

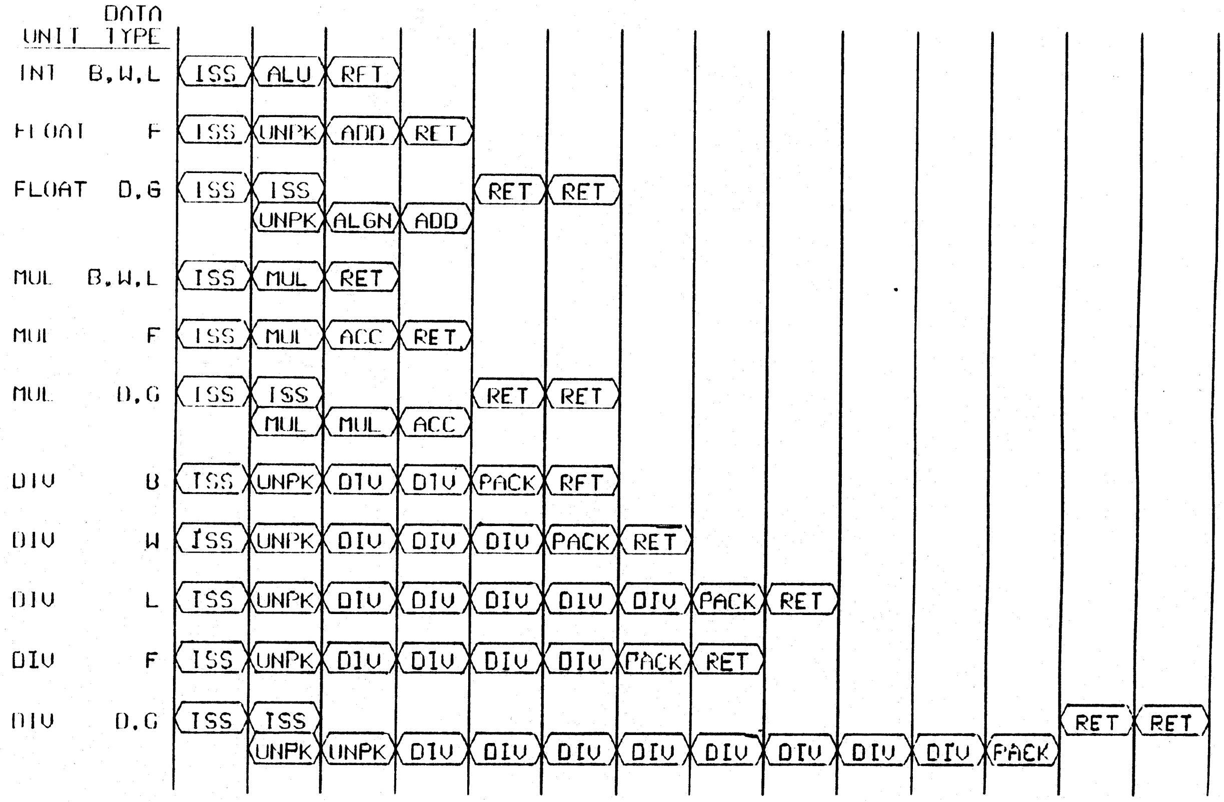

DATA
UNIT TYPE
INT B,W,L ISS ALU RET
FLOAT F ISS UNPK ADD RET
FLOAT D,G ISS ISS RET RET
UNPK ALGN ADD
MUL B,W,L ISS MUL RET
MUL F ISS MUL ACC RET
MUL D,G ISS ISS RET RET
MUL MUL ACC
DIV B ISS UNPK DIV DIV PACK RET
DIV W ISS UNPK DIV DIV DIV PACK RET
DIV L ISS UNPK DIV DIV DIV DIV DIV PACK RET
DIV F ISS UNPK DIV DIV DIV DIV PACK RET
DIV D,G ISS ISS RET RET
UNPK UNPK DIV DIV DIV DIV DIV DIV DIV DIV PACK

Figure 3–15 EBox Pipelining

The DIV cycles represent the DIV unit's ability to produce eight quotient bits. That is, if all of the significant bits of data are accumulated and divided by eight, the result would represent the number of divide cycles required. However, notice that in each case an extra cycle is required to accumulate the extra significant bits. For example, a LW operation is 32 bits divided by eight. This would require four DIV cycles and an extra cycle for the remaining significant bits.

3.3.8.3 Parallel Instruction Execution

Figure 3–16 illustrates the types of overlapping and piping parallelism available in the functional units. The figure illustrates three MULGs and two ADDGs. The retire cycles are represented by the destination GPRs being written. For example, in the first case (example a) R2 is retired in the first cycle and R3 in the second cycle.

The MULG instruction requires two 64-bit, G format sources. Since the MUL source data path is 32-bits wide, the low-order LW of each source is passed in the first cycle. The second cycle contains the high-order LW sources. This requires two issue cycles (i.e., two selections of data into the MUL functional unit).

MUL starts the multiply in Cycle 2 even though it does not have all the source data. It is able to initiate the operation since there are three multiply elements in the MUL unit, each capable of 32-bit KS 32-bit multiplication. The first MUL cycle multiplies the two low-order sources. In the second cycle the MUL performs the three remaining multiplies: high KS high, high KS low, low KS high. The partial products are then accumulated in

```
a  MULG3   (R0), (R0)+,R2
b  MULG3   (R0), (R0)+,R4
c  MULG3   (R0), (R0)+,R6
d  ADDG2   R1,R2
e  ADDG2   R6,R2
```

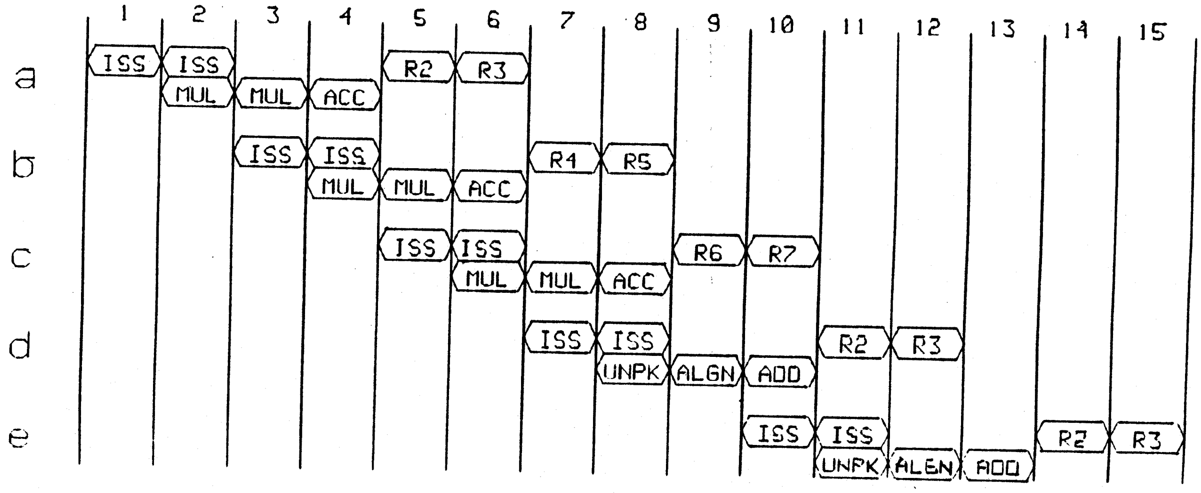


Figure 3–16 Macro Instruction Overlapping

the ACC cycle. The retire cycle consists of writing the first-half results to R2, and the second-half results to R3.

Since the MUL Unit is pipelined, the second multiply instruction is issued and overlapped in the second multiply cycle of the first instruction. The third multiply is issued and overlapped identical to the second. Essentially this is parallel macro instruction execution, since there are three macro instructions executing in the MUL pipe.

However, should the sources and destinations be memory locations, the operations would be using the full memory bandwidth to pass a QW to memory every cycle. This would consume some of the cache bandwidth, and affect the rate at which the MBox could supply sources.

As shown in Example d of Figure 3-16, the FLOAT Unit is then issued to execute the first ADDG2. Notice that in the first cycle FLOAT is only able to unpack, since the align and add operations require all the source data.

Note that there was no ISS cycle for the second ADD instruction in Cycle 9. This is a register conflict stall. The last ADD instruction requires R6 and 7, and R2 and 3 as sources. However the previous ADD instruction is writing its results to R2 and 3. In this case, issuing of the second ADD is delayed for one cycle to allow the result to be written to R2/R3. The R2/R3 content is bypassed back to the FLOAT Unit and allows the last ADD to unpack R2 in Cycle 11.

The execution of the five example macro instructions required 12 microcycles to complete. That is, cycles 2 - 13 where the FEU operations were performed (except for Cycle 9). Ten microwords were also executed.

3.3.9 EBox Microcode Overview

The EBox microcode provides the control required to execute the VAX instruction set, process interrupts and exceptions, and control for the entire system. EBox microcode is a synchronization point for the pipeline as it handles errors, interrupts, exceptions, traps, etc.

The functions executed by the EBox are controlled by generating specific sequences of microwords. Each microword within the sequence is encoded to perform the data routing and manipulation required.

The EBox microword is 150 bits wide, and is contained in a 4K word control store. The microword is comprised of 57 fields, some of which are overlapped. Note that microword fields are grouped according to function, and are laid out in a logical order rather than a physical order.

This microcode implementation is wider and has less storage than most VAX implementations. The wide microword allows simultaneous control of multiple operations to achieve parallel execution.

The control store is implemented in 1K x 4 and 4K x 4 STRAMS. Since the 1K STRAMs have a faster access time (7 nsec), they are used for certain fields of the microword (e.g., microbranching, next address, etc.). That is, due to the 1K functionality a new microword can be accessed each cycle.

3.3.9.1 Field Definition Introduction

The microword fields specify a sequence of bits within the microword; field encoding defines the particular field functions. The following subsections and tables summarize the major functions of the microword fields. That is, the field definitions only describe the general field functions; the field encoding is not included.

Generally, the following field definitions are grouped by related functions. For example, the Next Microaddress Generation Control group describes fields that are related to microaddress generation functions such as:

- how the next microaddress is generated
- define microbranch conditions
- validity checking on microbranch conditions
- specify microtrap types

3.3.9.2 Issue and Preselection Function Fields

Table Table 3-2 summarizes the Issue and Preselection function fields.

Table 3-2 Issue and Preselection Field Summary

Field Name	Description
UISSUE<22:20>	Specifies which unit will be selected for the next issued microinstruction. Possible selections are: INTUNIT, MULUNIT, FLOAT, etc.
UMEM_WAIT<80>	Synchronizes the microcode with write requests. It causes the next microword to stall until MBox write success or nosuccess is known. A nosuccess will cause a microtrap.
UBYPASS_DISABLE<29>	Used to disable the memory bypass logic. If a microword requires data that is not yet available, it is by default "issued with bypass" and is executed as soon as the next piece of data comes from the MBox. If the IBox is not suspended and is making memory requests, memory bypass must be disabled and the microword cannot be issued until the data has been written to the STREG
UNUM SRC<24:23>	Specifies the number of sources preselected. Possible values are none, Source 1 only, Source 2 only, or both.
UPTRSELA<26:25>	The register file addresses of Source1, Source2, and the destination may be saved. Possible values are hold current save pointers, load from source preselection registers and destination in the current microword, or load from integer result from the previous microword.
USPADDRMX_SEL<27>	The integer result for the save pointer may come from either the integer unit result or from the encoder position output (see encoder fields) for call/return.
USRC1 PTR<38:30>	Selects Source1 for the next microword. Selections may be: a STRAM or STREG location, the saved pointer, etc. The field can also be encoded to unwind the Rlog.
USRC2 PTR<47:39>	Selects Source2 for the next microword. Field encoding is similar to the USRC1 PTR field.
USRC_TRAP_SEL<146:145>	Controls the issue of microtraps from the source operands. It is needed for the fault handling of Field and CALLx instructions where the data is supplied from the IBox, but the EBox may not need it. In these cases the EBox branches and calculates the amount of data needed and checks for the fault again. Values are no disable, S1 Disable, S2 Disable, and S1 and S2 Disable

3.3.10 ALU Function Fields

Table 3-3 summarizes the ALU function fields.

Table 3–3 ALU Field Function Summary

Field Name	Description
UAMX_SEL<50:48>	Controls the AMUX inputs for the ALU and Shifter of the INTUNIT.
UBMX_SEL<52:51>	Controls the BMUX inputs for the ALU and shifter of the INTUNIT.
UALU_FUNCTION<57:53>	Directs the ALU to performs Boolean, binary, and BCD operations. The BCD operations are performed in one cycle.

3.3.10.1 Shifter Control Fields

The shifter performs a number of tasks and requires a number of fields to describe its use. Table 3–4 summarizes the field functions.

Table 3–4 Shifter Control Field Summary

Field Name	Description
USHF_FUNCTION<60:58>	Specifies Shifter functions. Selections include: nibble swap, byte swap, convert packed to numeric.
USHF_CNTMX_SEL<62:61>	Specifies the source of the shift count used during Shift instructions.
UCNTMX_SEL<64:63>	Specifies the source or value loaded into the counter. The counter can be used as a shift count, or as a loop counter in conjunction with a branch on count = 0.
ULIT<72:65>	Used as a shift count to control shifting, to load the CNT, or be the low-order 8 bits of the SHF_CNT input to the USHF_OUTMX field.
USHF_OUTMX_SEL<74:73>	The shift result can be either: normalizer output; shifter output; IPRs or PCs from the microsequencer and condition code logic; or the SHF_CNT.
UNORM0<65>	Used as a control field for an inverter which is part of the normalizer logic.
UNORM52<70:67>	This field determines whether to use the A or B operand and whether it is to be used in its normal order or bit-swapped. Other selections include: output from the encoder, its complement, etc.
UNORM76<72:71>	Determines whether the 32-bit data should be passed unchanged or have a bit called out by logic to be set or cleared.
UHOLD_POSITION<147>	Determines whether to latch a new value or hold the previous value.

3.3.10.2 BMUX Good Numeric

The CVTTP and CVTTP instructions control and branch on a latch that indicates whether numeric data is valid. Table 3-5 summarizes the field functions.

Table 3-5 BMUX Field Summary

Field Name	Description
ULOAD_OR_AND_ NUMERIC<60>	Specifies whether the input to the latch is the test on BMUX<7:0> or the test ANDed with the previous value of the latch.
UHOLD_NUMERIC<147>	Load or hold the good numeric latch.

3.3.10.3 Results and Destination Control

Table 3-6 summarizes the field functions.

Table 3-6 Result and Destination Field Summary

URETIRE TAG_A<77:76>	Determines the unit to be retired as none, INT, or MUL. It's used mainly by the FORK macro to generate a retire from INT with no concern whether it is from ALU or Shifter.
URETIRE TAG<77:75>	Specifies more exactly what is being retired: Vector, ALU, Shifter, Floating MUL, INT, MUL, FLOAT, or DIV.
URESMX_SEL<149>	This field selects either the ALU or the Shifter as the integer unit output.
URMX_SEL<78>	Determines whether to hold or load R register from an ALU or Shifter result.
UR1MX_SEL<148>	Selects either the ALU or Shifter as input to the R register.
UAUTONOMOUS<79>	Determines whether to post the result in the Result Queue.
UNUM DEST<81>	Specifies one, or no destination.
UDEST PTR<91:84>	Selects the destination. Selections include: an explicit STRAM or STREG location, the previous pointer, the previous pointer incremented or decremented by 1, etc. In addition, there is an encoding to unwind the Rlog.
UCTXRF<83:82>	Controls the size of data written to the register file. It is also used to specify data size for operand writes. The possible values are byte, word, long, and quad (which translates to long for the register file).

3.3.10.4 Virtual Address Control

These fields are used to control the virtual address generation portion of the data path. The VA and VB latches can also be used as general purpose latches since there are paths between them and the rest of the data path. In addition, bits of VA can be used for microbranching. Table 3–7 summarizes the field functions.

Table 3–7 VA Control Field Summary

Filed Name	Description
UVA_ALUMX_SEL<98:97>	Controls the VA ALU MUX input and can be either: VA, VB, Source 1, or Source 2.
UVA_ALU_FUNCTION<101:99>	Controls the VA ALU Choices are: pass; add 1,2,4,or 8; and subtract 1,2,or 4.
UMBOX_ADDRMX_SEL<102>	Controls whether the MBox address is loaded from the VA ALU MUX or the VA ALU. The VA ALU operation is always done. Thus the MBox address can be the value before or after the ALU operation.
UVA_SHFMX_SEL<92>	Selects the VA, or VA shifted left by one (fill with 0) as input to the VA MUX.
UVAMX_SEL<94:93>	Load VA with output from VA SHIFT MUX, the VA ALU result, the ALU result, or the shifter result.
UVBMX_SEL<96:95>	Load VB with from VA, from the VA ALU result, VB, or Source 1.

3.3.10.5 Condition Codes

Table 3–8 summarizes the field functions.

Table 3–8 Condition Code Field Summary

Field Name	Description
UCCK<106:103>	Determines how to affect the PSL condition code bits. Encoding defines the value of each condition code bit from some combination of current PSL condition code bits and ALU condition code bits. PSL V can be forced to 0 or 1, and PSL C can be forced to 0.
UCTX<108:107>	Used to control the data context of the condition code bits from the ALU. Encoding specifies: byte, word, longword, quad (this value is used when the microcode attempts to write the first longword of a quadword operand write).

3.3.10.6 Macrobranch Control

A number of fields are used to direct macrobranching. Table 3–9 summarizes the field functions.

Table 3–9 Macrobranch Control Field Summary

Field Name	Description
UMACROB<109>	Informs the CC unit whether or not to compare condition codes against the Macrobranch mask. All Macrobranch instructions will set this field.
UMACRO_B_CC_SEL<110>	Informs the CC unit whether the Macrobranch mask is compared to existing PSL condition code bits or the ALU condition codes bits being created in the current cycle.
UMACRO_BRANCH_SELECT<114:111>	These values are send to the CC unit for the conditional Macrobranch instructions. These values are used as masks to determine whether the branch should be taken.

3.3.10.7 Macrobranching and Displacement Operands

Branches have a displacement as the last operand, and an operand fault should take precedence over the branch. The mechanism for following this rule is basically as follows: For simple branches such as BNEQ, the microcode specifies UDEST_PTR as GPR15 (18F).

For branches with operand writes, the destination is already a GPR, or is a memory location that the Issue logic tracks with a GPR. The issue logic will not allow a retire until it is notified that the IBox is on the next instruction.

3.3.10.8 Macrobranching and Bad Branch Prediction Traps

The UTRAP field allows bad branch prediction traps. Simple branches use BRANCH_PRED; unconditional use UNCOND_BRANCH; and bit field branches like BBSS use BRANCH_NOMEM; which because of the algorithm disables write traps.

3.3.10.9 Next Microaddress Generation Control

Table 3–10 summarizes the field functions.

Table 3–10 Next Microaddress Control Field Summary

Field Name	Description
UNEXT ADDRESS<11:0>	Used in next microaddress generation as defined in the UNEXT_ADDR_SEL field.
UN20<2:0>	Used for validity checking of constraints when microbranching.
UNEXT_ADDR_SEL<13:12>	Determines how to generate the next address. Choices are: FRAM, next address field from the current microword ,pop an address from microstack and OR with next address field, use next address field, and push current on microstack
UBEN<18:14>	Provides up to 32 branch CODES, each of which creates a 3-bit value that is ORed into the low-order 3 bits of the next address as selected in the UNEXT_ADDR_SEL field. Each recipe specifies a signal for each bit positions.
USETUP<154>	Used by the microcoder to document the latency and make the branching flow clearer.
UTRAP<118:115>	Selects the type of microtraps allowed after the retire cycle. Some choices are: allow only memory management traps, disable write failures, allow trap for bad branch prediction, integer overflow, etc.

3.3.10.10 Flush Control

Table 3–11 summarizes the field functions.

Table 3–11 Flush Control Field Summary

Field Name	Description
UFLUSH<119>	If asserted, this bit will: invalidate the source list; invalidate the memory temps; flush the result queue; unbusy all functional units; clear memory interface; clear some blocking signals internal to issue and retire units.
UQUEUE_CTL<120>	If asserted, this bit will clear the source queues, destination queue, and fork queue.

3.3.10.11 MBox and IBox Control

These fields control read/write functions, and other miscellaneous I and MBox functions. Table 3–12 summarizes the field functions.

Table 3–12 M and IBox Control Field Summary

Field Name	Description
UMCF<125:121>	Specifies the I and MBox functions. for example: virtual read or write, read or write physical, set the FPD bit in the PSL, read MBox register, clear EBox error register, etc.
UMBOX_CTX<128:126>	Defines the data size context for op write and EBox memory references. Choices are: byte; word; long; and quad. It also is used to pass the mode to the MBox for probes. Choices are: kernel; executive; supervisor; user; or current.
UTAG<133:129>	The memory temps in the register file are written by the MBox when the microcode requests a memory read and an MBox register read.

3.3.10.12 Other Function Unit Control

Table 3–13 summarizes the remaining microword fields.

Table 3–13 Other Function Unit Field Summary

Field Name	Description
UMULCTL<136:134>	Defines the basic control for the Multiplier Unit.
UFLOATCTL<144:137>	Defines the basic control for the Float Unit.

3.3.11 Microsequencer Operations

The microsequencer selects between the next microaddress in the microflow, a microbranch, a microsubroutine call or return, exceptions, interrupts, and IBox forks. Much of the checking and selection control is part of the microword. In addition to the IBox passing the UPC (i.e., fork address) of the next instruction, other conditions for changing microcode flow are described in the following subsections.

3.3.11.1 Micro Branch On CPU Conditions

When the UBEN field specifies a branch, the UPC is formed by OR'ing the designated conditions into the low-order three bits of the UNEXT ADDRESS field Microbranches based on sources latched and held with the absence of an Issue Execute function (the instruction was not issued). Microbranches based on results are latched and held with a Hold Microword function (the previous instruction was not issued).

3.3.11.2 Subroutine Call And Return

A microstack is maintained to support microsubroutines. When a subroutine is called, the stack pointer is incremented and the previous UPC is placed on the microstack. The UPC specified in the NEXT ADDRESS field is the subroutine location.

On return from the subroutine, the top UPC on the microstack is OR'd with the NEXT ADDRESS field to locate the macro instruction following the subroutine call. The microstack pointer is then decremented.

3.3.11.3 Microtrap On Exception

If a macro instruction requires only one (or very few) cycles, microcode does not wait for exception conditions to become valid; it sets the MICROTRAP ENABLE field. The field is loaded into the Result Queue with the destination pointer. When an instruction is about to retire (indicating when the exception conditions are valid), the enable field causes the microsequencer to check the appropriate conditions and microtrap (change the microflow) if necessary.

When an exception is detected, the Issue unit stalls for one cycle while the first microword of the exception handling routine is accessed. In the next cycle the Issue unit forces an Issue Valid function. Normal microcode control resumes. The microcode pushes the PSL and PC, and flushes the E, I and M box pipes behind the instruction which trapped or faulted. It then generates the address of the macro exception handling routine.

3.3.11.4 Issue Detected Faults

Prior to issuing an instruction, the Issue Unit checks two fault bits:

- The Source Pointer Fault bit which indicates that a reserved addressing mode fault was detected by the IBox.
- The Source List Fault bit which indicates a memory access problem.

Also, the Issue unit will monitor the MBox Fault and Fault Valid bits for Opwrite and EBox write operations for memory problems. In addition, destination reserved addressing modes are flagged with the Destination Pointer Fault bit.

3.3.11.5 IBox Exception Forks

Some exception conditions are detected by the IBox (e.g., IBuffer request problems, reserved opcode). For these exceptions, the IBox passes the UPC for the error handling microcode rather than the UPC for instruction execution.

3.3.11.6 EBox Traps And Faults

When a trap is detected, the offending instruction is completed, the results and condition codes are written. The microtrap routine is then initiated.

When a fault is detected, the offending instruction is NOT completed, the results are not written and the condition codes are unpredictable. The microtrap routine is then initiated.

3.3.12 Interrupt Handling

There are conditions that are independent of the current CPU process, but which require immediate attention by the CPU. Since some conditions are more urgent than others, each is assigned an interrupt priority level (IPL).

When the microsequencer detects interrupts, it selects the highest priority interrupt and compares it against the IPL of the current process. The compare is performed between macro instructions (i.e., at fork time). If the interrupt priority is higher than the IPL, the microsequencer dispatches to the interrupt handling microcode.

As with microtraps, the Issue unit must stall one cycle and force an issue the next cycle when the microcode is ready. The microcode will push the PSL and PC onto the stack, and calculate the PC of the interrupt handling routine to dispatch the IBox.

The interval counter registers and software interrupt registers are in the microsequencer logic of each EBox, allowing each EBox to detect their assigned interrupts. The console interface and handshaking is handled in the JBox. Console interrupts are treated (by the EBox) as an I/O interrupt from the JBox.

AST level interrupts are detected by EBox microcode on MTPR, IPL, and REI instructions. The remainder of the interrupts are detected in the JBox. The JBox selects which processor will handle them, and sends a serial interrupt packet. The packet encodes the interrupt type, source, and IPL.

On each cycle, the Software Interrupt Status Register (SISR) is checked for outstanding software interrupts. For example, if SISR <02> were set, there would be an outstanding software interrupt at IPL level 2.

The highest software interrupt is arbitrated against other pending interrupts, and may be taken before the next instruction. The SIRR register is not implemented in the microsequencer. When MTPR writes SIRR, the microcode sets the appropriate bit of SISR, which is implemented as an HIR.

3.4 MBox Introduction

The Mbox is an independent functional unit that provides the CPU interface to main memory, I/O, and other processor subsystems. Its major functions are to:

- fetch Istream data on behalf of the IBox
- translate virtual addresses to physical addresses
- provide a relatively large data cache and translation buffer (TB)
- perform memory management functions including processing TB misses as they occur.
- process unaligned and page crossing references.

3.4.1 Pipeline Stages

As shown in Figure 3-17, the MBox can be viewed as having a 2-stage pipe. The first stage consists of the IBox and EBox input ports, the TB, and TB fixup unit. The second stage consists of the EBox, IBox, and Write Queue input ports, data cache, and the Data Traffic Managers including their output ports. The following subsections introduce the major hardware features.

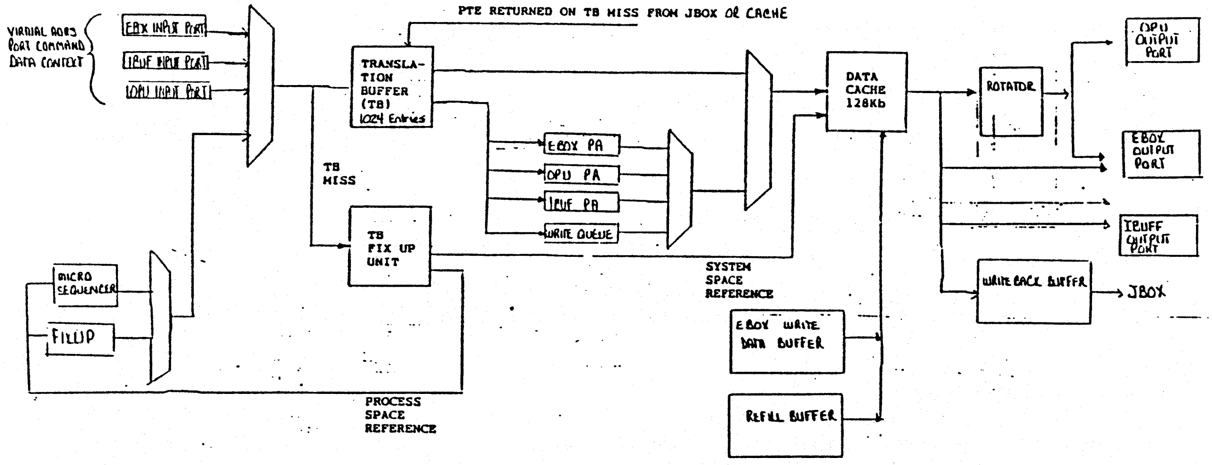


Figure 3–17 MBox Block Diagram

3.4.2 Address Translation

The MBox receives virtual addresses (VA) from the E and I boxes. The VAs are translated into physical addresses (PA) through the recently used Page Table Entries (PTE) stored in the Translation Buffer (TB).

The Translation Buffer (TB) has 1024 entries: 512 entries to describe Process Space (P0/P1), and 512 entries to describe System Space (S0). The TB consists of two units:

- o VA Tag Store - contains all the virtual bits that are not part of the address, and the Valid (V) bit.
- o PTE Store - contains the PTEs: Page Frame Number (PFN), the protection bits, and the Modify (M) bit.

3.4.3 TB Miss Handling

If the PTE required for addrerss translation is not in the TB, a TB Miss occurs. This is one of the most frequent disruptions to instruction processing. The miss requires that the PTE be fetched from the memory or the data cache. This service is normally provided by the EBox in most VAX systems. However, in this system the TB has a TB Fix-up Unit which can resolve a miss faster than Ebox microcode, and save recontructing complicated machine states.

As soon as the miss is detected, the fix-up unit begins the process of retriving the required PTE. Thus, the miss can be resolved early in the pipeline, often burying the retrieval delay under other operations. The fix-up unit has two ports: TB and data cache.

3.4.4 Data Cache Organization

The data cache has a capacity of 128K bytes, and is protected by ECC. It is 2-way set associative, with 64K bytes per set.

The cache operation is pipelined. Most read operations are processed at a rate of one per cycle. A write operation requires two cycles, the tag store must be checked and produce a hit before the write can be issued.

While most cache references are for longwords, a significant increase in performance is obtained since the cache is able to handle aligned quadword references. The cache can produce a quadword of read data every cycle; it can produce a quadword of write data on every second cycle. Cache refills and write-backs are done at a rate of 64 bits per cycle.

3.4.5 Write Queue.

The IBox Operand Store Unit (OPU) processes memory destination specifiers and sends the virtual addresses to the Mbox. There they are translated to physical addresses, and stored in an 8-entry FIFO buffer (Write Queue).

When the write data arrives from the Ebox, it is paired with the top entry in the Write Queue, and written in the cache. This allows the OPU to continue processing other specifiers without waiting for the write operation to complete.

To prevent invalid (stale) data from being read during operand read operations, read addresses are compared with entries in the Write Queue. If they match, the read is delayed until the write has been processed.

3.4.6 Data Traffic Manager

To be supplied.

3.4.7 Physical Organization

Figure Figure 3–18 describes the MBox MCU locations on the planar module. Figure Figure 3–19 describes the MBox physical partitioning. The followingm list summarizes the MCU content.

o Virtual Address Port (VAP) - This MCU contains the translation buffer, and the port arbitration decode and control logic. It has five MCAs and 26 STRAMs.

o Data Cache (DTA and DTB) - These two MCUs contain the data cache and the cache input and output buffers, including the refill buffer and rotator. Each MCU has three MCAs and 40 STRAMs.

o Cache Tag Unit (CTU) - This MCU contains the cache tag control to determine if the requested data is valid and in local cache (cache hit). It also contains the JBox command encode logic. This MCU has 18 1K X 4 STRAMs and 8 4K X 4 STRAMs.

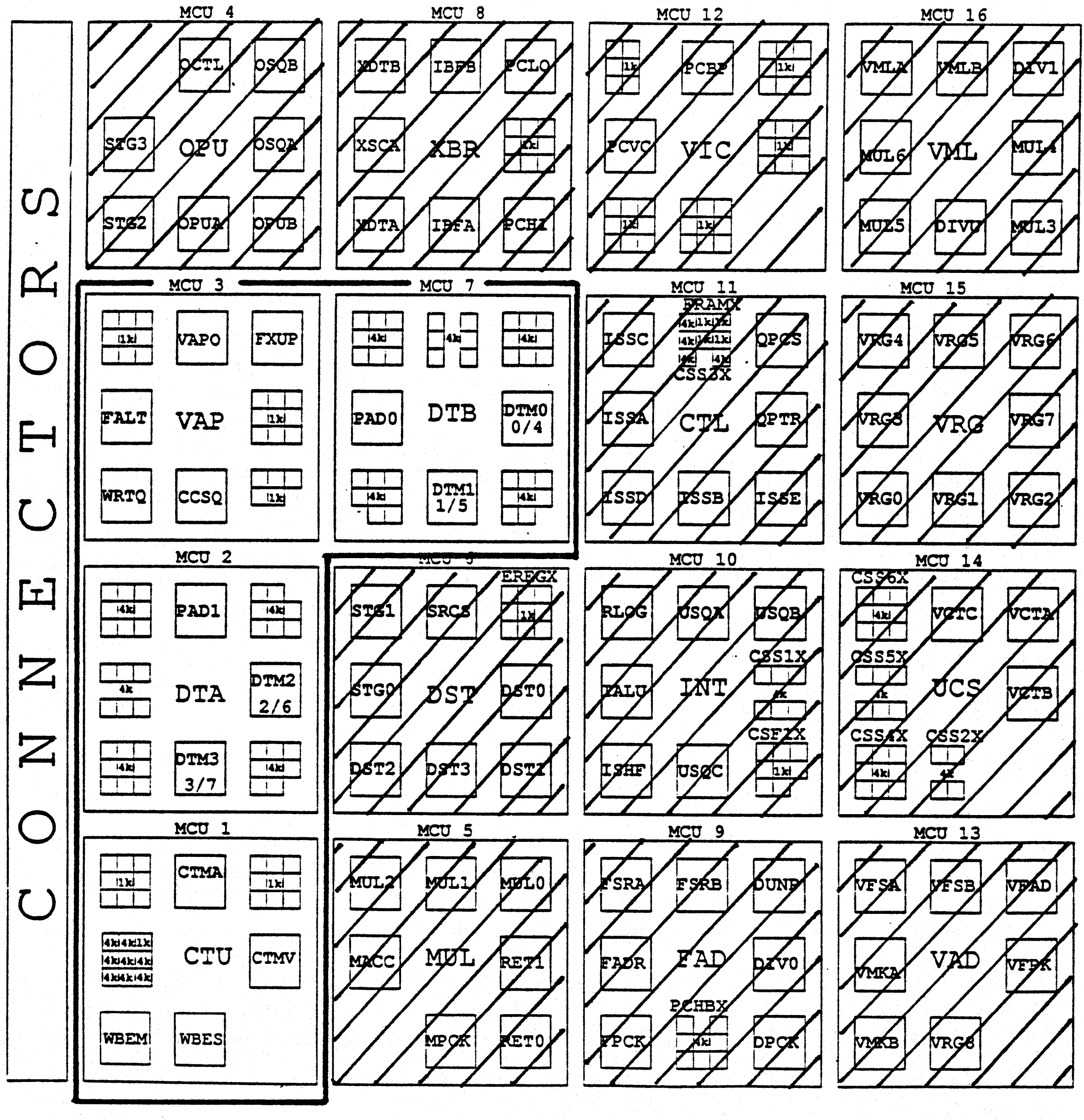


Figure 3–18 MCU Planar Module Locations

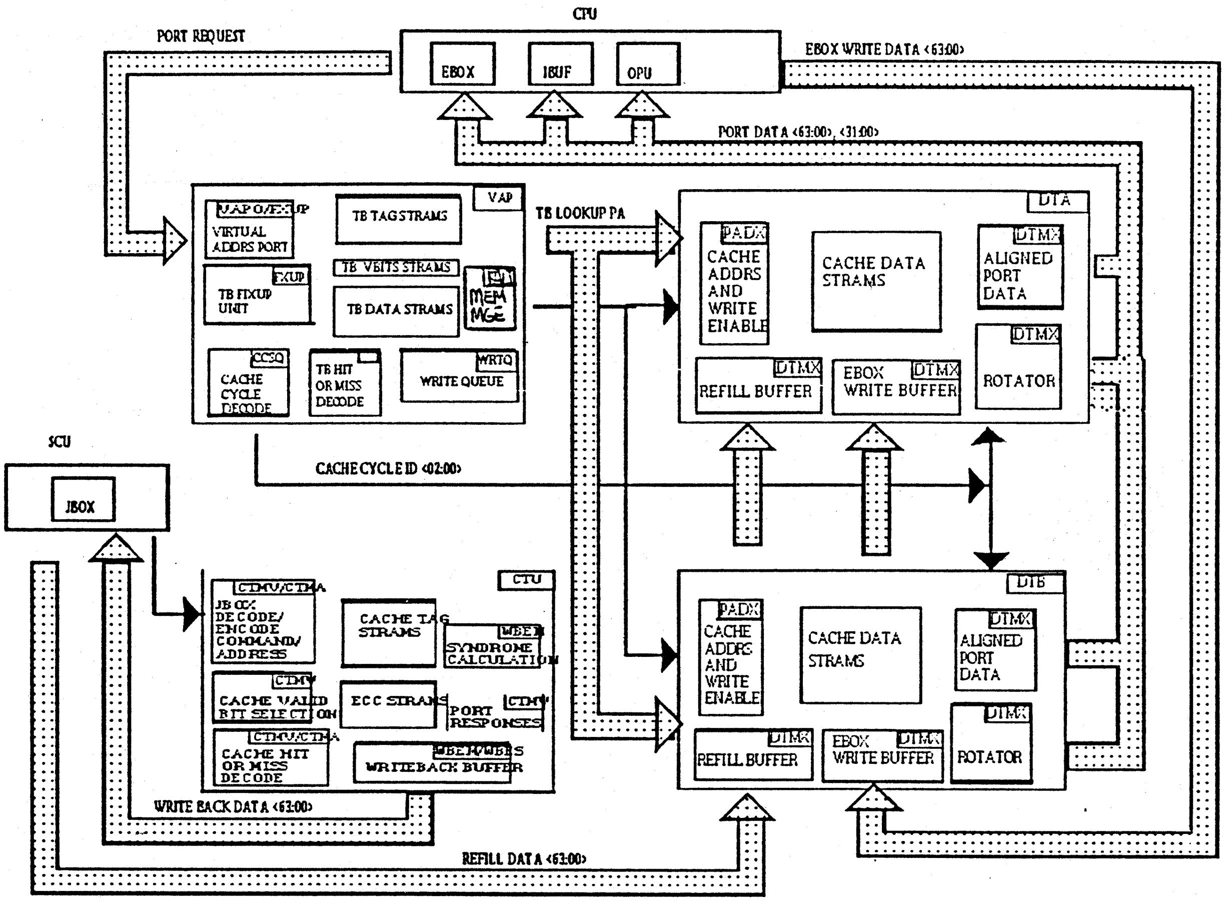


Figure 3–19 MBox MCUs

3.4.8 Virtual Address Port

The VAP contains the translation buffer and performs port arbitration and command decode. It consists of the following five MCAs:

- VAPO - This MCA is the virtual address port of the MCU. Its primary function is arbitration for the translation buffer. It latches VA <11:00>, and decodes the commands received from the two IBox ports (Instruction Buffer (IBUF) and Operand Processing Unit (OPU)), EBox, fixup, and sequencer ports.
- FXUP - This MCA contains the TB fixup unit and receives VA <31:12> from all the ports. The fixup unit accesses memory management registers and fetches valid page table entries for a port's virtual address that the translation buffer could not translate.
- FALT - This MCA performs the translation match and control. It handles memory management faults, and controls writing and invalidating translation buffer STRAMs. It receives cache data when accessing PTEs from memory.
- WRTQ - This MCA contains the write queue which has eight address entries and corresponding status information.
- CCSQ - This MCA decodes the cache operations and sends control signals to CTU, DTA and DTB.

The VAP STRAMs are as follows:

- TB STRAMs - There are 21, 1K X 4 STRAMs for the entire translation buffer. This includes the four copies for bits <15:09>.
- FIXUP STRAMs - There are six STRAMs devoted to the TB fixup unit.

3.4.9 Data Cache

The following is a listing of the MCAs on the DTA and DTB:

- PADX - These MCAs (PAD0 and PAD1) drive all the address lines and write enables for the cache data STRAMs. PAD0 is on DTB, and PAD1 is on DTA.
- DTMX - These MCAs (DTM0, DTM1, DTM2, and DTM3) buffer and control cache STRAM write and read data. DTM0 and DTM1 are on DTB, and DTM2 and DTM3 are on DTA. Each DTMX deals with two byte-slices of the quadword interface and byte-slices of the refill buffer and rotator.

There are 80, 4K X 4 STRAMs in the 128 Kbyte cache: 40 STRAMs on DTB, 40 on DTA.

3.4.10 Cache Tag Unit

The following is a listing of the MCAs on the CTU:

- CTMA - This MCA controls the cache tag STRAMs (along with CTMV). It performs address comparison and interfaces to the JBox. CTMA receives PA <32:06>. It drives the cache tag STRAM address lines, and is partially responsible for tag matching, generating cache hit and assembling a command and address to be sent to the JBox.
- CTMV - This MCA controls the 16 valid bits (one valid bit per longword in a 64 byte cache block), and generates port responses. It is partially responsible for generating cache hit, and assists in assembling the command and address to be sent to the JBox.
- WBEM - This MCA contains byte-slices of the write back buffer. Generates full ECC using the partial ECC from WBES. It compares the stored ECC against new ECC, and generates syndrome bits for bit correction. It generates ECC control bits and sends them to WBES. It differentially drives data to the JBox, and receives the command and address from the JBox and forwards it to CTMA/CTMV.
- WBES - This MCA contains byte-slices of the write back buffer. It generates partial ECC, that is sent to the WBEM. It receives ECC control signals for bit correction from the WBEM. It differentially drives data to the JBox, and receives the command and address from the JBox, and forwards it to CTMA/CTMV.

The following is a listing of the CTU STRAMs:

- TAG STRAMs - There is a total of 18, 1K X 4 tag store STRAMs: nine for cache SET0, nine for cache SET1. The format for each entry in the tag store consists of physical address bits <32:16>, valid bits <15:00>, a written bit, and two parity bits.
- ECC STRAMs - There is a total of 8, 1K X 4 STRAMs: four for ECC SET0, four for ECC SET1.

3.4.11 Memory Management

A virtual memory system consists of virtual address space and a memory mapping and protection mechanism.

3.4.11.1 Virtual Address Space

The virtual address space (Figure 3-20) seen by the programmer is a linear array of 4 Gbytes divided into a collection of 512-byte units called pages. The page is the basic unit of both relocation and protection.

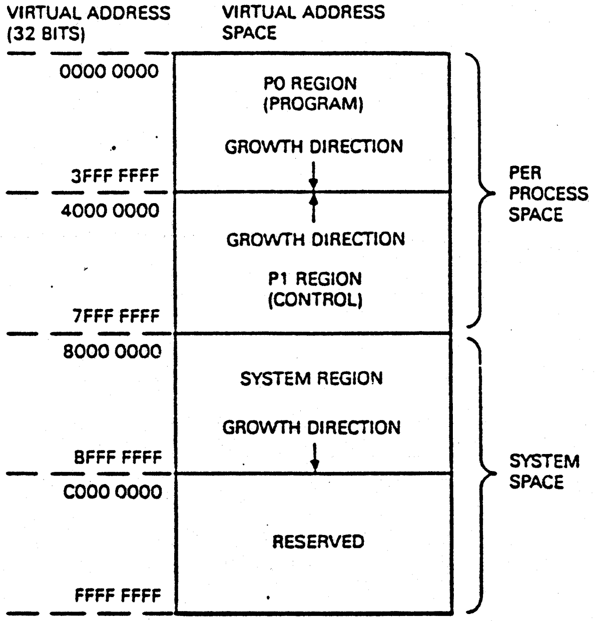


Figure 3-20 Virtual Address Space Allocation

Eight Gbytes is the maximum amount of physical memory. Physical memory exceeds the virtual address limits (4 Gbytes). Memory management provides the mechanism to map the active part of the virtual address space to the available physical address space. It also provides access protection for each page.

Virtual address bits <31:00> (Figure 3-21) specify the address space, that is divided into two address spaces of equal size.

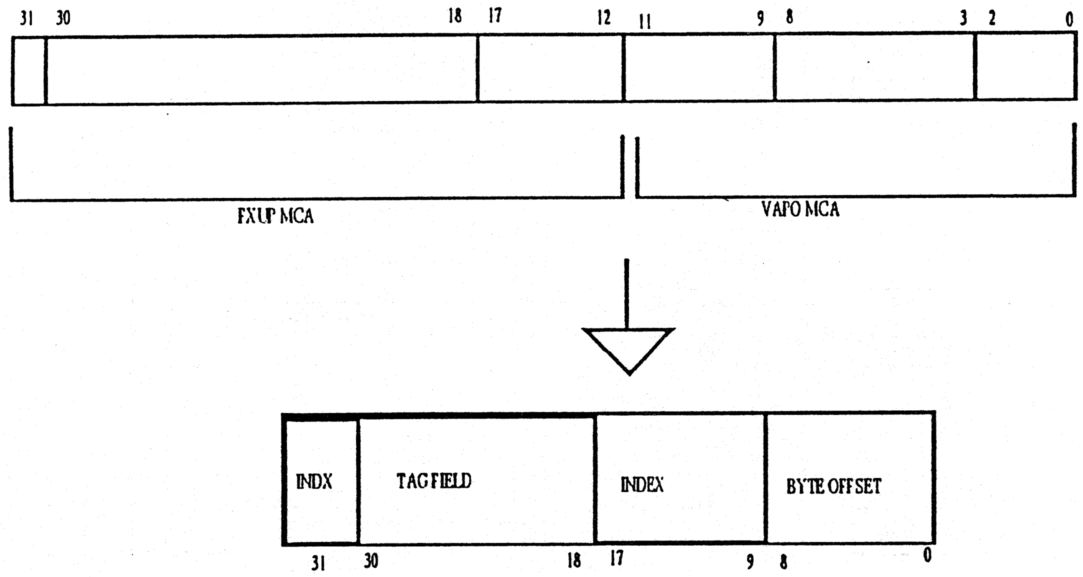


Figure 3–21 Virtual Address Format

The lower half of the address space, called process space, is divided into two regions, P0 and P1 space. The upper half of the address space, called system space, is divided into two regions, S0 and S1 space. The upper half of system space, S1, is currently unused and reserved for future expansion. The operating system resides in system space while the currently active process resides in P0 space. Both the process and operating system use P1 space.

3.4.11.2 Memory Mapping and Protection

The operating system controls the allocation of physical memory to the virtual address space through the use of mapping tables (Figure 3–22) in physical memory. The operating system maps inactive, but used parts of the virtual address space onto external storage media. There are three separate page tables that must be set up by the operating system before memory management can be enabled.

SPT - System page table defines S0 space
P0PT - P0 page table defines P0 space
P1PT - P1 page table defines P1 space

Each page table contains a list of 32-bit entries called page table entries (PTEs)(Figure 3–23). PTE bit definitions are listed in Table 3–14.

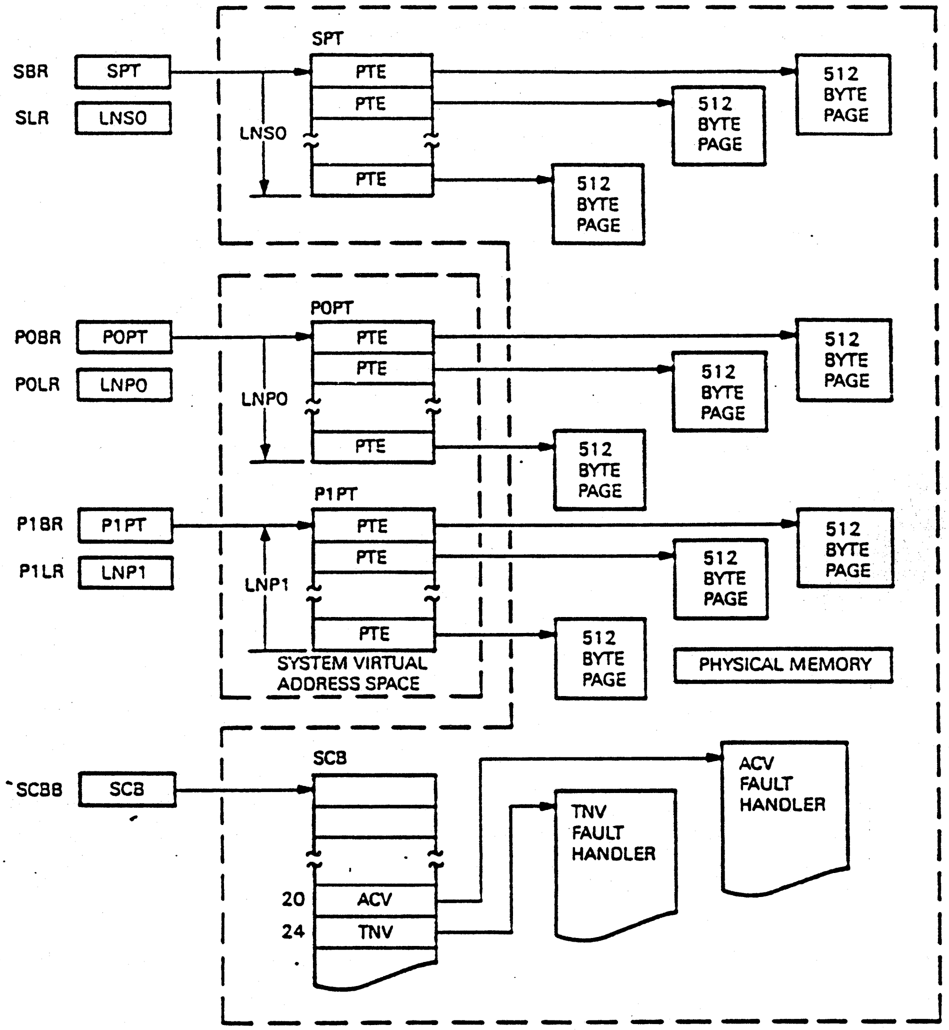


Figure 3–22 Memory Mapping

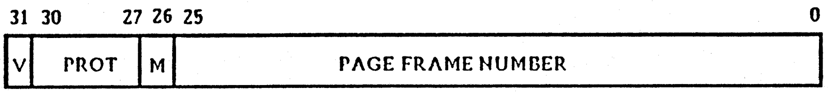


Figure 3–23 Page Table Entry (PTE)

Table 3–14 PTE Bit Definitions

Bit	Name	Definition
31	Valid	When set, indicates that the entry is valid and that the entry may be used for address translation. When clear, indicates an invalid entry that requires intervention by the operating system to correct the fault.
30:27	Protection	4-Bit protection code that specifies the type of access allowed as summarized in Figure 3-24
26	Modify Bit	When set, indicates that the page has been modified since being read into memory form the disk. When clear, indicates that the page has not been modified since being read into memory from the disk.
24:00	PFN	Specifies the page frame number in physical memory. During address translation, these bits are used to form the physical address of the page in memory.

CODE DECIMAL	 BINARY	MNEMONIC	K	E	S	U	COMMENT
0	0000	NA	-	-	-	-	NO ACCESS
1	0001		UNPREDICTABLE				RESERVED
2	0010	KW	RW	-	-	-	
3	0011	KR	R	-	-	-	
4	0100	UW	RW	RW	RW	RW	ALL ACCESS
5	0101	EW	RW	RW	-	-	
6	0110	ERKW	RW	R	-	-	
7	0111	ER	R	R	-	-	
8	1000	SW	RW	RW	RW	-	
9	1001	SREW	RW	RW	R	-	
10	1010	SRKW	RW	R	R	-	
11	1011	SR	R	R	R	-	
12	1100	URSW	RW	RW	RW	R	
13	1101	UREW	RW	RW	R	R	
14	1110	URKW	RW	R	R	R	
15	1111	UR	R	R	R	R	

- = NO ACCESS
R = READ ONLY
RW = READ/WRITE

K = KERNEL
E = EXECUTIVE
S = SUPERVISOR
U = USER

Figure 3–24 Protection Codes

Each PTE describes the location and protection of a 512 byte page in physical memory.

The operating system directly maps the system page table to physical memory. When a TB miss occurs for S0 address space, the MBox performs only one TB lookup operation. The operating system maps P0PT and P1PT page tables to system virtual address space. When a TB miss occurs for a process, P0 or P1 address space, the MBox performs two TB lookup operations.

3.4.11.3 Data Size

Virtual addresses can specify byte, word, longword, quadword, hexword and cache block boundaries (See Figure 3–25) and Figure 3–26).

WORD ADDRS BOUNDARY (2 BYTES)	LONGWORD ADRS BOUNDARY (4 BYTES)	QUADWORD ADDRS BOUNDARY (8 BYTES)	OCTAWORD ADDRS BOUNDARY (16 BYTES)	HEXWORD ADDRS BOUNDARY (32 BYTES)	CACHE BLOCK ADDRS BOUNDARY (64 BYTES)
1000	1000	1000	1000	1000	1000
1002	----	----	----	----	----
1004	1004	----	----	----	----
1006	----	----	----	----	----
1008	1008	1008	----	----	----
100A	----	----	----	----	----
100C	100C	----	----	----	----
100E	----	----	----	----	----
1010	1010	1010	1010	----	----
1012	----	----	----	----	----
1014	1014	----	----	----	----
1016	----	----	----	----	----
1018	1018	1018	----	----	----
101A	----	----	----	----	----
101C	101C	----	----	----	----
101E	----	----	----	----	----
1020	1020	1020	1020	1020	----
1022	----	----	----	----	----
1024	1024	----	----	----	----
1026	----	----	----	----	----
1028	1028	1028	----	----	----
102A	----	----	----	----	----
102C	102C	----	----	----	----
102E	----	----	----	----	----
1030	1030	1030	1030	----	----
1032	----	----	----	----	----
1034	1034	----	----	----	----
1036	----	----	----	----	----
1038	1038	1038	----	----	----
103A	----	----	----	----	----
103C	103C	----	----	----	----
103E	----	----	----	----	----
1040	1040	1040	1040	1040	1040

Figure 3–25 Address Boundaries

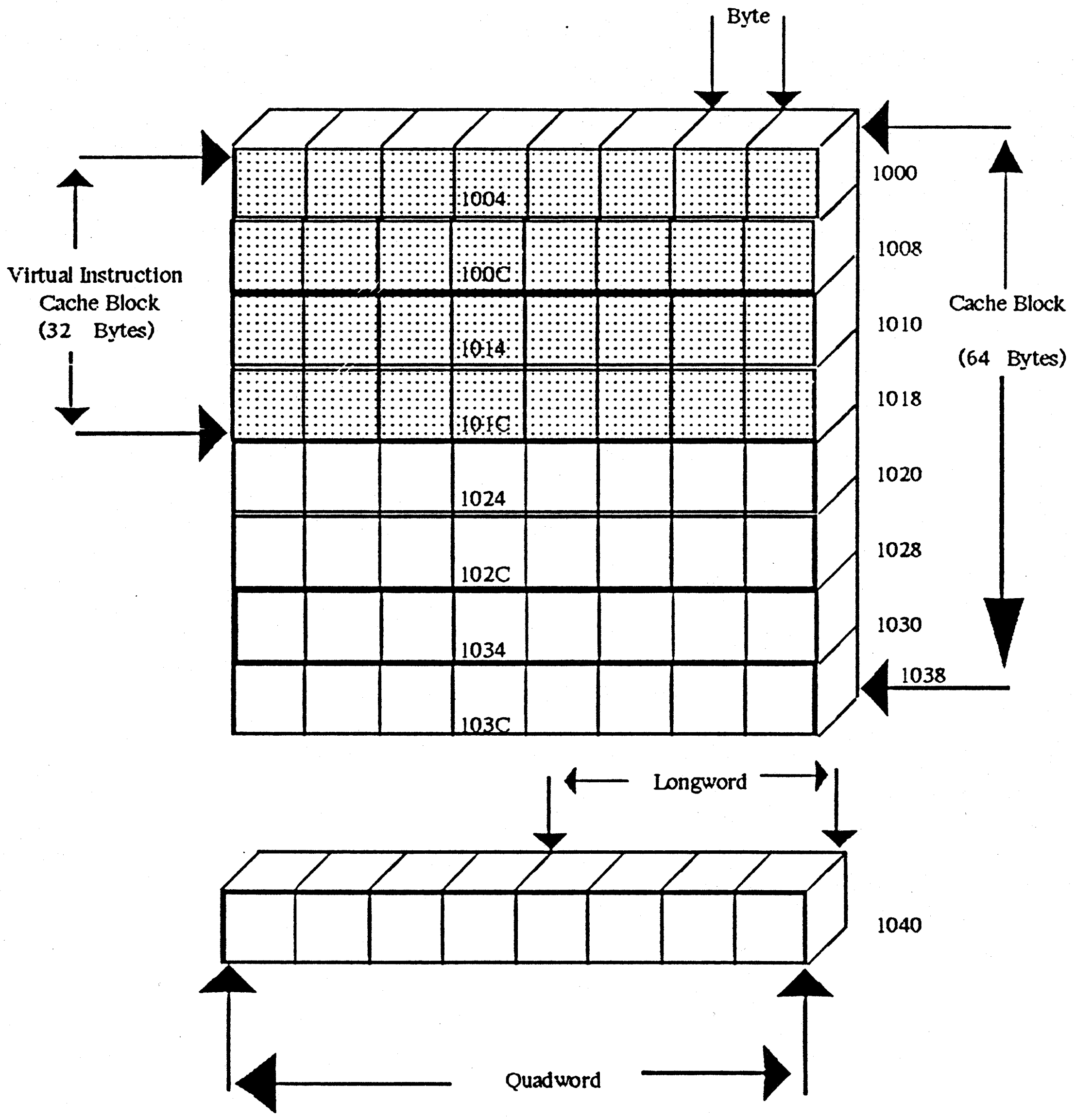


Figure 3-26 Cache Block

3.4.12 Translation Buffer

The Translation Buffer (TB) (Figure 3-27) contains 1024 locations (1K) and is divided into two halves with 512 locations for user space and 512 locations for system space. The TB, on the VAP MCU, is direct mapped and is made up of 23 1K X 4-bit STRAMs that are addressed by bits 31, and 17:09 of the virtual address.

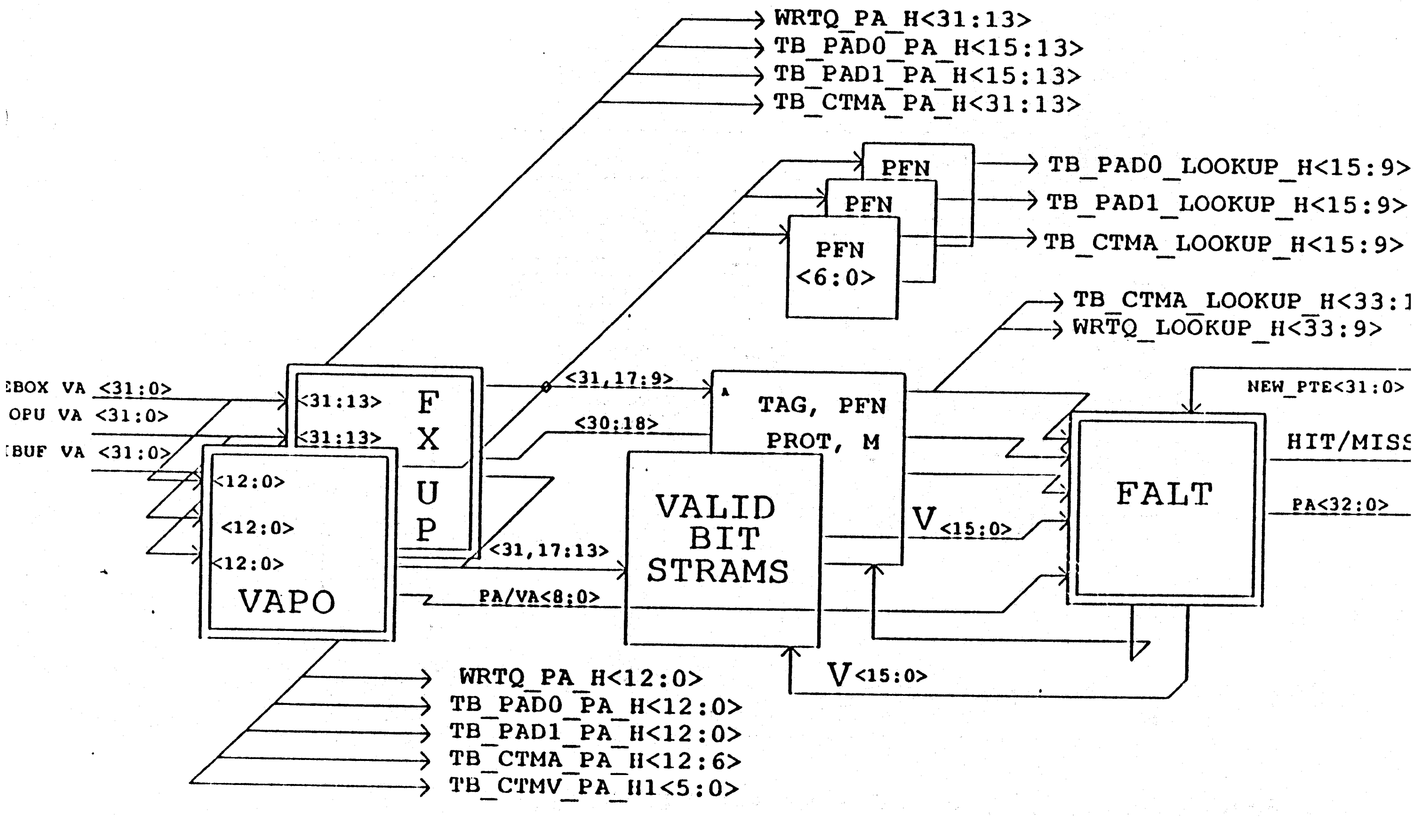


Figure 3-27 Translation Buffer Block Diagram

3.4.12.1 TB Lookup

TB lookup occurs after a port wins TB arbitration. During TB lookup, FALT determines if the valid PTE is in the translation buffer. Figure 3-28 illustrates the tag store containing the TB tags, PTE store containing the page table entries, and the valid bits.

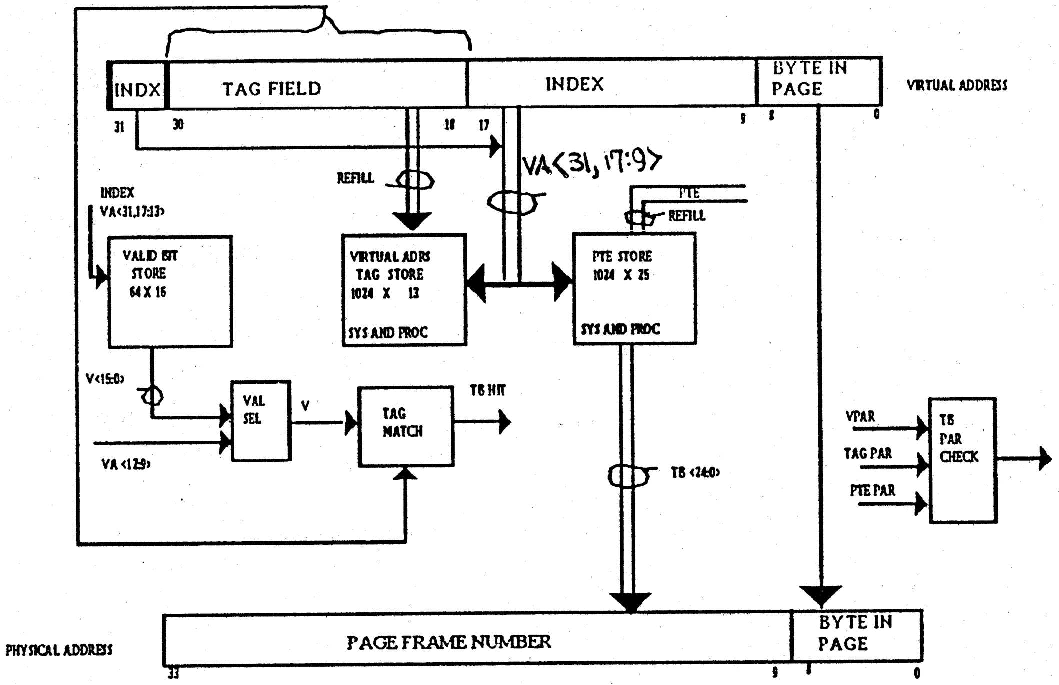


Figure 3-28 TB Tag Store, PTE Store, and Valid Bit Store

If the tag matches and the entry is valid, a TB hit results, and the physical address field of the page table entry is extracted to form PA <33:09>. The TB tag is shown in Figure 3-29, and the related bits are listed in Table 3-15.

TAG <30:18>	TAG PAR

Figure 3-29 TB Tag

Table 3-15 Tag Bit Definitions

Bit	Definition
TB Tag <30:18>	Contains bits 30:18 of the virtual page number. During translation the associated tag is accessed using VA 12:09, 31 and compared with VA<30:18> to determine if the PTE required to perform the virtual to physical address translation exists in the TB PTE store. During TB refill operations the TB tag is written from VA 30:18. Again using VA<12:09, 31> to select the associated tag. VA <31> selects either the system half or the process half of the TB. VA <31> is set to 1 for system half, and is 0 for process half.
TB Tag Par	A single address parity bit (odd parity) checks the integrity of the TB tag store during address translation. It is read and checked during translation, and generated and written during TB refills.

TB data (PTE) is shown in Figure 3-30, and corresponding bits are listed and described in Table 3-16.

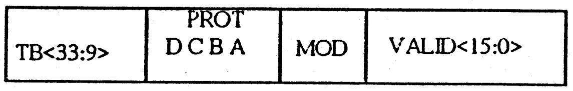


Figure 3-30 TB Data Format

Table 3-16 TB Data Format

Bit	Definition
TB <33:09>	Contains physical address bits <33:09> that specify the physical page frame number (PFN). During translation these bits are used in conjunction with VA<08:00> to form the required physical address. Like the tag, the associated PTE is accessed using VA<17:09> with VA<31>, selecting either the system half or the process half of the TB. During TB refill operations these bits are written from a PTE obtained by the TB fixup unit from local cache or main memory.
TB PROT <D:A>	This four bit field specifies the type of accesses allowed for the page described by the address segment of the PTE as summarized in Figure 3-24.
TB MOD	This single bit indicates that the referenced page has been modified.
TB VALID B	This bit specifies that the PTE is valid and the specified page is in the working set. During translation this bit must be found set to qualify for a TB hit and subsequent physical address generation.

Bit 31 selects either system space or process space, and bits 17:09 address the selected STRAMs. The 23, 1K X 4-bit STRAMs are divided into the following major areas.

o TB data field contains the page frame number (PFN), the protection bits and the modify bit.

o TAG data field is 13 bits wide and stores VA <30:18>. During a TB lookup, FALT compares these stored VA bits with VA <30:18> of the incoming VA, and determines if there is a TB hit or TB miss.

o Valid bit field uses four STRAMs. There are 64 entries of 16 valid bits. Virtual bits 31, and 17:13 address the four, 1K X 4 STRAMs. VA bits 12:09 select the appropriate valid bit. The selected valid bit determines a TB hit or miss.

3.4.12.2 TB Parity

Parity for the TB tag, TB data, and valid bits is shown in Figure 3-31.

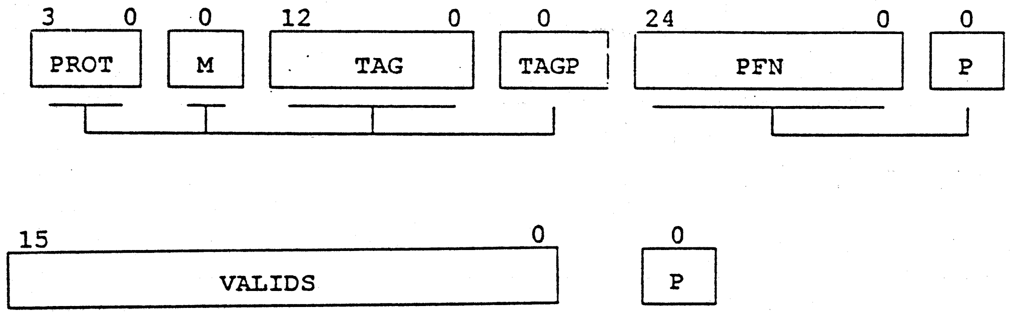


Figure 3-31 TB Parity

3.4.12.3 TB Hit

VAPO and FXUP (Figure 3-32 and (Figure 3-33) receive virtual addresses from the ports. They translate the virtual addresses to physical addresses using the translation buffer.

PADX uses the physical address for cache lookup. VA <08:00> directly form PA <08:00>, and specifies a byte in a 512-byte page. VA<31> selects either system space or process space, and enables the half of the translation buffer that will perform the translation. VA <30:09> perform the TB lookup. Simultaneously, VA <16:09> select an entry in both the address tag STRAMs and PTE STRAMs. FALT (Figure 3-34 control compares VA <30:17> with the contents of the tag entry.

Conditions for a TB hit are shown in Figure 3-35.

All parts of the TB simultaneously receive the virtual address so that, as the physical address is obtained, an associative lookup determines validity. Parity is stored and checked for the tag, valid bits, and TB data.

The TB allocates one valid bit per TB entry. They are arranged physically as 16 valid bits per valid entry. When invalidating the TB, FALT can clear 16 valid bits.

3.4.12.4 TB Miss

If the comparison fails between the TB tag and the virtual address and the required information is not stored in the translation buffer, there is a TB miss. FXUP retrieves the page table entry needed for the address translation from the system page tables in local cache or main memory. When the page table entry is written into the TB STRAMs, VAP retries the original request.

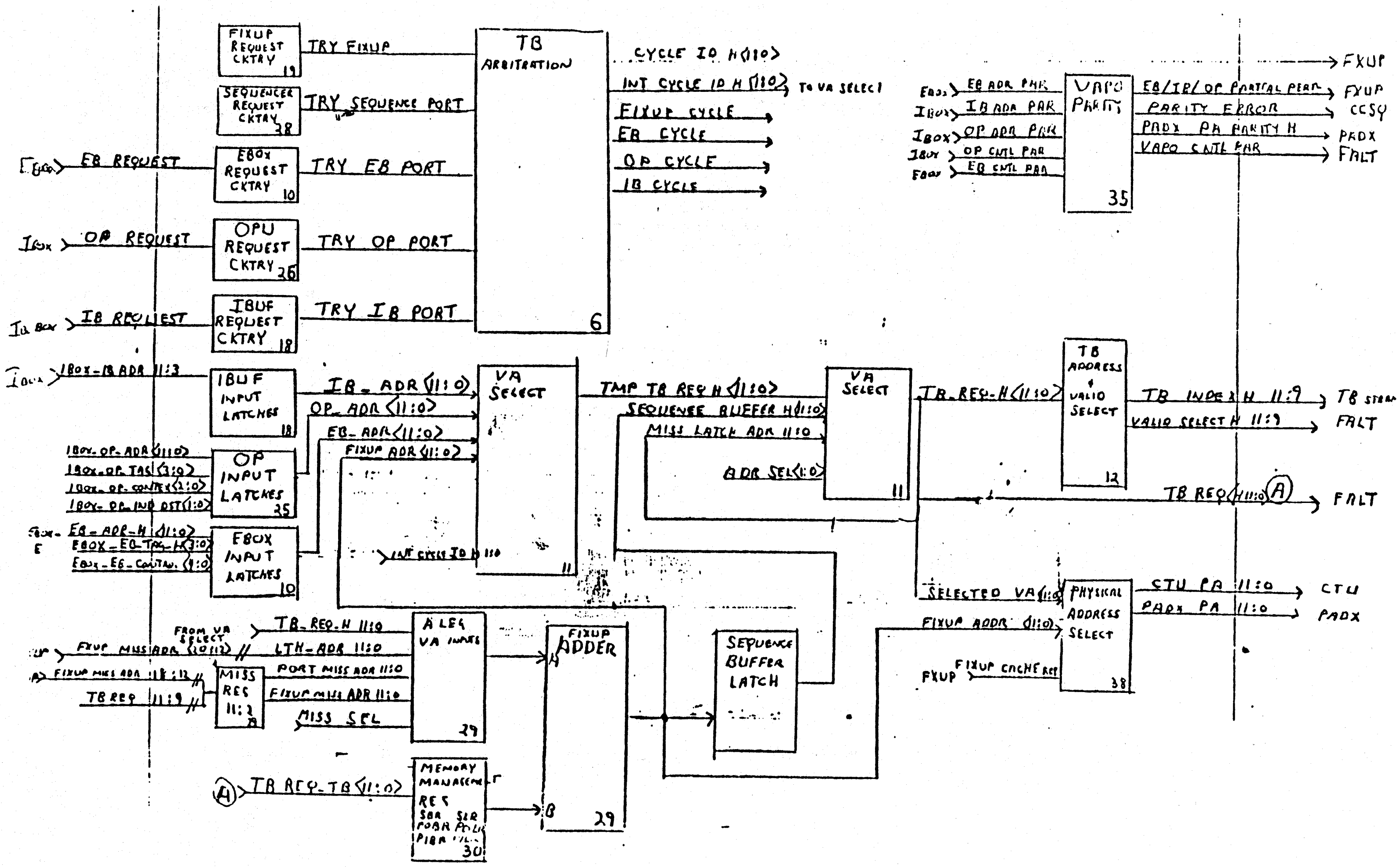


Figure 3–32 VAPO MCA Block Diagram

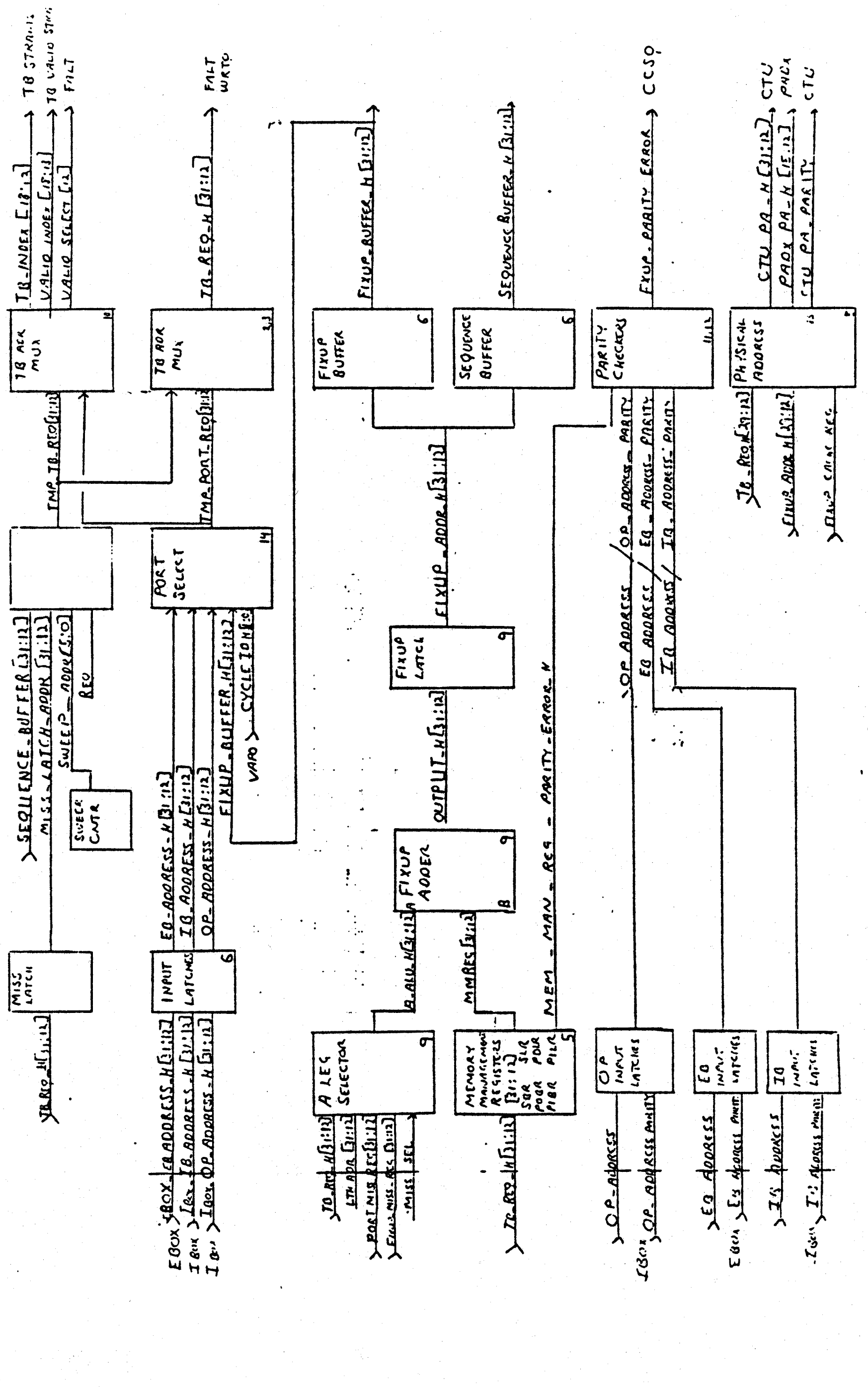


Figure 3-33 FXUP MCA Block Diagram

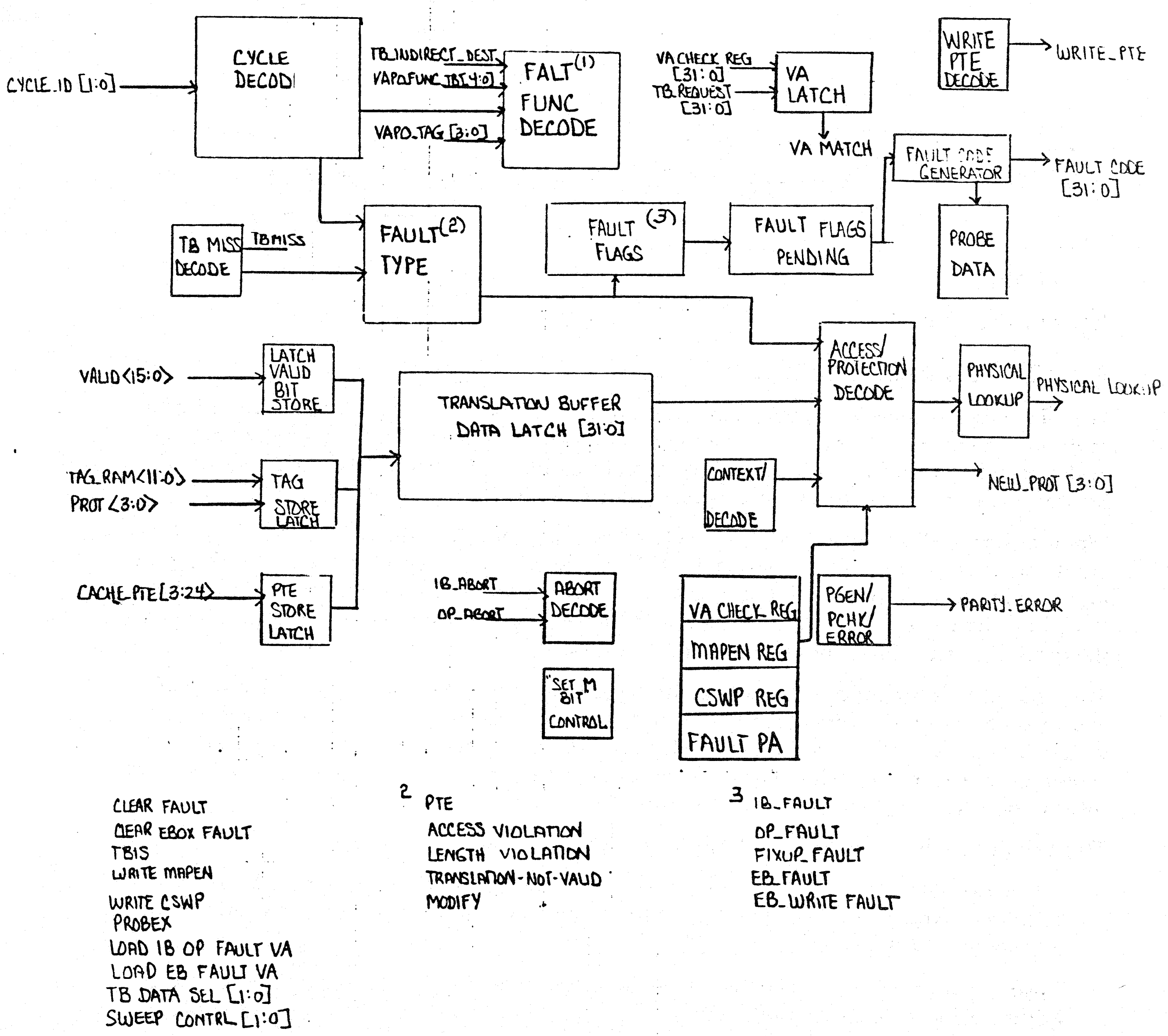


Figure 3–34 FALT MCA Block Diagram

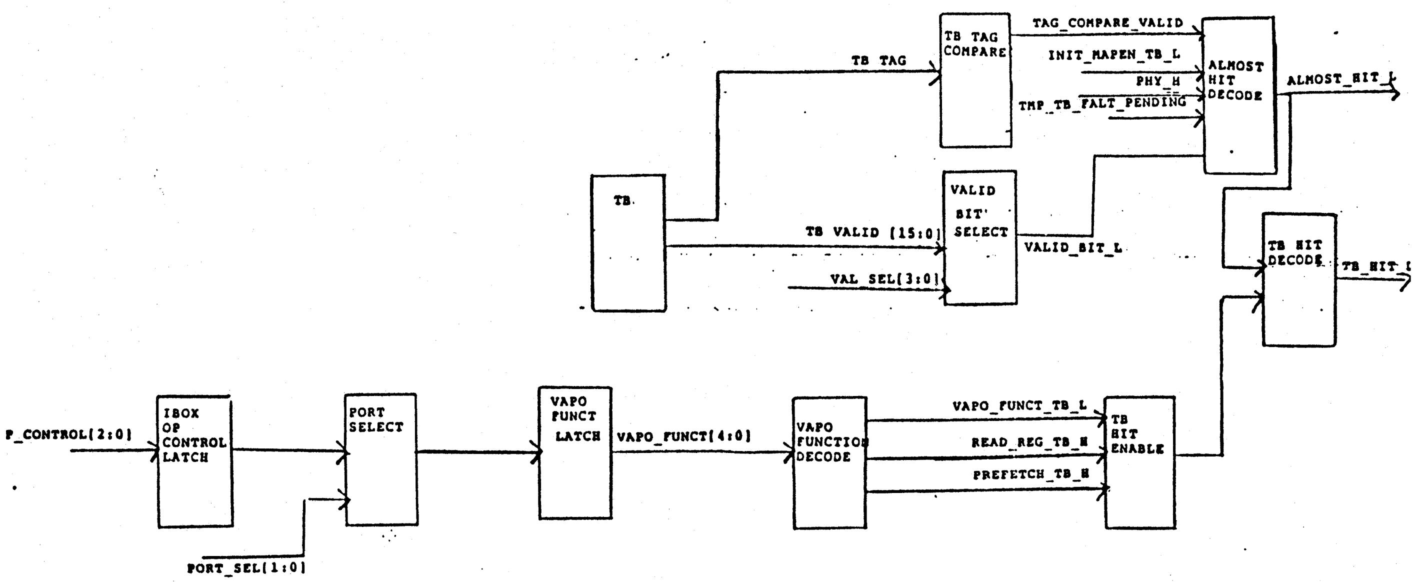


Figure 3-35 TB Hit

3.4.13 Translation Buffer Arbitration

The MBox can receive a request, command, and address from a port into an input buffer each and every cycle. There are five major ports (See Figure 3-17.) requesting access to the translation buffer. VAPO (TB arbitration) grants access to the TB on the basis of priority. The request with the highest priority gains control and is serviced first.

When a port VAP services a port, CCSQ asserts the grant line (EB_ACK), and drops this grant line if either of two conditions exist. The first condition is when the requesting port loses arbitration. In this case, the grant line is deasserted at the conclusion of arbitration (6 ns into the cycle). The input latches hold the request information, and the port inspects the grant line and does not send a second request.

The second type of condition is when the page table entry is not in the translation buffer (TB miss) or when the requested data is not present in cache (cache miss). A stall results: for a TB miss, a cache STRAM lookup is required; for a cache miss, a refill operation is required. Because the stall is not detected until late in the cycle, a buffer holds all request information for the port that won arbitration.

The priority of the ports,(Figure 3-36) from highest to lowest, is as follows:

- o TB Fixup Port - The TB fixup unit is an internal MBox port. It performs TB miss routines. When a miss is detected in the translation buffer, it sends a system virtual address to the TB to access user PTEs.
- o MBox Sequencer Port - The sequencer is an internal MBox port. Using the TB fixup adder, it increments, by a constant four, an unaligned address or an address of multiprecision operands.
- o EBox Port - The EBox port is a read/write port, that is, the address lines source address or data. It latches the request, a 32-bit address for either a virtual or physical reference, 5-bit function, 4-bit tag, and 4-bit context. The EBox requests include read or write a register (using the address lines to hold data and the tag field to select the register), flush P0/P1 half of the translation buffer, initialize the translation buffer, read and write data, and invalidate a single TB entry.

- OPU Port - The OPU port latches a 32-bit virtual address to prefetch all memory related operands. It latches the request, 5-bit tag field, 4-bit context, 3-bit function field, and 2-bit indirect field. The OPU requests include OPU read, OPU write, and OPU read with write check.
- Instruction Buffer Port (IBUF) - When the IBox encounters a miss in its virtual instruction cache (VIC), the IBUF port latches a request and VA <31:3>. The MBox responds with a 32-byte block of cache data.

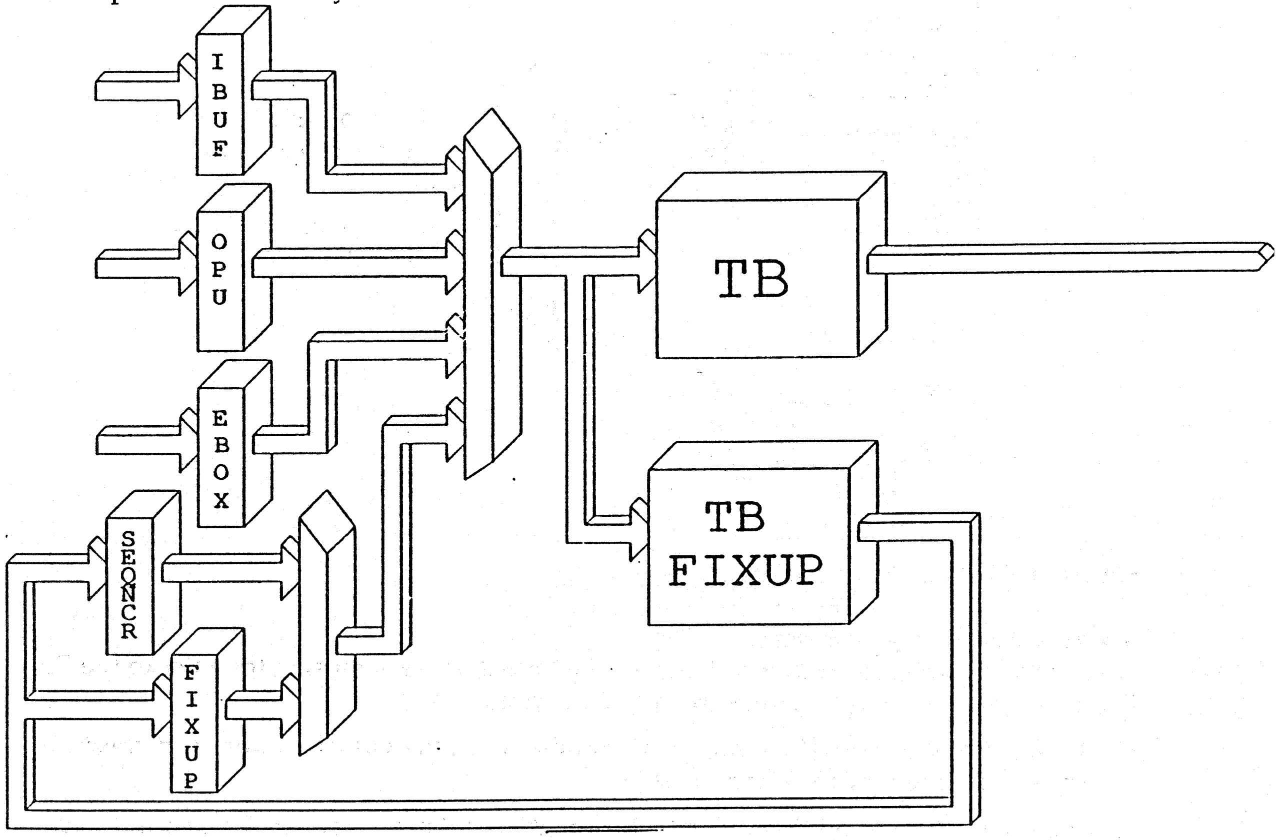


Figure 3–36 TB Ports

3.4.13.1 TB Fixup Port

The TB Fixup Port contains a TB fixup unit (miss processor), which is an address translation processor within the MBox. The miss processor is shown in Figure 3–37. It is responsible for fetching PTEs from cache whenever the TB detects a miss.

There are six architecturally defined registers in the TB fixup unit. these must be set up by the operating system to define the location and size of the system and process page tables. The TB fixup unit contains a register file that stores copies of the six memory management registers and a 32-bit adder.

- P0 base register (P0BR) - The P0 base register contains the system virtual address of the first entry in the P0 page table.
- P0 length register (P0LR) - The P0 length register contains a number defining the size of the P0PT (number of PTEs).
- P1 base register (P1BR) - The P1 base register contains the system virtual address of the first entry in the P1 page table.
- P1 length register (P1LR) - The P1 length register contains a number defining the size of the P1PT (number of PTEs).

- o System base register (SBR) - The system base register contains the physical address of the first entry in the SPT.
- o System length register (SLR) - The system length register contains a number defining the size of the SPT (number of PTEs).

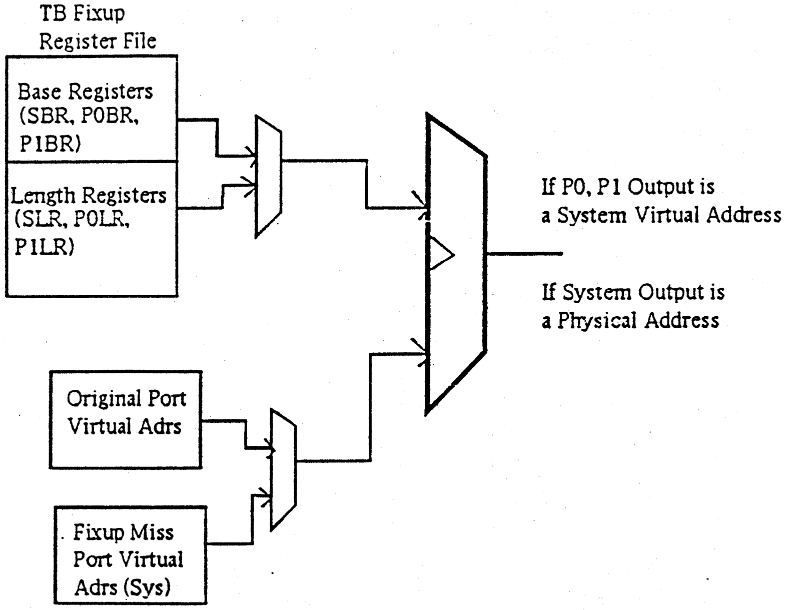


Figure 3–37 TB Miss Processor

3.4.13.2 Resolving a TB Miss

When the TB detects a miss, FALT passes the virtual address causing the miss to the TB fixup unit. If it is a system space memory reference,

1. TB fixup unit extracts the virtual page number from the virtual address and checks it against the appropriate length register.
2. If the virtual page number exceeds the length of the page table, FALT signals a fault and the TB fixup unit returns to the idle state.
3. If no length violation has occurred, the TB fixup unit multiplies the virtual page number by four (longword alignment) and adds the contents of the system base register to the virtual page number to form the physical address of the PTE in the system page table.
4. FALT reads the PTE from the system page table and checks bit 31 of the PTE to verify that it is a valid PTE.
5. If it is not valid, FALT signals a fault has occurred.
6. If it is valid, FALT checks the protection code, bits <30:17> of the PTE.
7. If access is allowed FALT writes the PTE into the TB STRAMs.

However, if the address was a P0 or P1 space address (Figure 3–38), then the address of the PTE is a system virtual address and the memory reference must be made through the TB to translate it to a physical address. This reference may miss in the TB. The TB fixup unit resolves double misses. One from the original memory request and one that it caused in the process of trying to fetch the required PTE.

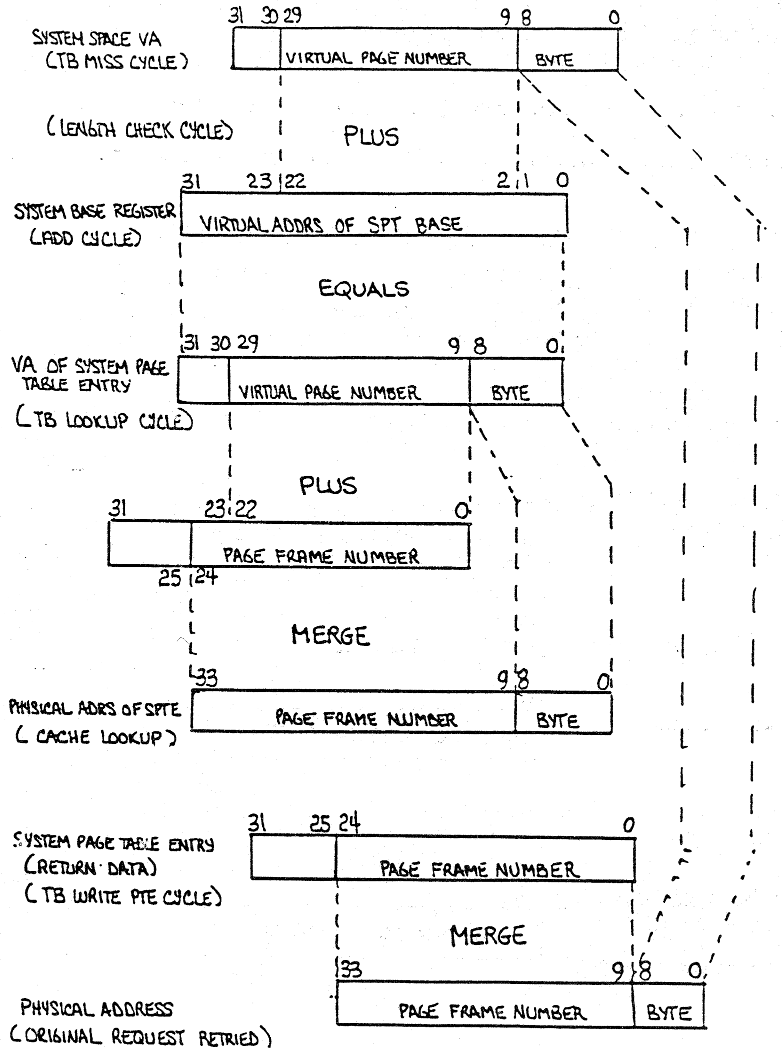


Figure 3–38 Per-Process Translation

FALT receives the PTEs, and tests for accessibility and translation-not-valid, and writes the PTEs in the TB tag STRAMs.

The process of fixing misses in the TB is not interruptible, in the sense that no other ports have access to the TB during miss processing. If all accesses to the cache are valid, then a single P0 or P1 miss takes the following seven cycles to complete:

- o TB Miss Detection - FALT compares the VA with the TB tag and determines that the PTE needed to translate the VA is not in the translation buffer.
- o Length Check - FXUP accesses the system length register and verifies that the virtual address is not outside the assigned boundary.
- o System Virtual Address Generation - FXUP accesses the system base register and calculates the new system virtual address.
- o System Virtual Address Lookup - FXUP accesses the TB tag and data STRAMs, and verifies that the virtual address is not outside the assigned boundary, generates physical address for cache lookup.

- Cache Lookup - PADX addresses the cache STRAMs. CTMA and CTMV determine if the requested data (PTE) is valid and present in cache. CTMA and CTMV enable the cache read data into the MBox fixup port output latch.
- Return Data - FALT receives the new PTE, checks to see if PTE is valid.
- Write TB Entry - FALT extracts and writes the TB, tag, and valid data into the TB STRAMs.

A double miss takes 13 cycles. An S0 miss (Figure 3–39), takes 6 cycles.

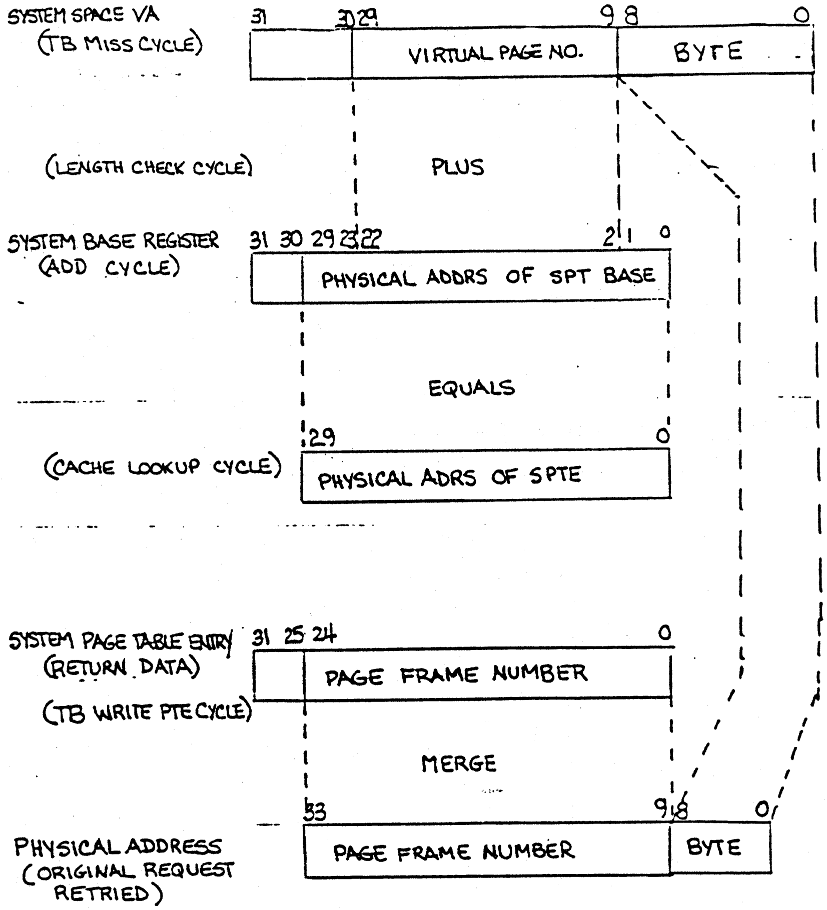


Figure 3–39 System Space Translation

The following is a listing of functions that support memory management:

- Load Process Context (LDPCTX) - Invalidates all P0 entries of the TB.
- Initialize TB (TBIA) and Invalidate Single TB Entry (TBIS) - The EBox issues a TBIA that is the move to processor register (MTPR) command to initialize the entire translation buffer. To invalidate a single TB entry, the EBox issues a TBIS.
- Write and read register - The EBox issues a write register to load data in the memory management registers (in the TB fixup unit register file). The EBox puts the data in the EBox address input, and selects the appropriate register using the EBox tag field. The memory management registers and the fault registers are available to the EBox with the read register command.
- Probe read and probe write commands - The EBox issues the probe read and write commands to check the read or write accessibility of the page.

3.4.13.3 Memory Management Faults

The TB fixup unit resolves single and double misses in the TB. It is responsible for only reporting and detecting length violations.

The EBox resolves length violations, as well as the following exceptions:

- o Access-control-violation faults (length violations) are detected as a result of the TB lookup or update.
- o Translation-not-valid faults are detected as a result of the TB lookup or update.

The faulting parameters are held in two registers, fault parameters register and the fault virtual address register. The fault parameters register is shown in Figure 3-40. On a read fault, the MBox sends a response to the port and asserts the fault line. Faults occur on pre-translated writes. The MBox returns the faults when the EBox attempts to do the write.

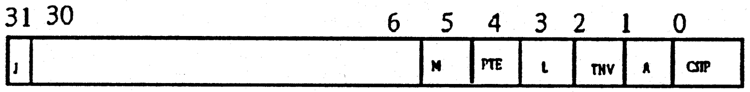


Figure 3–40 Fault Parameter Register

Table 3–17 lists the fault parameter register bits and their definitions.

Table 3–17 Fault Paramteter Bit Definitions

Bit	Definition
J<31>	Indicates that the OPU was jammed.
<30:06>	TBD
M<05>	Indicates a modify intent reference.
PTE<04>	Indicates a PTE reference violation.
L<03>	Indicates a length violation.
TNV<02>	Indicates translation not valid.
A<01>	Indicates an access violation.
CSIP<00>	Indicates a cache sweep in progress.

In all cases, FALT holds the fault parameters and faulting virtual address in the fault registers that can be read by the EBox.

If FALT detects a PTE that does not have the M-bit set and it is a write operation, TB fixup sets the M-bit, and the MBox continues. The operating system is notified that the M-bit needed to be set (modify fault). For the modify fault, none of the fault bits are set in the fault parameter register.

3.4.13.4 TB Fixup Functions

The TB fixup unit initiates a PTE fetch with a read operation. When cache returns the PTE to the TB, VAP checks to see if the original request is a write operation. If it is, and FALT does not detect any memory management problems, the TB fixup unit performs a set-m-bit operation, and signals cache. Cache holds the PTE until the translation buffer signals that the M-bit needs to be set, or that the PTE has been written into the TB and the M-bit does not need to be set.

3.4.14 Sequencer Port

The sequencer port uses the TB fixup unit adder to add a constant four to the previous address. The adder is set up to add the constant to a virtual address that is either:

o unaligned

o an address for a data size greater than a longword

A port (EBox and OPU) issues a request, starting address, and context (data size). As the starting address is involved with TB lookup, the sequencer passes it through the adder (the address is incremented by 4). If the new address is required, the sequencer issues a request that locks out all other ports and the new address accesses the TB. The MBox repeats this process until all required addresses are produced.

The port (EBox or OPU) request is stored in the sequencer function buffer, which tracks the port requests. If the OPU requests an unaligned read with write check, the function (read with write check) comes from the sequence port during the next cycle.

The TB fixup processor remains idle until a TB miss occurs. The mircoword in idle sets up the adder to perform the increment.

3.4.15 EBox Port

The EBox port (Figure 3–41) has an address input latch <31:00>, a data input latch <31:00>, a function <04:00>, and a tag <03:00>.

3.4.15.1 EBox Virtual Address

The EBox port has a 32-bit address, EBOX_EB_ADDRESS <31:00>, input latch for EBox reads and explicit writes. The latch holds both virtual and physical references. It is also used as the data path for transferring register contents to the memory management registers (P0BR, P0LR, P1BR, P1LR, SBR, SLR, MAPEN) during a move to processor register (MTPR) instruction.

The EBox has access to the MBox readable and writeable registers that are listed in Table 3–18 and Table 3–19, respectively. The EB_TAG<3:0> select the register. The EBox sends write register data over the virtual address lines, and receives read register data from the EBox output port (data traffic manager).

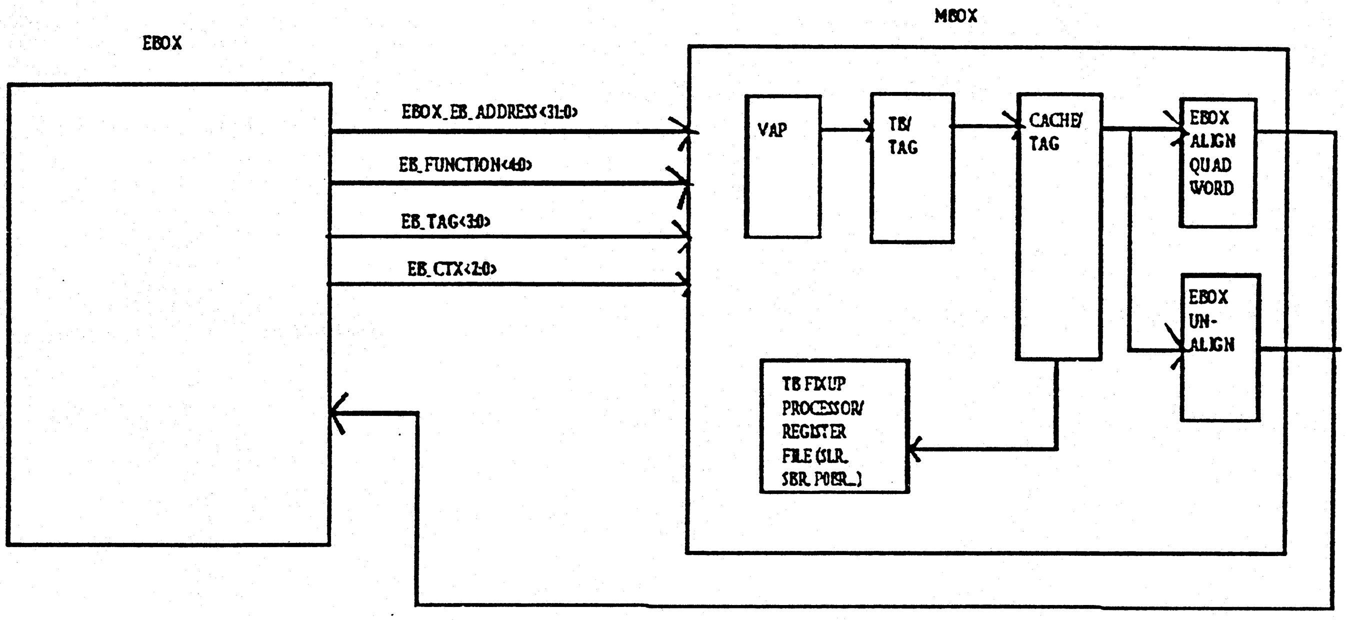


Figure 3–41 EBox Port Request

Table 3–18 MBox Readable Registers

Register	Register Number	Description
Fault Param	F	Fault Parameter
Fault VA	1	Faulting Virtual Address

Table 3–19 MBox Writeable Registers

Register	Register Number	Description
CSWP	1	Cache Sweep
MAPEN	6	Map Enable
P0BR	8	P0 Base Register
P0LR	9	P0 Length Register
P1BR	A	P1 Base Register
P1LR	B	P1 Length Register
SBR	C	System Base Register
SLR	D	System Length Register

3.4.15.2 EBox Data Input Latch

The EBox data path is 32 bits wide and performs unaligned transactions. The data traffic manager/rotator aligns the data. The MBox can write any size data, a byte to a quadword, for the EBox (see Table 3–20.)

Table 3–20 EBox Data Sizes

OPU Context	Data Size
0	longword
1	byte
2	word
3	quadword
4	octaword
5	block

Byte and word writes require three cache cycles:

- o TB lookup - Virtual address port performs a TB lookup using the latched EBox virtual address and generates a cache lookup address.
- o Merge - Data traffic manager merges the cache read data with the byte or word.
- o Write - Data traffic manager outputs the data to be written into the cache location.

Unaligned longword writes, that do not cross a quadword boundary (address bit 2 equals 0) require three cycles.

- o TB lookup - Virtual address port performs a TB lookup using the latched EBox virtual address and generates a cache lookup address.
- o Rotate - The rotator, as determined by the physical address and the context, rotates the cache read data.
- o Merge - Data traffic manager merges the rotated data with the longword.
- o Write - Data traffic manager outputs the data to be written into the cache location.

Aligned longwords or aligned quadwords can be written in two cycles, lookup and write. There are two types of writes, independent of the size word, can occur, explicit writes and op-writes.

3.4.15.3 Explicit Writes

An explicit write is one in which the EBox sends both the address and data, along with the write command. The data size can be from a single byte up to a quadword. Explicit writes occur during instructions that have suspended the IBox (PUSHR, MOVC3, etc). The data traffic manager receives the data a half cycle before the MBox receives the address.

An explicit write of an aligned longword takes three cycles. In the first cycle, VAP and FXUP (translation buffer) translates the virtual address and decodes the port function. In the second cycle, CTMA and CTMV perform a cache lookup in the cache tag store and determine the status of the targeted block. In the last cycle, PADX enables the cache data STRAMs and CTMA and CTMV enable cache tag STRAMs for a write operation. CCSQ sends a cache cycle id that indicates it is a cache write operation.

Unaligned writes require an additional cycle to rotate and merge data.

Explicit writes, when unaligned, are limited to longword size only. This prevents the first portion of the write to occur before the access check is performed on the subsequent incremented address. Potential page crossing is determined with the starting address and context. If true, the page crossing danger flag is asserted and accompanies the physical address to the cache. This prevents the cache write from occurring until the write check is made on the second address.

3.4.15.4 OP Writes

An OP write command requires the use of a separate request and arbitration mechanism since the front end of the MBox is effectively bypassed.

An OP write is one that the operand processing unit (OPU) has sent an address that corresponds to an operand memory write. VAP translates the address and stores it the write queue. The EBox executes the instruction, and sends the data with a command. CCSQ removes the physical address from the write queue, as the write operation is performed. If no fault occurs, WRTQ asserts MBOX_Q_FAULT_VALID_H without MBOX_Q_FAULT_H.

3.4.15.5 EBox Functions

The function field from the EBox, EB_FUNCTION <04:00>, accompanies IB_REQUEST<00>. EBox functions are listed in Table 3–21.

Table 3–21 EBox Functions

Function	Mnemonic	Description
19	TBCHK	TB Check
18	RR	Read Register
17	WR	Write Register
16	LDPCTX	Load Process Context
15	TBIA	TB Invalidate All Entries
14	TBIS	TB Invalidate Single Entry
10	CF	Clear Fault
9	PW	Probe for Write
8	PR	Probe for Read
7	WP	Write Physical
6	RP	Read Physical
5	WUL	Write Unlock
4	RL	Read Lock
3	RWC	Read with Write Check
2	W	Write
1	R	Read

3.4.15.6 EBox Port Parity

The EBox sends four parity bits (Figure 3–42) with the EBox virtual address lines. Also shown in Figure 3–42, is the one parity bit sent with the EBox port request information.

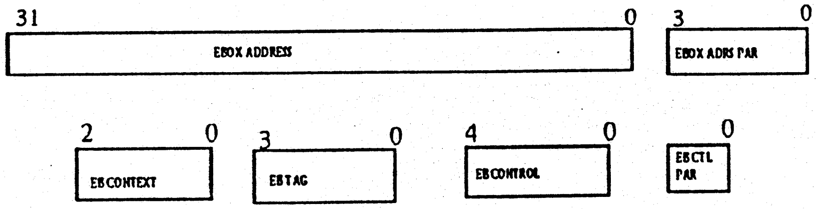


Figure 3–42 EBox Parity Bits

3.4.16 IBox Ports

There are two ports that the IBox uses to access memory data from the MBox, the operand processing unit (OPU) and the instruction buffer (IBUF).

3.4.16.1 OPU Port

The OPU port (Figure 3-43) has an address input latch <31:00>, function <04:00>, tag <03:00>, OP indirect destination <01:00>. It uses its MBox port to prefetch all memory related operands. The port is similar to the IBUF port, in that, there is no data input to the MBox. The OPU asserts its request line, OP_REQUEST<00>, and sends a 32-bit virtual address, OP_ADDRESS <31:00>.

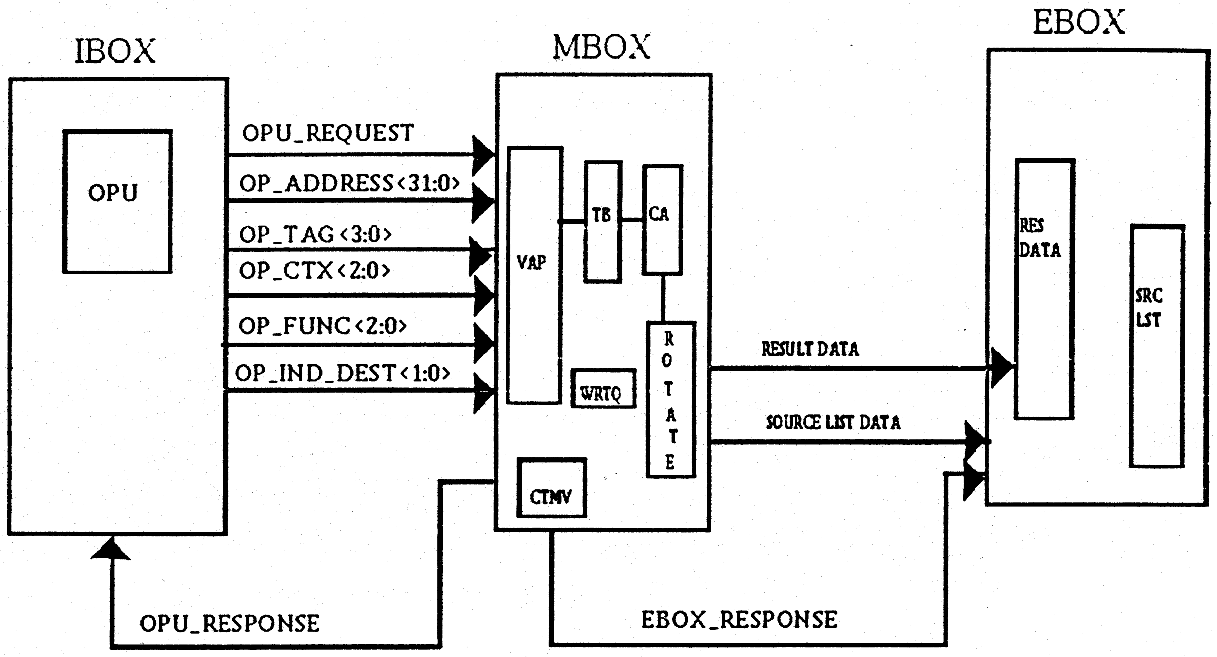


Figure 3–43 OPU Port Request

The MBox responds with IB_OP_ACK, and acknowledges that the command and address has been received. Byte addressing can occur from this port so the address is the full 32-bits. The data traffic manager (OPU port) returns OPU data in an aligned format, in that, the requested byte occupies the rightmost position with increasing bytes to the left. The OPU data sizes are listed in Table 3-22.

Table 3-22 OPU Data Sizes

OPU Context	Data Size
0	longword
1	byte
2	word
3	quadword
4	octaword
5	block

The data is output from the data traffic manager/rotator.

The OPU commands are listed in Table 3-23.

Table 3-23 OPU Commands

Function	Mnemonic	Description
0	R	Read
1	RWC	Read with write check
2	RWCNC	Write check, no conflict check
3	WC	Write check
4	RNB	Read, no block
5	RWCNB	Read with write check, no block
6	RNOP	Read, force TB hit, force cache hit
7	WCNB	Write check, no block

3.4.16.2 Read

For an OPU read command, the OPU sends the following information:

- Request - Initiates an OPU request.
- Function - Indicates an OPU read.
- Address - VA <31:00> indicate the location of either operand data or an indirect address.
- Tag - OP_TAG<03:00> indicate the storage location in the source list (register file in the EBox).
- OP Indirect - OP_INDIRECT <01:00> (Table 3-24) indicates that the data is to be returned to either the EBox or to the OPU. The MBox responds to the corresponding port, EBox, or OPU.

Table 3–24 OP Indirect Decode

OP Indirect Function	Destination
0	Operand fetch to the EBox
2	Indirect fetch for read operand
3	Indirect fetch for write operand

3.4.16.3 Read with Write Check

The OPU read with write check command is similar to the OPU read. However, during an OPU read-with-write-check, the write queue checks for memory locations that are used for both the source and the destination. The cache lookup address for a read is inserted into the write queue (Figure 3–47) with status information. The MBox selects the cache lookup address used when the EBox completes execution and sends the data to be written into cache. The data traffic manager returns read data to the EBox in an aligned format, in which the requested byte is in the right-most byte location. CTMV generates MBOX_EB_RESPONSE.

3.4.16.4 Write Check

No Conflict Check.

3.4.16.5 Write Check

Write checks allow the IBox to prefetch beyond those specifiers that write to a memory location. When the OPU decodes a memory destination operand, it sends the virtual address to the MBox along with a function code to indicate write check. CCSQ inserts the translated address (cache lookup) and six status bits into the write queue. CCSQ delays the write until the EBox executes the instruction and sends the data to the MBox.

3.4.16.6 Read

No Block.

3.4.16.7 Read with Write Check

No Block.

3.4.16.8 Read, Force TB Hit, Force Cache Hit

The MBox forces a TB hit to support vector reads. The read-no-op enables a steady stream of data to the vector unit in cases where the stride is some small number other than one. The vector unit does not have to calculate which reads to perform.

The MBox also forces a TB hit when the EBox sends read and write physical functions.

The MBox forces a cache hit when FALT detects a page fault in the translation buffer.

The MBox forces a cache when the JBox determines that the valid data is in the write back buffer and not in cache (bypass).

3.4.16.9 Write Check, No Block

3.4.16.10 OPU Port Parity

The OPU sends four parity bits (Figure 3-44) with the OPU virtual address. Also shown in Figure 3-44, is the one parity bit sent with the OPU port request information.

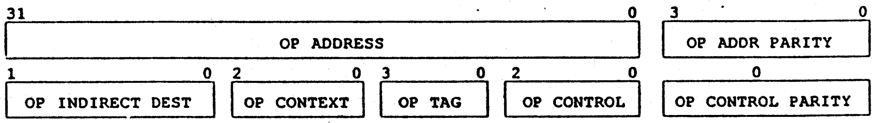


Figure 3-44 OPU Parity Bits

3.4.17 Instruction Buffer Port

The IBUF port is shown in Figure 3-45. It requests data from the MBox when it encounters a miss in the virtual instruction cache (VIC). The IBUF sends a virtual address and asserts IB_REQUEST. The IBox insures that only one outstanding request occurs, so the MBox always accepts the request. A signal to acknowledge the receipt of the request is not necessary. Because the address is quadword aligned, the MBox ignores VA bits <02:00>.

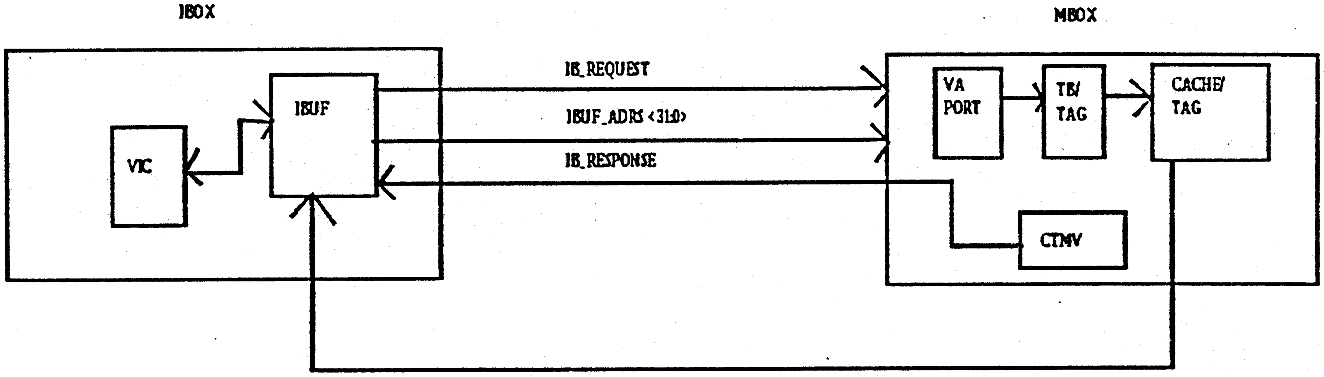


Figure 3-45 IBUF Port Request

The MBox performs the lookup operation in two cycles. In the first cycle, VAPO and FXUP translate the virtual address to a physical address. In the second cycle, the physical address accesses the cache tag and data STRAMs.

The data traffic manager (aligned port) returns a full block of data over the 64-bit wide data path. CTMV generates MBOX_IB_RESPONSE. The 32 bytes (4 quadwords) are time-multiplexed at 8 bytes per cycle. The IBUF port does not do wrapped reads. If the virtual address is not a block boundary (VA <04:03> does not equal 0), the MBox responds with the requested quadword first, and then sends the remaining quadwords, if any, in that block.

The instruction buffer requests a block of data for the virtual instruction cache when all of the following conditions exist:

- o VIC miss
- o Instruction buffer extension register (IBEX), which is a quadword register between the VIC and the IBUF, is empty or will be empty at the end of the cycle
- o IBEX2 is empty

- No flushes are present
- No requests are outstanding
- No page faults exist

The request for a block of instruction stream data is quadword aligned. If the request is for the second quadword in a VIC block, the MBox responds with the second, third, and fourth quadword of the block. The second, third, and fourth quadwords are marked valid, but the first is marked invalid.

3.4.17.1 IBUF Port Parity

The IBUF sends four parity bits (Figure 3-46), with the IBUF virtual address.

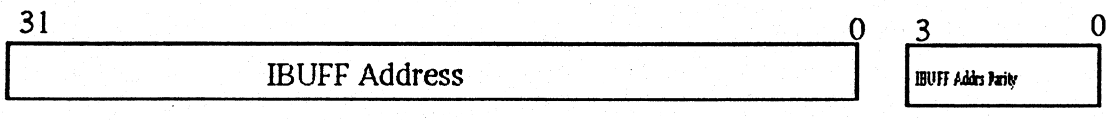


Figure 3-46 IBUF VA Parity Bits

3.4.18 TB Outputs

PADX, CTMA, CTMV, and WRTQ latch the TB lookup (physical) address from the output of the TB STRAMs.

3.4.18.1 Holding Latches

PADX contains holding latches, EBox, OPU, and IBUF, that hold the TB lookup address. PADX uses the TB lookup address to drive address lines for the cache data STRAMs, and CTMA and CTMV use the lookup address to drive address lines for the cache tag STRAMs.

3.4.18.2 Write Queue

When the operand processing unit (OPU) processes memory destination specifiers and sends the virtual addresses, CCSQ stores the TB lookup (physical) address in a FIFO called the write queue. The write queue is in the WRTQ MCA.

The write queue (Figure 3-47) has eight entries.

The write queue assists the IBox in prefetching instructions. When the OPU decodes a memory write operand, it sends the virtual address to the MBox along with the appropriate command. The EBox issues an OPU write command at the conclusion of execution and sends the result data. The physical address for the result data, along with the status bits, are at the top of the write queue. CCSQ pairs the write data in the EBox write data buffer with the top translated address entry in the write queue. EBox data and parity are written into the cache location addressed by the contents of the write queue.

The write queue status bits help control the operation of the write queue (Table 3-25)

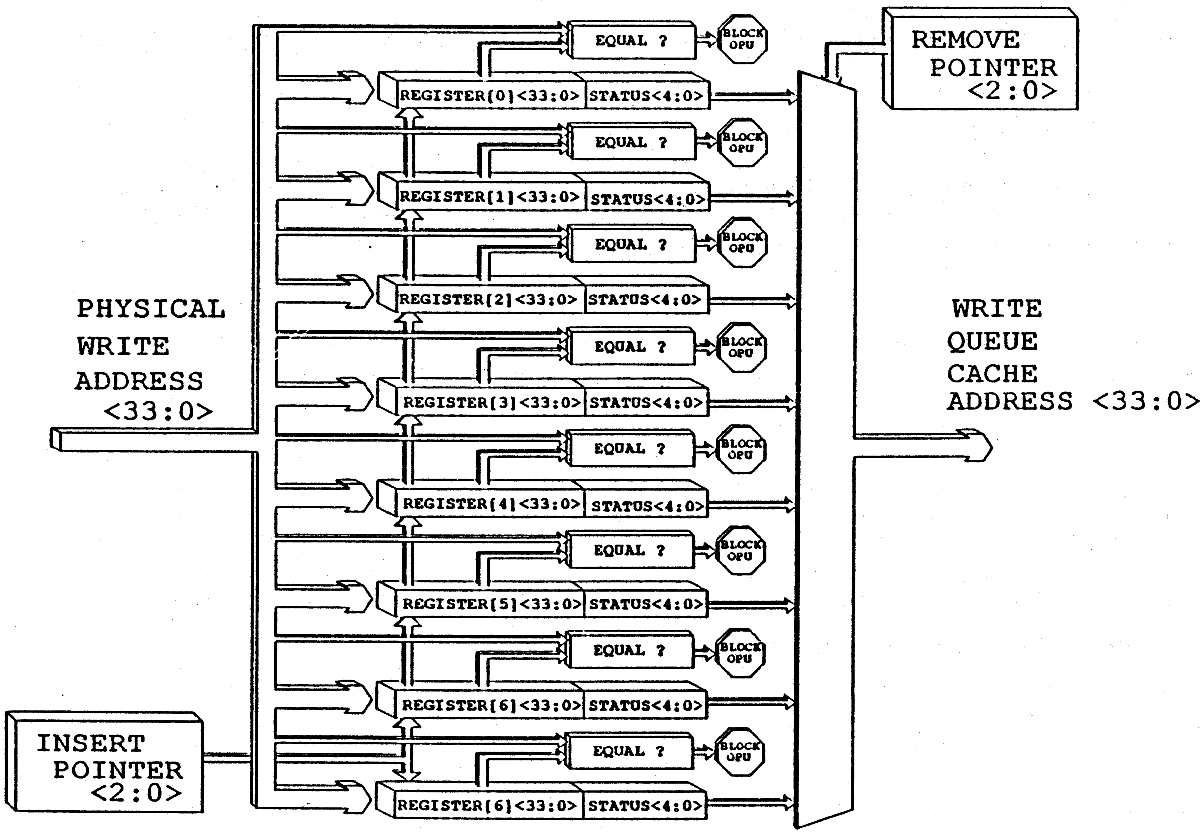


Figure 3–47 Write Queue Block Diagram

Table 3–25 Write Queue Status Bits

Status Bit	Purpose
Fault	Indicates that a memory management fault occurred during translation.
Twice	Asserted when an address is needed twice. Addresses are produced with longword increments. Unaligned address that cross quadword boundaries require the address to be used twice.
PCD	Asserted when there is a potential for an operand to cross a page. It prevents the write from proceeding until the full operand is write checked.
Valid	Asserted when entry in the write queue is valid.
Last	Marks the last in a series of addresses for a single operand.
Blocked	Marks those valid entries that had a conflict with a read address. Each address bit for each entry has an XOR gate for read address conflict checking. The resolution is down to the quad word. Blocked references are not allowed to proceed, nor are additional OPU port requests accepted until the block is resolved. Whenever the write queue pops an entry, upon the completion of an OP WRITE request, the block is cleared.

The write queue consists of valid bits <07:00> and two pointer registers. The valid bits mark each valid entry. The insert pointer register (INSERT_POINTER<02:00>) selects the queue location for the new OP write addresses. The remove pointer register (REMOVE_

POINTER<02:00>) selects the next write queue entry to be removed or popped from the queue.

The OPU can perform operand reads and process other specifiers without waiting for the write operation to complete. To prevent stale data from being read, the WRTQ compares addresses with entries in the write queue. If they match, the CCSQ delays the read operation until the write operation has been processed.

The code in Example 3–1 provides two memory references:

```
R1 = 1000

INCL (R1)                ;Read with write check at 1000
COM  (R1)                ;Read with write check at 1000
```

Example 3–1 Write Queue/OPU Block

On a cycle by cycle basis,

1. TB translates request 1
2. Write queue latches the write address for request 1
3. CTMs latch the read part of request 1
4. Assuming a cache hit, a cache read is started for address 1000, and the data is sent to the EBox.
5. TB translates request 2
6. Write queue latches the write address for request 2
7. CTMs latch the read part of request 2
8. Write queue detects a conflict between the first write to 1000 and the second read to 1000, this stalls the second read.
9. EBox initiates the write (result from the increment longword) at 1000 for request 1. The write queue pops the top entry, the address for result data, for request 1. The cache-lookup is done to store the write data. MBox performs the write cycle for the request 1. The conflict no longer exists. EBox initiates the write (complimented data) at 1000 for request 2.
10. Assuming a cache hit, MBox performs the cache lookup at 1000 for request 2, and sends the operand data to the EBox.

Two writes take place at address 1000. However, two separate write queue entries for 1000 are used.

The twice, or new_quad, bit in the status portion of each write queue entry signifies that the entry should be used twice when a sequence of addresses has been created by the sequence port in the TB. In this case, when an attempt to pop or remove a write-queue entry and the twice bit is set, the write queue uses the address twice before popping it.

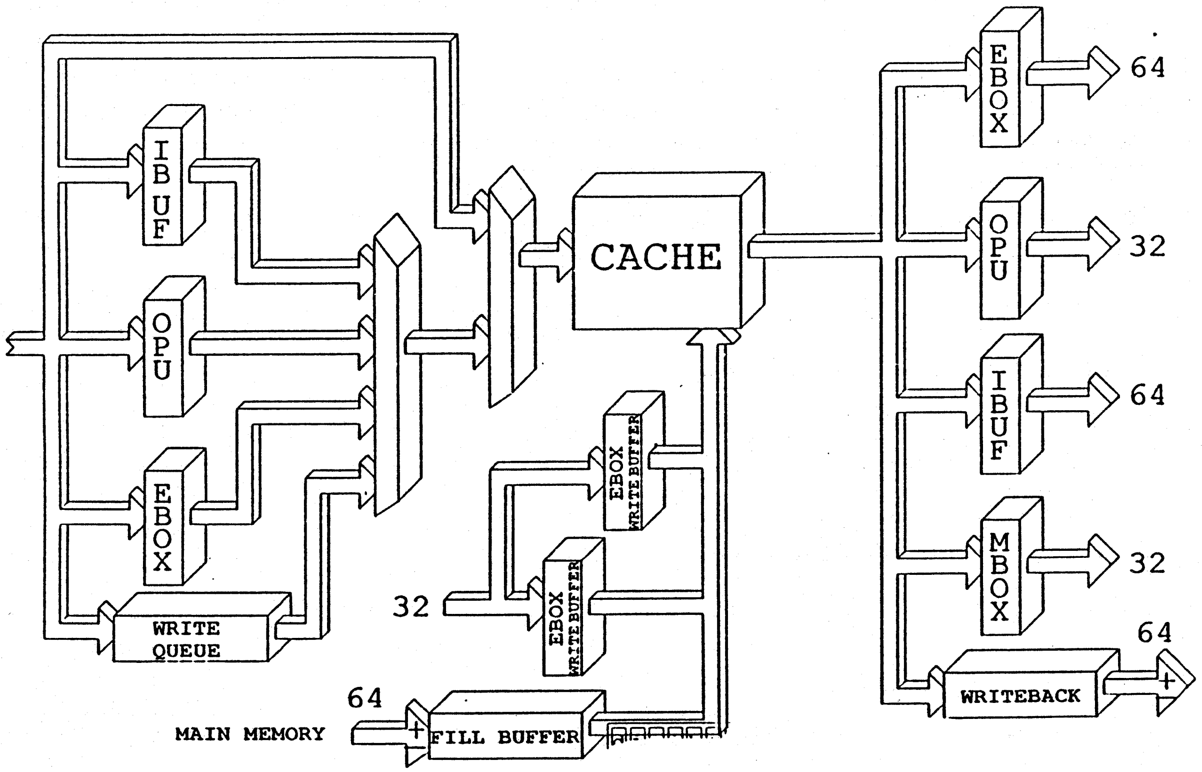


Figure 3–48 Cache Ports

3.4.19 Data Cache

The MBox data cache (Figure 3–48) is 128 Kbytes. The cache is two-way set associative or consists of two separate cache sets, with each cache having 64 K bytes. Each set is 8 K lines deep. Each line is 8 bytes wide. The block size is 64 bytes with a valid bit per longword. The fill size equals the block size and a refill is accomplished in 8 consecutive cycles, writing 8 bytes per cycle. There are 80, 4 K X 4-bit STRAMs in the data cache.

The cache tag store is two-way set associative. With a block size of 64 bytes, the 64 K byte set requires 1024 tag entries. Each entry has physical address bits <32:16>, a written bit, and 16 valid bits.

Caches cycles are are decoded in the CCSQ MCA (Figure 3–49).

3.4.19.1 Cache Lookup

Cache lookup follows the successful translation of the virtual address into a physical address. During cache lookup, CTMA and CTMV determine if the data referenced by the physical address is in data cache. Figure 3–50 illustrates the two data cache sets and their associated cache tag stores. The cache tag is shown in Figure 3–51, and corresponding bits are listed in Table 3–26. The cache data is shown in Figure 3–52, and corresponding bits are listed in Table 3–27.

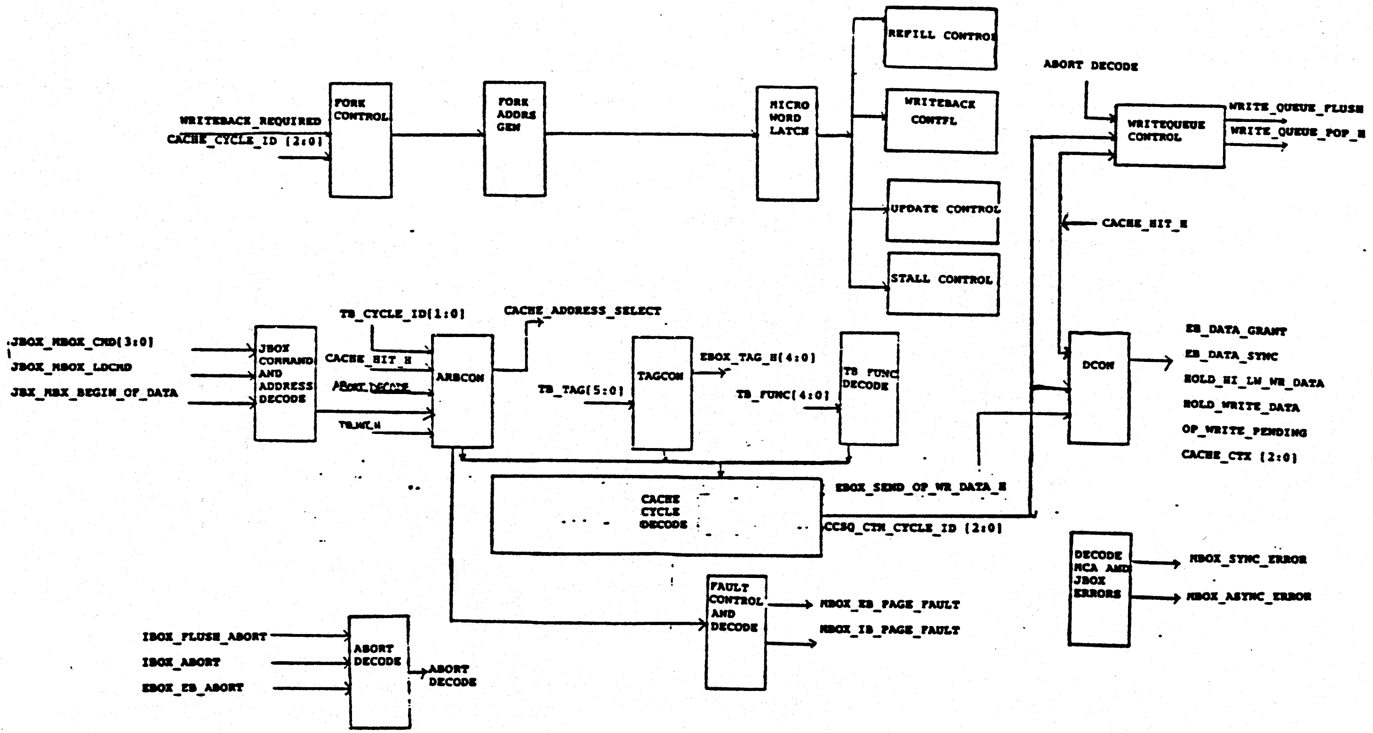


Figure 3–49 CCSQ MCA Block Diagram

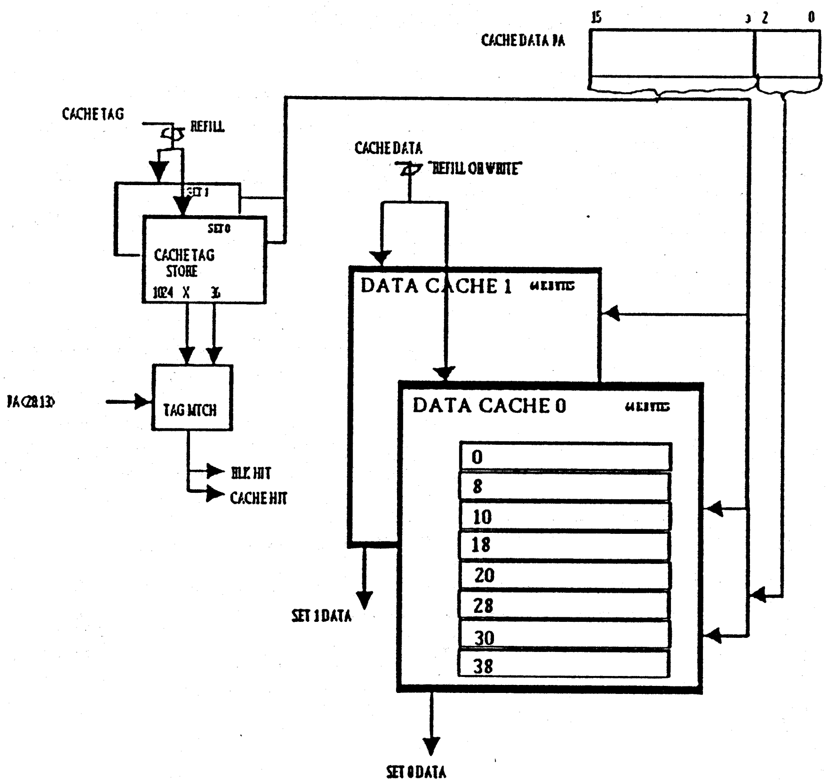


Figure 3–50 Cache Sets 0 and 1

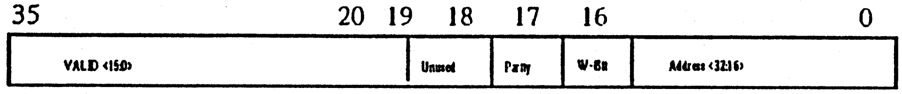


Figure 3–51 Cache Tag Format

Table 3–26 Cache Tag Bit Definitions

Bit	Definition
VAL <15:00>	This 16 bit field marks the validity of each of the 16 longwords in the associated data block: VAL 0 for LW0, VAL 1 for LW1, VAL 2 for LW2, etc. All 16 bits are normally set when the block is refilled and checked during lookup to determine if the longword requested is valid in cache (Cache Hit).
TAG PAR	Parity bit for the tag. During refills it is generated and written and during lookups it is read and checked. Calculates odd parity on the entire tag contents.
TAG W BIT	This bit is set whenever one of the longwords in the associated cache block is modified. It is used during write backs to determine the need to write back the cache block to the arrays.
ADR TAG	16-bit physical address field that is loaded with PA <32:16> during a cache refill. During lookup these bits are compared with PA <32:16> to determine if the block being accessed is currently stored in cache.
LRU	The LRU bit is used to implement a least recently used algorithm to determine that cache to select during a refill. There are 1024 tags for each data cache. There are 1024 LRU bits to handle selection for both caches.

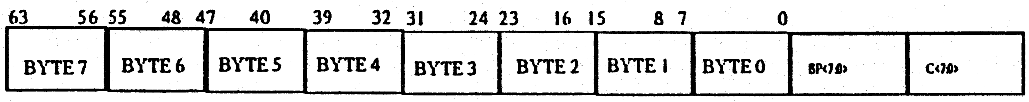


Figure 3–52 Cache Data Format

Table 3–27 Cache Data Bit Definitions

Data <63:00>	**64 bit data longword. There are 16 longwords associated with each tag address.**
BP<7:00>	Byte parity bits for each of the eight bytes within a quadword.
C<7:00>	An 8-bit check code calculated on the data segment of the longword.

3.4.19.2 Cache Hit

CTMV (Figure 3–53) and CTMA determine a cache hit or miss condition. The following conditions must be met for a cache hit:

- o Cache is enabled (on)
- o Cache tag matches the physical address
- o Valid bit for the requested longword is set in the tag
- o No cache tag errors were detected

- Physical address maps to internal memory space

3.4.19.3 Cache Miss

When the cache lookup fails, there is a cache miss. This condition requires refilling the data cache and updating the cache tag store. The least recently used (LRU) (Figure 3–53) control selects a cache block. For each cache block, there is a cache tag. The tag contains a written bit, that when set initiates a cache write back to write the cache block into main memory. During a cache refill, a block of eight quadwords, 64 bytes, from main memory is written into the selected cache.

3.4.19.4 Cache Arbitration

The port with access to the translation buffer in the previous cycle has access to the data cache (Figure 3–54)

in the current cycle. The flow, from the TB to cache, occurs in consecutive cycles. The cache portion does not require more than port ID, function, and address. There are, however, conditions that require special control and arbitration, which alter the flow of arbitration, such as misses, unaligned multiprecision writes, and write backs. Figure 3–49 illustrates cache cycle decode and arbitration.

The following ports generate cache requests:

- The virtual instruction cache (VIC) in the IBox accesses cache for VIC refills. Starting with the quadword address received in the virtual address port, the data traffic manager sends either a full or partial (depending on the starting address bits) 32-byte VIC block, 64 bits per cycle. The MBox (data traffic manager) sends data most likely to be needed. The IBox flushes the VIC on every return from exception and interrupt (REI) instruction.
- The operand processing unit sends addresses to the MBox for source operands to be retrieved from cache and sent to the EBox.
- The operand store unit sends addresses to the MBox to be paired with result data from the EBox, and initiates a write reference to memory.
- The operand unit requests data to generate addresses for indirect memory references. The data traffic manager returns the data to the OPU.
- For instructions that access memory without explicit specifiers, the EBox microcode usually requests the data directly. Examples of this include string instructions, privileged instructions, and interrupts and exceptions. For some frequent operations like PUSHL, the IBox creates the destination address, even though it is an implicit operand, one that does not appear as a specifier. For these cases the OPU has logic to create the address, usually from the stack pointer (SP).
- The MBox performs its own memory management functions. The translation buffer fixup unit processes translation buffer misses as they occur, without causing a microcode trap. The MBox generates its own cache references to read and write page table data. It also has a sequencer that increments addresses for unaligned references and page crossing references. The TB fixup unit also does cache flushes, both for cache invalidation and memory validation.

The SCU can also access the cache for cache refill, write back, and invalidate.

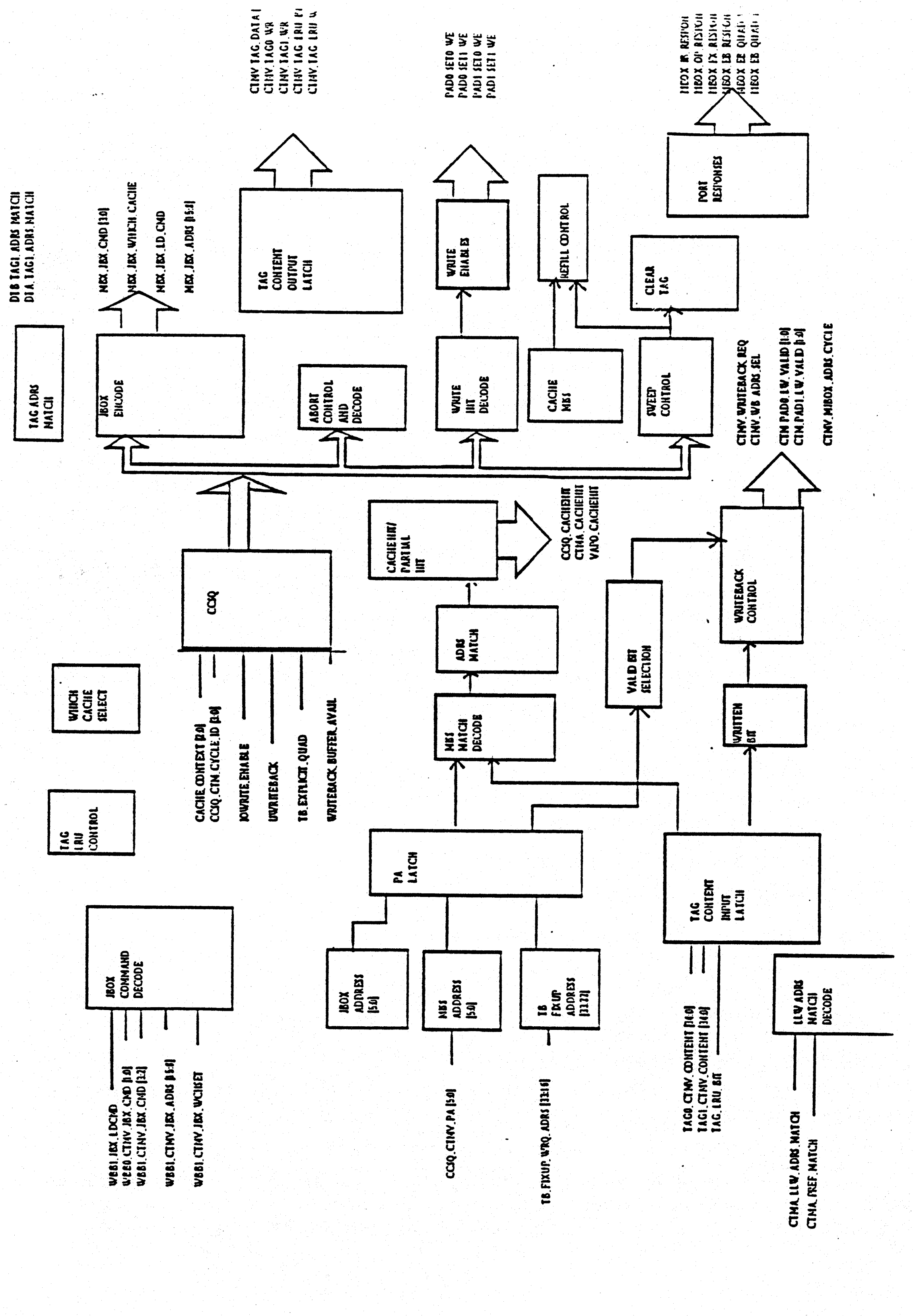


Figure 3–53 CTMV Block Diagram

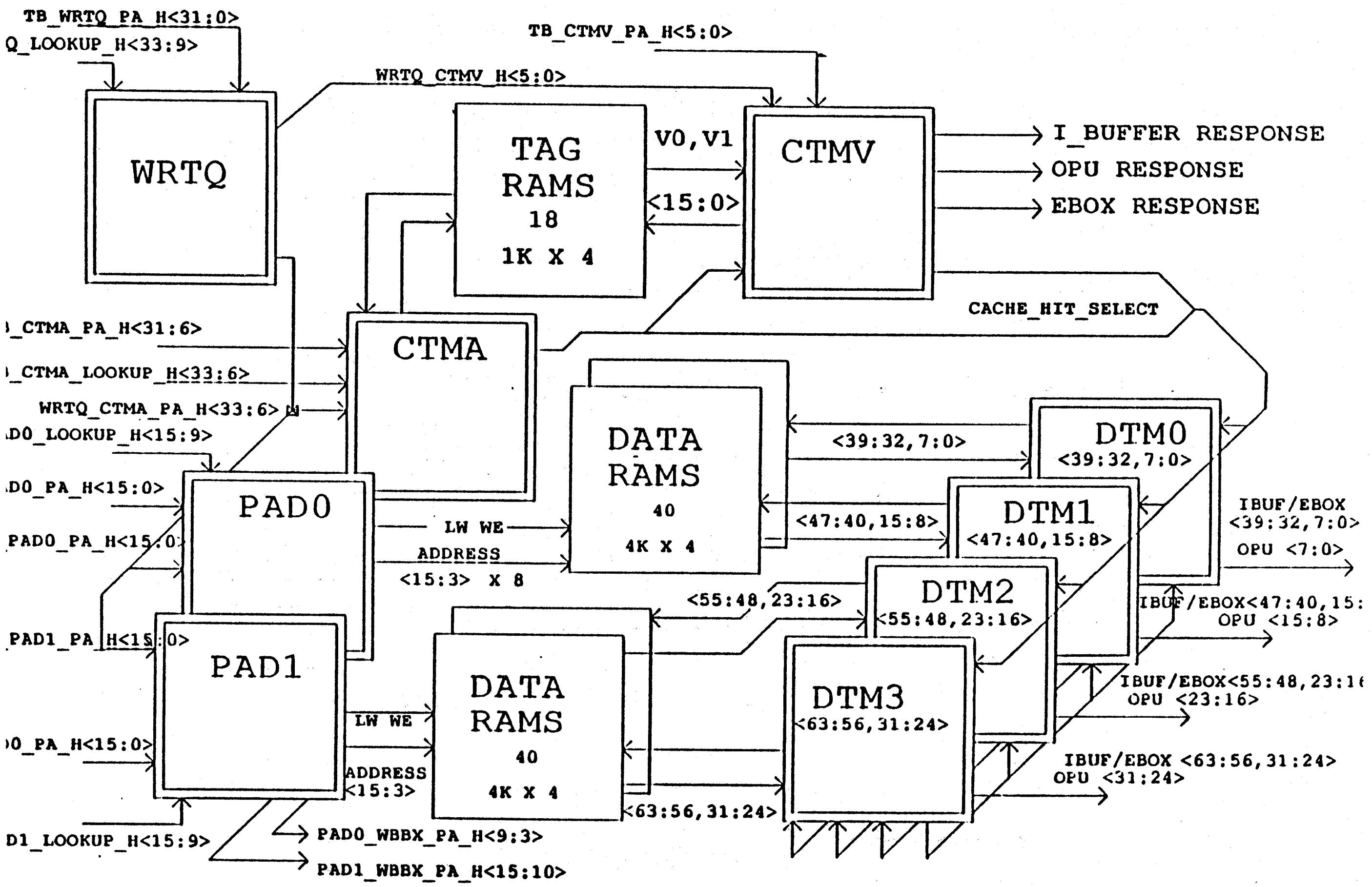


Figure 3–54 Data Cache Block Diagram

The cache bandwidth varies with the type of reference. The cache can do a read every cycle, and a write every other cycle. Internal cache is 64 bits wide. Types of references and corresponding bandwidths are as follows:

- o Refills and Write Backs - 64 bits per cycle.
- o Virtual Instruction Cache Refills - 64 bits per cycle
- o Back-to-Back Longword Writes - 64 bits in two cycles
- o Aligned Quadword Operands/Read - 64 bits per cycle
- o Aligned Quadword Results/Write - 64 bits in two cycles

3.4.19.5 Cache Address Sources

VA bits <15:09> require a TB lookup translation before addressing the cache. There are four sets of STRAMs for bits <15:09>.

The TB lookup output latch accesses the input address latch for the cache STRAMs. TB lookup bits <15:3> address the 4 K X 4 data cache STRAMs. Bits <02:00> determine the first byte of the data, and the context selects the number of bytes or data size. Simultaneously, TB lookup bits <15:06> address the cache tag to determine a cache hit.

The cache address sources (Figure 3–55) are as follows:

- o JBox - Sends read and write miss refill, invalidate, and induced write back addresses.

- TB Fixup Unit - Transfers a translated address from the translation buffer for user and system PTEs.
- Write Queue - Holds pre-translated OPU write addresses.
- EBox - Sends virtual or physical addresses.
- OPU - Sends virtual or physical addresses.
- Instruction Buffer - Sends virtual or physical addresses.

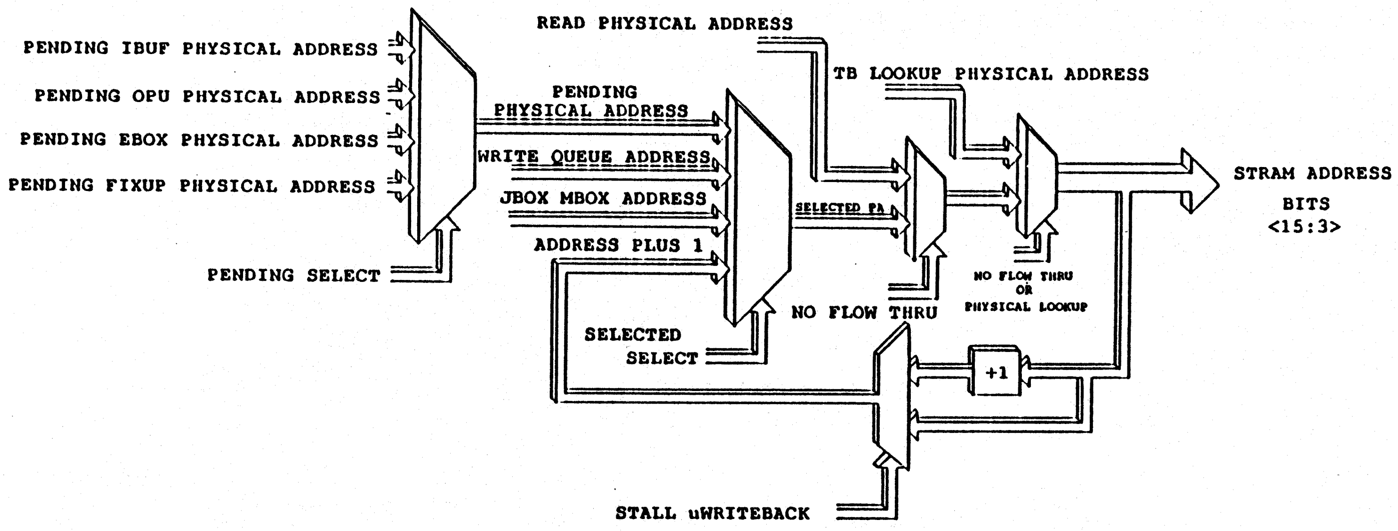

READ PHYSICAL ADDRESS
PENDING IBUF PHYSICAL ADDRESS
TB LOOKUP PHYSICAL ADDRESS
PENDING OPU PHYSICAL ADDRESS
PENDING PHYSICAL ADDRESS
PENDING EBOX PHYSICAL ADDRESS
WRITE QUEUE ADDRESS
STRAM ADDRESS BITS <15:3>
PENDING FIXUP PHYSICAL ADDRESS
JBOX MBOX ADDRESS
SELECTED PA
ADDRESS PLUS 1
PENDING SELECT
NO FLOW THRU OR PHYSICAL LOOKUP
NO FLOW THRU
SELECTED SELECT
+1
STALL uWRITEBACK

Figure 3–55 Cache Address Sources

3.4.19.6 Cache Data Sources

There are two external sources of data to the data cache, the refill buffer and the EBox write buffer, and one internal source for PTE updating.

Data Traffic Manager/Refill Buffer The JBox sends refill data to the MBox (Figure 3–56).

The data comes from main memory and is a full block, 64 bytes, 8 bytes per cycle for eight consecutive cycles.

If the requested block was valid and written in another CPUs cache, JBox writes the block to main memory and sends it to the requesting CPU. The existing partially valid and written blocks must be merged with main memory data. JBox performs wrapped refills, and the requested quadword appears first.

The MBox receives the data into a refill buffer, and sends the data to the requesting port and the cache STRAMs. WBEM and WBES receives and checks parity as it comes from the JBox.

JBox Parity Figure 3–57 illustrates the JBox data parity.

DTA and DTB writes byte parity with the refill data. The data traffic manager generates the ECC check bits and stores the bits in a separate bank of STRAMs.

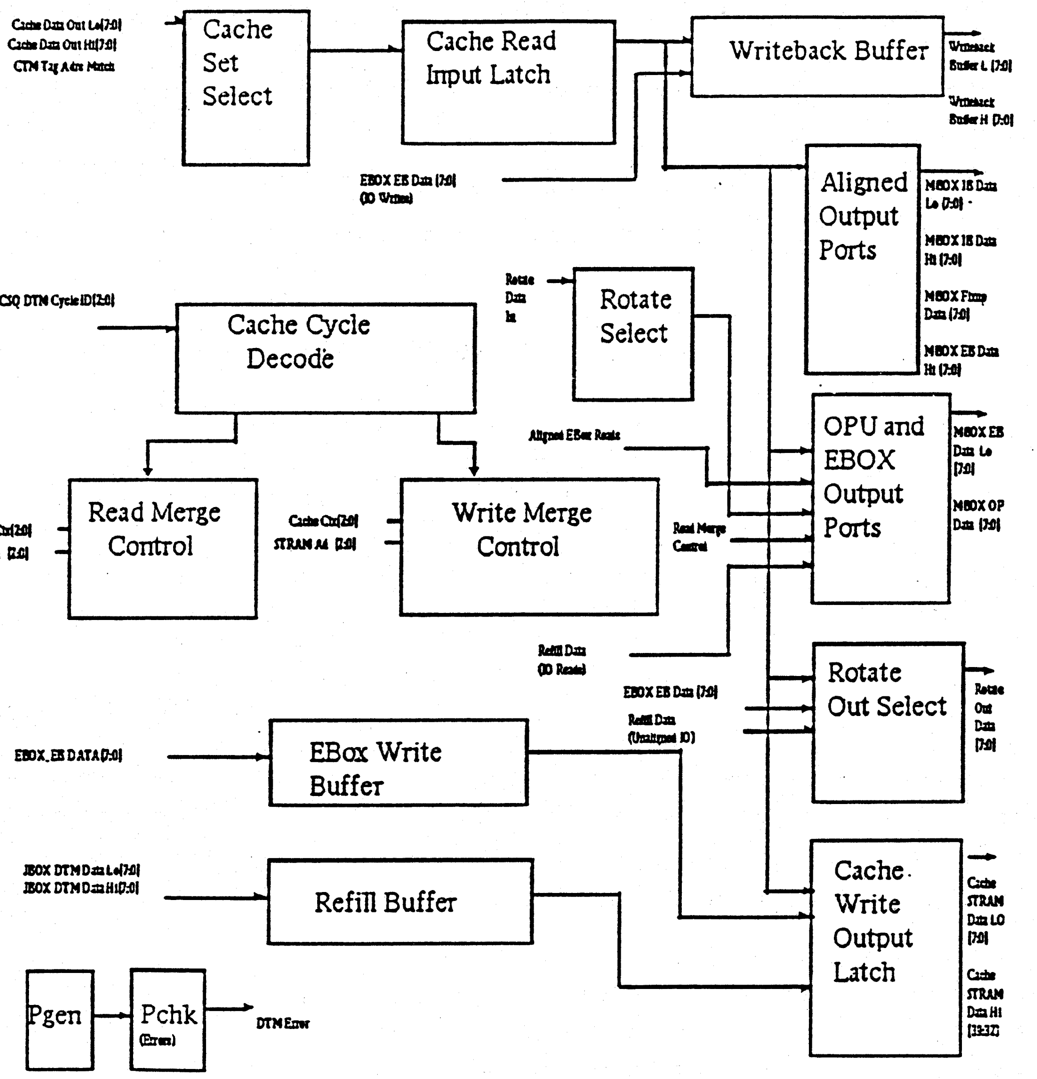


Figure 3–56 DTMX MCA Block Diagram Buffer

Data Traffic Manager/EBox Write Data Buffer There is an 8 byte EBox write data buffer (Figure 3–56) in the MBox that receives and holds up to a quadword until the write can complete. The data path is 4 bytes, 32 bits, in width. The data traffic manager/rotator right justifies the EBox data, independent of starting address and context.

Explicit longword writes require a TB cycle and two cache cycles. Pre-translated writes or OP writes, where the physical address is at the top of the write queue, require two cache cycles.

3.4.19.7 Cache Outputs

CTMA and CTMV latch the translated address, and access the cache tag STRAMs and determine cache hit or miss conditions. DTM0, DTM1, DTM2, and DTM3 latch the cache output, and when appropriate, rotate the cache data. Table 3–28 list each DTMX and corresponding bytes.

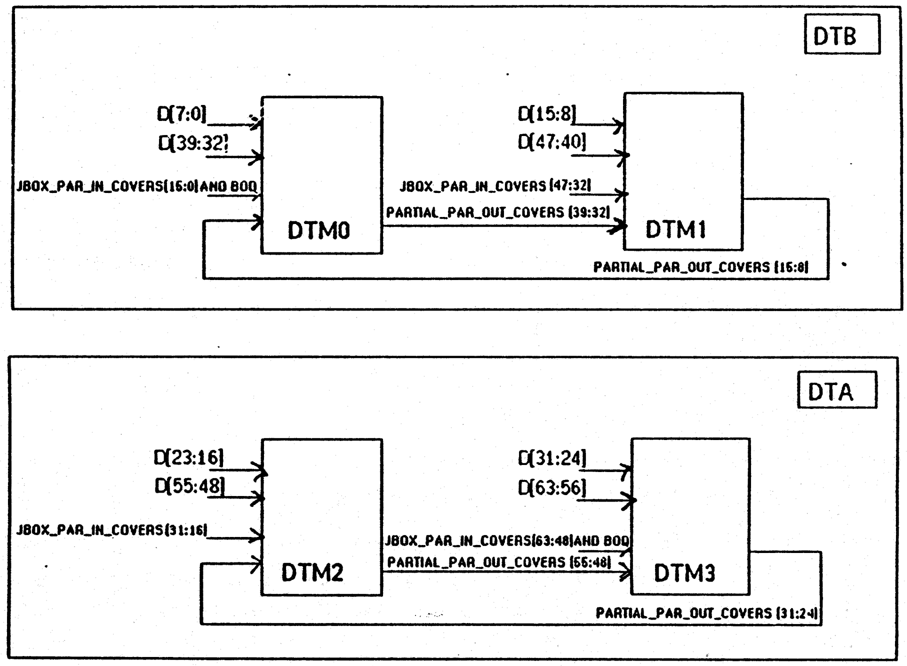

DTB
D[7:0]
D[39:32]
JBOX_PAR_IN_COVERS[16:0] AND BOQ
DTM0
D[15:8]
D[47:40]
JBOX_PAR_IN_COVERS [47:32]
PARTIAL_PAR_OUT_COVERS [39:32]
DTM1
PARTIAL_PAR_OUT_COVERS [16:8]
DTA
D[23:16]
D[55:48]
JBOX_PAR_IN_COVERS[31:16]
DTM2
D[31:24]
D[63:56]
JBOX_PAR_IN_COVERS[63:48] AND BOQ
PARTIAL_PAR_OUT_COVERS [55:48]
DTM3
PARTIAL_PAR_OUT_COVERS [31:24]

Figure 3–57 JBox Data Parity

Table 3-28 DTMX Byte Slices

DTMX	Bytes
DTM0	Byte 0, Byte 4
DTM1	Byte 1, Byte 5
DTM2	Byte 2, Byte 6
DTM3	Byte 3, Byte 7

Four ports receive cache data:

- TB Fixup Unit - The TB fixup unit issues longword aligned reads from the cache to fetch PTEs. Data is returned along with a response signal.
- Instruction Buffer - The data path to the instruction buffer is 8 bytes wide, or 64 bits plus byte parity. The MBox returns the correct number of quadwords, from the starting address, to the top of the block. The maximum number is 4 consecutive cycles of 8 bytes per cycle. No alignment is necessary for this port. The data traffic manager (aligned port) returns a quadword on a quadword boundary.
- Operand Processing Unit - The OPU port is used for indirect addressing of operands. The data is byte addressable and alignment is performed when address bits <01:00> are non-zero.

  When alignment is required, there is one additional cycle to rotate the data into the proper position. Byte 0 contains the byte found at the starting address and bytes 1, 2, 3 are to the left.

  When unaligned operands cross the quadword boundary, the MBox performs two cache reads. The rotation logic receives the starting address and context, and predicts cross quad conditions. The data traffic manager holds the appropriate bytes until the complete operand is assembled. Unaligned operands that cross a quad word boundary require two additional cycles if the data is in the cache.
- EBox Port - The EBox aligned quad output port is 8 bytes wide. The data traffic manager sends the data using the quad word path for quad word aligned reads.

  There are two response lines, one for bytes <03:00> and one for bytes <7:4>. All operands with context less than quad word use the lower four bytes.

3.4.20 Data Traffic Manager/Rotator

The rotator (Figure 3-58) rotates a longword, 8 bytes, by one, two or three bytes. Output data is aligned, in that, byte 0 of the requested data is in the right most position.

Each DTMX MCA (Figure 3-59) generates one rotate byte of the longword.

An example of a shift by one byte is shown in Figure 3-60, and an example of a shift by two bytes is shown in Figure 3-61.

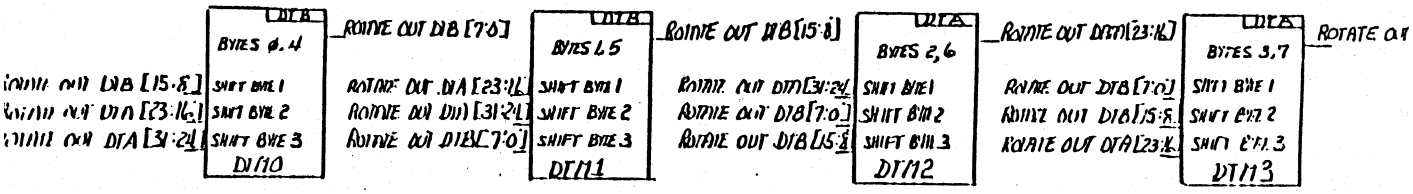


Figure 3–58 Data Traffic Manager/Rotator

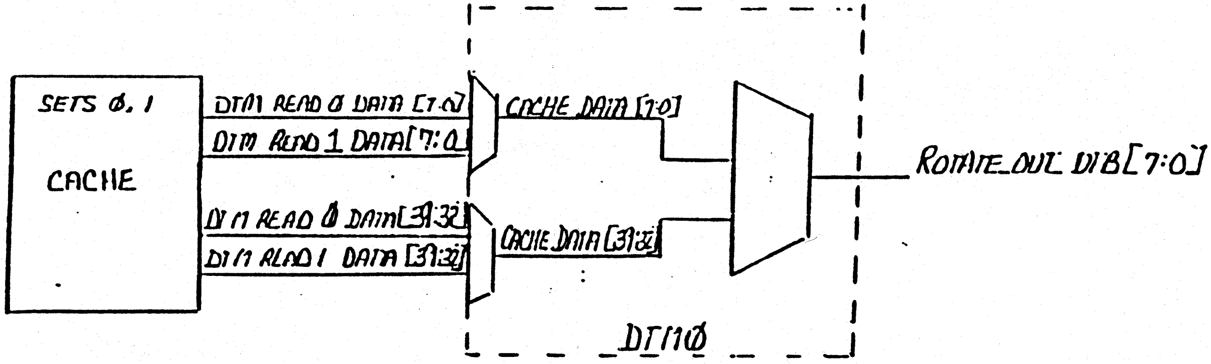


Figure 3–59 Rotate Byte Selection

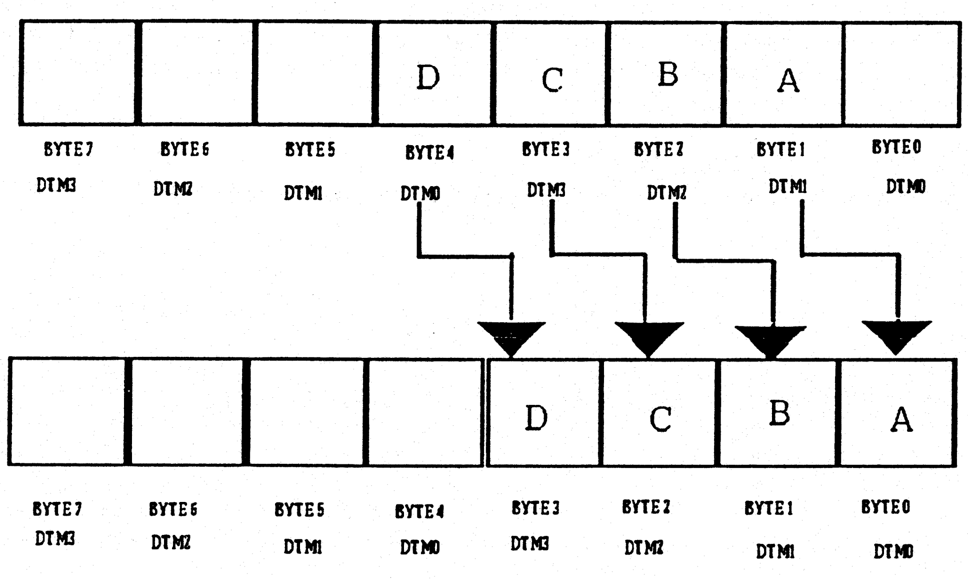


Figure 3–60 Shift by One Byte

3.4.21 MBox Operations

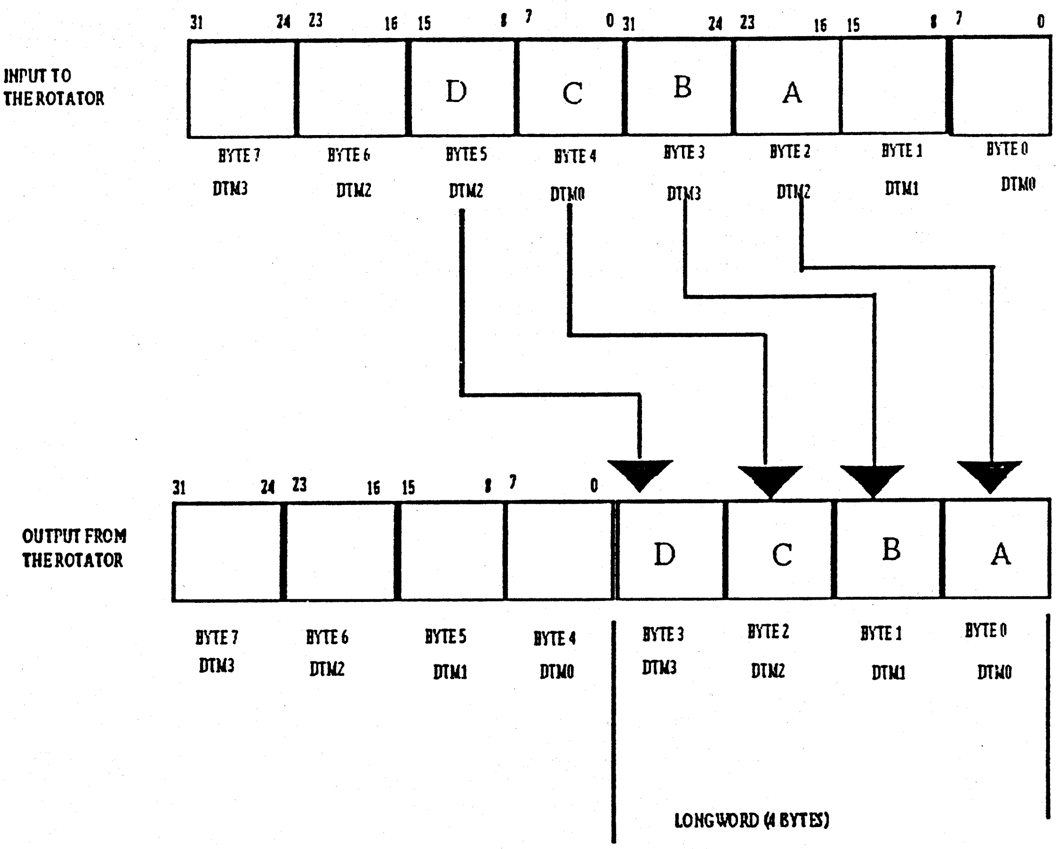


Figure 3–61 Shift by Two Bytes

3.4.21.1 I/O Reads and Writes

The MBox does not cache CPU writes and reads to I/O space. The MBox detects an I/O address, and assembles a command and address for the JBox. For a write, the MBox places the EBox register data as the top entry in the write back buffer. The MBox receives a SENDDATA signal from the JBox, and sends the data in a single cycle. For a read, the JBox places the data in the refill buffer.

3.4.21.2 Cache Write Operations

Table 3–29 lists the cache condition and corresponding write operation.

Table 3–29 Cache Conditions

Cache Condition	Write Operation	Description
Valid/Written	Aligned	This is a two cycle operation. The write takes place immediately after the lookup cycle. Partially valid blocks do exist so the write cycle may have required the assertion of the valid bit.
Valid/Written	Unaligned	Unaligned writes to valid blocks require a merge cycle between lookup and write. During the lookup cycle the MBox holds the line that hit and merges the EBox data via the data traffic manager/rotator into the cache data. The sequence is lookup, merge, and write.
Valid/Not Written	Aligned	This write would be accomplished in two cycles. However, due to the multiprocessor environment, the policy is to send a request to the JBox called status update. The JBox ensures that this block is not valid and written in another CPUs cache before the write can complete. the JBox responds with a command giving permission. Only then can the write complete along with the update to the tag.
Valid/Not Written	Unaligned	This situation is similar to the previous. The data requires a merge cycle when permission to write is received from the JBox.
Not Valid/Written	Aligned	Invalid condition.
Not Valid/Written	Unaligned	Invalid condition.
Not Valid/Not Written	Aligned	The MBox supports write allocates to previously invalid blocks provided it is longword aligned. This is accomplished in two cycles but does require that a command and address is sent to the JBox to update its copy of the tags. Any cache consistency conflicts are handled by the JBox without effecting the writing CPU.
Not Valid/Not Written	Unaligned	This is a complete miss and the CTMs assemble a command and address and sends it to the JBox. The MBox rotates the EBox data into the correct position, so that when the data is returned, it is merged into the refill data as it is written into the cache.

In addition, the cache writes can occur under a cache miss. There is a limited number of conditions under which a write under a miss can occur. That is, only when the completion of the write does not involve a change in status.

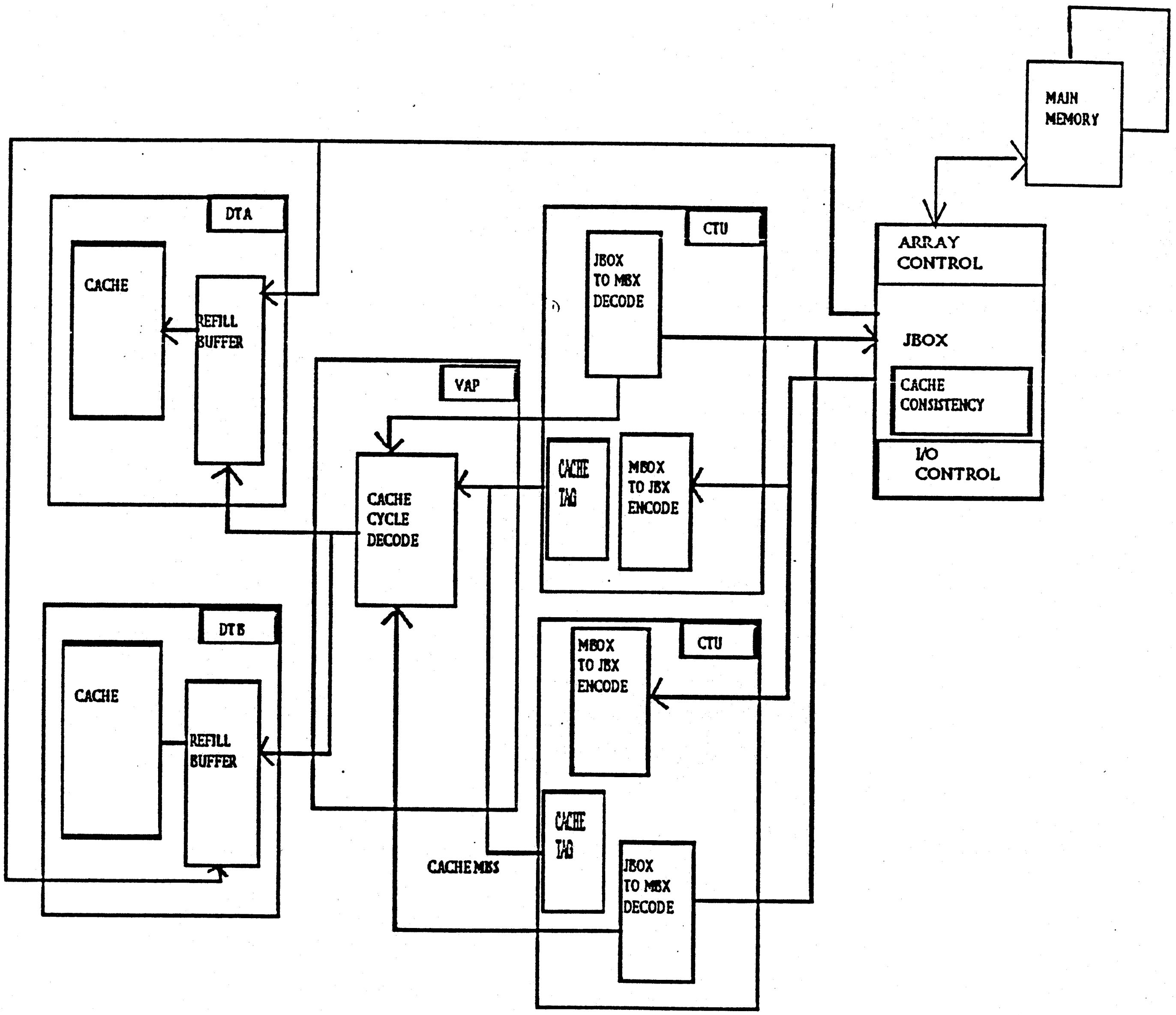


Figure 3–62 Refill Operation

3.4.21.3 Cache Refill

When the IBox, EBox, or TB fixup unit generates a read or write request that results in a cache miss, the MBox requests a block of data (64 bytes) from the JBox. The read is wrapped, and the requested quadword returns first.

The sequence of refill events (Figure 3–62) to obtain data in main memory is as follows:

1. CTMA and CTMV detect a cache miss and assemble a command and address to request a block of data from main memory.
2. MBox asserts MBOX_JBOX_LOAD_CMD.
3. JBox accesses main memory, obtains the data, and assembles a corresponding command and address to return data to the MBox.
4. JBox asserts JBOX_MBOX_LOAD_CMD.

5. JBox loads data in the refill buffer in 8 consecutive cycles.
6. Simultaneously, when the last cycle is complete, CTMA and CTMV updates the cache tag.

The command and address is time multiplexed across two consecutive cycles. The first cycle contains the address components sufficient to begin the transaction (cache tag address, row address bits, etc.) and the second cycle contains the remaining bits.

3.4.21.4 Cache Write Back

WBEM and WBES contain and control the write back buffer, and are shown in Figure 3–63 and Figure 3–64, respectively. Write back buffer byte-slices are shown in Figure 3–65

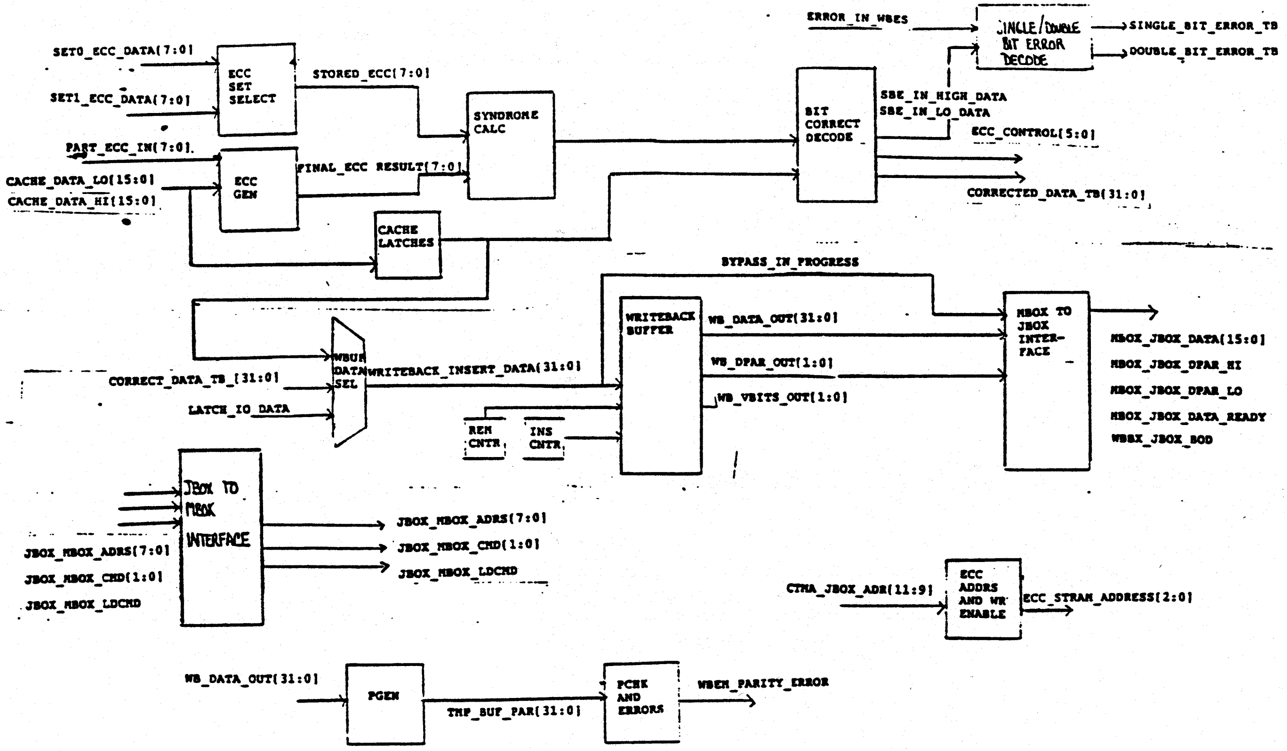


Figure 3–63 WBEM MCA Block Diagram

Write back buffer byte slices are shown in Figure 3–65.

A write back is a transaction in which the MBox sends valid and modified cache data to the JBox to be written into main memory. Write backs occur when the MBox or JBox issue the following commands.

Read Miss with Write Back A read miss with write back condition occurs when an attempt to read cache results in a miss, and there is valid data in cache. It takes 20 cycles to resolve a read miss with a write back. During nine of the 20 cycles, the cache unloads valid data that is to be written in main memory.

The sequence of events to handle a read miss is as follows. Figure 3–66 illustrates a write back operation.

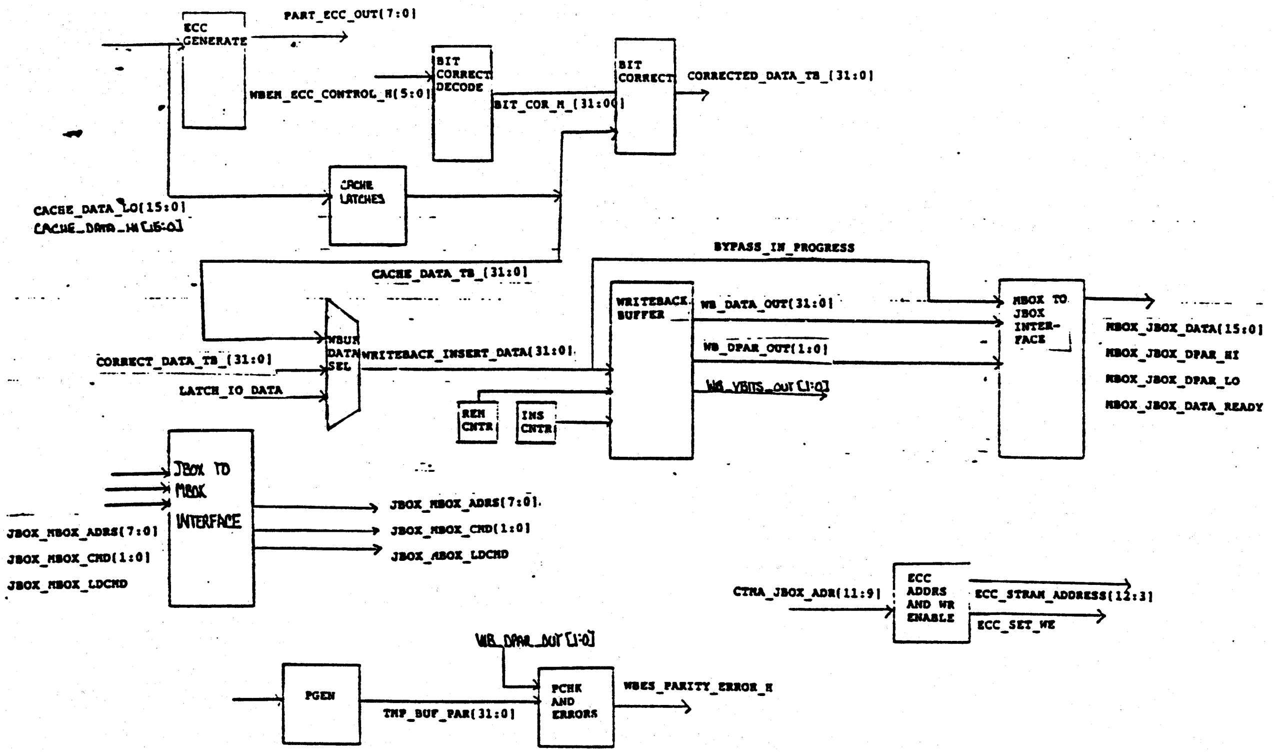


Figure 3–64 WBES MCA Block Diagram

DTA	DTA	DTB	DTB	DTA	DTA	DTB	DTB
<63:56>	<55:48>	<47:40>	<39:32>	<31:24>	<23:16>	<15:8>	<7:0>

CACHE QUADWORD

WBES				WBEM			
<63:56>	<55:48>	<31:24>	<23:16>	<47:40>	<39:32>	<15:8>	<7:0>

WRITEBACK INSERT QUADWORD

Figure 3–65 Write Back Buffer Byte Slices

1. CTMV and CTMA, using the least-recently-used decode, selects the cache and detects a read miss. If there is no valid data in the cache block, CTMV and CTMA assembles a command and address to refill cache.
2. If there is valid data in the cache block, CTMA and CTMV assembles a command and address, read-miss-with-writeback.
3. The write back is performed first. Cache block is sent to the JBox in 8 consecutive cycles. At the end of the eight cycles, the cache block is available to other CPUs.

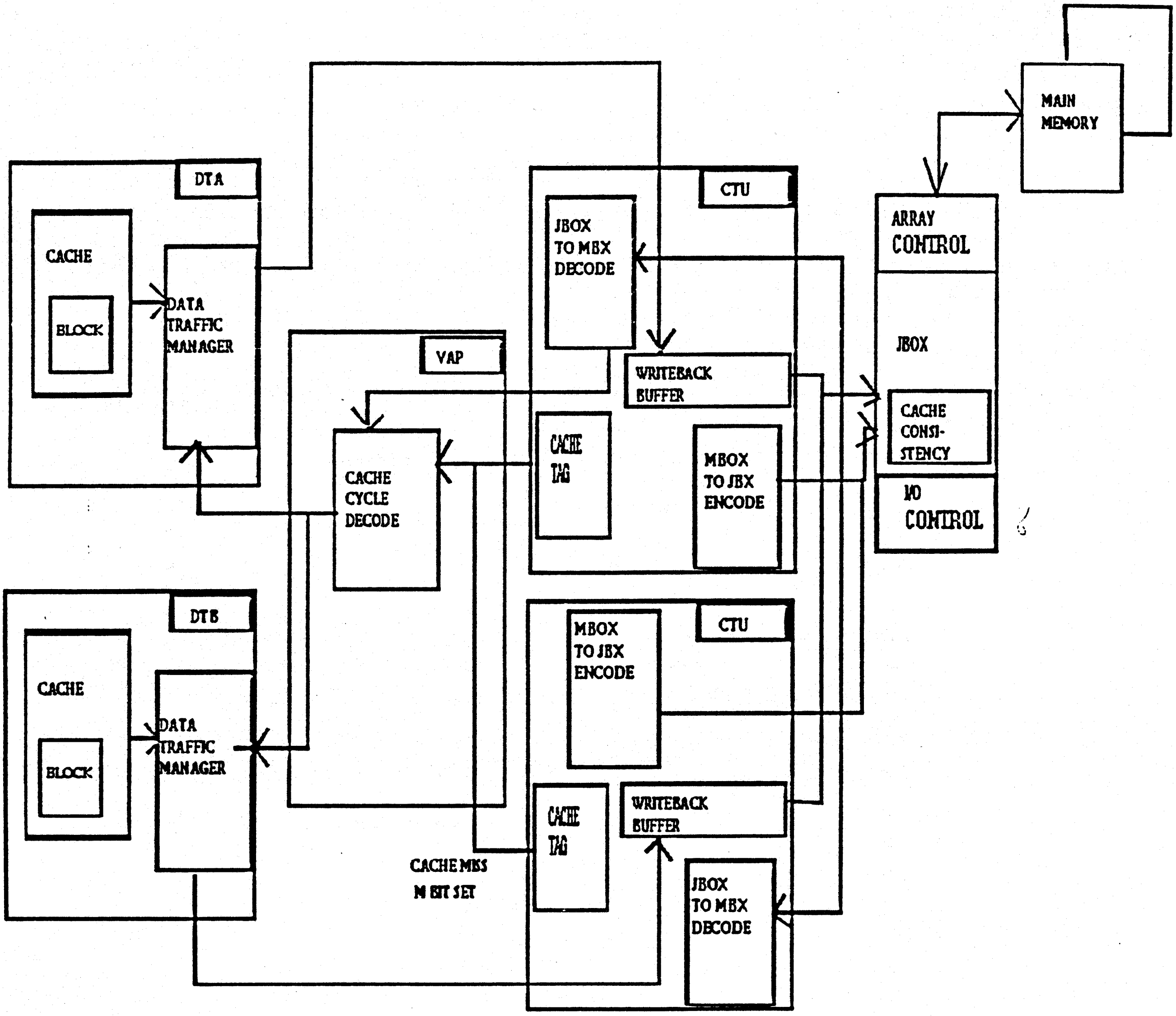


Figure 3–66 Write Back Operation

4. The cache refill occurs in 8 consecutive data cycles. When the last data cycle completes, CTMA and CTMV update the cache tag.

Write Miss with Write Back Writes to the cache may cause a write back similar to a read miss. This occurs when there is a write miss and both blocks (two-way set associative) contain valid and modified data that must be written in main memory. Aligned longword writes that miss are normally written without delay, unless both blocks are to be written.

JBox Induced Write Back A JBox induced write back occurs when processor "A" requests a block of data that is valid and written in the cache of processor "B". The JBox sends a command and address to processor "B" requesting that cache block of data. As a result, the JBox induces a write back from processor "B". The JBox writes the data back to main memory, and sends it to processor "A". If the refill to processor "A" was the result of a write miss, then the JBox invalidates the cache block in "B". If not, the cache block in "B" remains valid.

Cache Sweep Aside from detecting and reporting its own errors, the MBox has an additional requirement during the recovery phase of error handling. The MBox reports errors to the EBox. The EBox then traps the micro-machine or interrupts the instruction stream at an instruction boundary, and issues a cache sweep.

At the conclusion of the cache sweep, the EBox stops the CPU clocks, scans the retained state of the CPU, and restores it so that processing can continue when the clocks are turned back on. Cache sweep unloads all written blocks of data to main memory.

CTMV and CTMA reads an entry in the TAG store. If the entry is not valid, the counter increments to the next tag address and read the next entry. If the entry is valid and not written, CTMV assembles the full 34-bit address, and sends an invalidate command to the JBox, so that the tag store copy in the JBox can be swept. FALT determines if the block of data is written. CTMV assembles the address and write back command. CTMV continues until all 2K tag store entries are invalidated.

Throughout the sweep, the port from the JBox to the sweeping MBox remains open. The MBox arbitrates for the JBox port during idle breaks in the sweep routine.

4
SCU and Main Memory Functional Overview

4.1 System Control Subsystem Introduction

The System Control Unit (SCU) interconnects the CPU, I/O subsystem, main memory arrays, and SPU. The unit also provides I/O device, SPU, and inter-CPU interrupt and exception handling. As shown in Figure 4–1, the SCU is logically partitioned into three major functional units.

Junction Box (JBox): provides up to four ports which interface up to four CPU subsystems, and includes address and data crossbars, and cache consistency logic.

I/O Control Unit (ICU): provides up to four ports into the JBox which interface to the I/O system bus and bus adapters. The ICU controls the I/O subsystem through the XMI busses. The SCU can be expanded to include two ICUs (i.e., ICU0 and ICU1) with:

o each ICU capable of handling up to two XMI busses,

o each ICU connected to its XMIs through the JBOX-XMI Data Interface (JXDI) cable, and

o each JXDI connected to an XMI-JBox Adapter (XJA) module in the XMI cardcage.

Array Control Unit (ACU): provides up to two ports into the JBox which interface to the main memory arrays. The SCU can also be expanded to include two ACUs (i.e., ACU0 and ACU1) providing the two ports (one port per ACU). The memory control unit is functionally integrated into the ACU. The memory is logically divided between the two ports, with each port supporting one Main Memory Unit (MMU). Each ACU provides dynamic timing signals to the Memory Array Cards (MACs) and Daughter Array Cards (DACs) of an MMU. The SCU can support up to two MMUs for a maximum main memory capacity of 512 Mbytes (with 1Mbyte DRAMs).)

The SCU provides 8-byte (quadword) wide data interconnects, and address and data crossbars between the MBox, ICU, and ACU. The interconnects and crossbars allow simultaneous transactions between functional unit ports. That is, the SCU has the potential of maintaining all 10 ports active, providing there are no port conflicts. However, this type port activity requires extensive communication, validity checking, and parallel operations.

Another major SCU function is to manage memory access. Since memory data is distributed in the CPU caches, the data in main memory may not be valid. The SCU tracks the location of valid data, and insures that a memory port request results in a valid read or write operation. It assures cache consistency across all CPU caches, and assumes

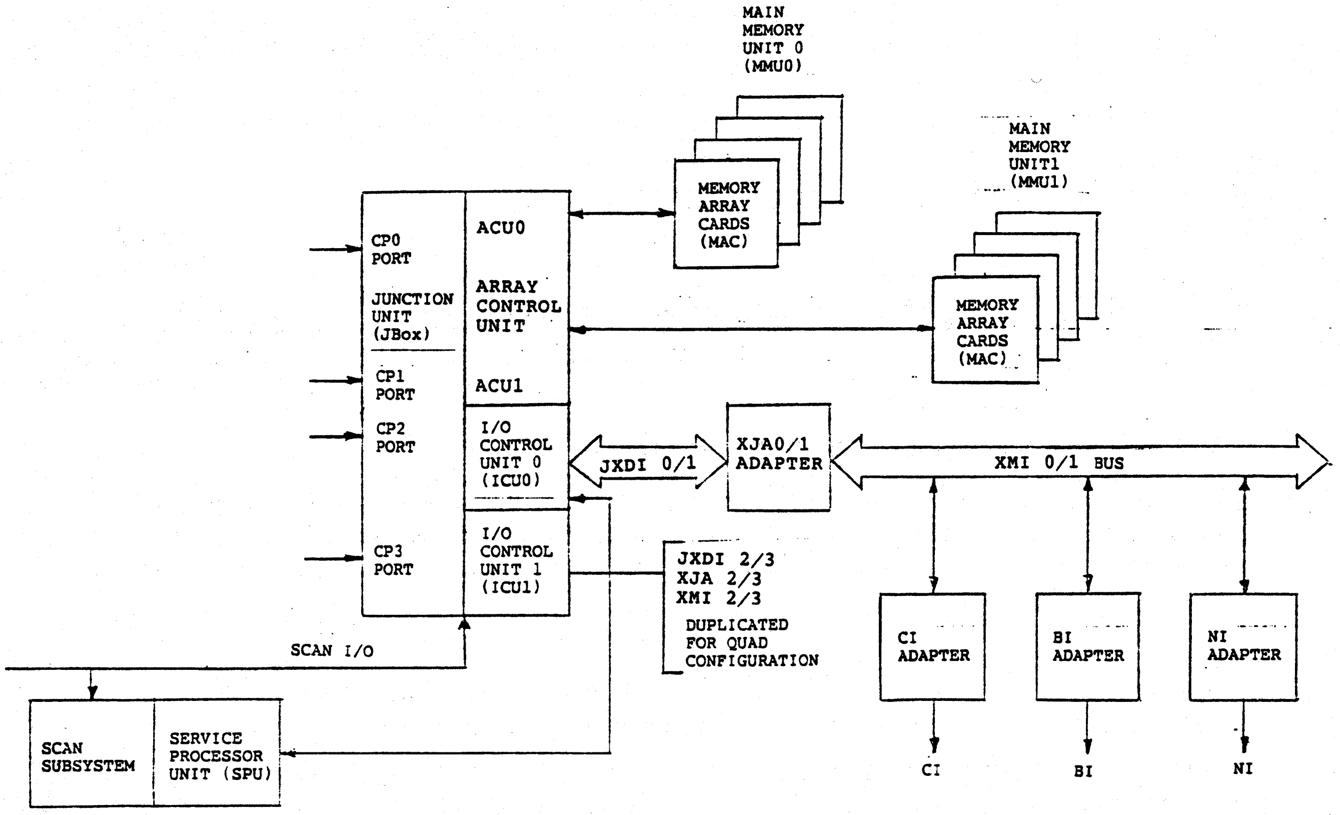


Figure 4–1 Basic SCU Subsystem

responsibility for blocks which may require invalidating as a result of I/O write operations to memory.

Interrupts from the I/O devices, SPU, and between CPUs are distributed by the SCU. Except for inter-CPU interrupts, they are distributed in either round-robin fashion, or directed to a single CPU.

The SCU connects the SPU to the remainder of the system. Communication is performed through SCU registers which are assigned I/O space addresses. SCU registers are also used to configure memory and I/O, and for interrupt and exception status.

4.1.1 Data and Address Interconnects

The SCU interconnects between the MBox, ICU, and ACU are independent 8-byte (quadword) wide data interfaces (i.e., one quadword interconnect for each data transfer direction). A quadword of data can be transferred over each interconnect on every clock cycle (16ns), providing a raw Bandwidth of 500 Mbytes/sec. Note that the data transfer packet size is identical to the CPU block size.

The address interface between the JBox and MBox is 2-bytes wide. A complete address transfer requires two cycles. The address bits on this interface are Physical Address (PA) 33:02.

The address from the ACU to the memory system is divided into two components:

- Multiplexed Row Address/Column Address: sourced by the SCU and applied to the Memory Array Cards (MACs) in the Main Memory Units (MMUs). This field is up to 13-bits wide, depending on the DRAM chip size in use.

- o Memory Port, Segment, and Bank Select fields: these fields are sourced by the SCU and are applied to the Main Memory Control (MMCx) chips in the ACU.

There are no lines to transfer the return address from the memory system to the ACU. The memory returns an index field which allows the JBox to associate data returned from memory with one of 12 address registers.

The JBox to ICU address interface is also 2-bytes wide. A complete address transfer requires two cycles.

4.1.2 JBox Tag Rams

To maintain cache consistency between the CPU write back caches, the SCU maintains duplicate Tag RAM arrays. The Tag RAMs are located in the JBox with a Tag RAM associated with each CPU cache.

Whenever a CPU makes a reference to the JBox, or I/O is accessing memory:

- o all Tag RAMs are examined in the JBox,
- o the received address is matched against the content of the appropriate Tag RAM,
- o the status of the addressed block determined, and
- o appropriate action, if any, to maintain cache consistency is determined.

The caches in the CPUs are 2-way set associative, with each set having 1K tag entries. For economy of RAMs, 4K x 4 RAMs are used in the global tag arrays in the JBox. Two, 1K sections of the 4K RAM are used for the tags, and the other 2K space is unused. Thus doing a tag lookup for both sets requires two cycles.

4.1.3 Physical and Functional Overview

The SCU requires four MCUs for a basic configuration which supports two CPUs, one Main Memory Unit (MMU), and two XJAs. Incorporating two additional MCUs provides the maximum configuration which supports four CPUs, two MMUs, and four XJAs.

Figure 4–2 presents a basic SCU functional block diagram. Figure 4–3 presents the physical MCU layout for a fully populated SCU planar module, including all port interconnect connectors. Figure 4–4 identifies the MCA functions within each MCU.
)

The SCU MCU functions are summarized in the following subsections.

4.1.3.1 DAx MCU Description

There can be two DAx MCUs (Figure 4–5), DA0 and DA1. The MCUs transfer I/O commands to the CCU (Cache Consistency Unit) MCU.

The MCUs contain the SCU registers and provide a path between the data crossbar and the memory array cards. Each MCU contains eight MCAs and handles the following address and data slices:

- — Bytes 0 - 3 of the data to the DSXX (Data Switch or crossbar) MCAs
- — two of the four address bytes to the TAG MCU
- — one of the two nibbles of the SPU interface

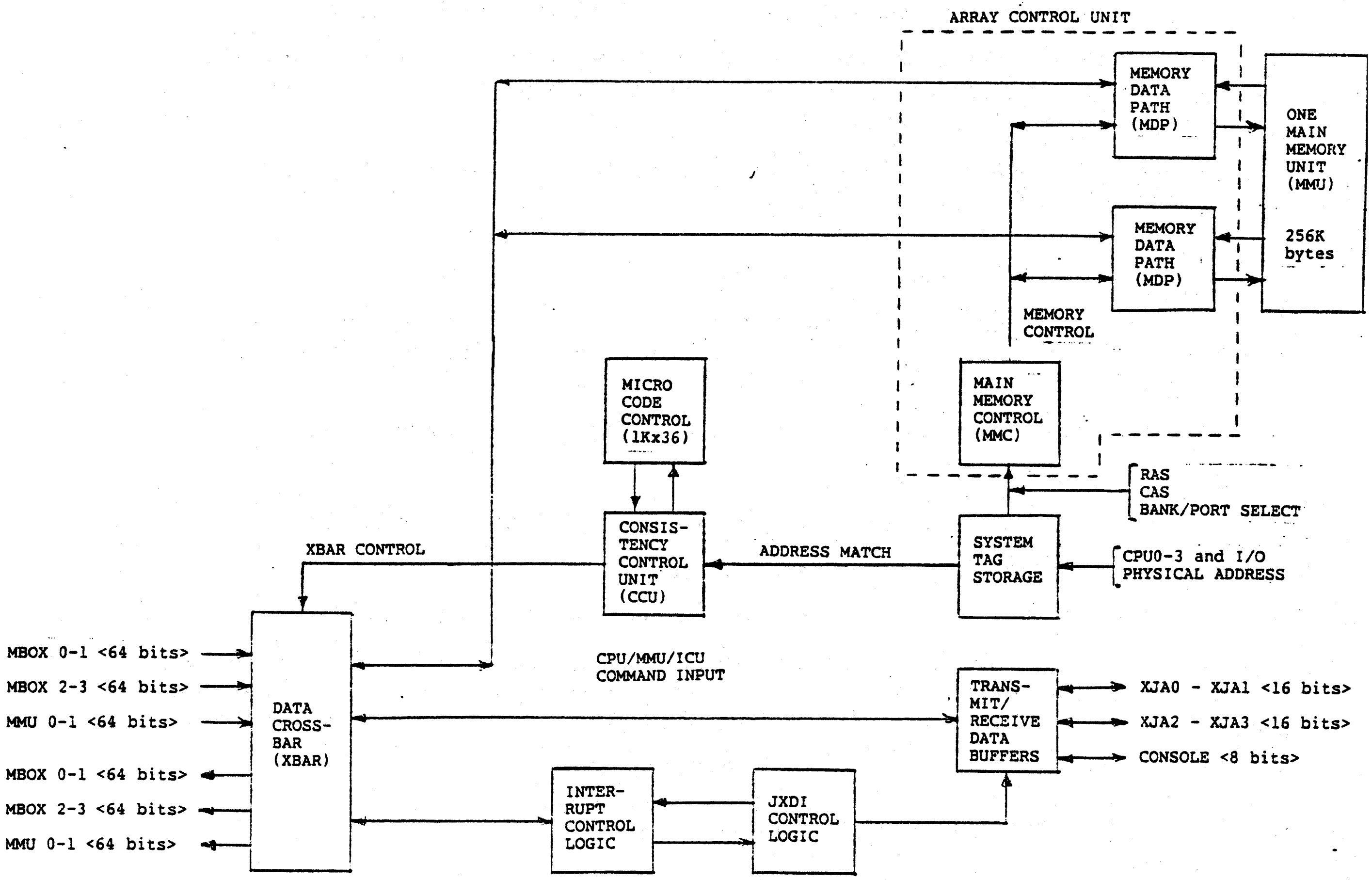


Figure 4–2 Basic Functional Block Diagram

— one of the two bytes of the JXDI interface

Each MCU contains eight MCAs. The MCA functions are summarized in Table 4–1.

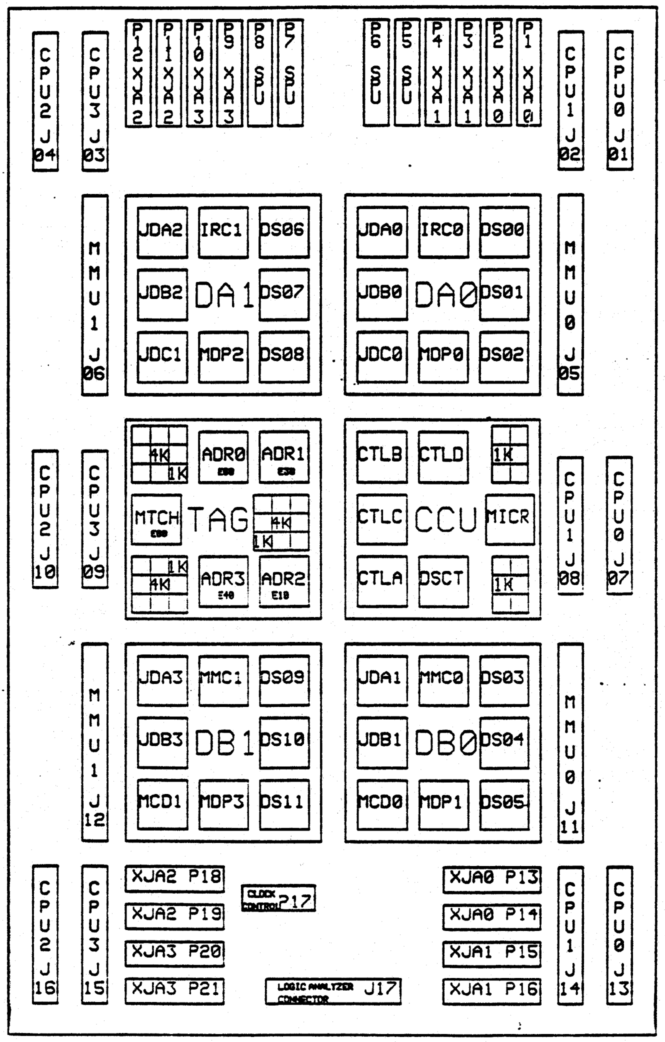


Figure 4-3 MCU Planar Module Layout

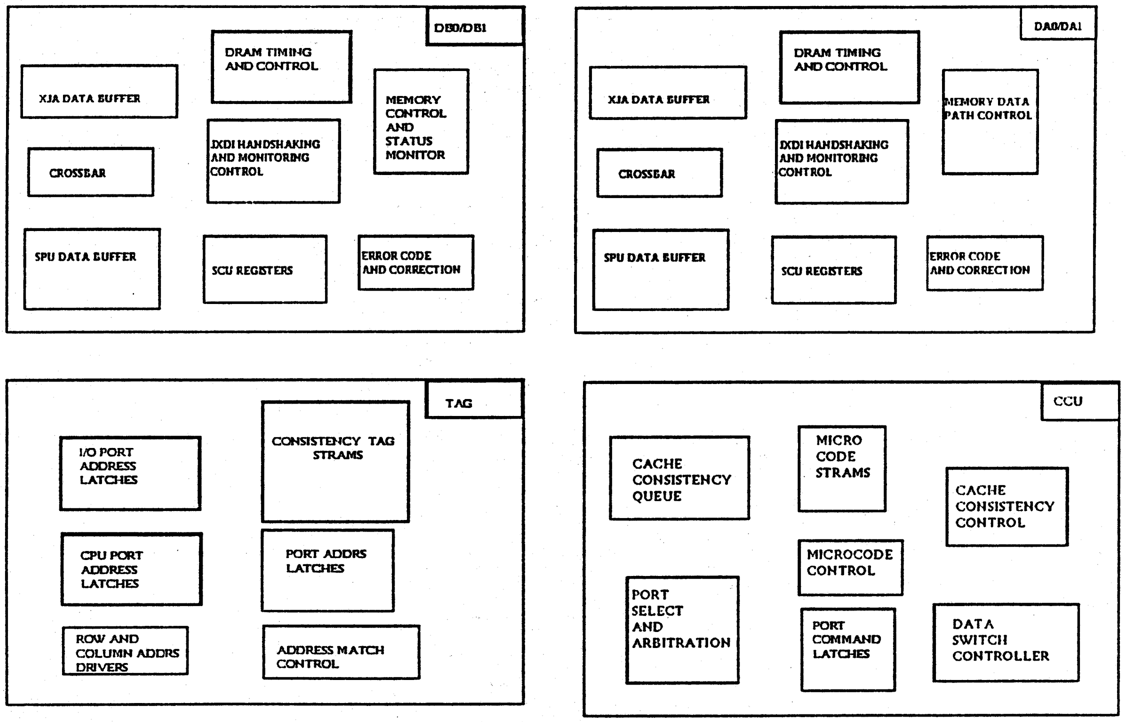


Figure 4–4 MCU/MCA Functions

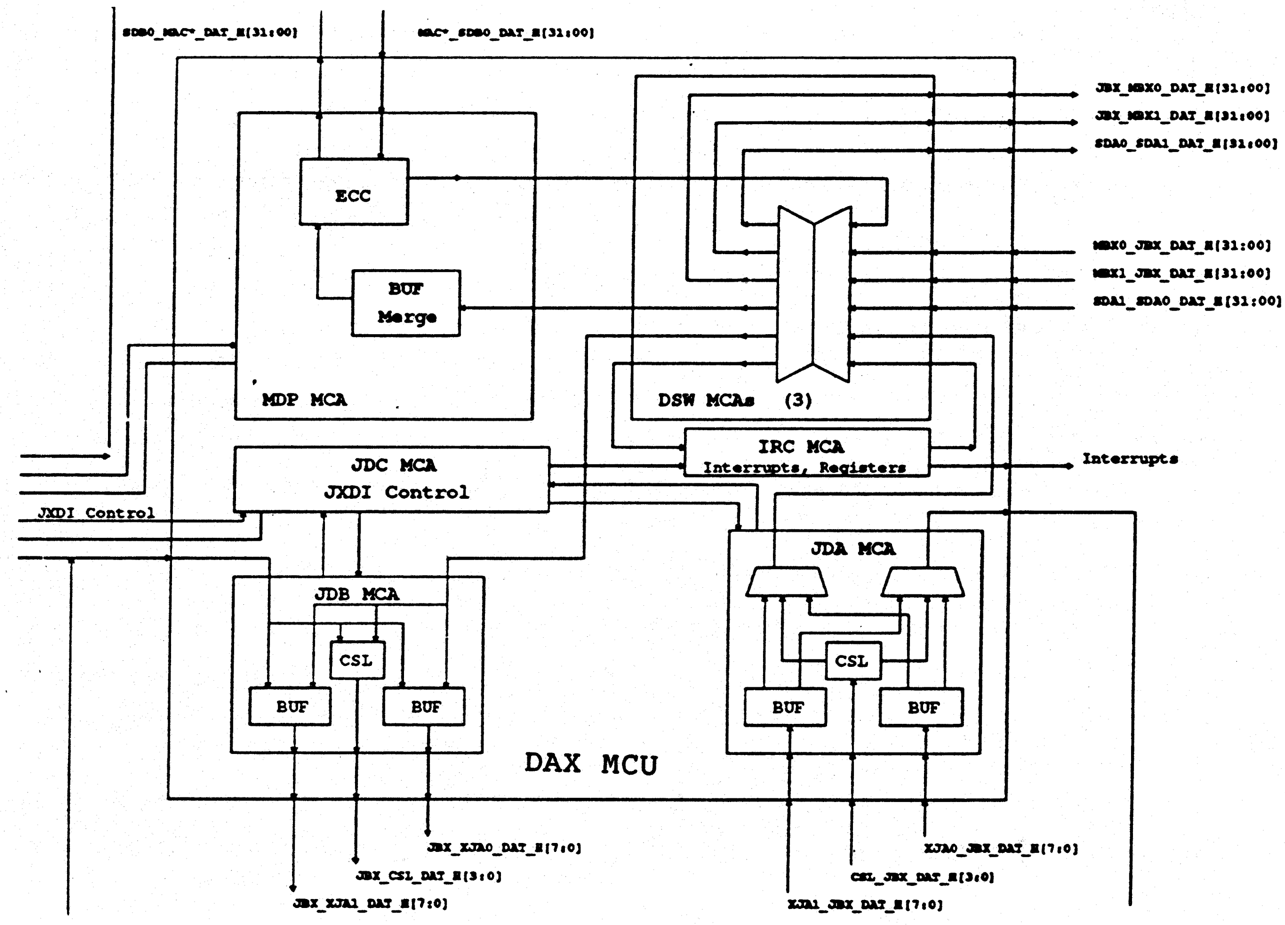


Figure 4-5 DAx MCU Block Diagram

Table 4-1 DAx MCA Descriptions

TYPE	DESCRIPTION
JDA0	The XJA and SPU data buffer MCA receives one byte of the interface from each of the two XJAs. It receives one nibble of the SPU interface. It sends half of the address bits to the TAG MCU. It buffers and sends the data from the XJAs (through a 4-byte wide path) to the group of DSXX. It sends commands from the I/O to CCU.
JDB0	This MCA is similar to the JDAX and deals with signals in the reverse direction.
JDC0	This MCA controls the operation of the JDAX and JDBX and monitors their errors. It coordinates the handshaking signals to and from the JXDI, and to and from the CCU. It sources the clocks that are transmitted to the XJAs.
DSxx	The Data Switch MCAs (DS00, DS01, and DS02) are 64-bit block multiplexers which provide crossbar capability for 4 bytes of data. They support CPUs 0 and 1, memory port 0, and XJAs 0 and 1. For register reads and writes, they have a 4-byte wide path to and from the IRC.
IRC0	This MCA contains the SCU registers and interfaces to the crossbar. It contains the interrupt logic for I/O to CPU interrupts and inter-CPU interrupts. It handles the handshaking signals for the SPU interface.
MDP0	This MCA provides a 4-byte wide path between the data crossbar and the memory array cards. It handles ECC and read-modify-write operations. It provides check bit generation for write data. It detects and corrects single bit errors and detects double bit errors. It generates data patterns during BIST. It contains the byte merge logic.

The DA1 MCU contains identical MCAs and functions for the expansion ports:

o JDA2

o JDB2

o JDC1

o DS06, DS07, DS08

o IRC1

o MDP2

4.1.3.2 CCU MCA Descriptions

The CCU MCU contains the cache consistency unit, and the JBox control unit. It tracks valid data locations and manages data to and from the ports. The MCU contains six MCAs and 18, 1K STRAMs. The STRAMs contain the SCU microcode, and a microPC history buffer. The MCAs are described in Table 4-2.

Table 4–2 CCU MCA Descriptions

TYPE	DESCRIPTION
CTLA	The Control A MCA receives requests (load commands) for data movement. It contains the port arbitration logic. It generates an index that points to a command (in CTLB) and an address (in TAG).
CTLB	The Control B MCA receives and stores twenty port commands and distributes the commands to other ports.
CTLC	The Control C MCA sends commands to the Data Switch Controller MCA (DSCT) It sends cache consistency information to a queue (in CTLD) and sends consistency commands to the ports.
CTLD	The Control D MCA contains the consistency cache queue. The microcode accesses the queue that contains cache check information.
DSCT	The Data Switch Controller MCA controls the Data Switch MCAs (DSXX) located in the DBx MCU
MICR	This MCA controls the SCU microcode.

4.1.3.3 DBx MCU Descriptions

The DB0 and DB1 MCUs provide control signals for the memory array cards and monitor their status. Each contains eight MCAs (Figure 4–6), and handles the following addresses and data slices:

o bytes 4 - 7 of the data to the DSXX MCAs

o two of the four address bytes of the TAG MCU

o one of the two nibbles of the SPU interface

o one of the two bytes of the JXDI interface

The MCAs are described in Table 4–3.

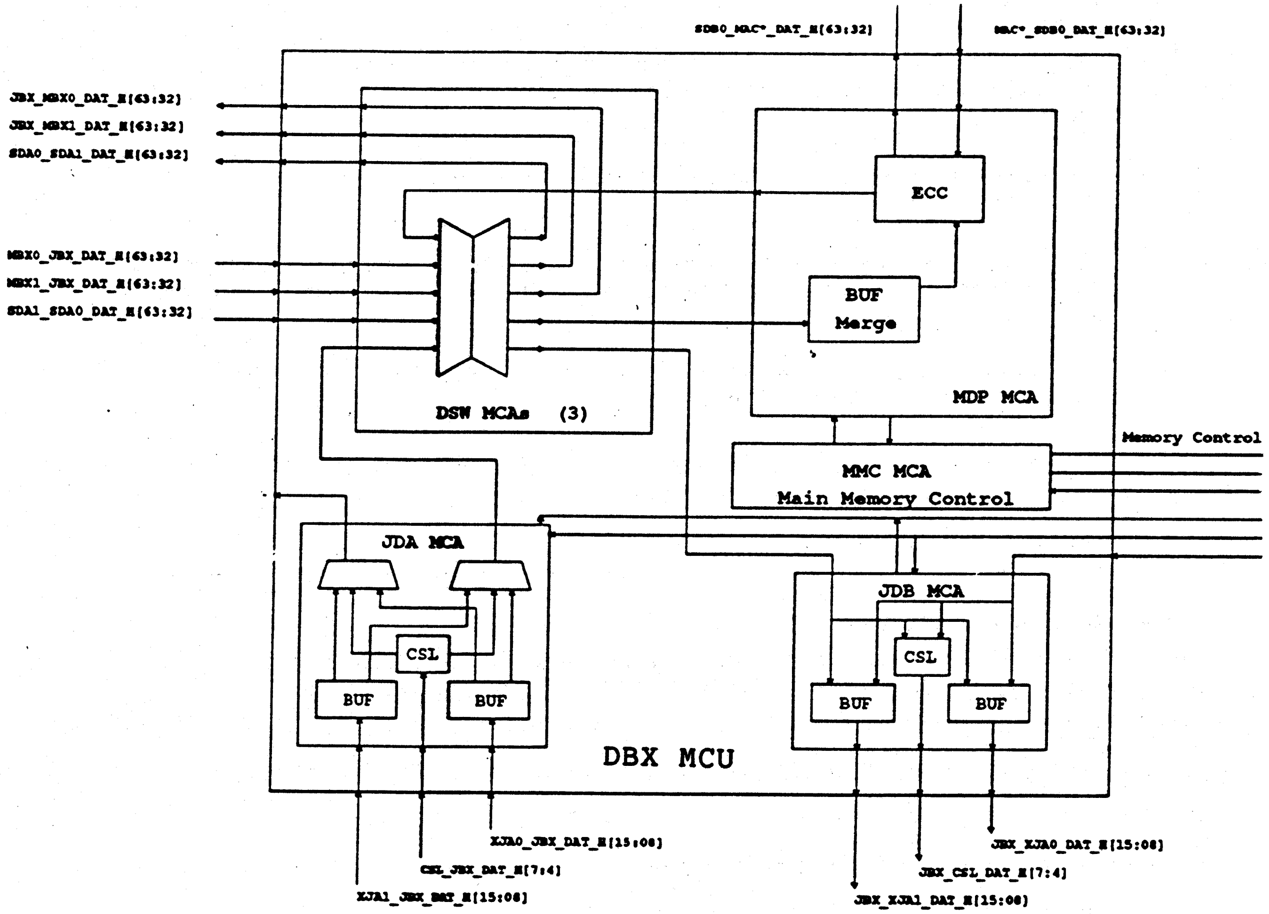


Figure 4-6 DBX MCU Block Diagram

Table 4–3 DBx MCA Descriptions

TYPE	DESCRIPTION
JDA1	The XJA and SPU data buffer MCA receives one byte of the interface from each of the two XJAs. It receives one nibble of the SPU interface. It sends half of the address bits to the TAG MCU. It buffers and sends the data from the XJAs (via a 4-byte wide path) to the group of DSXX. It sends commands from the I/O to CCU.
JDB1	This MCA is similar to the JDA1 and deals with signals in the reverse direction.
DSxx	The Data Switch MCAs (DS03, DS04, and DS05) are 64-bit block multiplexers which provide crossbar capability for 4 bytes of data. They support CPUs 2 and 3, memory port 1, and XJAs 2 and 3. For register reads and writes, they have a 4-byte wide path to and from the IRC.
MMC0	The Main Memory Control MCA provides the control signals to the memory array cards and receives status from them. It provides the command, control and status interface to the JBox. It provides the data path control and DRAM control commands to the MCD. It provides the error detection on all MMC control lines and supports BIST operations.
MDP1	The Memory Data Path 1 MCA provides a 4-byte wide path between the data crossbar and the memory array cards. It handles ECC and read-modify-write operations. It provides check bit generation for write data. It detects and corrects single bit errors and detects double bit errors. It generates data patterns during BIST. It contains the byte merge logic.
MCD	The Memory Control DRAM MCA provides DRAM timing and control, as well as commands to the MMC MCA during Self Test.

The DB1 MCU contains identical MCAs and functions for the expansion ports:

- JDA2
- JDB2
- JDC1
- DS06, DS07, DS08
- MDP2

4.1.3.4 TAG MCA Descriptions

The TAG MCU receives and transmits addresses to and from the ports. It controls the tag STRAMs and determines if there is an address match. As shown in Figure 4–7, the MCU contains five MCAs, 24, 4K, and 3, 1K STRAMs. The MCAs are described in Table 4–4.

Table 4–4 TAG MCA Descriptions

TYPE	DESCRIPTION
MTCH	The Match MCA drives the addresses and data to the TAG STRAMs. It receives addresses from the TAG STRAMs and matches these addresses with the addresses received from the ports. It sends address match signals to the CCU.
ADRx	Each address MCA (ADR0, ADR1, ADR2, and ADR3) deals with one quarter.of the address bits. Each receives one quarter of the address field from the four CPU ports and two I/O ports and transmits one quarter of the address field to the same ports. They source row and column addresses to the MMUs. The address signals from each port are double buffered to accommodate the frequent occurrence of a write back accompanying a refill.

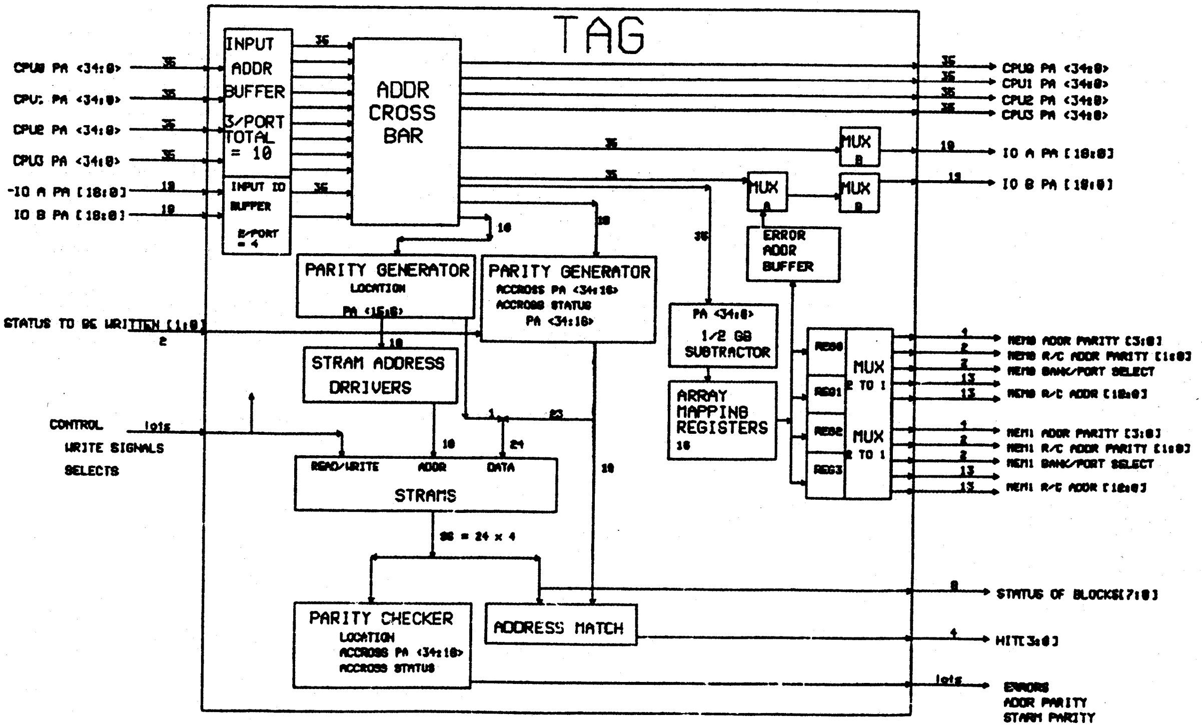


Figure 4–7 TAG MCU Block Diagram

4.1.4 CPU Port

From the perspective of a CPU the SCU resembles a memory controller. Since the CPUs implement write back caches, the SCU ensures cache data consistency. The SCU accomplishes this through the use of duplicate consistency tag stores for each CPU. The JBox portion of the SCU implements the cache consistency logic.

Data paths between the CPU and the JBox are 64-bits wide, and terminate and originate in the MBox. The address interfaces between the boxes are 2-bytes wide. A complete address transfer requires two cycles. Commands are transferred between the boxes, with each box containing encode and decode logic. Table 4–5 lists the JBox to MBox port commands. Table 4–6 lists the MBox to JBox port commands.

Table 4–5 JBox to MBox Commands

Command	Description
0000	Read Refill
0001	Write Refill
0010	Read Refill Linked
0011	Write Refill Linked
0100	Write Refill Lock
0101	Write Refill Unlock
0110	Write Back
0111	Longword Write Update
1000	Read I/O Register
1001	Write I/O Register
1010	Unused
1011	Longword Write Update Linked
1100	Invalidate
1101	Write Refill Link Lock
1110	Unlock
1111	Unused

Table 4–6 MBox to JBox Commands

Command	Description
0000	Get Data Written
0001	Get Data Read
0010	Get Data Invalidate
0011	Return Data Read
0100	Return Data Written
0101	OK to write
0110	Invalidate Read Block
0111	Return I/O Register Data
1000	Return Read Error Status
1001	Lock Acknowledge
1010	Memory Read NXM
1011	I/O Read NXM
1100	Lock Denied
1101	Invalidate Written Block
1110	Unused
1111	Unused

4.1.5 Managing Memory

One of the SCU's functions is to keep the subsystems running in parallel. Its main function, however, is to manage memory access. Since memory data is distributed in the CPUs' caches, the data in main memory may be invalid.

The SCU tracks valid data and ensures that a port's request results in a valid read or write. It does this by keeping duplicate cache tag stores for the four CPU caches. On each memory transaction, the SCU checks its tags for hits.

When a CPU writes to a memory location, the SCU ensures that the location is invalidated if it is present in any other cache. For this purpose, the MBox notifies the SCU the first time it writes to a cache block.

Main memory is split between two memory ports on the SCU. Each port has two segments. All four segments can be cycled in parallel, in that, there can be up to four memory references handled simultaneously. When a CPU or I/O port makes a memory request, the SCU passes it on to the right segment. The segments are interleaved on block boundaries. Block 0 is in segment 0, block 1 is in segment 1, block 4 is in segment 0, and so on. Each block is 64 bytes long and matches the size of the CPU cache block.

The SCU offers a crossbar connection between its ports and allows simultaneous transfers if there are no conflicts. Each transaction is done at a rate of 8 bytes/cycle.

In the case of conflicts, the SCU stores pending requests in a set of double buffers for later processing. This accommodates the common sequence of a cache refill request followed by a write back request. The CPU can send two requests and can continue to process other local cache references.

4.1.6 SPU Port

The SPU (Figure 4–8) is a uVAX driven, BI-based, dedicated controller that provides service and maintenance support for the computer system. It is the operator console and initializes controller startup, and shutdown. It monitors the system and tests and diagnoses hardware faults.

The I/O controller (ICU) interfaces to the SJAs (via the JXDI cable) and the service processor (SPU), and implements the Central System Interrupt Arbiter.

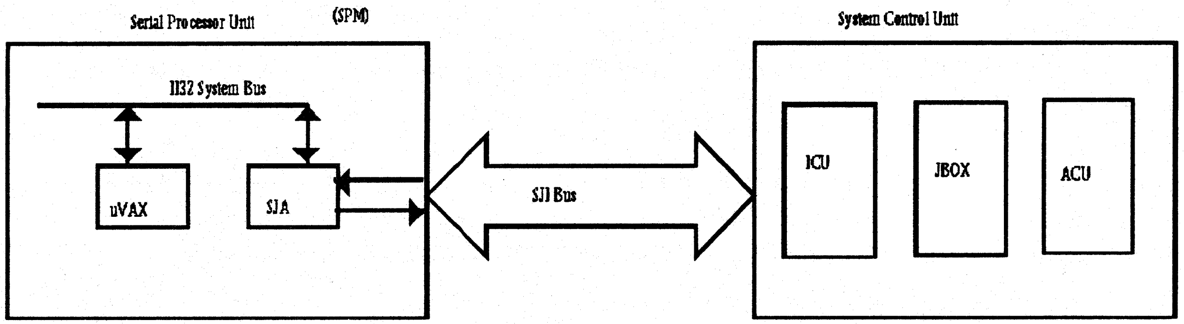


Figure 4–8 SPU Port

The SCU connects the service processor (console) to the remainder of the system. Communication is performed through SCU registers. The SCU registers have I/O space addresses and configure memory and I/O. They also contain interrupt and exception status.

The SPU to JBox adapter contains the following registers:

- Two RX, RXCS and RXDB, registers that transfer data received from the console terminals to the CPUs.
- Two TX registers, TXCS and TXDB (one per CPU). They transfer data received from the CPUs to the console terminals.
- TODR, time of day register, interfaces the TOY clock of the SPM to the CPU.
- MWBR, memory window base register, contains a base address for windowing bus transactions into main memory and I/O space.
- DX (DMA Transfer) Registers, DXCS, DXSPU, DXMEM, DXCNT. All DMA transfers are quadword oriented. The addresses are automatically aligned to quadword boundaries.

- SJCS, (SPU-JBox Interface) Register - This register centralizes the interrupt enabling for all interrupt flags which pertain to the SPU. It contains the interrupt enables for power fail interrupts directed to the CPUs, the Reboot mechanism for forcing an SPM reset, a mechanism for logging the ID of the primary CPU and a bit to allow a software reset of the SJA. This register is not accessible to the CPU and may not be modified (directly) by the operator.
- Flag (Interrupt Flag Register) - centralizes the interrupt flags which pertain to the SPU.
- RXFCT (Receive Function Request) Register - This register allows the II32 (internal SPU system bus)to transmit information to the CPUs. The format allows up to 64K possible function codes plus an eight bit parameter passing field. The RXPRM register may be used to pass up to 32 additional bits of parameter information. A JBox read of this register will generate an interrupt.
- RXPRM (Receive Function Parameter) Register - This register in conjunction with RXFCT register allows passing up to 32 additional bits of parameter information from the II32 to the CPUs.
- TXFCT (Transmit Function Request) Register - Allows the CPUs to transmit control information to the II32. The format allows up to 64K possible function codes plus an eight bit parameter passing field. The TXPRM register may be used to pass up to 32 additional bits of parameter information.
- TXPRM (Transmit Function Parameter) Register - Conjunction with TXFCT allows passing parameter information to CPUs.
- Reboot Register - A JBox write of a 1 to bit 0, REBOOT, sets bit 16 in the SJCS register.
- XJA Register - Allows the II32 to signal pending clock shutdown and scanning to the rest of the system.
- Direct Access Registers - Command, ADDR, DATA_HI, DATA_LO allow the II32 to gain direct access to the JBox interface. Most transactions between the II32 and the JBox are handled automatically by the SJA.

The console registers reside in I/O space. EBox microcode generates physical references to implement the move to and from processor register instructions which reference these registers.

The Aquarius specific registers reside in I/O space. These registers provide access to the console functions such as error insertion, error reporting, and local storage media. These registers may be mapped in system virtual space. The CPU and SCU reference these registers using physical addresses.

SJA to JBox interface protocol includes transfers from the JBox to the SJA, transfers from the SJA to the JBox, handshake parity, and transactions that conform to the basic formats for a DMA read/write packet, I/O read/write command packet, ECC command packet, and interrupt command packet.

The JBox to SJA commands are summarized in Table 4–7.

Table 4–7 JBox to SJA Commands

Command	Code	Description
Read Register	0000	JBox wants to read a console register that resides physically in the console subsystem. The I/O packet is used.
Write Register	0001	JBox wants to write a console register that resides physically in the console subsystem. The I/O packet is used.
Return DMA Read	0010	JBox delivers SPU read data that was requested in a previous read request that referenced memory space. The DMA packet is used.
Return I/O Read	0011	JBox delivers to SPU read data that was requested via a previous read request that referenced I/O space. The I/O packet is used.
Return Read Error	0100	The JBox notifies the SPU when read data (requested via a previous read request) that referenced I/O or memory space encountered an error condition. This may be due to a NXM, DBE or fatal XJA or memory errors. The DMA packet is used.
Write Error Reg	0101	The JBox reports to the SPU of an ECC incident involving a memory access requested by the SPU. The data block includes the address where the error was located and an error syndrome (total eight bytes). The ECC packet is used.
Read Lock Denied	0110	The JBox notifies the SPU that a read lock request that referenced memory space encountered an existing lock and the requested data will not be returned. The DMA packet is used.

The SJA to JBox commands are summarized in Table 4–8

Table 4–8 SJA to JBox Commands

Command	Code	Definition
DMA Read Request	0000	SPU wants to read a valid memory space address. The SPU can only have a single read request outstanding at a given time. The DMA packet is used.
DMA Write	0001	SPU wants to write a valid memory space address. The DMA packet is used.
DMA Read Lock Request	0010	The SPU wants to read lock a valid memory space address. The SPU can only have a single read request outstanding at a given time. The DMA packet is used.
DMA Write Unlock	0011	The SPU wants to write unlock a valid memory space address. This must match a previous DMA read lock request. The DMA packet is used.
I/O Read Request	0100	The SPU wants to read a valid I/O space address. The I/O packet is used.
I/O Write	0101	The SPU wants to write a valid I/O space address. The I/O packet is used.
Request Return Read	0110	The SPU delivers read data that was requested via a previous read register request. The I/O packet is used.
Interrupt TRX	0111	The SPU wants to interrupt the operating system due to console terminal receive. The SPU can select which CPU wants to interrupt via the ID field. The interrupt packet is used.
Interrupt TTX	1000	The SPU wishes to interrupt the operating system due to console terminal transmit. The SPU can select which CPU wants to interrupt via the ID field. The interrupt packet is used.

4.1.7 ICU Port

The I/O controller (ICU) port serves as an interface to the XJAs (through the JXDI cable) and the service processor (SPU). It also implements the Central System Interrupt Arbiter.

The JDAx, JDBx, and JDCx provide the functionality referred to as the ICU.

The XJA communicates with the ICU using three basic types of transactions:

1. DMA - These are XMI transactions that select the XJA as the responder node. These can be reads, writes, read locks, or write unlocks. They can be quadword, octaword, or hexword in length. The XJA can have up to four DMA type read transactions (accepted from the XMI) outstanding at any given time. Read data returned from the ICU is forwarded on the XMI as read data response transactions.
2. CPU - The CPUs can access the I/O portion of VAX physical address space via CPU type transactions. These transactions are received by the XJA from the JXDI and are forwarded to the XMI with the XJA as the commander. CPU are longword in length and the XJA can only accept a single CPU type transaction at a time.

3. Interrupt - The XJA fields interrupt transactions from the XMI and forwards them to the ICU using interrupt type transactions. The resulting SCB offset vector fetch initiated by an AQUARIUS CPU uses a CPU type transaction.

The XMI to JBox adapter (XJA) and the ICU provide an information path between the JBox and I/O devices. I/O is done by the I/O control units (ICUs). There can be up to two ICUs, each capable of handling up to two XMI busses. To communicate with the XMI bus, a JXDI (JBox XMI data interface) connects an XJA module to the ICUs. The JXDI has 16 data lines in each direction, and cycles every 16 ns. It has a total Bandwidth of 125 Mbytes/sec in both directions. Address, command, and data are time multiplexed onto these wires.

The JBox to ICU address interface is 2-bytes wide, and the complete address is transferred during two cycles.

XJA to ICU packet types are as follows:

- Status
- Interrupt
- DMA Read
- CPU Read Data Return
- DMA QW Write
- DMA OW Write
- DMA HW Write

ICU to XJA packet types are as follows:

- Status
- CPU Read
- DMA QW Read Return
- CPU Write
- DMA OW Read Return
- DMA HW Read Return

The ICU to XJA commands are:

- CPU Read Request
- CPU Write Request
- DMA Read Data Return
- DMA Read Lock Data Return
- DMA Read Locked Status
- DMA Read Error Status

The XJA to ICU commands are:

- DMA Read Request
- DMA Read Lock Request
- DMA Write Request
- DMA Write Unlock Request

- o CPU Read Data Return
- o CPU Read Error Status
- o CPU Write Complete
- o Interrupt Request

4.1.8 Interrupts

The EBox microcode handles interrupts. The central system arbiter (in the IRC0 MCA) (Figure 4-9) distributes I/O device interrupts, service processor unit interrupts, and CPU interrupts in a round robin fashion or broadcast.

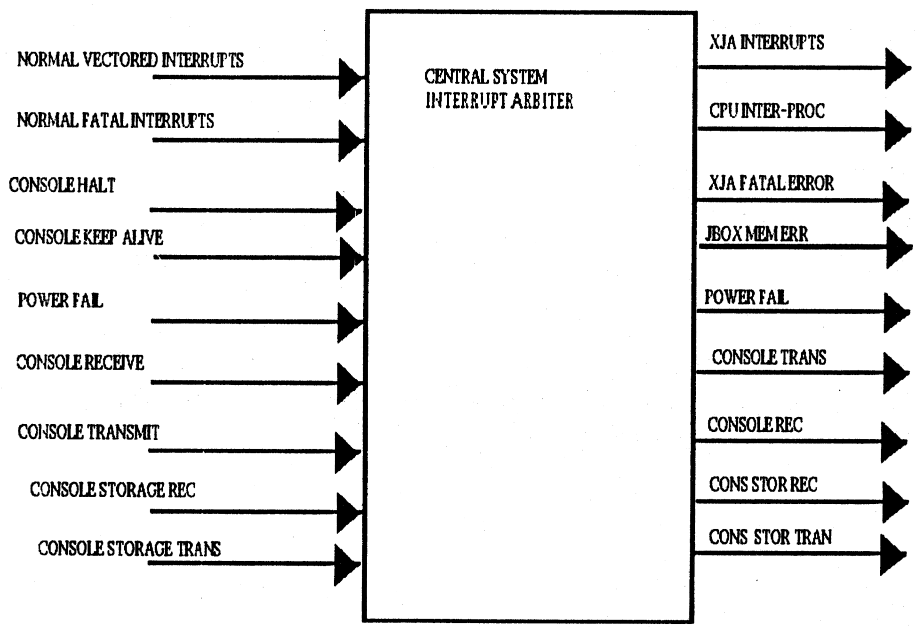


Figure 4-9 Central System Interrupt Arbiter

4.1.8.1 I/O Interrupts

Each XJA (XJA0, XJA1, XJA2, and XJA3) has five levels of interrupts IPL 14 thru 17, and IPL 1D. Each XJA assembles an interrupt packet (Figure 4-11) that is translated by IRC0. IRC0 determines which XJA generated an interrupt and translates it into different levels of interrupt. For an interrupt level of 14, 15, 16, or 17, the XJA assembles an interrupt packet. For an interrupt level of 1D, the XJA does not send a packet and asserts XJA_FATAL_ERROR (Figure 4-10).

The JBox sends out (through differential wires) a serial transmission that indicates that there is an interrupt and the level of the interrupt. The EBox branches on the interrupt bits to determine which interrupt is to be serviced.

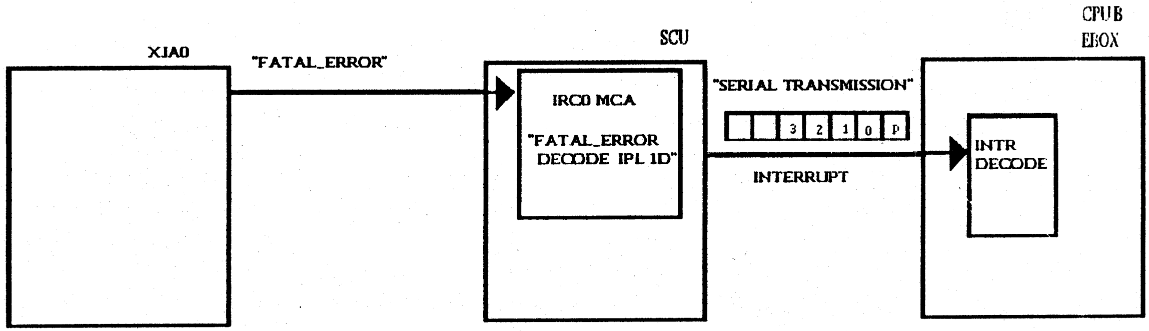


Figure 4–10 XJA Fatal Error

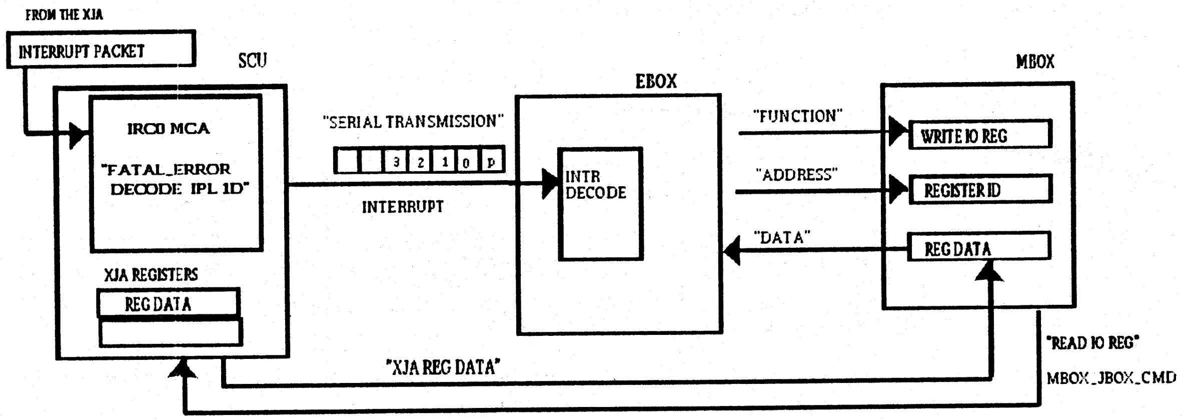


Figure 4–11 Interrupt Packet Transmission

4.1.8.2 SPU Interrupts

The SPU assembles and passes an interrupt packet to the JBox. SPU interrupts are shown in Figure 4–9. The JBox translates the packet and generates a CPU interrupt. Figure 4–12 illustrates the power fail interrupt path.

4.1.8.3 Inter-Processing Interrupts

Inter-processing interrupts occur in a multiprocessor environment. Two registers support inter-processing interrupts, CPU configuration register and the IP interrupt register. For example, if CPU A decides to interrupt CPU B, CPU A writes into the interprocessing register, and then the JBox generates inter-processing interrupts (Figure 4–13).

The CPU interval timer does not interrupt the JBox.

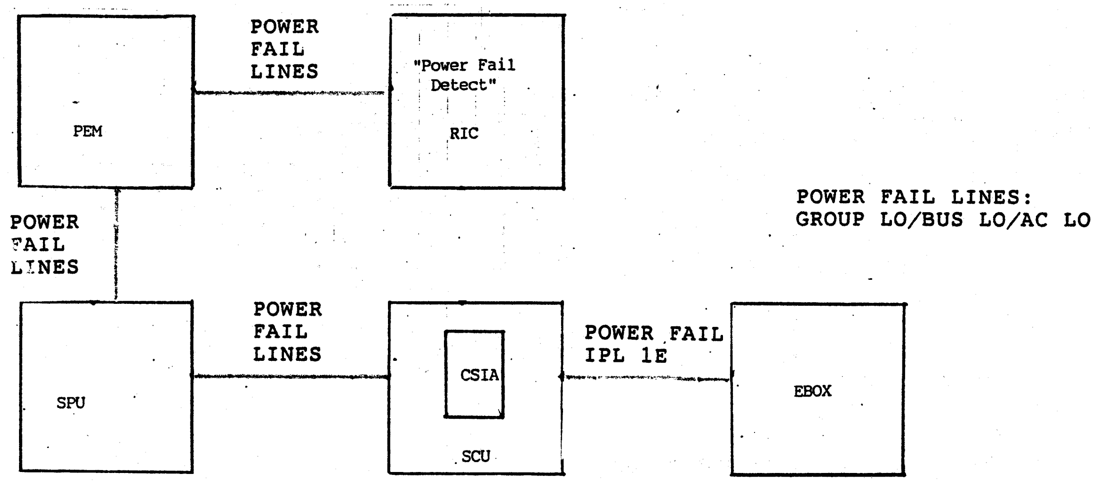


Figure 4-12 Power Fail Interrupt

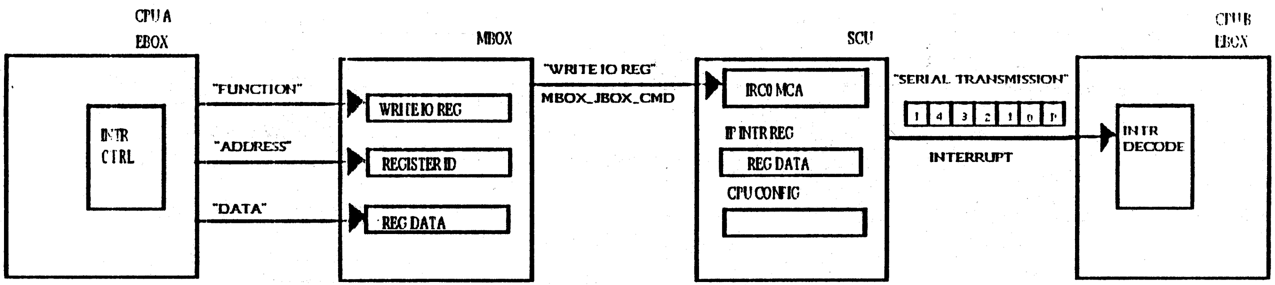


Figure 4-13 Inter-processor Interrupt

4.1.9 Interlocks

The JBox supports interlock read and write requests from the ports. An interlock inhibits access until the block is unlocked. The JBox tracks the interlock addresses by storing them in the 2K - 3K range of one of the TAG STRAMs. The size of the interlocked block is a cache block (64 bytes).

4.1.10 ACU Port

The ACU contains the main data path control and ECC logic for the memory subsystem. The data path is divided between two MDPs. The address interface has two components: multiplexed row address and column address, and memory port select, segment select, and bank select fields.

The JBox to memory interface can be broken into four categories:

o JBox to memory command

o Memory to JBox command

o Data movement

o DRAM address

The JBox communicates with memory by sending commands to the MMC that owns the segment of memory needed. This is done through the use of segment command buffers. Each MMC contains two command buffers.

Memory communicates with the JBox when any of the following conditions exist:

- o a read request was made and the data is ready to send
- o an error was detected during the transfer of read data
- o an error was detected during the transfer of write data
- o a command buffer is available

There are three types of memory to ACU interfaces, command, data, and row/column addressing. Data is divided between two MDPx (ACU). Row and column address is provided by the JBox.

Memory interfaces to the service processor unit with the MMCx. This interface initializes the memory and provides a way to switch the memory between three different timing modes, normal, step, and standby.

Data is transferred to and from the memory subsystem at a rate of one quadword per clock cycle. This transfer rate is maintained through the ACU and to and from the Main Memory Unit (MMU). The transfer size can be specified as 1, 2, 4, or 8 quadwords.

Associated with each quadword of write data are mask bits which define the write status of each byte of data. I/O can request data on quadword, octaword, hexword boundaries.

4.2 Memory Subsystem Functional Overview

The memory subsystem is a non-bussed, high Bandwidth system implemented in various gate array and board technologies. As shown in Figure 4–14 a memory subsystem is comprised of an ACU and a Main Memory Unit (MMU). The ACU, located on the SCU, contains the main control and ECC logic. The MMU is comprised of four memory modules which contain the Dynamic RAMs (DRAMs), and gate arrays used for logic level translation and data buffering.

The subsystem is block oriented; that is, all accesses to or from memory are either in 64-byte increments or a subset of that increment. The memory elements (drams) are arranged such that any access activates 64 bytes of DRAM content. The 64 bytes are distributed across four memory modules.

Each memory module consists of:

- o a Memory Array Card (MAC), which acts as a mother board, and
- o two, removable Daughter Array Cards (DAC).

A module contains 64 Mbytes using a 1 Mbit DRAM implementation. The minimum subsystem configuration is four memory modules or 256 Mbytes (one MMU). A second MMU may be added to provide a maximum memory size of 512 Mbytes. Memory size is increased by a factor of four with the implementation of 4 Mbit DRAMs. Figure 4–15 illustrates the physical MAC placement on the SCU planar module.

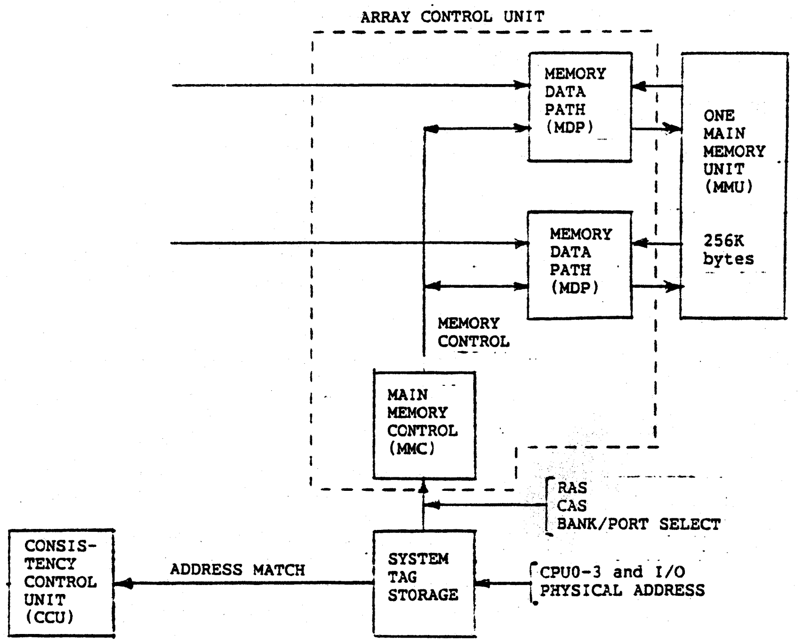


Figure 4–14 Basic Memory Subsystem Block Diagram

4.2.1 Maximum Configurations

A fully configured AQUARIUS memory will have the following parameters at a 16 nsec clock cycle.

- 4-way interleaving
- 512 Mbyte memory size (1 Mbit DRAM)
- 888 Mbyte/sec read/write Bandwidth
- 280 nsec read latency (from receive command to transmit data to the ACU)
- memory expansion support to 8 Gbyte

A fully configured ARIDUS memory will have the following parameters at a 24 nsec clock cycle.

- 4-way interleaving
- 512 Mbyte memory size (1 Mbit DRAM)
- 888 Mbyte/sec read/write Bandwidth (read only or write only Bandwidth limited to 666 Mbyte/sec by the transfer Bandwidth)
- 369 nsec read latency (from receive command to transmit data to the SCU)
- memory expansion support to 8 Gbyte

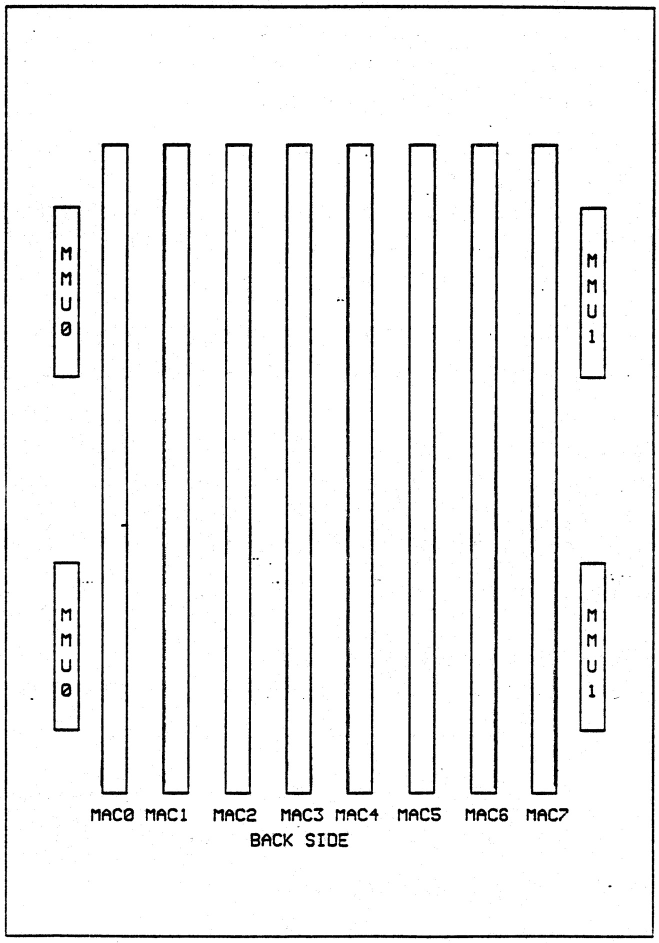


Figure 4–15 MAC Planar Module Placement

4.2.2 Operational Overview

All requests for memory are channeled through the ACU. The ACU monitors the subsystem Segment Command Buffer status bits to determine if a requested memory operation can be initiated. The memory subsystem has two segment command buffers. Each segment command buffer status bit in the memory system is associated with a memory segment. Once a command buffer has been loaded, the memory subsystem will execute it.

Provided the memory segment command buffer requested is free, the ACU will load the buffer with cycle type, starting quadword, number of quadwords, bank number, and index. The index is used to associate commands sent to memory with commands and addresses resident in the SCU.

On a write command the SCU will set up the data path from the sender through the data switch (crossbar) and to the selected MMU. At this point the SCU sets the memory segment command buffer status busy. This will stay busy until a segment command buffer available signal is received from the memory system.

The memory system has two segment command buffers. Once a segment command buffer has been loaded the memory system will execute it. During execution the index (and row/column select bit) is used (by the memory system) to obtain the row or column address from the JBox. In this way all memory addressing is handled by the JBox.

In the case of a read, the memory subsystem will access data from the DRAM and hold it in a read buffer. It then requests service to transfer the read data to the requester by transferring a READ READY command to the SCU. The index is also sent as part of the ready command, which is used by the SCU to determine the requester and its address.

4.2.2.1 Data Transfers

A write or read command to memory can specify 1,2,4 or 8 quadwords. Mask bits, which define the write status of the associated data, are transferred with each quadword of write data.

Read data is wrapped such that the starting quadword is specified. Wraps are different for I/O and CPU read requests. CPU wraps start with a quadword from 0-7, continues to 7, then wraps to 0, continuing until the quadword prior the starting quadword. I/O wraps follow the XMI bus specification. Wraps are done within octaword (16 byte) boundaries.

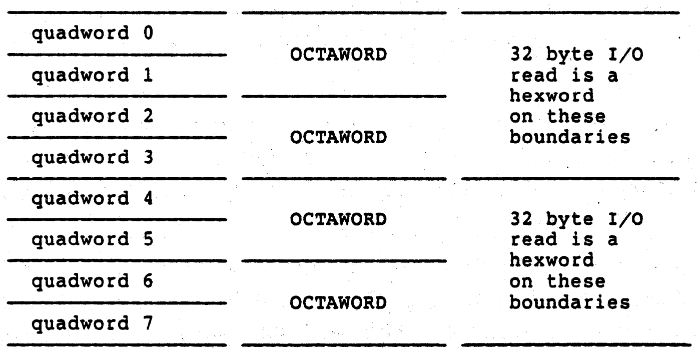
Table 4-9 summarizes the data transfer sizes and the associated mask bits.

Table 4-9 Data Transfer Size Summary

TRANSACTION	SIZE
CPU Read	8 quadwords
CPU Write	8 quadwords, 1 mask bit/longword
I/O Read	1, 2, 4, or 8 quadwords
I/O Masked Write	1 or 2 quadwords, 1 mask bit/byte
Non-masked I/O Write	4 or 8 quadwords, no mask bits, all bytes are valid

4.2.2.2 Wrap-on-Read Sequences

Only read data can be wrapped. As shown in Figure 4-16, the maximum wrapped read transfers are 64 bytes for a CPU and 32 bytes for I/O.



quadword 0	OCTAWORD	32 byte I/O read is a hexword on these boundaries
quadword 1		
quadword 2	OCTAWORD	
quadword 3		
quadword 4	OCTAWORD	32 byte I/O read is a hexword on these boundaries
quadword 5		
quadword 6	OCTAWORD	
quadword 7		

Figure 4-16 Wrapped Read Data

CLOCK CYCLES							
1st	2nd	3rd	4th	5th	6th	7th	8th
0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	0
2	3	4	5	6	7	0	1
3	4	5	6	7	0	1	2
4	5	6	7	0	1	2	3
5	6	7	0	1	2	3	4
6	7	0	1	2	3	4	5
7	0	1	2	3	4	5	6

Figure 4–17 CPU Wrap Sequence

Figure 4–17 illustrates a CPU hexword wrap sequence.

Figure 4–18 illustrates possible I/O wrap sequences for hexword 0 and hexword 1.

Possible I/O wrap sequences hexword 0

CLOCK CYCLES							
1st	2nd	3rd	4th	5th	6th	7th	8th
0	1						
1	0						
2	3						
3	2						
0	1	2	3				
1	0	3	2				
2	3	0	1				
3	2	1	0				

Possible I/O wrap sequences hexword 1

CLOCK CYCLES							
1st	2nd	3rd	4th	5th	6th	7th	8th
4	5						
5	4						
6	7						
7	6						
4	5	6	7				
5	4	7	6				
6	7	4	5				
7	6	5	4				

Figure 4–18 I/O Wrap Sequences(Hexword 0 & 1)

4.2.2.3 Clock System

During normal system operation the memory subsystem operates from the system clock. The memory module receives STRAM clocks and gated B clocks from the SCU for synchronizing data transfers. Each memory module contains a clock system used only for battery back up or single step/scan operation.

STRAM clocks are used to clock out read data from the 16 DRAM Data Path (DDP) MCAs that form the MMU data path. Each STRAM clock is routed to two DDPs such that eight STRAM clocks are required for each MMU. With a two-MMU configuration, 16 STRAM clocks are required. All STRAM clocks are programmed the same.

The gated B clock is derived on an MCA through a gated B clock macro. The output of the macro is the system B clock. The gated clocks are sent from the MMC to the input latches on the memory modules. The clock is used for all transfers from the SCU module to the memory modules. Data, commands, and addresses are transmitted with a B phase latch and received on the memory module with the falling edge of the gated B clock.

4.2.3 Hardware Overview

The ACU logic, located on the SCU module, contains the data path and control for the memory system. The data path portion is divided between two Memory Data Path (MDP) MCAs. Each MDP provides a 4-byte wide path in each direction. In addition, the MDPs provide error coding/decoding, byte merging, and parity generation/checking. The Main Memory Control (MMC) MCA and Memory Control DRAM (MCD) MCA provide control for the data path and memory modules. Note that DRAM cycle timing is controlled by the MCD, except during the two backup timing modes (i.e., Standby and Step).

The following paragraphs describe the major logic hardware and the associated functions.

Memory Module: A module consists of and MAC and two DACs. Overall the module provides DRAM data storage, read and write data buffering DRAM data integrity during power fail, and DRAM control timing during single step operation. The MAC provides one half of the memory module DRAM storage as well as the required gate arrays and MSI logic. The DACs each provide one quarter of the module DRAM storage.

Memory Data Path: The MDP MCA is located in the ACU and provides check bit generation for write data, and determines SBE/DBE on read data. In addition, it provides the byte merge path, and generates data patterns during Self Test.
Main Memory Control: The MMC MCA is located in the ACU and provides the command and control interface to the JBox, and data path control to the MMU. In addition, the MCA provides DRAM control commands to the MCD MCA on the MMU, as well as DRAM control and address commands to the DCA MCAs during step mode operation.
Memory Control DRAMs: The MCD MCA is located in the ACU and provides DRAM timing and control (except in Step Mode), and commands to the MMC MCA during Self Test.
DRAM Data Path: The DDP in located on the MMU and provides read and write data path buffering, level translation, and the DRAM bypass path.
DRAM Control and Address: The DCA MCA is located on the MMU contains the CAS mask registers, and provides level translation. In addition, it provides DRAM control during Step Mode, EEPROM control, and buffer control signals to the DDP MCAs.

4.2.4 SCU to Memory Interface

The SCU to Memory interface can be broken into various descriptions. This breakdown is as follows:

- JBox to Memory Command
- Memory to JBox Command
- Data Movement
- DRAM Address

From the memory perspective this interface is duplicated twice, once for each ACU.

4.2.4.1 SCU to Memory Command Interface

The SCU communicates with the memory by sending commands to the MMC that owns the segment of memory needed. This is done through the use of segment command buffers. Each MMC contains two command buffers. Table 4–10 provides a general description of buffer fields.

Table 4–10 Buffer Field Descriptions

FIELD NAME	DESCRIPTION
CTLPAR	Parity, odd parity across all bits, valid on every clock cycle
BANKADDR	Bank Address, bank 0 or 1 and segment command buffer select
CMD	Command, memory operation to perform
INDEX	Index, used to locate address in SCU
LDCMD	Load Command, indicates the CMD, BANKADDR, INDEX, and CTLPAR are valid.
BUFAVAIL	Buffer Available, indicates that the SCU can accept commands.
SENDDATA	Send Data, memory can transmit read data.
CYCLESTAT	Cycle Status, specifies whether a request should be canceled.
LENGTH	Indicates number of quadwords in transfer, decoded as:1, 2, 4, 8 quadwords.

Once the MMC receives a command as indicated by the LDCMD bit it will load the command into the appropriate segment command buffer by decoding the msb of the bankaddr bits. When the MMC is ready to accept another command it will assert its BUFAVAIL signal for one clock cycle.

4.2.4.2 Memory to SCU Command Interface

The memory communicates with the SCU under two conditions:

o a read request was made and the data is ready to send

o an error was detected during the transfer of read data

Table 4–11 provides a summary description of the status information transferred to the SCU.

Table 4–11 Memory Status Information

FIELD NAME	DESCRIPTION
CMD	Command, if 0 = return read data; if 1 = error data
SEGMENT	segment number.
LDCMD	Load Command, used by the SCU to load command and segment.
BUFAVAIL	Buffer Available, indicates the availability of each segment command buffer.
WRITEOK	Write OK, indicates that memory received data from the SCU without a parity error.
READOK	Read OK, indicates that no ECC errors occurred during transfer of read data through the MDP.
CTLPAR	Parity, odd parity across all bits, valid on every clock cycle.

The MMC will send the command to the JBox and wait for a SENDDATA signal. The data movement protocol described next illustrates the timing between SENDDATA and the movement of data.

4.2.4.3 Data Movement Protocol

Figure 4–19 illustrates the timing relationship between commands and the movement of data. The DSCT is the data switch controller located in the SCU.

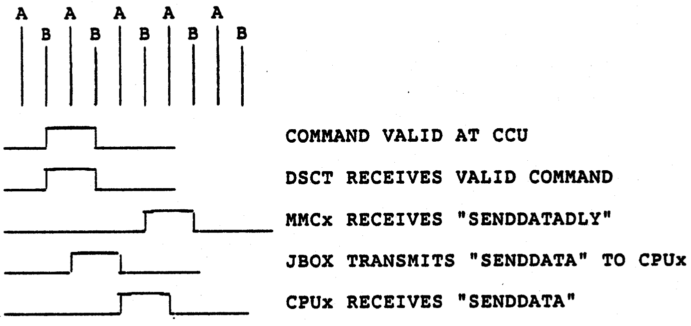

A
B
A
B
A
B
A
B
A
B
COMMAND VALID AT CCU
DSCT RECEIVES VALID COMMAND
MMCx RECEIVES "SENDDATADLY"
JBOX TRANSMITS "SENDDATA" TO CPUx
CPUx RECEIVES "SENDDATA"

Figure 4–19 Tranfer Timing Relationships

The objective of the above timing is to have a fixed number of cycles between the DSCT receipt of a data transfer command to the time data gets to the data switch.

The JBox transmits SENDDATADLY to the MMC delayed by one cycle to take into account the cycle it takes for SENDDATA to traverse the JBox to CPU cable.

The MMC delays SENDDATADLY received from the Box by another cycle to take into account the cycle it takes for data to traverse the CPU to Box cable.

4.2.4.4 Address Interface

The MMC receives starting quadword information from the Box at the same time it receives the command information. The MMC receives the starting quadword address and its parity, and parity on address bits <33:03>

The SCU stores all DRAM row and column addresses for all memory operations. The MMC provides control signals to the SCU for transmitting the row or column address at the proper time within the DRAM cycle.

4.2.5 Memory Subsystem Organization

the following subsections describe the general subsystem organization and interleaving configurations.

4.2.5.1 Dynamic RAM Organization

An MMU contains all DRAMs associated with a single memory subsystem. The DRAMs are distributed across four memory modules to support a single data path between the MMU and the ACU. When there is an access to memory 160 drams on each memory module will be activated or a total of 640 drams. This represents 64 bytes of ECC encoded memory.

The MMU has two segments. Each segment has two banks. Control is such that the two segments operate independently but share a common data path. The data lines are common; address lines are not common between segments. This organization permits two way interleaving allowing access to both segments simultaneously.

For each segment there are two banks. Only one bank may be active. This is controlled by the RAS signal. And, although the WE and CAS signals are common between the two banks they will have effect only on the bank with RAS active.

There are eight unique CAS lines per segment. The CAS line is used to determine whether a write operation will be permitted to occur.

4.2.5.2 Gate Array Organization

The MMC MCA receives the command information from the JBox. After decoding the information the MMC passes DRAMs commands to the MCD MCA. In addition, the MMC has control of the entire data path. The MCD provides all control timing for the DRAMs.

During write operations data is passed to the two MDPs. Each MDP will operate on a longword providing ECC check bits on write data. Data is then passed onto the MMU where it is stored in a buffer (Write Buffer 0).

During read operations data is received by the MDPs from the MMU. The MDP will decode the 7 check bits, correcting single bit errors and detecting double bit errors. Data is then passed on to the JBox.

The MMU contains all of the required data buffering. The data buffers can store 128 bytes of read or write data in two, 64-byte write buffers. On write operations data passes through the MDP and is stored in the MMU. Once the transfer is complete data can be written to the DRAMs. On read operations a block of data is read into the MMU read buffer. The data is transferred out eight bytes at a time through the MDP's until the entire data block has been transferred.

Memory address decoding is handled by the JBox. An index is passed with every command sent to the MMC. If the requested memory segment is ready, the MMC transmits the index and a row/column select bit to the ADRx MCAs. The ADRx MCAs use the index to select the row/col address. The address is applied directly to the MMU.

4.2.5.3 Interleaved Operation

There can be two memory subsystems connected to the SCU. Each subsystem is comprised of an ACU and a MMU. The two subsystems are completely independent. If the DRAMs are the same type in both systems, the two subsystems can have alternating addresses, and in effect be interleaved. In addition each memory subsystem contains two segments that can be interleaved. Table 4–12 illustrates interleaving possibilities with the memory subsystems.

Table 4–12 Interleaving Efficiency

MEM SYS 0	MEM SYS 1	MAX INTERLEAVE
256 Mb	256 Mb	4-way
256 Mb	1 Gb	2-way
1 Gb	1 Gb	4-way
1 Gb	4 Gb	2-way
4 Gb	4 Gb	4-way

Optimum memory performance is achieved when the size of both memory subsystems are equal. Where the two subsystems are of different sizes the interleaving must be two-way for each subsystem, with each occupying either the upper or lower portion of memory.

4.2.5.4 MMU Organization

Each 8-byte transfer to the MMU will be distributed among 4 memory modules. The data path for each longword is between one MDP and two memory modules. Therefore 4 bytes of data, 7 ECC check bits and a mark bit are divided between two memory modules. Figure 4–20 illustrates the data partitioning between MDP and memory modules. The figure also illustrates how each quadword transfer is divided and stored. For example, on a write operation MM0 (memory module 0) will contain 160 data bits corresponding to 20 bits from each of eight quadwords.

4.2.5.5 Memory Module Data Organization

As shown in Figure 4–21 there are four buffers distributed across four DDP gate arrays (one on each memory module). Two buffers store write data (MACWR BUF 0 and MACWR BUF 1). The other two buffers store read data (MACRD BUF O and MACRD BUF 1). Each of these buffers store 160 bits of data corresponding to a maximum of eight data transfers between the ACU and the MMU.

The upper portion of the figure illustrates the DRAM bit assignment on a single memory module. The figure also shows that the first four words are stored on the two DAC modules (each DAC module receiving 10 bits of the word). In addition, the output of each DDP is divided between a DAC and the MAC. Note that in this context a word contains 20 bits.

	MDP1		MDP0	
	MM3	MM2	MM1	MM0
each quadword	word	word	word	word
	79 60	59 40	39 20	19 0
quadword 0	longword 1		longword 0	
quadword 1	longword 3		longword 2	
quadword 2	longword 5		longword 4	
quadword 3	longword 7		longword 6	
quadword 4	longword 9		longword 8	
quadword 5	longword 11		longword 10	
quadword 6	longword 13		longword 12	
quadword 7	longword 15		longword 14	

```
Memory Module 0  bits 19:0
  - 16 data bits
  -  4 check bits

Memory Module 1 bits 39:20
  - 16 data bits
  -  3 check bits
  -  1 mark bit

Memory Module 2  bits 59:40
  - 16 data bits
  -  4 check bits

Memory Module 3 bits 79:60
  - 16 data bits
  -  3 check bits
  -  1 mark bit
```

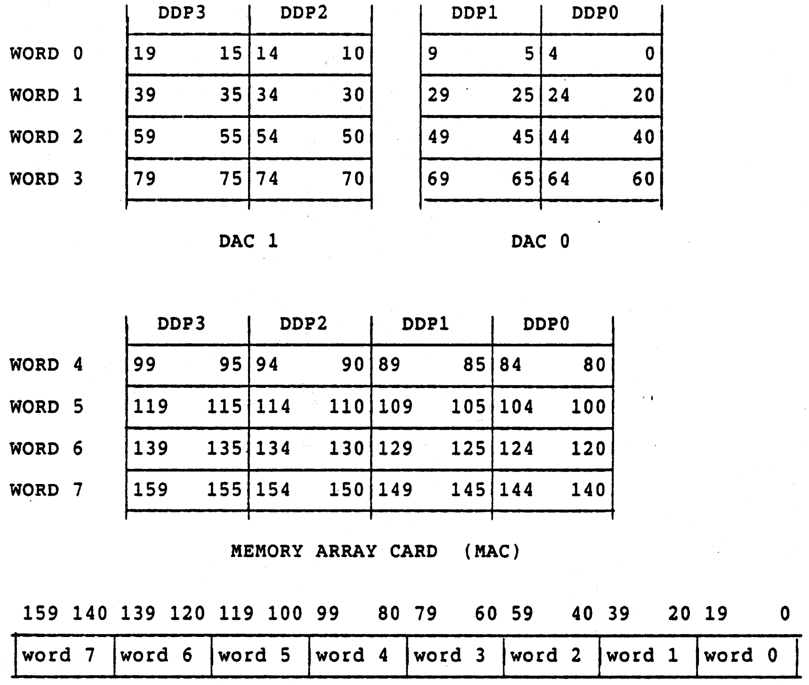
Figure 4–20 MMU Organization

4.2.6 Memory Operation

The memory subsystem stores data on block boundaries of 64 bytes. All accesses to memory activate 64 bytes of DRAMs and the bits bits required for error correction support. The memory subsystem performs the following operations:

- read data
- write data
- read-modify-write data
- write-read data
- write-pass data
- refresh

There are specific requirements within the above operations. For example, read operations must provide data wrapped on quadword boundaries. Write data operation must support a minimum write of four bytes. This corresponds to the minimum write that a CPU will require.



	DDP3	DDP2	DDP1	DDP0
WORD 0	19 15	14 10	9 5	4 0
WORD 1	39 35	34 30	29 25	24 20
WORD 2	59 55	54 50	49 45	44 40
WORD 3	79 75	74 70	69 65	64 60
	DAC 1		DAC 0	

	DDP3	DDP2	DDP1	DDP0
WORD 4	99 95	94 90	89 85	84 80
WORD 5	119 115	114 110	109 105	104 100
WORD 6	139 135	134 130	129 125	124 120
WORD 7	159 155	154 150	149 145	144 140

MEMORY ARRAY CARD (MAC)

159 140	139 120	119 100	99 80	79 60	59 40	39 20	19 0
word 7	word 6	word 5	word 4	word 3	word 2	word 1	word 0

Figure 4–21 Module Data Organization

The read-modify-write operation (I/O only) must support a minimum write of one byte. The write-pass operation must receive data from one CPU and transmit to a another CPU wrapped on quadword boundaries, while also storing the data in memory. The write-read operation must receive data from one CPU, store valid longwords into memory, read the entire data block from memory and then transmit wrapped data to a another CPU.

4.2.6.1 Read Data Operation Summary

Memory operations are in 64 byte quantities. When a read data cycle executes 640 DRAMs will be read and loaded into a read buffer (READ BUFFER 0). Data in READ BUFFER 0 is transferred to READ BUFFER 1. This allows a second read operation from the other segment to continue..

READ BUFFER 1 drives an 8:1 mutiplexer used to wrap data out 80 bits per clock cycle. The first 80-bit word to be transferred is determined by the Starting Quadword field in the command buffer. The number of subsequent 80-bit words to transfer are determined by the Number of Quadwords field in the command buffer.

The transfer of the 80-bit data word is from four memory modules to two MDPs. Each MDP operates on 40 bits, performing single/double bit error detection and single bit error correction as required. Each MDP transmits a longword per clock cycle to the JBox through the control/command interface between the ACU and the MMC. The MMC completes the operation by updating the segment command buffer availability status to the SCU.

4.2.6.2 Write Data Operation Summary

Initially the JBox commands to the selected segment command buffer. The memory subsystem has the capability of writing from 1 to 16 longwords. Data is transferred from the ACU in 8-byte wide (quadword) increments. In the ACU this data path is split into two, 4-byte (longword) paths. Each 4-byte path feeds into an MDP.

In the MDP MCA the check bit generate logic appends a 7-bit ECC code to the 32 bit data path. A 40th bit (mark bit) is added that is used in conjunction with the ECC code. The ACU will provide 80 bits of data to the MMU.

The 80 bits of data will be transmitted to the MMU. At the MMU data is loaded into a write buffer (WRITE BUFFER 0) through a 1:8 demux. Demux selection starts at 000 (binary) and increments to a number determined by the number of quadwords specified by the command information received from the JBox.

During the transfer of data the MMC will set the CAS Mask Register bits using the two CAS mask control signals. For each valid longword in WRITE BUFFER 1 a CAS mask bit is set. The CAS mask bit, if set, will allow CAS to be applied to that set of 40 DRAMs which will enable writing.

The MMU under MMC and MCD control completes the DRAM timing sequence, and updates the data and segment command buffer status for the JBox

4.2.6.3 Read-Modify-Write

A read-modify-write data operation starts as a write command to the memory subsystem. The MMC MCA decodes the index to determine if the read operation is to be for I/O or CPU. Since an I/O write can be a byte write the MMC will examine the mask bits and determine whether a byte write is to be written. If so, the MMC must execute a read-modify-write cycle.

If a byte write is requested, the MMC reads the block from memory to retrieve sufficient data for check bit generation. During the initial stages of the read operation the I/O write data is transferred through the WRITE BUFFER 0, WRITE BUFFER 1, through the DRAM bypass path, into the READ BUFFER 0, and subsequently into READ BUFFER 1.

From READ BUFFER 1 the data is transferred through the wrap multiplexer and into the I/O merge buffer in the MDPs. The I/O merge buffer holds the I/O write data until the required bytes to generate ECC check bits can be read from the DRAMs.

When data is ready at the DRAMs it is loaded into READ BUFFER 0, then to READ BUFFER 1. One or two quadwords will be transferred from READ BUFFER 1 through the wrap multiplexer into the MDP for byte merging with the I/O write data. The I/O write data bytes are combined with the necessary read data bytes so that new check bits can be generated. Data is now ready to be loaded into WRITE BUFFER 0.

The MMU under control of MMC and MCD transfers the write data to WRITE BUFFER 1. All valid longwords in WRITE BUFFER 1 are written to the DRAMs. The MMC and MCD MCAs complete DRAM timing sequence, and update the data and segment command buffer status for the JBox.

4.2.6.4 Write-Read Data

A write-read data is a mixed mode operation. Data is first written to a location, and then read out from the same location. During the write cycle of the operation not all DRAMs are written. Invalidated longwords (from a CPU) are not written. During the read phase of the operation all DRAMs are read.

Under MMC control, the MDPs receive data one quadword per clock cycle from the MMU. It checks for errors (and corrects single bits), and generates longword parity. Data is transferred from the MDPs to the JBox. The MMC MCA completes the operation by updating the segment command buffer status.

4.2.6.5 Write-Pass Data

A write-pass data operation uses the same write timing as a normal write cycle. During this operation the DRAM bypass path is used to pass data directly to the DRAM read buffers immediately after the DRAM write buffers are loaded. During the write cycle all data is valid and all DRAMs are written. The data located in the DRAM read buffers is unloaded the same as in a read operation.

The MDPs under MMC control receives data one quadword per clock cycle from the MMU. It checks for errors, correcting single bit errors as required, and generates word parity. The data is then transferred from the MDPs to the JBox. The MMC completes the operation, updating the segment command buffer status.

4.2.6.6 Refresh Data Operation

Refresh operations are initiated approximately every 12 microseconds. A refresh flag originates at each of the memory modules. The MMC receives one of these signals and uses it as a command request for refresh. On completion of active cycles, the MMC will execute a refresh cycle to all DRAMs in the MMU. This cycle does not involve the data or address path but does use the RAS and CAS control signals.

4.2.6.7 Refresh Types

There are three different refresh types; MMC, DCA and Standby. MMC refreshes are used during normal system operation. When the system enters step mode operation DCA refreshes are used. During power fail or scan operation Standby refreshes are used.

4.2.7 Memory Timing

The memory subsystem can operate in one of three modes: Normal, Step and Standby. The system clocks cannot be stopped while the memory is in Normal mode. While in Step mode the system clocks can run normally or burst. Standby mode is used when the output of the MMC will not be stable. This occurs during scan operation and power fail.

4.2.7.1 Step Mode Operation

STEP mode is enabled by the SPU whenever normal system clocks will be stopped. During normal system operation the MMC provides all DRAM control signals. When stopping the clocks is necessary, control of the DRAM is provided by the DCA in STEP mode. During this mode DRAMs are controlled by the DCA on each memory module rather than being controlled by the MMC.

The MMC uses the DRAM control lines (RAS, CAS, WE) as command lines to each of the DCAs. Thus each memory module will be working independently and consequently asynchronously. Note that all four memory modules will always receive the same command.

In this mode the RAS lines serve to select the bank of DRAMs and start the command cycle. In addition, all RAS lines are logically ORed together to provide a load signal to the DCA Command buffer. The CAS and WE lines carry a 4 bit command to the DCA which is stored in the DCA Command buffer.

4.2.7.2 Standby Operation

During scan operation or during system power loss the memory modules go into standby operation. Standby operation maintains refreshes to the DRAMs, ensuring that the contents of the memory are not compromised.

Standby operation is initiated by the SPU as part of the power down sequence. Standby can be entered only after the SPU has switched the memory system into STEP mode. After each DCA has entered step mode the SPU sends a standby enable function to the memory modules. The memory modules switch to Standby operation if no DRAM cycles are in progress. Otherwise, Standby is be entered following DRAM cycle completion.

After power is restored, the memory modules remain in Standby until after initialization. Each memory module is initialized by the SPU with an initialization signal. As part of the initialization the SPU will set conditions such that the MMC will be in STEP mode.

After scan operations to the MMC are complete, the SPU negates the standby enable function to the memory modules. Following an acknowledgment function each memory module is then in step mode. Exiting STEP mode is initiated by the SPU to enable normal memory operation.

4.2.8 Error Strategy

Error Strategy is broken down into detection, reporting and recovery. Error detection on the command, address and data transfers is performed using of parity. Error detection on data from the memory modules is performed using ECC. The following error types are detected:

- write data errors
- read data errors
- control errors
- address errors
- protocol errors.

4.2.8.1 Write Data Error

A write data error is defined as an error that occurs during the transmission of the data from the JBox to a memory module. Detection of this error is done by the MDP MCA using odd parity. Write data errors are reported by the MMC to the JBox. In addition, any data received with bad parity is marked as bad data before being written to memory.

4.2.8.2 Read Data Error

A read data error is a SBE (single bit) or MBE (multiple bit) detected on a longword of data read from the MMU. Detection of this error is done by the MDP which contains error correction code logic. In the case of correctable errors (SBE) MMU data is corrected and passed to the destination (through the JBox) with odd parity. For uncorrectable errors (MBE), data is passed to the destination as is but with bad (even) parity.

4.2.8.3 Control Errors

Control errors are broken down into two classifications:

- MMC to memory module errors, and
- MMC to MDP errors.

The MMC provides odd parity on all control signals to the MDPs and checks parity on DRAM control signals sent to the memory modules. Whenever parity errors are detected on the control signals, the MMC will flag the JBox that a fatal control error has occurred.

4.2.8.4 Address Errors

Two types of address errors are detected:

- longword address errors, and
- row/col address errors.

Longword address errors are handled by the MDP including address parity in the ECC check bit generation. The address used to generate address parity is defined down to the longword. This ensures that if data is written to the wrong data buffer (on the memory module DDPs) an ECC error will be detected on reading. The syndrome can be used to distinguish between an address parity error, data error or check bit error.

The row/col address errors are handled by the JBox providing odd parity with the row/column address sent to the memory module. Each memory module calculates parity and if it is correct toggles an acknowledgment signal to the MMC.

4.2.8.5 Protocol Errors

The following protocol errors are detected:

- MMC receives Beginning of Data (BOD) without receiving a write command from the JBox.
- MMC does not receive a BOD after receiving a write command.
- Timeout before receiving cancel/ok status.

5
I/O Subsystem Description

5.1 Chapter Objective

The chapter objective is to introduce and provide a functional overview of the I/O Subsystem hardware components. Included is a summary of typical I/O configurations, configuration rules, and supported (and potentially supported) devices, adapters, and controllers. All I/O specifications were used as resource material.

5.2 Subsystem Introduction

As shown in Figure 5-1, the I/O subsystem is defined as:

- o XMI Bus - system I/O bus
- o XJA adapter
- o JXDI - interface between the I/O ports of the ICU and the XJA

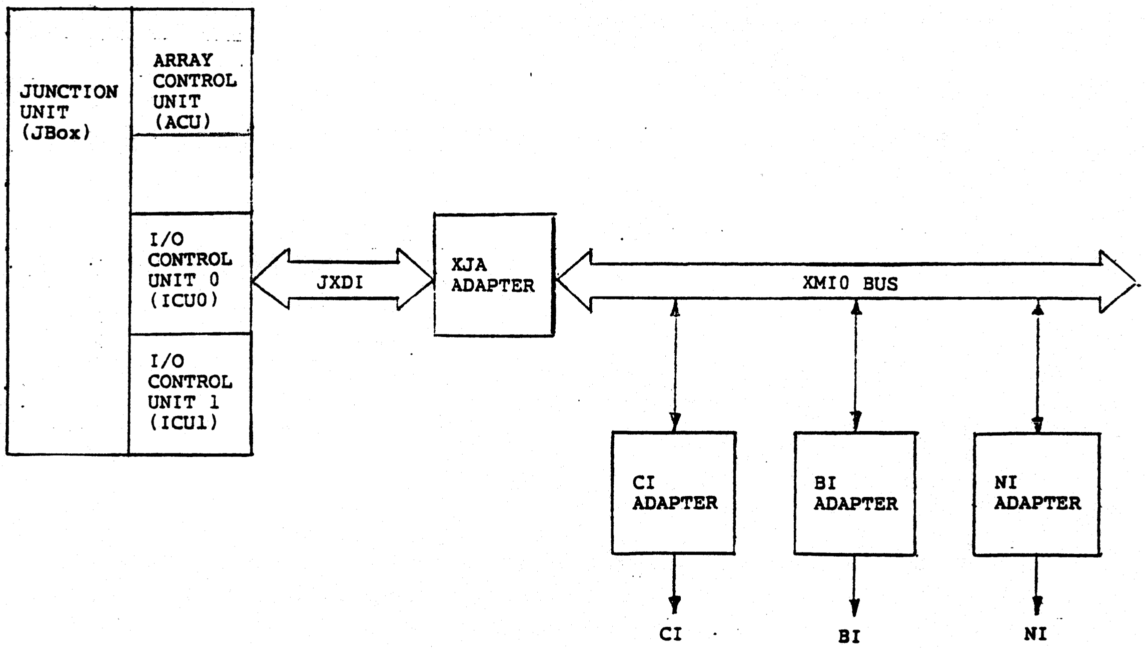


Figure 5–1 Basic I/O Subsystem Block Diagram

The XMI to SCU Adapter (XJA) together with the ICU and JXDI interface provide the data and control paths between the CPU and I/O devices over the XMI bus.

5.2.1 XMI Overview

The XMI bus consists of the bus protocol, backplane (including the card cage), and the logic to implement the protocol. The XMI is a limited length, pended, synchronous bus with centralized arbitration. Several transactions can be in progress at a given time, allowing efficient use of bus bandwidth. Arbitration and data transfers occur simultaneously, with multiplexed data and address lines.

The bus supports quadword, octaword, and hexword read and write operations to memory. In addition, the bus supports longword read and write operations to I/O space. These longword operations implement byte and word modes required by certain I/O devices.

The XMI operates on a 64ns bus cycle, with a raw bandwidth of 125 Mbyte/Second. The usable bandwidth, which depends on transaction length, is specified in Table 5-1.

Table 5–1 XMI Bandwidth

OPERATION	BANDWIDTH
Longword Read	31.25 Mbytes/sec
Quadword Read	62.5 Mbytes/sec
Octaword Read	83.3 Mbytes/sec
Hexword Read	100.0 Mbytes/sec
Longword Write	31.25 Mbytes/sec
Quadword Write	62.5 Mbytes/sec
Octaword Write	83.3 Mbytes/sec
Hexword Write	100.0 Mbytes/sec

The XMI uses the same connector and module technology as used on the VAXBI bus. Similar to the VAXBI bus, all XMI interfaces use a set of custom components with predefined etch to interface to the bus. AQUARIUS will use only the 14-slot XMI card cage.

Individual sets of three clocks are distributed radially to each XMI node from a central source on the backplane (i.e., Clock/Arbiter Module). The clock signals provide the clock waveforms to the node-specific logic and the required control lines.

5.2.2 XJA Overview

The XJA is implemented on an extended T-series module to conform to the XMI specifications. The XJA is implemented primarily in a single LSI Sea-of-Gates CMOS gate array, and three AMCC Q3500 series bipolar gate arrays. The standard XMI chip set (XLATCH - DC530, and XCLOCK - DC531) is used to interface to the XMI.

The XJA module resides in the XMI card cage and provides the physical connection to the XMI. Two TBD pin connectors on the module front edge provide the physical connection to the JXDI.

The XJA communicates with the ICU and the XMI through the following transaction types:

- DMA Transactions - XMI transactions that select the XJA as the responder node will be forwarded to the ICU as DMA type transactions. DMA transactions can be reads, writes, read locks, or write unlocks, and can be quadword, octaword or hexword in length.
- CPU Transactions - AQUARIUS CPUs can access the I/O portion of VAX physical address space through CPU type transactions. These transactions are received by the XJA from the JXDI and forwarded to the XMI with the XJA as the commander.
- Interrupt Transactions - The XJA will field interrupt transactions from the XMI and forward them to the ICU using interrupt type transactions. The resulting SCB offset vector fetch initiated by an AQUARIUS CPU will be a CPU type transaction.

5.2.3 JXDI Overview

The SCU to XJA data interconnect (JXDI) is the logical definition of the 12-foot cable connecting the SCU and XJA. All JXDI signals are unidirectional, series-terminated ECL 100K levels, differential. The JXDI is protected by parity and implements a retry mechanism.

The method of data transfer is asynchronous with clocks supplied by the transmitter at a cycle time equal to the nominal CPU cycle time (i.e. 16 nsec). In general, the JXDI is symmetrical in that the same data and control signals are sent and received by the ICU and XJA.

5.2.4 SCU Overview

The SCU is the logical interface between the CPU, main memory, and I/O subsystem. The ICU portion of the SCU interfaces to XJAs through the JXDI cable, interfaces to the SPU, and implements the central system interrupt arbiter.

From the perspective of an AQUARIUS CPU or an XJA, the SCU resembles a memory controller. Since AQUARIUS CPU's implement write-back caches, the SCU must insure cache data consistency. This consistency is insured by using a duplicate cache tag store for each CPU.

5.2.5 System Physical Address Space

Figure 5-2 illustrates the division of AQUARIUS physical address space into memory space and I/O space.

All AQUARIUS memory will reside in the main memory system. XMI memory modules will not be visible from the perspective of an AQUARIUS CPU. XMI memory will only be visible from that particular XMI with a corresponding address's hole in the AQUARIUS main memory not accessible from that XMI. The address hole will be equal in size to the amount of memory residing on the XMI and defined by the memory module starting address register.

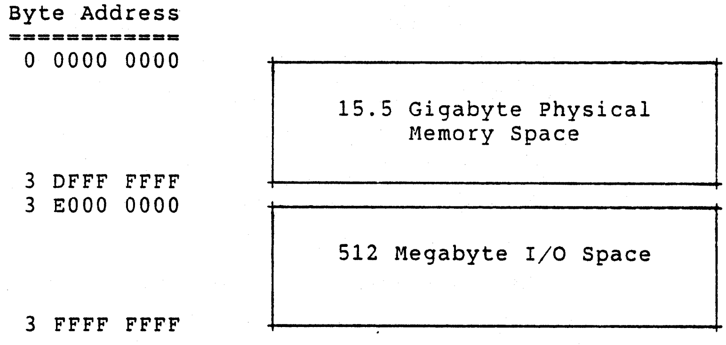


Figure 5-2 AQUARIUS Physical Address Space

5.2.6 XMI Addressing

The XMI has a 40-bit physical address space. When the MSB of an XMI address bit <39> is asserted, I/O space is assumed and address bit <38:29> are ignored. This yields a 512 Mbyte I/O address space and a 512 Gbyte memory space. The XMI data lines (XMI_D_L) during an XMI command address cycle are mapped to address bits as follows:

XMI_D_L[28:0] -> XMI_ADDRESS_L[28:0]
XMI_D_L[57:48] -> XMI_ADDRESS_L[38:29]
XMI_D_L[29] -> XMI_ADDRESS_L[39]

The XMI address bits are mapped to AQUARIUS address bits in the XJA as follows:

XMI_ADDRESS[33:0] -> AQUA_ADDRESS[33:0]

This mapping allows XMI devices to have access to the entire AQUARIUS memory space. Addresses received by the XJA from the XMI with XMI_ADDRESS[39] clear and XMI_ADDRESS[38:34] nonzero, will not be accepted (i.e., NO ACK). In addition, the XJA Configuration Register allows the size and starting address of AQUARIUS memory to be set at any 64 Mbyte boundary.

Addresses received by the XJA from the XMI with XMI_ADDRESS<39:34> zero, and XMI_ADDRESS<33:29> = 1F#16 will also be receive a NO ACK since AQUARIUS does not implement memory in the last 512 Mbytes of the 34-bit physical address space.

5.2.7 XMI I/O Space

The XMI specifies that when XMI_ADDRESS<39> is set, XMI_ADDRESS<38:28> are ignored and the address is assumed to be in the 512 Mbyte XMI I/O space. The I/O address space is divided into XMI Private Space, XMI Node Space, and XBI Window Space.

XMI transactions that reference I/O space will not be passed on by the XJA to the ICU. This prevents an XMI from accessing other XMI I/O space. Figure 5-3 illustrates the division of XMI I/O space as seen from the perspective of an XMI node in an AQUARIUS system.

```
Byte Address
============

80 0000 0000    +---------------------------------+
                |     XMI Private Space           |  24 Mbytes
80 0180 0000    +---------------------------------+
                |     XMI Node Space              |  16 x 512 Kbytes
80 0200 0000    +---------------------------------+
                |     XBI1 Window Space           |  32 Mbytes
80 0400 0000    +---------------------------------+
                |     XBI2 Window Space           |  32 Mbytes
80 0600 0000    +---------------------------------+
                |     XBI3 Window Space           |  32 Mbytes
80 0800 0000    +---------------------------------+
                |     XBI4 Window Space           |  32 Mbytes
80 0A00 0000    +---------------------------------+
                |          Reserved               |  192 Mbytes
80 1600 0000    +---------------------------------+
                |     XBI5 Window Space           |  32 Mbytes
80 1800 0000    +---------------------------------+
                |     XBI6 Window Space           |  32 Mbytes
80 1A00 0000    +---------------------------------+
                |     XBI7 Window Space           |  32 Mbytes
80 1C00 0000    +---------------------------------+
                |     XBI8 Window Space           |  32 Mbytes
80 1E00 0000    +---------------------------------+
```

Figure 5-3 XMI Node Space Address Allocation

5.2.8 XMI Private Space

XMI Private Space is a 24-MByte address region containing the reset address as required by the uVAX architectural subset. References to XMI private space will be serviced by resources local to a node, such as local device CSRs and boot ROM, and will not be broadcast on the XMI.

The XJA will not use any of its allocated XMI node Private Space. All XJA XMI required registers and XJA specific public registers are located in XMI Node Space and are accessible from the XMI. XJA specific private registers are located in the reserved space of AQUARIUS I/O space.

5.2.9 XBI Window Space

XBI Window Space consists of eight 32 Mbyte address regions used for XMI to BI transaction windowing. Longword length references directed to an XBI Window Space will be re-issued on the appropriate BI.

XMI transactions are translated into the corresponding VAXBI transaction. XMI devices can only access I/O space on a VAXBI (e.g. VAXBI memory space locations are not accessible from the XMI).

5.2.10 AQUARIUS I/O Space

The I/O space as seen from the perspective of an AQUARIUS CPU is illustrated in in Figure 5-4. The partitioning allows support of multiple XMIs. Note that XMI Private Space is omitted from the map since it is only visible locally to a specific node and used only by uVax processor nodes.

Byte Address		
3 E000 0000	XMI0 Node Space	16 x 512 Kbytes
3 E080 0000	XMI1 Node Space	16 x 512 Kbytes
3 E100 0000	XMI2 Node Space	16 x 512 Kbytes
3 E180 0000	XMI3 Node Space	16 x 512 Kbytes
3 E200 0000	XBI0 Window Space	32 Mbytes
3 E400 0000	XBI1 Window Space	32 Mbytes
3 E600 0000	XBI2 Window Space	32 Mbytes
3 E800 0000	XBI3 Window Space	32 Mbytes
3 EA00 0000	XBI4 Window Space	32 Mbytes
3 EC00 0000	XBI5 Window Space	32 Mbytes
3 EE00 0000	XBI6 Window Space	32 Mbytes
3 F000 0000	XBI7 Window Space	32 Mbytes
3 F200 0000	XBI8 Window Space	32 Mbytes
3 F400 0000	XBI9 Window Space	32 Mbytes
3 F600 0000	XBIA Window Space	32 Mbytes
3 F800 0000	XBIB Window Space	32 Mbytes
3 FA00 0000	XBIC Window Space	32 Mbytes
3 FC00 0000	XBID Window Space	32 Mbytes
3 FE00 0000	XJA0 Private Space	512 Kbytes
3 FE08 0000	XJA1 Private Space	512 Kbytes
3 FE10 0000	XJA2 Private Space	512 Kbytes
3 FE18 0000	XJA3 Private Space	512 Kbytes
3 FE20 0000	Jbox/SPU Register Space	~~24~~ 30 Mbyte
3 FFFF FFFF		

Figure 5-4 AQUARIUS I/O Address Space

5.2.11 JBox/CSL Register Map

The final 30 Mbytes of I/O address space is dedicated to AQUARIUS specific registers that control the JBox and allow the AQUARIUS CPU's to communicate with the SPU. Figure 5-5 illustrates the JBox specific registers that reside physically in the IRC MCA of the JBox.

SPU-resident registers are described in the SPU Subsystem Description. XJA private space registers are summarized in the XJA description.

Byte Address	31 0
3 FE20 0000	JBOX_INTR_CTRL
3 FE20 0004	JBOX_IPINTR_CTRL
3 FE20 0008	JBOX_ERR_SUMMARY
3 FE20 000C	Reserved
3 FE2F FFFF	

Figure 5–5 JBox Register Map

5.2.12 I/O Space Configuration

The I/O space allocation allows for a maximum of 14 BI's on an AQUARIUS system. The exact physical location of a given XBI at a given XMI node address is defined through the contents of the PAMM (Physical Address Memory Map) STRAM structure. The structure is located in the TAG MCU on the SCU module.

This 1K deep STRAM structure is loaded by the SPU at system initialization time. Each STRAM location defines where a given 512K byte block of I/O space resides.

5.3 XMI Bus Description

5.3.1 XMI Definitions

The terms described in Table 5-2 are XMI-specific and are used to describe XMI transactions, data quantities, and transfer types.

Table 5-2 XMI Term Definitions

Node	A node is a hardware device which connects to the XMI backplane. The largest XMI system configuration will support 14 nodes.
Transfer	A transfer is the smallest quantum of work which occurs on the XMI. Typical examples of transfers are: the command cycle of a read, the command and following data cycles of a write.
Transaction	A transaction is a logical task that is being performed and is composed of one or more transfers (e.g. read). For a read operation the transaction consists of a command transfer followed some time later by a return data transfer.
Commander	The commander is the node that initiated the transaction in progress. In any write transaction, the commander is the node that requested the write; for reads, the commander is requesting the data. A commander in a transaction holds for the duration of the transaction. However, in some cases it may appear that the commander changes. For example, a commander initiates a read transaction. It is the responder (data source) that initiates the return data transfer, but the node that requested the data is still the commander.
Responder	The responder is the complement to the commander in a transaction.
Transmitter	A transmitter is the node that is sourcing the information on the bus. For example, the commander is the transmitter during the command transfer, and the receiver during the return data transfer.
Receiver	The complement to the transmitter, it is the receiver of data being moved during a transfer.
Naturally Aligned	Refers to a data quantity whose address could be specified as an offset, from the beginning of memory, of an integral number of data elements of the same size. In naturally aligned data the low-order address bits are zero. All XMI reads and writes transfer a naturally aligned block of data.
Wraparound Read	Defined as an octaword or hexword read operation where read data is returned in a specific pattern in which the specifically addressed quadword is returned first, independent of alignment. The remaining data in the naturally aligned block of data containing the addressed quadword is returned in subsequent transfers. See Examples 1 and 2.

Example 1:

Read octaword, byte address 00000018 (hex)

00000018 First Quadword
00000010 Second Quadword

Example 2:

For purposes of defining the wrapping order for hexword reads, a hexword read is decomposed into two octaword reads, with the addressed octaword read data returned first. Within each of the octawords the wrapping order is the same as described above for octawords. Return data for the second octaword maintains the same wrapping order used in the first octaword.

Read hexword, byte address 00000018 (hex).

00000018 First Quadword
00000010 Second Quadword = First Octaword

00000008 Third Quadword
00000000 Fourth Quadword = Second Octaword

The XMI protocol requires that all octaword and hexword reads, both normal and interlocked, be wrapped.

5.3.2 Bus Arbitration

The XMI protocol can architecturally support up to 16 nodes. However, actual XMI implementations will support 14 nodes. At a given time, any or all of the nodes may request use of the XMI. Arbitration cycles occur in parallel with data transfer cycles using a set of lines dedicated to bus arbitration.

When a node requires ownership of the bus, it asserts one of its two request lines; XMI CMD (commander) REQ L or XMI RES (responder) REQ L (each node has a dedicated pair of request lines that are connected to the central arbiter). The XMI CMD REQ L line is used by nodes to initiate XMI transactions (i.e. act as a commander) while the XMI RES REQ L is used by nodes to return data to a commander (i.e. act as a responder). The XMI arbiter maintains two independent round-robin queues; one for each of the request types. The responder requests are given higher priority than commander requests.

During any given cycle, all nodes have the opportunity to request the bus. The arbiter receives all the requests and decides which node shall be granted the bus. In the next cycle the selected node begins its transfer.

The XMI has two additional arbitration control signals: XMI HOLD L and XMI SUP (suppress) L. Assertion of XMI HOLD L guarantees that the current XMI transmitter will be granted ownership of the bus in the next cycle, independent of the value of any other outstanding requests. The XMI HOLD L signal is used for multi-cycle transfers, allowing the current transmitter to acquire consecutive cycles.

If a node can not maintain bus traffic (e.g., a CPU backing-up on cache invalidate operations due to XMI writes, or a memory queue becoming full), that node can temporarily suppress the start of additional XMI transactions. The node suppresses additional cycles by asserting the XMI SUP (suppress) L signal. XMI SUP L blocks all commander requests but allows responder requests to continue to be serviced.

The XMI arbitration scheme consists of three priority classes: Hold, Responder Arbitration, and Commander Arbitration. Hold has the highest priority and guarantees that the current transmitter will be granted the bus in the next cycle. The next priority class is responder requests, followed by the commander requests. Within the responder and commander classes, priority is distributed in a round-robin manner.

5.3.3 Bus Integrity

The XMI Bus contains a number of features to enhance the integrity and reliability of the bus. All bus information transfer lines are parity protected, and bus confirmation signals are ECC protected. The XMI bus protocol permits detection and recovery of all single-bit error conditions on these signals. In addition the XMI defines timeout conditions that may be used to detect and diagnose failures.

5.3.4 Node Identification

Each node that interfaces to the XMI bus has an identification number (ID = 1 to 14). The node ID is provided by the node's XMI NODE ID<3:0> H lines which are hardwired in the backplane such that the physical slot number equals the node ID.

5.3.5 XMI Signal Line Descriptions

The XMI signal lines are separated into five groups:

1. Arbitration - The XMI arbitration request signals are a pair of dedicated lines from each node to the central arbiter used to request access to the XMI. When a node wishes to gain access to the XMI, it asserts the appropriate request line.
2. Information - The use of this field is multiplexed between command and data information. On data cycles the lines represent 64 bits of read or write data. On command cycles the lines represent command code, address, and mask information.
3. Response - This group of signals lines are used by the receiver to notify the transmitter of data transfer status.
4. Control - This group of signal lines provide the timing, power status, reset, and error and error status.
5. Miscellaneous - These lines provide the node identification field and spare XMI lines.

Table 5-3 describes each XMI signal line.

Table 5-3 XMI Signal Line Descriptions

XMI LINE	LINE DESCRIPTION
ARBITRATION LINES	
XMI CMD REQ[n]	The Commander Request lines are used by XMI commanders to initiate new transactions. Requests are asserted by each node at the start of each XMI cycle, indicating a request for the next cycle. In general, after making a request a node may not negate that request until XMI GRANT has been issued.
XMI RES REQ[n]	The Request Response line is used by responders to service Read or Interrupt Acknowledge (IDENT) transactions. The responder requests have higher priority than the commander requests.
XMI GRANT[n]	The GRANT lines are a set of dedicated lines from the central arbiter to each node used to indicate when a node has been granted the bus. XMI GRANT[n] L is asserted by the arbiter at the end of each XMI cycle, indicating the node to be granted the bus during the next cycle.
XMI HOLD L	The Hold line is a wired-OR signal used to implement multi-cycle transfers. The XMI node currently granted the bus may assert its XMI HOLD L line to guarantee that it will be granted the bus in the next cycle, independent of request priority.
XMI SUP L	The Suppress line is a wire-ORed signal used to control the initiation of new XMI transactions. Assertion of XMI SUP blocks all commander requests, suppressing the initiation of new XMI transactions.

Table 5-3 (Cont.) XMI Signal Line Descriptions

XMI LINE	LINE DESCRIPTION
ARBITRATION LINES	
	XMI SUP L may be asserted by any node at the start of each XMI cycle to control arbitration for the cycle after the next XMI cycle.
INFORMATION	
XMI D<63:00> L	The Data lines are multiplexed between data and command information. The lines transfer 64 bits of read or write data on data cycles, and command, address, and mask information on command cycles. Figure 5-5 specifies all command cycle fields.
XMI F<3:0> L	The Function lines encode the function being performed on the bus in the current cycle. See Figure 5-5 for field encodings.
XMI ID<5:0> L	During the command cycle and return data cycles, the ID field contains the commander's ID. The ID is used to identify the source of the request on the command cycle, and to associate returning data with the commander that issued the request on return data cycles. An individual XMI Commander ID can have only one outstanding transaction at any time. Each XMI node is allocated four commander IDs, enabling it to have up to four transactions in progress at any given time. Fixed ID codes are required for the identification of interrupt sources and to provide for XMI to VAXBI address translation. See Figure 5-5 for field encodings.
XMI P<2:0> L	The three parity bits protect the XMI D, XMI F, and XMI ID fields. XMI P<2> is computed over XMI F<3:0> and XMI ID<5:0>. XMI P<1> and XMI P<0> are computed over XMI D<63:32> L and XMI D<31:00> L, respectively. Even parity is used, where the Exclusive OR of all bits including the parity bit is a zero.
RESPONSE	
XMI CNF<2:0> L	The three confirmation lines are used by the receiver to notify the transmitter of the status of the data transfer. The coding provides single-bit error detection and correction, double-bit errors are not detected. See Figure 5-5 for field encodings.

Table 5–3 (Cont.) XMI Signal Line Descriptions

XMI LINE	LINE DESCRIPTION
CONTROL	
XMI TIME[n] H	The XMI TIME[n] H lines are a set of 15 clock signals that provide a time reference for XMI nodes. The TIME H signal is a 46.9Mhz square-wave.
XMI TIME[n] L	These lines are the complement of XMI TIME[n] H, and are provided so that the clock decoder (XCLOCK) can use the same polarity edge for all generated clocks.
XMI PHASE[n] H	This signal is a pulse that windows the edges of XMI TIME[n] H and XMI TIME[n] L that represent the beginning of the XMI cycle.
XMI AC LO L	XMI AC LO L is asserted when the line voltage is below minimum specifications.
XMI DC LO L	XMI DC LO L warns of the impending loss of DC power and is used for initialization on power restoration. A node uses XMI DC LO L to force its circuitry into an initialized state.
XMI RESET L	XMI RESET L is asserted by nodes that need to initialize the system to the power-up state. XMI RESET L is received by the Clock/Arbiter module and passed to a reset module. After assertion of XMI RESET L, the reset module sequences PRIM AC LO L and PRIM DC LO L as in the case of a true power-down/power-up sequence. The Clock/Arbiter module buffers and synchronizes these signals and drives XMI AC LO L and XMI DC LO L.
XMI BAD L	XMI BAD L is asserted by a node until it passes its selftest. On power-up XMI BAD L is negated only when all nodes have passed self-test.
XMI FAULT L	XMI FAULT L is a Wire-ORed signal used to indicate that a node has detected an unrecoverable error and results in a system-wide machine check. To signal a fault condition, a node asserts XMI FAULT L for one full XMI cycle, starting at the beginning of the cycle. Nodes receiving XMI FAULT L will latch the signal at the end of the XMI cycle.
XMI DEFA H	This default signal is driven by the arbiter during idle XMI cycles to default the bus. XMI DEFA H is routed to the XCLOCK's DEFAULT H pin in slot 1 of the XMI card cage.
XMI DEFB H	Same as XMI DEFA H except this signal is driven to slot 14 in the 14-slot of the XMI card cage.
XMI ERR DEF H	This signal is used to assure that the XMI is properly configured for defaulting the bus. It connects to the XCLOCK DEFAULT H pins in slots 2 through 13. A backplane pull-up resistor asserts this line unless a module is plugged into slot 1 or 14 (XMI modules tie the XMI ERR DEF H pin to ground). XMI ERR DEF H is received by the arbiter and while asserted the arbiter will not grant access to the XMI.

Table 5–3 (Cont.) XMI Signal Line Descriptions

XMI LINE	LINE DESCRIPTION
MISCELLANEOUS	
XMI NODE ID<3:0> H	Each slot on the XMI backplane is wired with a unique four-bit ID code. This code will be used by each node to define their commander IDs and CSR addresses. XMI NODE ID<3:0> H corresponds to bits XMI ID<5:2> of a node's commander IDs.
XMI SPARE0 L	The XMI SPARE 0 L signal is a wired-OR line reserved for DIGITAL use.
XMI SPARE1 L	Identical to SPARE 0

5.3.6 XMI Signal Field Descriptions

Figure 5-6 illustrates all XMI signal fields and the field encodings.

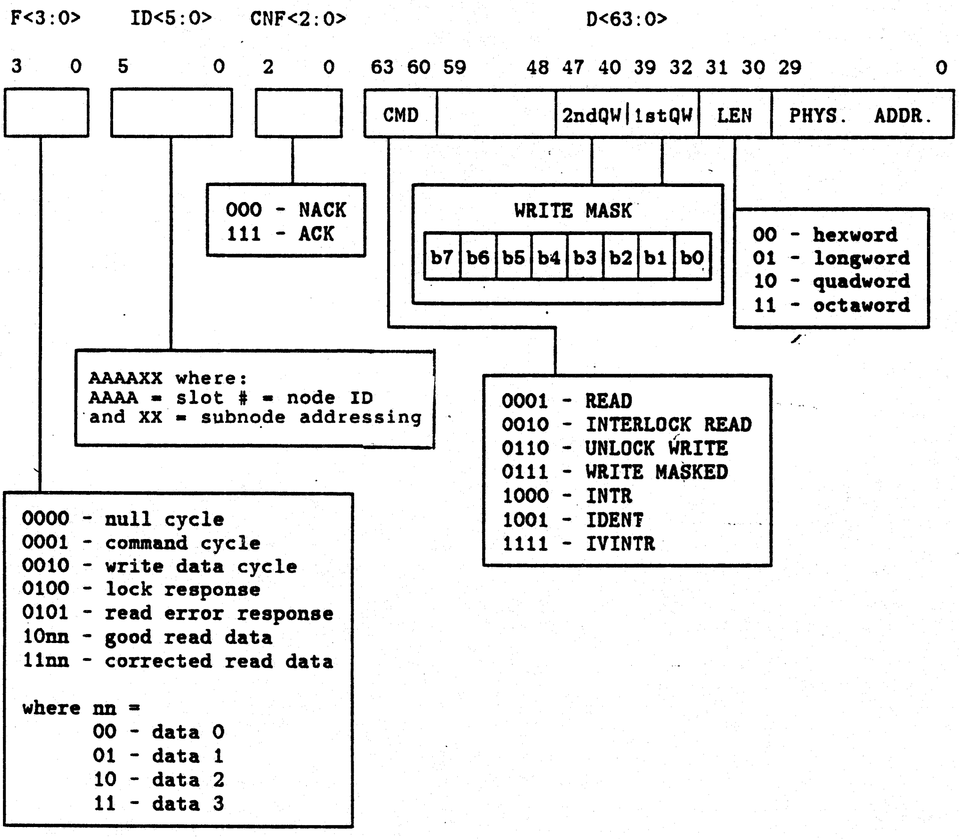


Figure 5–6 XMI Field Descriptions

5.3.7 Command Field Descriptions

The command field specifies the type of transaction to be executed. The following subsections describe each of the transactions.

Table 5-4 specifies the supported XMI transactions, and the associated address space. Table 5-5 summarizes the command field codes.

Table 5–4 XMI Transaction Types

TRANSACTION	DATA TYPE	ADDRESS SPACE
Read	Longword	I/O space
Read	Quadword	Memory space
Read	Octaword	Memory space
Read	Hexword	Memory space
Interlock Read	Longword	I/O space
Interlock Read	Quadword	Memory space
Interlock Read	Octaword	Memory space
Interlock Read	Hexword	Memory space
Write Masked	Longword	I/O space
Write Masked	Quadword	Memory space
Write Masked	Hexaword	Memory space
Write Masked	Octaword	Memory space
Unlock Write	Longword	I/O space
Unlock Write	Quadword	Memory space
Unlock Write	Octaword	Memory space

Table 5–5 XMI Command Summary

TRANSACTION	MNEMONIC	COMMAND CODE
Read	READ	0 0 0 1
Interlocked Read	IREAD	0 0 1 0
Unlock Write Masked	UWMASK	0 1 1 0
Write Masked*	WMASK	0 1 1 1
Interrupt	INTR	1 0 0 0
Identify	IDENT	1 0 0 1
Implied Vector	IVINTR	1 1 1 1

*Mask is ignored on hexaword write operation to memory space

5.3.7.1 Read Transaction

Read transactions (0001) are used to move a longword, quadword, octaword, or hexword of data from a responder to a commander. The data is naturally aligned and wrapped. A read transaction is initiated by the commander driving the XMI address and function lines to represent a longword, quadword, octaword, or hexword read transaction.

The read command cycle is decoded by the nodes on the bus. The node which decodes the address, latches the address and command. This device is the responder. Some time later, when the responder has the requested data, it initiates a return data transfer.

Multiple transfers may be necessary to transfer all quadwords in a given octaword or hexword transaction. The commander, which has been monitoring the bus traffic waiting for its return data, latches the information. The commander issues its own ID in the ID field during the command cycle. The responder returns the same ID with the return read data to allow the commander to identify the return read data it requested.

5.3.7.2 Interlock Read Transaction

The Interlock Read transaction (0010) is used to access to a shared data structure in memory. It is identical to the read transaction, but with additional functionality. The effect of an interlocked transaction depends on the state of the interlock bit in the memory. If the memory is already locked, it responds to this read request with a Locked Response and no data is returned. This signifies to the commander that the shared memory structure is not available.

However, if the memory is not locked receipt of this request locks the memory to further interlocked read requests to the referenced location. The interlocked request provides the data contained in the addressed location(s) to the commander. A corresponding transaction Unlock Write is required to remove the memory lock.

5.3.7.3 Unlock Write Transaction

The Unlock Write transaction (0110) is the complement to the Interlock Read, and is used to relinquish a lock on a shared data structure in memory. When the Interlock Read has completed the requested access, it relinquishes the lock with an Unlock Write. Should an Unlock Write transaction be directed to a location not currently locked, the responder will perform the write operation.

5.3.7.4 Write Masked Transaction

The Write Masked transaction (0111) moves a pattern of bytes from a commander to a responder for insertion into a longword, quadword, or octaword. The longword, quadword, or octaword block is naturally aligned.

The commander gains the XMI and sends a command cycle specifying a Longword, Quadword, or an Octaword Write Masked command, a byte mask, and the address. It immediately follows this with one or two cycles of write data. All nodes on the XMI decode the address. The node decoding the address becomes the responder. The responder accepts the command, address and data and performs the requested write.

The mask field accompanying the command and address is unrestricted. Each bit in the 16-bit mask field corresponds to a byte of data in the associated one or two quadwords. If a mask bit is set to 1, the associated byte is written; if set to 0, the byte is not written.

Write Mask transactions are guaranteed to be masked in XMI memory space (ADR<29> = 0). A Write Masked transaction directed to an I/O space (ADR<29> = 1) location may be implemented as a full longword write. The support of maskable writes in I/O space is node implementation specific.

5.3.7.5 Interrupt Transaction Types

The XMI supports the following types of interrupt transactions:

- o Interrupt Request (INTR - 1000)
- o Interrupt Acknowledge (IDENT - 1001)
- o Implied Vector Interrupt (IVINTR - 1111)

The INTR and IDENT transactions are used to implement device interrupts. An I/O node will issue an INTR transaction to the XJA in order to interrupt a processor at a specified IPL. In response to the INTR a node will issue an IDENT transaction directed to the interrupting I/O node soliciting an interrupt vector.

The IVINTR transaction is used to implement single cycle interrupt transactions where the interrupt priority and the interrupt vector value are implied by bits in the interrupt type field. The IVINTR transaction is used to implement:

- o interprocessor interrupts IPL = 14H, vector = 80H
- o Write Error interrupts IPL = 1DH, vector = 60H.

Since the value of the interrupt vector is indicated by the value of the IPL field, IVINTR transactions do not require a corresponding interrupt acknowledge cycle.

5.3.7.6 Invalidate Operations

All cache-resident nodes on the XMI are required to monitor write traffic and perform cache invalidates if the XMI write compares with a block stored in cache. The XMI also has the concept of a cache invalidate transaction that does not result in the update of main memory. A commander can perform an invalidate operation by issuing a Quadword Write Mask or Octaword Write Mask command with the mask field equal to all zeros. The size of the region to be invalidated is specified in the length field.

An invalidate operation maintains all Write Masked transaction requirements, including supplying the appropriate write data cycles consistent with the transaction length. Since the write data will be discarded by the responder, the value of XMI D<63:0> during these data cycles is unspecified. However, the value of XMI D<63:0> must be consistent with XMI P<1:0>. Invalidate operations are not allowed in I/O space, since I/O space devices may implement masked writes as full longword writes.

5.3.8 Mask Field

The mask field is located in D<47:32> during the Command cycle. It is used to supply byte-level mask information for the XMI write masked and unlock write transactions. The correspondence between bits in the mask and bytes of the data during a write transaction is shown in Figure 5-7.

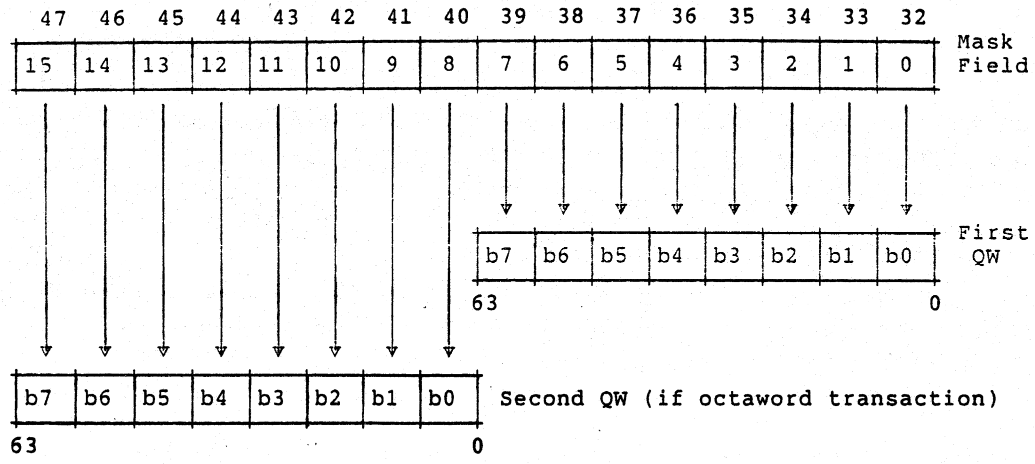


Figure 5–7 Mask Field Layout

During read transactions this field is a don't care status. Commanders can drive any data pattern in this field, and Responders must not depend on a certain defined pattern (such as all zeros). Note that correct parity must still be generated over this field even though its contents are a don't care status.

The maximum length of a write transaction is an octaword, which requires 16 mask bits in the upper longword of the command. Mask bits which are ones specify that the corresponding bytes of the following quadwords are to be replaced with the analogous byte of the data. Mask bits which are zero disable modification of the corresponding bytes in memory.

For Longword and Quadword length writes, the unused mask bits (D<47:36> and D<47:40>, respectively) are unspecified and are ignored by responders (other than to check parity). For all non-write commands, all bits in the mask field must be zero.

5.3.9 Length Field

The Length field, located in D<31:30>, is used to define the number of words in the XMI data transfer. The Length field is encoded as follows:

- 00 - hexword
- 01 - longword
- 10 - quadword
- 11 - octaword

Longword length transactions may only be used in I/O space. Quadword, Octaword and Hexword transactions may only be used in Memory space. Hexword length may only be used for read or interlock read transactions.

5.3.10 Address Field

The low-order 30 bits of the command cycle, D<29:00>, define the address of an XMI read or write transaction. The number of significant bits in the address depends on the transaction type and length. The low-order four bits are significant address bits or don't care, depending on the function being requested.

For longword length transactions, ADR<1:0> are significant for a VAXBI word-mode or byte-mode transaction in I/O space (ADR <1> for word mode; ADR<1:0> for byte mode. Quadword and Octaword write transfers are assumed to be naturally aligned, allowing the low-order address bits to be don't care status.

In the case of reads the situation is different since the memory does wraparound reads. Even though the operand size would indicate that some number of lower bits can be don't care, they are significant since all wrapped reads need to identify the quadword to be transferred first. On longword reads to I/O space ADR<1> is significant since there is no explicit READ WORD function on the XMI.

When a read is directed to a word-oriented device ADR<1> becomes significant since it specifies which word is to be read from that device. The relationship between the high and low words, the state of ADR<1> and the data bits is:

- ADR<1> = 1 = high word = D<31:16>
- ADR<1> = 0 = low word = D<15:00>

In the case of a longword oriented device ADR<1> is ignored as an address bit, and a full longword of data is returned for a read operation.

5.3.11 Node Specifier Field

The Node Specifier Field is located in D<15:0> during Command cycle of an interrupt transaction (INTR, IDENT, IVINTR) and is used to specify the source or destination of an interrupt. There is a direct correlation between the field bit position and the node ID. That is:

- D<15> = Node 15
- D<14> = Node 14
-
-
- D<1> = Node 1

o D<0> = Node 0

5.3.12 Function Field Description

The following subsections describe each of the functions encoded in this field.

5.3.12.1 Null Cycle

A function code of 0000 indicates a null cycle (i.e., an unused XMI cycle) which is ignored by all XMI Responders. During this cycle the values of D<63:00> and XMI ID<5:0> are unspecified. However, the value of XMI P<2:0> should be consistent with the values of DATA and ID to avoid unnecessary logging of bus errors. The default value of XMI P<2:0> will produce correct parity when DATA and ID are not driven.

5.3.12.2 Command Cycle

The command field, located in D<63:60>, defines the specific transaction being initiated by the Command cycle. A function code of 0001 identifies an XMI Command cycle. The XMI Command cycle is used by a Commander to initiate an XMI transaction. During this cycle the Commander drives its Commander ID on XMI ID<5:0> and drives command information on D<63:00>.

5.3.12.3 Write Data Cycle

A function code of 0010 identifies an XMI Write Data Cycle. Write Data Cycles immediately follow the XMI Command cycle during an XMI write transfer. During this cycle the Commander drives its Command ID on XMI ID<5:0> and drives write data on D<63:00>. The full 64 bits of data are used during quadword length or larger writes. For longword length writes, only the lower longword, D<31:00>, is used. In this case the value of the upper longword is unspecified. In either case the full 64 bits of data are used when checking XMI P<2:0>.

5.3.12.4 Locked Response Cycle

The Locked Response is used to indicate that the location specified in an Interlock Read transaction was already locked. During this cycle the Responder drives 0100 on XMI F<3:0> and the Commander's ID on XMI ID<5:0>. The value of the data bits, D<63:00>, are unspecified; however, they must contain a value consistent with XMI P<2:0>. A Locked Response signals the termination of an Interlock Read transaction. When issued, it is always the first and only read response to the transaction.

5.3.12.5 Read Error Response Cycle

The Read Error Response indicates that a Read, Interlock Read or IDENT transaction completed unsuccessfully due to an error condition at the responder node. The Read Error Response may be used for an uncorrectable memory error or a reference to non-existent location on VAXBI. During this cycle the Responder drives 0101 on XMI F<3:0> and the Commander's ID on XMI ID<5:0>. The value of the data bits, D<63:00>, are unspecified; however, they must contain a value consistent with XMI P<2:0>. A Read Error Response signals the termination of the transaction, and thus no further read responses are provided.

5.3.12.6 Read Data Response Cycles

Function codes 1000 - 1111 are used to identify return data in response to a READ, INTERLOCK READ or IDENT transaction. The Good Read Data response (GRDn, codes 1000 - 1011) indicates that the quadword of data is error-free. The Corrected Read Data response (CRDn, codes 1100 - 1111) indicates that the corresponding quadword of data stored in memory contained a single-bit error which was successfully corrected using ECC prior to shipment on the XMI. Both types of read data responses contain a sequence ID located in XMI F<1:0> which is used to identify when a read data cycle has been lost due to an XMI parity error.

During a read data response cycle, the Responder drives the Commander's ID on XMI ID<5:0> and read data on D<63:00>. The full 64 bits of data are used during quadword and octaword length reads. For longword length reads, only the lower longword, D<31:00>, is used. In this case the value of the upper longword is unspecified. In either case the full 64 bits of data are used when checking XMI P<2:0>.

5.3.13 XMI Support Components

Several semicustom components have been developed to support the XMI bus. The following subsections provide an overview of the components.

5.3.13.1 XCLOCK and XLATCH Chips

The primary interface to the XMI is provided by two DIGITAL-designed CMOS1 standard-cell components: XCLOCK and XLATCH. An XCLOCK and seven XLATCHes reside in the XMI Corner of each XMI node.

The XCLOCK is a 44-pin CMOS standard-cell chip that contains the logic to decode the signals required to control the XLATCHes. Included is a set of clocks for use by the node-specific logic . The XCLOCK component also drives and receives the XMI CNF<2:0> L signals and XMI HOLD L.

The XLATCH is a 44-pin CMOS standard-cell chip that contains the logic required to interface the XMI bus to a multiplexed bus within an XMI node. Each XLATCH part provides 11 tristate and 1 open drain transceivers. Seven XLATCHes are used on each XMI module. The XLATCH components drive and receive data from the XMI and controlled by signals from the XCLOCK.

5.3.13.2 Clock/Arbiter Module

Clock generation and arbitration among XMI nodes is provided by the XMI Clock/Arbiter module. The module is a daughter board consisting of custom chips and is mounted in the center of the XMI backplane. The XMI arbiter and clock generator chip (XARB) is implemented in a CMOS gate array. The module functions as the XMI's central arbiter and also provides two polarities of a 46.9 MHz clock signal, and a 15.6 MHz phase clock signal. (See Figure 5-8.)

Transactions on the XMI take place in discrete cycles delineated by the central clocks. The XMI cycle is nominally 64 ns, which corresponds to 15.6 MHz. Fifteen sets of CMOS clock signals (each consisting of an XMI TIME H and XMI TIME L) are generated by the central clock system (clock/arbiter module). Fourteen sets are used by the XMI nodes, and the fifteenth is used by the arbiter.

All clocks in the system are generated off edges of the XMI TIME H and XMI TIME L waveforms. To minimize clock skew, only falling edges of TIME H and L are used. XMI TIME H and XMI TIME L are complementary square-wave signals operating at 46.9 MHz (that is, 3 x the 15.6 MHz XMI frequency from a CMOS crystal oscillator.

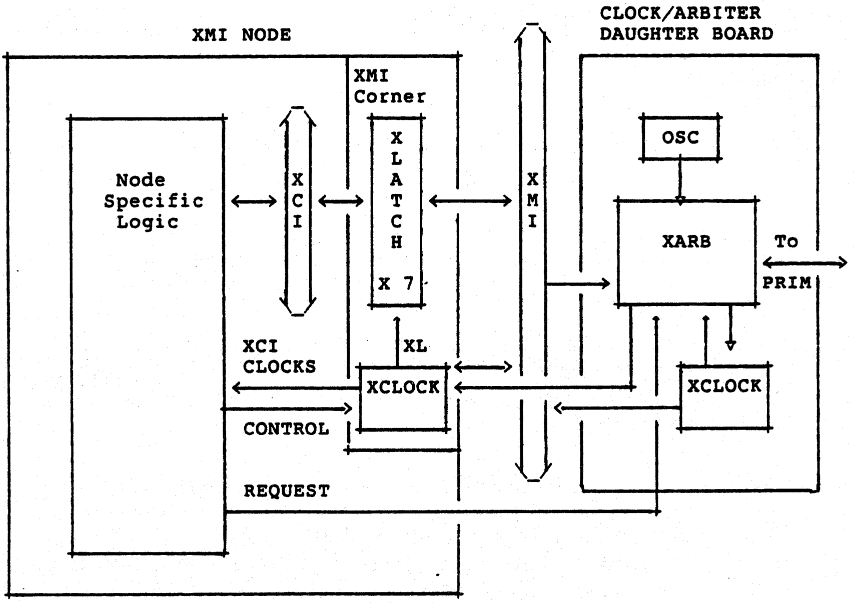


Figure 5–8 Basic Clock/Arbiter Block Diagram

The module implements two arbitration queues, a responder queue and a commander queue. Priority increases with the node ID with a responder having a higher priority than a commander.

The module also implements the:

- suppress function where commander requests can be disabled by a responder node by asserting its XMI SUP L line
- HOLD function where any current bus master can gain access to the XMI during the next bus cycle by asserting the XMI HOLD L line.

In addition, the module is supplied with +5 Vdc to insure that the system clocks are available at all times for battery-backed-up memories.

5.4 JXDI Description

All JXDI signals are uni-directional, series terminated ECL 100K levels, differential (except where noted). The method of data transfer on the JXDI is asynchronous with clocks supplied by the transmitter at a cycle time equal to the nominal AQUARIUS CPU cycle time.

Note that for the ICU transmitter the 16 nsec clock source is CLOCK_B. For the XJA transmitter the 16 nsec clock source is a crystal oscillator.

5.4.1 JXDI Signal Descriptions

Table 5–6 JXDI Signal Line Descriptions

SIGNAL LINE	DESCRIPTION
XJA To ICU	
XJA_DATA_H <15:0>	These 15 data lines transfer command, address, and data information to the ICU.
XJA_PAR_H <1:0>	This field provides odd parity over XJA_DATA_H<15:0>. PAR_H<0> corresponds to DATA_H<7:0>, and PAR_H <1> corresponds to DATA_H <15:8>. A parity bit is asserted when the number of data bits asserted is even.
XJA_CMDAVAIL_H	The Command Available signal is asserted on the cycle before the XJA starts transmitting a transaction packet of information to the ICU. The XJA is only allowed to assert XJA_CMDAVAIL if an ICU buffer is available as indicated by ICU_BUFEMPTD_H (buffer emptied). In the XJA there is at least six JXDI cycles between successive XJA_CMDAVAIL_H's.
XJA_XFERACK_H	The Transfer Acknowledge signal is asserted after the last transfer of a transaction to indicate that an ICU to XJA transaction packet has been received without errors. The signal allows the ICU to purge the corresponding transmit buffer. If XJA_XFERACK_H is asserted, XJA_XFERRETRY_H (transfer retry) should not be asserted. There will always be at least seven JXDI cycles between successive XJA_XFERACK_H's. The ICU will queue up these before its CLKJ synchronizers. That way if SCU clocks are stopped or run at slow speed, an XJA_XFERACK_H will not be lost. There are also at least two JXDI cycles of separation between an XJA_XFERRETRY_H and an XJA_XFERACK_H. This insures that the ICU cannot become confused as to which applies to which packet. The ICU can transmit up to two transactions to the XJA before receiving an XJA_XFERACK_H or XJA_XFERRETRY_H - either one or the other - for each transaction sent. If the first transaction has bad parity and the second does not, then XJA_XFERRETRY_H will be sent first, followed by XJA_XFERACK_H; and visa versa. That is, the ICU takes the first ACK/NAK (XFERACK/XFERRETRY) to correspond to the first transaction sent, the second to the second transaction sent.
XJA_XFERRETRY_H	The Transfer Retry signal is asserted following the last transfer of a transaction to indicate that a parity error was detected by the XJA receive logic during an ICU to XJA transaction. On the receipt of XJA_XFERRETRY_H, the ICU will attempt to retry the failed transaction TBD times before flagging a fatal error. If XJA_XFERRETRY_H is asserted, XJA_XFERACK_H should not be asserted. XJA_XFERRETRY is also used by the XJA to force a retry of an ICU to XJA transfer when the XJA is busy with an XJA to ICU transfer. This is due to the fact that the interface between the XDC and XDE chips is a bidirectional bus that can only handle a transfer in one direction at a time. If the XJA asserts XJA_XFERRETRY for this reason (i.e. a collision) the XJA will hold off further XJA to ICU transfers until the retried ICU to XJA transfer is allowed to complete.

Table 5–6 (Cont.) JXDI Signal Line Descriptions

SIGNAL LINE	DESCRIPTION
XJA To ICU	
	There will always be at least one JXDI cycle between successive XJA_XFERRETRY_Hs. The ICU will queue up these before its CLKJ synchronizers. Thus if the SCU clocks are stopped or run at slow speed, an XJA_XFERRETRY_H will not be lost. There are also at least 2 JXDI cycles of separation between an XJA_XFERRETRY_H and an XJA_XFERACK_H. This insures that the ICU cannot become confused as to which applies to which packet. The ICU can transmit up to two transactions to the XJA before receiving an XJA_XFERACK_H or XJA_XFERRETRY_H, either one or the other, for each transaction sent. If the first transaction has bad parity and the second does not, then XJA_XFERRETRY_H will be sent first, followed by XJA_XFERACK_H; and visa versa. That is, the ICU takes the first ACK/NAK (XFERACK/XFERRETRY) to correspond to the first transaction sent, the second to the second transaction sent.
XJA_BUFEMPTD_H	The Buffer Emptied signal indicates that the XJA has successfully emptied a JXDI receive buffer by either initiating the appropriate XMI transaction or by carrying out the specified internal operation (e.g., register write etc.). The XJA asserts this signal for one JXDI cycle every time it has emptied a JXDI receive buffer. The XJA has two receive buffers. The ICU can track the status of these two buffers by using this signal to increment a counter. The counter is decremented by one for each ICU transaction transmitted. A count of zero stops the transmitter. Simultaneous attempts to increment and decrement the counter cancel each other. The counter is preset by SPU_RESET_H to a count of 2; a count of 3 is detected as a fatal error. If a parity error or a collision (see XJA_XFERRETRY_H description) was detected on an ICU to XJA transaction, XJA_BUFEMPTD_H will be asserted with XJA_XFERERR_H. If this occurs, the resulting XJA_BUFEMPTD_H may occur nearly coincident with another occurring as described above; the two cannot overlap but may occur in adjacent cycles. The ICU will queue up these before its CLKJ synchronizers. Thus if the SCU clocks are stopped or run at slow speed, an XJA_BUFEMPTD_H will not be lost. The result is a one-to-one relationship between each ICU_CMDAVAIL_H (or ICU transaction) and XJA_BUFEMPTD_H, regardless of receive errors.

Table 5–6 (Cont.) JXDI Signal Line Descriptions

SIGNAL LINE	DESCRIPTION
ICU To XJA	
ICU_DATA_H<15:0>	This is the data word transferred from the ICU to the XJA that carries command, address and data information.
ICU_PAR_H<1:0>	This field is the odd parity over ICU_DATA_H<15:0>. PAR_H<0> corresponds to DATA_H<7:0>, and PAR_H<1> corresponds to DATA_H<15:8>. A parity bit is asserted when the number of asserted data bits is even.
ICU_CMDAVAIL_H	This signal is asserted on the cycle before the ICU starts transmitting a transaction packet of information. The ICU is only allowed to assert ICU_CMDAVAIL if an XJA buffer is available as indicated by XJA_BUFEMPTD_H.

Table 5-6 (Cont.) JXDI Signal Line Descriptions

SIGNAL LINE	DESCRIPTION
ICU To XJA	
	There will be at least 12 JXDI cycles between successive ICU_CMDAVAIL_H's. If the ICU is transmitting a packet and, before it can finish, receives XJA_XFERRETRY_H pertaining to that packet, it will finish the current transmission before restarting it or starting another.
ICU_XFERACK_H	This signal is asserted after the last transfer of a transaction to indicate that an XJA to ICU transaction packet has been received without errors. This signal allows the XJA to purge the corresponding transmit buffer. If ICU_XFERACK_H is asserted, ICU_XFERRETRY_H should not be asserted. Successive ICU_XFERACK_H's are issued at least six JXDI cycles apart. This insures that the freerunning XJA synchronizers for this signal will not be overrun. There are also at least two JXDI cycles of separation between an ICU_XFERACK_H and an ICU_XFERRETRY_H. This insures that synchronized versions of these signals do not occur during the same cycle, confusing the XJA as to which applies to which of two XJA transmit packets.
ICU_XFERRETRY_H	This signal is asserted after the last transfer of a transaction to indicate that a parity error was detected by the ICU receive logic during an XJA to ICU transaction. On the receipt of ICU_XFERRETRY_H, the XJA will always retry the failed transaction. The ICU may count TBD retries then flag a fatal error. If ICU_XFERRETRY_H is asserted, ICU_XFERACK_H should not be asserted. Successive ICU_XFERRETRY_Hs are issued at least six JXDI cycles apart. This insures that the freerunning XJA synchronizers for this signal will not be overrun. There are also at least two JXDI cycles of separation between an ICU_XFERRETRY_H and an ICU_XFERACK_H. This insures that synchronized versions of these signals can't occur during the same cycle, confusing the XJA as to which applies to which of two XJA transmit packets.
ICU_BUFEMPTD_H	This signal indicates that the ICU has successfully emptied a JXDI receive buffer by sending the transaction to the JBox. The ICU asserts this signal for one JXDI cycle each time it has emptied a JXDI receive buffer. The ICU has two receive buffers. The XJA can track the status of these two buffers by using this signal to increment a counter. The counter is decremented by one for each XJA transaction transmitted. A count of zero stops the transmitter. Simultaneous attempts to increment and decrement the counter cancel each other. The counter is preset to a count of two; a count of three is detected as a fatal error. If a parity error was detected on an XJA to ICU transaction, ICU_BUFEMPTD_H will be asserted with ICU_XFERERR_H. The result is a one-to-one relationship between each XJA_CMDAVAIL_H (or XJA transaction) and ICU_BUFEMPTD_H, regardless of receive errors. Successive ICU_BUFEMPTD_H's are issued at least six JXDI cycles apart. This insures that the freerunning XJA synchronizers for this signal will not be overrun.

Table 5-6 (Cont.) JXDI Signal Line Descriptions

SIGNAL LINE	DESCRIPTION
Miscellaneous	
ICU_CLKJ_H<2:0>	This signal is a 16 nsec period, 50% duty cycle clock signal the ICU sends to the XJA for receiving on the JXDI data lines.
XJA_CLKX_H<2:0>	This signal is a 16nsec period, 50% duty cycle clock signal the XJA sends to the ICU for receiving on the JXDI data lines.
SPU_RESET_H	This signal is a single-ended TTL level signal that is sourced by the SPU and used during initialization.
SPU_CLKSTOP_H	This signal is a single-ended TTL level signal sourced by the SPU. It is used to warn the XJA of impending SCU clock stoppage. On receiving this signal, the XJA will finish the current JXDI transmit transaction, if there is one, but will not transmit any new ones to the ICU, even if signaled to retry by ICU_XJA_XFERRETRY_H.
ICU_LOOP_H	This signal, when asserted, forces the XJA to directly loopback all JXDI signals sourced by the ICU as their opposite number on those lines sourced by the XJA. For example, ICU_CLKJ_H>2> loops as XJA_CLKX_H<2>; SPU_RESET_H loops to become XJA_FATALERR_H.
XJA_PRESENT_H	This is a non-differential ECL signal hardwired on the XJA module to be asserted. The ICU receives this signal into a SCAN LATCH that can then be read by the SPU to indicate the presence or absence of an XJA in a particular system configuration.
XJA_FATALERR_H	This signal, when asserted, indicates to the ICU that this XJA has detected a fatal error and may or may not be capable of responding to further CPU requests. When asserted, this signal indicates that the ICU should request an IPL1DH interrupt from all four Aquarius CPU's irrespective of the JBOX_ INTR_CTRL register. This signal will be asserted and negated asynchronously with respect to XJA_CLKX_H and can be received (synchronized) directly through a scan latch as, once asserted, it will remain so until the operating system clears a corresponding status bit in an XJA status register.

5.4.2 Field Definitions

The following subsections describe the encoding of each field and the related JXDI transaction.

5.4.2.1 Command Field Coding

The Command Field is specified as data lines DATA_H<3:0> on Word 0 of a JXDI transaction. Table 5-7 defines the command field coding.

Table 5–7 Command Field Codes

CODE<3:0>	COMMAND
0 0 0 0	READ_REQUEST
0 0 0 1	READ_LOCK_REQUEST
0 0 1 0	READ_DATA_RETURN
0 0 1 1	READ_LOCK_DATA_RETURN
0 1 0 0	WRITE_REQUEST
0 1 0 1	WRITE_UNLOCK_REQUEST
0 1 1 0	Reserved
0 1 1 1	Reserved
1 0 0 0	INTERRUPT_REQUEST
1 0 0 1	READ_LOCKED_STATUS
1 0 1 0	READ_ERROR_STATUS
1 0 1 1	WRITE_COMPLETE
1 1 0 0	Reserved
1 1 0 1	Reserved
1 1 1 0	Reserved
1 1 1 1	Reserved

Reserved commands with good parity decode as minimum length packets in the receiver and result in the receiver issuing XFERACK and BUFEMPTD.

The WRITE_COMPLETE command is used only on CPU type JXDI transactions. The command informs the ICU that a given CPU write command has completed and the ICU can send another CPU type transaction to the XJA.

READ_LOCK_REQUEST and WRITE_UNLOCK_REQUEST will not be issued by the ICU to the XJA. The XJA will support receiving these functions only for the XJA Kludge Mode. The ICU does not support receiving READ_LOCK_DATA_RETURN or READ_LOCKED STATUS.

5.4.2.2 Length Field Coding

The Length Field occupies data lines DATA_H<5:4> of Word 0 of the JXDI transaction. Table 5-8 specifies the data types codes.

Table 5–8 Length Field Codes

LENGTH <5:4>	DATA TYPE
0 0	Hexword (32 bytes)
0 1	Block (64 bytes)
1 0	Quadword (8 bytes)
1 1	Octaword (16 bytes)

Longword length XMI transactions that reference main memory will be translated into QW length transactions. All CPU type JXDI transactions are longword in length and therefore do not have the length field defined.

Block transactions are split into two separate JXDI transactions, each a hexword in length, each assigned a one bit sequence value to distinguish between halves, but with the JXDI command word in each containing the Block length encode.

5.4.2.3 Mask Field Coding

The fourth cycle of a JXDI DMA write transaction contains the byte mask bits for the following write data. Bit 0 corresponds to Byte 0 etc.. When DMA write transactions are less than 16 bytes in length, the unused mask bits are de-asserted.

In all cases the JXDI parity bits reflect the correct parity over the JXDI data fields. For DMA hexword writes, the mask field will be all ones since in these transactions all 32 bytes are valid. That is, the memory controller may interpret any of these bits as a longword (not byte) mask.

The DATA_H<15:12> field of Word 2 on a JXDI CPU Write type transaction contains the byte mask for the following longword of write data. DATA_H<15> is the mask bit for byte 3, <14> for byte 2, <13> for byte 1 and <12> for byte 0.

5.4.2.4 Identity Field Coding

The DATA_H<13:08> field of Word 0 on a JXDI transaction contains a unique identity code (ID) used to identify the source of the request and the destination of the returned read data.

During DMA type JXDI transactions, DATA_H<13:08> contains the value of the corresponding XMI_ID<5:0> field of the corresponding XMI transaction. This field is encoded as follows:

DATA_H<13:10> = XMI Node ID of the commander
DATA_H<09:08> = which of four possible outstanding commander requests.

During CPU type JXDI transactions, DATA_H[13:08] is encoded as follows and specifies the request source.

DATA_H<13:08> = 0 0 0 0 0 0 = AQUARIUS CPU
DATA_H<13:08> = 0 0 0 0 0 1 = SPU

5.4.2.5 IPL Field Coding

During INTR type JXDI transactions, DATA_H<7:4> is encoded with the Interrupt Priority Level (IPL) as follows:

<5:4> = 0 0 = IPL14H
<5:4> = 0 1 = IPL15H
<5:4> = 1 0 = IPL16H
<5:4> = 1 1 = IPL17H

5.4.2.6 Address Interpretation

All addresses are naturally aligned. The SCU supports wrapped read requests on quadword boundaries. Table 5-9 specifies JXDI address interpretation.

Table 5–9 JXDI Address Interpretation

TRANSACTION	ADDRESS BITS <5:0>
Read Longword	A A A A A A
Read Quadword	A A A D D D
Read Octaword	A A W D D D
Read Hexword	A W W D D D
Write Longword	A A A A A A
Write Quadword	A A A D D D
Write Octaword	A A D D D D

Where: A = Address, D = Don't care, and W = Wraparound bits (starting quadword).

For longword length transactions, ADR<1:0> are only significant when dealing with a VAXBI word-mode or byte-mode transaction in I/O space. ADR<1> is required for word mode and ADR<1:0> is required for byte mode. Quadword and Octaword write transfers are assumed to be naturally aligned, allowing the lower bits of the address to have don't care status.

In the case of reads the situation is different since the main memory does wraparound reads on quadword boundaries. On longword reads to I/O space, ADR<1> is significant. This is because there is no explicit READ WORD function on the XMI. When a read is directed toward a word-oriented device ADR<1> significant since it specifies which word is to be read from that device. The relationship between the high and low words, ADR<1>, and the data bits is:

ADR<1> = 1 = high word = D<31:16>
ADR<1> = 0 = low word = D<15:00>

In the case of a longword oriented device ADR<1> is ignored as an address bit. A longword of data is returned for a read operation.

The XJA asserts ADR<1:0> based on the contents of the Mask field in the DATA_ H<15:12> field of Word 2 on a JXDI CPU type transaction as specified in Table 5-10.

Table 5–10 Mask Field XMI Address Interpretation

MASK FIELD	XMI ADDRESS
DATA_ H<15:12>	
0 0 0 0	0 0
0 0 0 1	0 0
0 0 1 0	0 1
0 1 0 0	1 0
1 0 0 0	1 1
0 0 1 1	0 0
1 1 0 0	1 0
1 1 1 1	0 0

NOTE
The JXDI Mask field is valid and significant for both CPU write and read type transactions.

5.4.2.7 Sequence Field Coding

The Sequence Bit (S) encodes hexword halves transmitted in both directions as part of the Block transaction as follows:

S = 0 = First half (low order hexword)
S = 1 = Second half (high order hexword)

5.5 XJA Description

The XJA communicates with the ICU and the XMI through three transaction types: DMA, CPU, and Interrupt.

5.5.1 DMA Transactions

XMI transactions that select the XJA as the responder node will be forwarded to the ICU as DMA type transactions. DMA transactions can be reads, writes, read locks, or write unlocks. DMA transactions can be quadword, octaword or hexword in length. The XJA can have up to four DMA type read transactions accepted from the XMI outstanding at any given time. Read data returned from the ICU will be forwarded on the XMI as read data response transactions.

The JXDI command prefix code for DMA type transactions is DMA.

5.5.2 CPU Transactions

AQUARIUS CPUs can access the I/O portion of VAX physical address space through CPU type transactions. These transactions are received by the XJA from the JXDI and are forwarded on to the XMI with the XJA as the commander. CPU type transactions are longword in length and the XJA can only accept a single CPU type transaction at a time.

The JXDI command prefix code for CPU type transactions is CPU.

5.5.3 Interrupt Transactions

The XJA will field interrupt transactions from the XMI and forward them to the ICU using interrupt type transactions. The resulting SCB offset vector fetch initiated by an Aquarius CPU will be a CPU type transaction.

The JXDI command prefix code for interrupt type transactions is INTR.

5.5.4 JXDI Transaction Descriptions

Table 5-11 describes each JXDI transaction. Depending on the command direction over the JXDI, two conditions apply to each transaction direction.

If the command direction is from the XJA to ICU, the XJA will:

o not purge the assembled transaction until it receives ICU_XFERACK

o re-transmit the transaction if it receives ICU_XFERRETRY.

If the command direction is from the ICU to XJA, the ICU will:

o not purge the assembled transaction until it receives XJA_XFERACK

o re-transmit the transaction if it receives XJA_XFERRETRY.

Table 5–11 JXDI Transaction Descriptions

COMMAND	DESCRIPTION
XJA to ICU	
DMA$READ_REQUEST	In response to the receipt of an XMI read request that references AQUARIUS main memory, the XJA will assemble and transmit a DMA$READ_REQUEST transaction to the ICU.
DMA$READ_LOCK_REQUEST	In response to the receipt of an XMI interlock read request that references AQUARIUS main memory, the XJA will assemble and transmit a DMA$READ_LOCK_REQUEST transaction to the ICU. If the ICU detects that the requested data is locked by another entity (i.e., CPU or another I/O device) the ICU will respond to this request with a DMA$READ_LOCK_STATUS JXDI transaction.
DMA$WRITE_REQUEST	In response to the receipt of an XMI write request that references AQUARIUS main memory, the XJA will assemble and transmit a DMA$WRITE_REQUEST transaction to the ICU. The command will be followed by a specified number (in the length field) of data cycles.
DMA$WRITE_UNLOCK_REQUEST	In response to the receipt of an XMI unlock write request that references AQUARIUS main memory, the XJA will assemble and transmit a DMA$WRITE_UNLOCK_REQUEST transaction to the ICU. The command is followed by a specified number (in the length field) of data cycles.
CPU$READ_DATA_RETURN	In response to the receipt of an XMI read data response cycle that corresponds to a previously transmitted XMI read data request, or a previously accepted CPU$READ_REQUEST that referenced local XJA registers, the XJA will assemble and transmit a CPU$READ_DATA_RETURN transaction to the ICU. The command cycle is followed by three reserved cycles, then four data cycles (CPU's can only access I/O space with longword length transactions).
CPU$READ_ERROR_STATUS	In response to the receipt of an XMI read error response cycle that corresponds to: a previously transmitted XMI read data request, an XMI commander, retry timeout, response timeout, or a previously accepted CPU$READ_REQUEST that referenced local XJA registers and incurred an error, the XJA will assemble and transmit a CPU$READ_ERROR_STATUS transaction to the ICU.
CPU$WRITE_COMPLETE	On completion of a previously requested CPU$WRITE_REQUEST or CPU$WRITE_UNLOCK_REQUEST the XJA will assemble and transmit a CPU$WRITE_COMPLETE transaction to the ICU. This indicates to the ICU that the XJA has completed the requested XMI or local write successfully, and the ICU can now send another CPU request to the XJA.

Table 5-11 (Cont.) JXDI Transaction Descriptions

COMMAND	DESCRIPTION
XJA to ICU	
INTR$REQUEST	In response to the receipt of an XMI interrupt request, the occurrence of a condition that requires the generation of an error interrupt, the receipt of an XMI write error interrupt, or interprocessor interrupt, at an IPL that does not have a request currently outstanding, the XJA will assemble and transmit a INTR$REQUEST transaction to the ICU.
ICU To XJA	
CPU$READ_REQUEST	This command is generated in response to an I/O space read request from an AQUARIUS CPU or the SPU. In response to the receipt of a CPU$READ_REQUEST transaction that does not reference local XJA registers, the XJA will arbitrate for the XMI, assemble an XMI C/A field and attempt to transmit an XMI read command.
CPU$WRITE_REQUEST	This command is generated in response to an I/O space write request from an AQUARIUS CPU or the SPU. In response to the receipt of a valid CPU$WRITE_REQUEST transaction that does not reference local XJA registers, the XJA will arbitrate for the XMI, assemble an XMI C/A field and attempt to transmit an XMI write command.
DMA$READ_DATA_RETURN	This command is generated in order to return read data from main memory that was requested through a previous DMA$READ_REQUEST command. In response to the receipt of a DMA$READ_DATA_RETURN transaction, the XJA will arbitrate for the XMI and attempt to transmit an XMI read data response cycle(s).
DMA$READ_LOCK_DATA_ RETURN	This command is generated in order to return read data from main memory that was requested through a previous DMA$READ_LOCK_REQUEST command. In response to the receipt of a DMA$READ_LOCK_DATA_RETURN transaction, the XJA will arbitrate for the XMI and attempt to transmit an XMI read data response cycle(s).
DMA$READ_LOCKED_STATUS	This command is generated by the CCU in response to a DMA$READ_LOCK_REQUEST command where the requested data was previously locked by another entity (i.e., CPU, SPU or another XJA). In response to the receipt of a DMA$READ_LOCKED_STATUS transaction,the XJA will arbitrate for the XMI, and attempt to transmit an XMI read locked response cycle.
DMA$READ_ERROR_STATUS	This command is generated by the CCU in response to a DMA$READ_REQUEST or a DMA$READ_LOCK_REQUEST command where the requested data, when read from main memory, encountered an uncorrectable error. I n response to the receipt of a DMA$READ_ERROR_STATUS transaction, the XJA will arbitrate for the XMI, and attempt to transmit an XMI read error response cycle.

5.5.5 XJA Functional Overview

The XJA is partitioned into a single CMOS gate array, two unique AMCC gate array types, and the two standard XMI corner chips.

5.5.5.1 JXDI Data Path Array

The XJA Data path ECL gate array (XDE) implements one byte of the two-byte wide JXDI. It is basically a level and speed shifter data path. The pair of XDCs transfer two two bytes of data from the ECL JXDI at 16 nsec, to the TTL levels of the XDC array (XJA Data Path) at 32 nsec.

5.5.5.2 JXDI Control Array

The JXDI Control ECL gate array (XCE) controls the JXDI protocol sequencing and interfaces the JXDI control to the XDC array. The XCE implements several hardwired state machines to sequence the JXDI interface. XCE interacts with three asynchronous clock systems, as well as requests from the ICU and the XDC array. In addition, the XCE arbitrates the use of the CBI bus, and implements the JXDI error recovery mechanism.

5.5.5.3 Data Path Gate Array

The XJA Data path CMOS gate array (XDC) implements all of the required data path interface functions to the XLATCH chips. The XDC implements a 3-hexword DMA write buffer, four XMI receive C/A buffers, two hexword DMA return read data buffers, a single XMI transmit C/A buffer, and the XMI required and XJA specific registers.

The XDC array also implements the main XJA control functions and interfaces to the XCLOCK chip and the JXDI control chip (XCE). The XDC array is the most complex of the chips since it must implement the full XMI protocol functionality, and XJA error recovery functionality. The following paragraphs provide an overview of the XJA arrays and the major functional logic within the XDC array. (See Figure 5-9.)

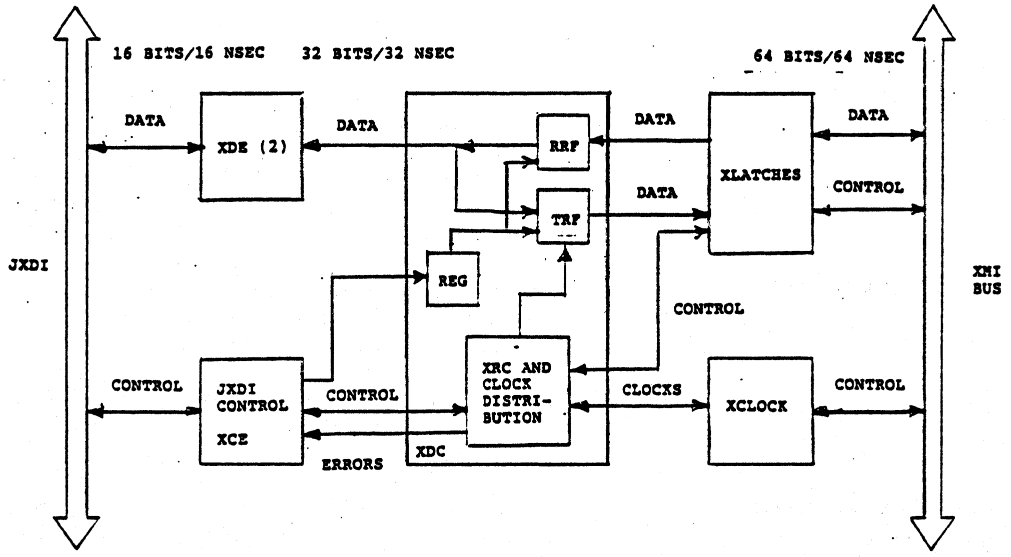


Figure 5–9 Basic XJA Block Diagram

- o XMI Receive Logic (XRC) - The XRC receives XMI transaction cycles, checks XMI parity, and performs XMI protocol checking to insure bus integrity. The XRC generates the XMI CNF code for XMI commands that select the XJA. If the XJA is selected, the XRC initiates the Receive Control Machine (RCM)

- Receive Register File (OF) - The OF contains four XMI C/A buffers and three hexword XMI write data buffers. The XRC loads OF when it receives valid XMI commands. Return read data XMI cycles are also stored in OF. OF contains the JXDI command generator for CPU, INTR and status type XJA to ICU transactions. RCM controls the status of the buffers in OF and sends this status to XRC.
- XJA Registers (REG) - The REG implements the XMI required and XJA specific registers. The XMI required registers are implemented as visible from the XMI and the CPUs. In general, the XJA specific registers are only visible from the CPUs. REG is also responsible for most of the XJA error handling in conjunction with the Receive and Transmit Control Machines.

  All XJA registers will ignore any masking information on writes. Masked writes to these registers will be treated as longword writes. All XJA registers will ignore locking information on read locks and write unlocks. No logical locking mechanism will be set and these transactions will complete as generic reads and writes.
- Transmit Register File (TRF) - The TRF receives JXDI transactions from the ICU and buffers them for transmit on the XMI. It contains a two buffers, each capable of handling a hexword Return Read Data transaction. TRF is loaded under control of the JXDI Control array (XCE). TRF is unloaded to either the REG or the XMI under the control of the Transmit Control Machine (TCM). Status of the TRF buffer is controlled by TCM which sends the status to the XCE array.
- Receive Control Machine (RCM) - The RCM provide the control required to receive XMI transactions, and notify the XCE array that a given transaction is to be sent to the ICU. RCM controls the status of the buffers in OF and sends the status to XRC. If an XMI transaction references the XMI registers, the RCM passes control to TCM.

  The RCM handles the XJA DMA flow control. If no DMA C/A or write buffers are available in the XJA for servicing XMI DMA requests, the XRC will NOACK any DMA XMI transactions that select the XJA. RCM will assert XMI_SUP_L when all available DMA C/A or write buffers are full. Thus up to three additional DMA read requests may receive XMI NOACK response from XRC before RCM can assert SUP_L. The XJA will always accept return read data XMI transactions (provided the XJA has a CPU read outstanding) and XMI transactions that select the XJA's XMI visible registers.
- Transmit Control Machine (TCM) - the TCM provides the control required to generate XMI command and response sequences due to either:

  the receipt of a valid JXDI transaction, or
  in response to an XMI read request that referenced the XJA's XMI registers.

  The TCM also maintains control of all accesses to the REG.

5.5.5.4 XJA Ramp Features

Since the XJA is not part of the Scan System, it will be tested at system initialization time through a functional diagnostic running in either the SPU or one of the CPUs. The diagnostic will report success or failure to the SPU (or CPU) through interrupts and an error bit in an XJA error register.

The XJA will implement an transient error recovery mechanism to conform to the XMI. All data paths will be parity protected. When parity errors are detected on the XMI, the XJA will attempt to retry the XMI transaction. Only after a TBD timeout will the XJA assume that the transaction failed and raise an error interrupt.

5.6 I/O Configuration Descriptions

The following subsections describe the preliminary I/O configuration rules, configurations, and I/O controllers and devices.

5.6.1 I/O Configuration Rules

AQUARIUS will use the CI bus as the primary mass storage access path, and will be considered a CI-based system. The CI will be provided by the XCD (XMI to CI adapter) There is a maximum of 16 XCD - CI ports allowed on an AQUARIUS system. All three CI couplers will be supported. However, the CI-Switch is recommended to support connectivity and the aggregate bandwidth requirements.

There will be up to four XMI busses on four separate 14-slot XMI card cages. An XMI card cage cannot be repeated to another card cage, establishing a limit of one card cage per XMI. There is a physical limit of nine XMI nodes due to unusable slot restrictions of the current 14-slot XMI backplane.

The XBI adapter will provide AQUARIUS with the capability of using the BI bus. BI expansion is not required for AQUARIUS systems and will be considered an add-on. The BCA (BI to CI adapter) will be supported, but the recommended CI connection is the XCD.

5.6.2 Absolute Minimum I/O Configuration

Absolute minimum describes the I/O configuration necessary to provide one CI and one NI interconnect as shown below. This is not a high availability configuration in that if any single component fails, the entire AQUARIUS system becomes unavailable.

XMI0 Card Cage
XCD ---> CI
XNA ---> NI

5.6.3 Minimum High Availability Configuration

To achieve the high availability goals, redundant data paths must be implemented for the system disk as shown below. Each XMI cardcage will require it's own power supply that can be independently powered off. The external I/O components will also be redundant (i.e., two HSC70s, and a shadow set for the system disk). In this way any single failure will recover to the redundant path, and will not effect system availability.

XMI0 Card Cage
XCD ---> CI ---> HSC-1 ---> System Disk A-port
XNA ---> NI

XMI1 Card Cage
XCD ---> CI ---> HSC-2 ---> System Disk B-port
XNA ---> NI

5.6.4 Maximum High Availability Configuration

Maximum describes the I/O configuration required to provide 16 CI and two NI interconnects as shown below. It is physically possible to connect more than 16 CIs on a system. However, 16 is the maximum number supported under VMS. It is recommended that the system disk be configured in the high availability connect, and that each card cage be individually powered.

XMI0 Card Cage
XCD0 ——> CI0
XCD1 ——> CI1
XCD2 ——> CI2
.
.
.
XCD7 ——> CI7
XNA0 ——> NI0

XMI1 Card Cage
XCD8 ——> CI8
XCD9 ——> CI9
.
.
.
XCD15 ——> CI15
XNA1 ——> NI1

5.6.5 Typical AQUARIUS Configurations

AQUARIUS Product Management has defined the following typical configurations:

Cluster Compute Server (shown below) is a VAXcluster node that provides only additional computes without more I/O capacity.

XMI-0 Card Cage
XCD ——> CI0 ––> existing cluster
XNA ——> NI0 ––> existing Ethernet

Cluster Expansion Node (shown below) is a VAXcluster node that is added for new applications and includes the components to support those applications.

XMI-0 Card Cage
XCD ——> CI0 ——> HSC70 ——> 7 RA90 disks
XCD ——> CI1 ——> HSC70––> 6 RA90 disks
XNA ——> Ethernet

New Applications (shown below) is a single system that would be the beginning of a cluster or a stand-alone system.

XMI-0 Card Cage
XCD ——> CI0——> HSC70 ——> 7 RA90 disks, 3 TA79 tapes
XCD ——> CI1——> HSC70 ——>13 RA90 disks
XNA ——> Ethernet

XMI-1 Card Cage
XCD ——> CI2——> HSC70——>12 RA90 disks,3 TA79 tapes
XCD ——> CI3——> HSC70——>14 RA90 disks
XNA ——> Ethernet

These configurations do not include terminals, printers, or other unit record devices which will be connected through the NI (Ethernet), and are customer specific.

5.6.6 Supported Components

Table 5-12 summarizes the supported I/O adapters, controllers, and devices.

Table 5–12 I/O Supported Components

COMPONENT	DESCRIPTION
Adapters	
XBI	XMI to BI Adapter
XNA	XMI to NI Adapter
XCD	XMI to CI Adapter
DRX	XMI to 32-bit parallel interface - similar to DRB32
Controllers	
BCA	BI to CI Device
DRB32	General Purpose Parallel Device
HSX50	XMI TO SI disk and tape controller
KDB50	UDA Lookalike for the BI bus
DEBNT	BI to NI Device for the BI bus
DMB32	Provides Local Terminals

Table 5-12 (Cont.) I/O Supported Components

COMPONENT	DESCRIPTION
Devices	
RA90	Fixed media, 1.2 Gbyte DSA disk
RA82	Fixed media, 622 Mbyte DSA disk
RA81	Fixed media, 466 Mbyte DSA disk
RA60	Removable media, 204 Mbyte DSA disk
TA90	New 3480 look-alike cartridge magtape
TX79	Enhanced TA78
HSC50	Hierachical Storage Controller up to 6 disk controllers
HSC70	Hierachical Storage Controller up to 9 disk controllers
SC008	Star coupler up to 16 nodes 1 channel
CI Switch	(PLEIADES) CI Connectivity up to 64 nodes and 8 channels

5.6.7 Communication Requirements

The I/O system will permit connection of several thousand terminals and computers distributed worldwide. This will be accomplished by using DEC standard Ethernet based products interfaced through the XNA Ethernet adapter. Ethernet concentrators will be the method of attaching terminal servers and print servers.

Custom communication device requirements will be met by using the BI bus interconnect, (XBI) or the XNAT. Local terminal lines will not be configured in the basic CPU cabinet. Any local terminal connections, other than the console terminal any local terminal connections will be considered add on to the basic AQUARIUS system.

It is also a requirement of the DECNET and LAT software to allow multiple Ethernet adapters on one operating system, and to have autofailover capabilities. This goes along with the reliability theme of having no single point of failure affect system availability.

6
Power and Control Subsystem Overview

6.1 Chapter Objective

The chapter objective is to introduce and provide specifications, and a functional overview of the AQUARIUS:

- power subsystem
- Power Control Subsystem (PCS)
- Operator Control Panel (OCP)

The majority of the chapter contents are abstracts or edits of all available power, power component, and control subsystem specifications.

Note that some power subsystem components have not been included in this description because of Engineering revisions. In addition, some components included may may not be part of the final design.

6.2 AQUARIUS Power Subsystem

6.2.1 Ac Power Distribution

As shown in Figure 6-1, the facility ac utility power is applied to the Utility Port Conditioner (UPC). The UPC is a free-standing device used to convert the 3-phase ac power input to an isolated, regulated 280 vdc.

An H7390 (Ac Front End) is required in place of the UPC. The H7390 performs the same functions as the UPC.

In either input case, the regulated 280 vdc is distributed to sets of Dc to Dc Converters and air movers in the CPU, SCU, and XMI cabinets.

6.2.1.1 UPC Overview

The UPC contains:

- an integrated DEC power controller function
- a delta to wye isolation transformer to provide power for single phase loads (e.g., BBUs, service outlets, etc.).

The UPC is a buy-out product from Liebert Corp. Since the UPC is still in the design stage only the basic specifications are included in this description. Details will be provided in the Power Subsystem Technical Description.

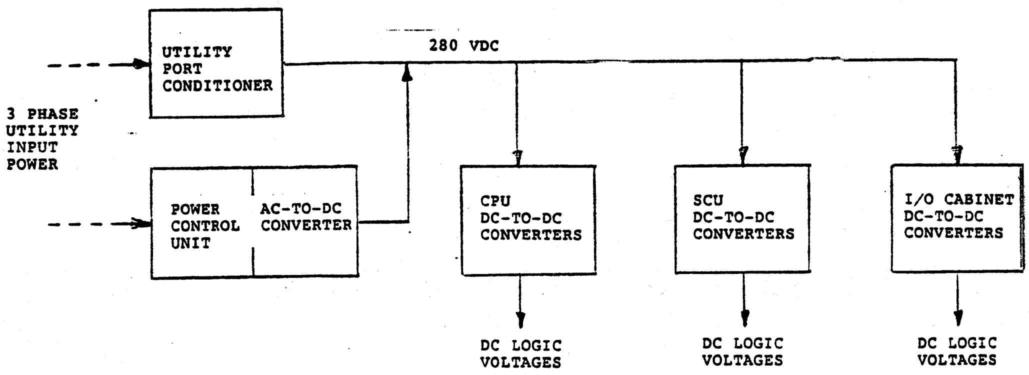


Figure 6–1 Basic Ac Power Input Block Diagram

The UPC can power a single dc load of 20 KW. With load sharing two UPC outputs can be ORed together to power a quad configuration. In this configuration, if one UPC fails the other UPC can support the total common load without interruption. (That is, common load = SCU, SPU, and memory subsystem.)

A sense circuit in the UPC monitors the input line voltage to ensure proper phase sequencing. If improper phase sequencing is detected, the UPC output voltage is inhibited.

The UPC will be delivered in two operating variations. Variation 1 operates from a nominal 416 v RMS source; Variation 2 operates from a nominal 208 v RMS source. However, each variation is further subdivided into several nominal ranges as specified in the following list. The list also specifies the maximum line current for each range.

o Nominal 416 v

  380 v nominal line-to-line: 303 - 418 v RMS, 3-phase delta, 3-wire without neutral, at 50 Hz

  400 v nominal line-to-line: 311 - 438 v RMS, 3-phase delta, 3-wire without neutral, at 50 Hz

  415 v nominal line-to-line: 327 - 457 v RMS, 3-phase delta, 3-wire without neutral, at 50 Hz

o Nominal 208 v

  202v nominal line-to-line: 174 - 220 v RMS, 3-phase delta, 4-wire mid-point earthed, at 50 or 60 Hz

  208 v nominal line-to-line: 159 - 229 v RMS, 3-phase delta, 3-wire without neutral, at a nominal 60 Hz

In addition, over the input voltage ranges the ratio of real power to apparent power will be greater than 0.90 at all power levels between 50% and 100% of full load.

The UPC output is applied to the CPU cabinet and distributed to the SCU and XMI cabinets through a set of H7287 Diode OR Modules, or

The PCS status, control, and measurement signals for the UPC are described in Table 6-8.

6.2.2 Dc Power Components

6.2.2.1 H7287 Diode OR Description

The function of the H7287 Diode OR module is to logically OR two, 280 vdc inputs from two UPCs. The module consists of three fused OR circuits. The basic module block diagram and output distribution is shown in Figure 6-2.

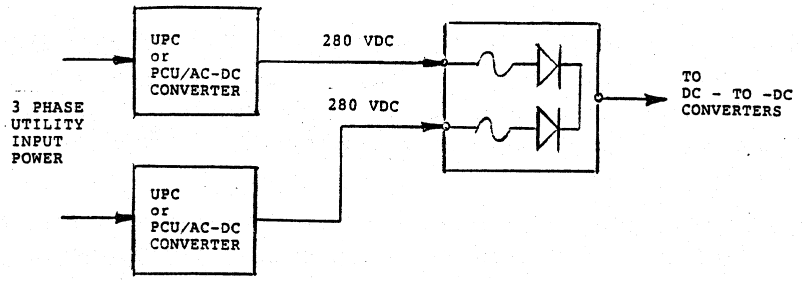


Figure 6–2 Basic Diode OR and Output Distribution

The Module operates over an input voltage range of 0 to 325 vdc +/-(TBD) and can withstand a 368 vdc, 1-second overvoltage surge. The input fuses are rated for 40 A, 20 A, and 20 A at a minimum of 500 vdc. The OR circuits are rated at maximum input and output currents of 30 A, 10 A, and 10 A, with maximum surge currents of 35 A, 15 A, and 15 A respectively.

6.2.2.2 H7380 Dc-DC Converter Description

The H7380 Dc-Dc converter reduces the 280 vdc input to a nominal 3.4 vdc or 5.2 vdc output at an output current between 20 A and 240 A. Five converters can be operated in parallel (load sharing) to provide N+1 redundancy. The total power required could be supplied by four converters. If one of the five converters should fail, the output of the remaining four will increase to provide the necessary power.

A startup voltage of +15 vdc at a maximum of 300 mA is supplied by an H7382 Bias Supply to initiate converter operation. An internal H7380 supply and H7382 input are diode ORed internally to the converter. Once the converter is delivering power the internal supply replaces the H7382 input.

Constant Current and Pulse Overcurrent limiting modes are used to prevent excessive output current causing module damage. Each mode is specified below:

- o Constant Current Limiting Mode: In this mode a maximum current limit of 300 A is set within each regulator. Once the limit is reached, drawing additional current causes a decrease in output voltage.
- o Pulsed Overcurrent Limit Mode: In this mode the output is turned off for a fixed (TBD) time period after a current limit of 265 A is reached. The converter then turns on with a slowstart. If the overload is still present, the output will turn off again.

  This on/off action will continue indefinitely or until the overload is removed. At that point the H7380 output will automatically recover to normal operation.

In addition, input overcurrent protection is provided by two fuses: one on the input high side, and one on the return side. The module is also equipped with thermal sensors and thermal fuses to protect against overtemperature conditions.

The front panel of each converter has the two LED indicators described below:

- o Module OK: When ON this green LED indicates that the module is operating correctly and is delivering current between 20 A +/-TBD% and 240 A +/-TBD%.

- Over Current: When ON this yellow LED indicates the module has exceeded the output threshold and is delivering excessive current. If the LED is pulsing, it indicates that the module is in pulsed overcurrent mode and delivering more than 265 A +/-TBD%

Logic power for a dual CPU is supplied by ten, 240-amp H7380 converter modules located in the CPU cabinet. Five converters supply -5.2 vdc, while the remaining five modules supply -3.4 vdc. A quad configuration would require 20 H7380s (for the CPUs only).

Four H7380s supply logic power for the SCU cabinet. Two supply -5.2 vdc to the SCU and memory, while the remaining two supply -3.2 vdc to the SCU only. In addition, five H7380s (two of which are located in the CPU cabinet for BBU) supply +5.0 vdc to the memory. Therefore a dual processor system requires 19 H7380s, while a quad-processor requires 29 H7380s.

The RIC modules provide the error amplifiers, voltage references, clock and control signals required for converter operation. During system initialization, the RIC configures the H7380s for the required power outputs of:

-5.2 vdc at 240 amps (CPU)
-3.2 vdc at 240 amps (CPU)
-5.2 vdc at 120 amps (SCU and Memory)
-3.2 vdc at 120 amps (SCU)
+5.0 vdc at 120 amps (Memory)

In addition, the RIC monitors several H7380 status signals, and controls several functions. The H7380 - RIC status and control signals are described in Table 6-9.

6.2.2.3 H7382/H7383 Bias Supply Description

The H7382 is a dc-to-dc converter module that reduces the 280 vdc input to +5, +15, and -15 vdc outputs. The converter module is used as the:

- primary source of power to the control and monitoring circuits of the PCS,
- as the startup power input for the H7380 converters.

The module has a green LED indicator on its front panel. When the LED is ON it indicates that the module is operating correctly (Module OK). In addition, the module has a MODULE OK output signal. When asserted, the signal indicates that all outputs are greater than 90% of the nominal voltage rating.

Two modules may be operated in parallel to increase the effective reliability through an output diode OR.

The module has output overload protection. That is, the module will not sustain damage when operating into a shorted load. The module will recover when the short is removed.

The module can also receive an input synchronizing signal (SYNC IN). SYNC IN allows two modules to synchronize their clocks. If SYNC IN is not present, the module free-runs on its internal clock.

The internal module clock is turned off in the event of an Output Overvoltage Protection (OVP) condition. Resetting the OVP circuit requires removing, then applying the input power.

The H7383 converter module is similar to the H7382 except that it reduces the 280 vdc input to +5, +15, -15, +24, and -24 vdc outputs.

6.2.2.4 H7214/7215 Power Module Descriptions

The H7214 and H7215 power modules are dc-dc converters located in the XMI I/O cabinet. A pair of these modules constitute one I/O power supply. One pair of converters is capable of powering one 14-slot XMI backplane or the 6-slot console BI backplane. Each module is described in the following subsections.

6.2.2.5 H7214 Power Module

The H7214 is a switching power module and provides:

- o a regulated +5 vdc output at 120 A
- o an unregulated +13.5 vdc output at 0.5 amps.

The module has one green LED located on the rear of the module. When ON, the LED indicates proper operation of the module.

The +5 vdc output has a crowbar to prevent the voltage from exceeding a maximum fault voltage of between 5.7 and 7.0 vdc. The 13.5 vdc output has short circuit protection provided by a series PTC (positive temperature coefficient) resistor.

The module input and output signals are described in Table 6-1.

Table 6-1 H7214 Signal Descriptions

SIGNAL DESCRIPTION	
INPUT SIGNALS	
Channel Number Inhibit	When high, this signal inhibits all module outputs. The signal also resets the module to a ready state allowing the output power to be restored when the signal is negated.
Clock	This signal is a 33 KHz pulse train used to synchronize the module.
OUTPUT SIGNAL	
Channel Number OK	When asserted, the signal indicates that the module is operating properly. When negated, the signal indicates that the module is not within the specified regulation.

6.2.2.6 H7215 Power Module

The H7215 power module provides the following outputs: -5.2, -2.0, +12.0, and -12.0 vdc. The input and output signals are described in Table 6-2.

Table 6–2 H7215 I/O Signal Descriptions

SIGNALS	DESCRIPTION
INPUT SIGNALS	
Sync	This signal is a 33 KHz clock pulse train used to synchronize the module with other power modules in the system.
Channel Inhibit	When asserted, this signal inhibits all module outputs. The signal also resets the module to a ready state so that the output power is restored when the signal is negated.
OUTPUT SIGNALS	
CHANNEL OK	When asserted, this signal indicates that the module is operating properly. When negated, it indicates that the module is not within its specified regulation.
Over Temperature	When asserted (L), this signal indicates that an over temperature condition exists in the module.

6.3 Power Control Subsystem

The Power Control Subsystem (PCS), provides an interface between the Service Processor Unit (SPU) and the power subsystem. Its primary functions are:

- Monitoring the status of the power system, cooling system including the Water Cooling Unit (WCU), and local environment and alerting the SPU of power and environmental fault conditions
- Controlling power system power-on and power-off sequences
- Powerfail handling including AC LO and DC LO
- Battery backup operation
- Controlling the DEC POWER BUS and TOTAL OFF BUS
- Controlling the Operators Control Panel (OCP)

6.4 PCS Hardware Overview

The PCS will continuously monitor the state of the environment and the power subsystem. Under normal operating conditions, the PCS will execute commands it receives from the SPU, and will inform it of any faults that have occurred in the system.

If the PCS detects a fault condition, and exception reporting is enabled, it will inform the SPU by issuing an exception report. If necessary, an Automatic Shut Down (ASD) sequence will be initiated. The ASD starts a counter. If the fault does not clear by the time the counter times out, the system is shut down through the Total Off bus.

In a quad CPU system, the PCS will support powering off one dual CPU while the other dual CPU continues operating. This is performed through a PCS software command to trip the proper circuit breaker. Manual resetting of the circuit breaker is required to restore power to the powered off machine.

The PCS also supports power down for console module replacement. Loss of a CPU group or one UPC will not cause a fatal power system fault and continued operation of the remaining CPU group is possible.

As shown in Figure 6-3 the major components of the PCS are:

- o Power and Environmental Monitor module (PEM)
- o Signal Input Panel (SIP)
- o Regulator Intelligence Card Bus (RICBUS)
- o Regulator Intelligence Cards (RICs)

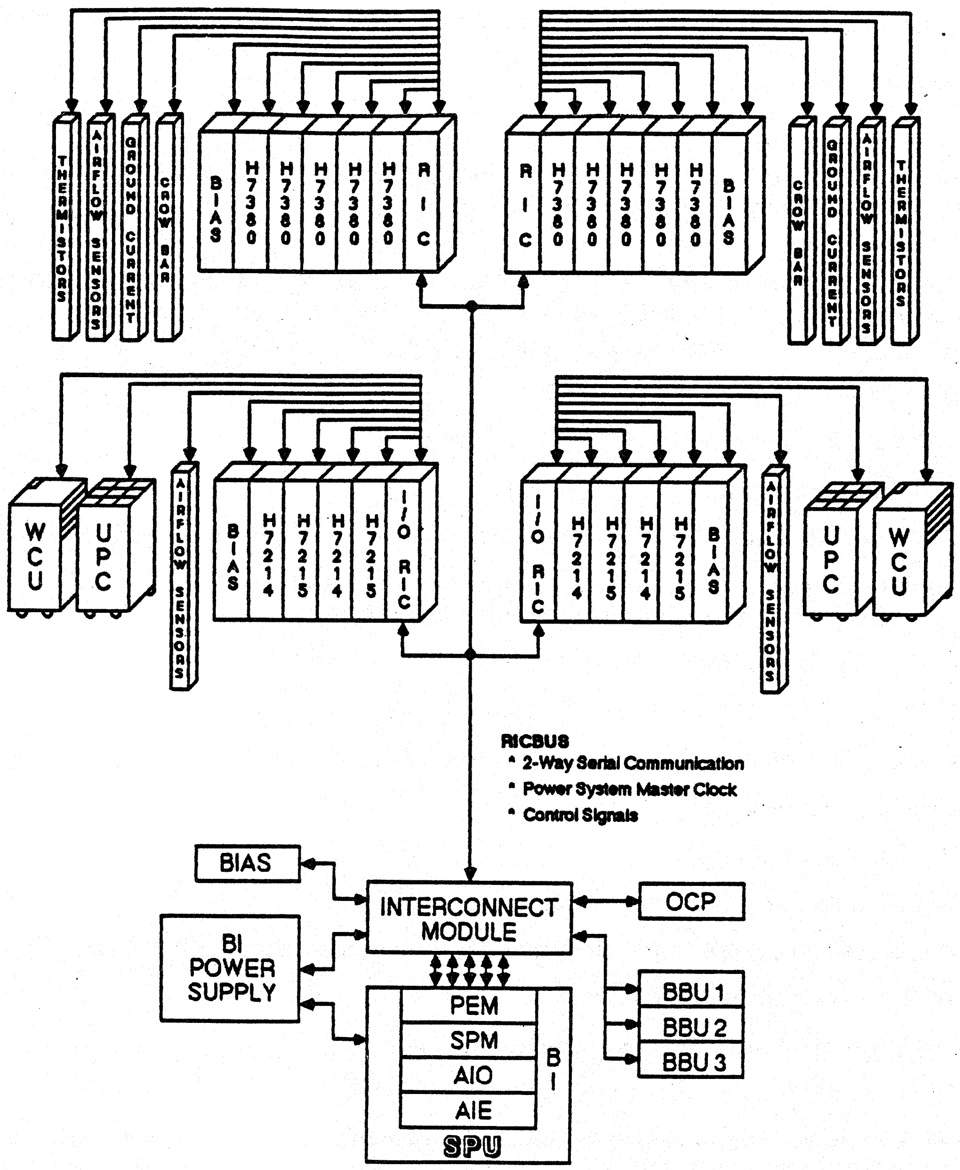


Figure 6-3 PCS Hardware Block Diagram

The PCS consists of a number of H7388 and H7389 microprocessor-based RICs distributed throughout the power system under the control of the PEM. Information exchanged is between the RICs and the PEM using the serial data link of the RICBUS.

The status of a power module, thermistor, or sensor can be determined locally by the RIC and relayed to the PEM over the RICBUS either as a response to a command or through an interrupt. The PEM can command a RIC to power-up or power-down its associated modules, as well as margin the output voltage (i.e., +/- 5%).

The PEM is in turn controlled by the Service Processor Module (SPM) which resides in the SPU (BI) backplane. The PEM communicates with the SPM over a portion of the internal SPU bus (II32 bus). This bus is connected to the PEM through a cable connected to 30 BI Bus user pins on both modules.

The following paragraphs describe the parameters which are monitored and measured by the PCS. However, these parameters may be read by the SPU only if exception reporting is enabled.

NOTE
In this context *monitor* implies that the sensor returns a go/nogo state, and *measure* implies that the sensor returns an actual value (e.g., output from an analog-to-digital converter).

General PCS measurements are listed below. In all cases parameter measurements which fall outside the acceptable limits will cause an exception report to be issued. Those cases which may initiate an ASD are identified.

- All H7380 regulator bus voltages
- Cabinet air temperature; fault may initiate an ASD.
- Ground current
- WCU coolant and cabinet air temperatures; fault may initiate an ASD.

The monitored parameters are listed below. In all cases faults will cause an exception report to be issued. Those cases which may initiate an ASD are identified.

- H7380 regulator GROUP LO status signals
- I/O regulator MOD (module) OK status signals
- Cabinet fan rotation; fault may initiate an ASD.
- Crowbar status signals
- BBU status signals
- WCU coolant level; fault may initiate an ASD
- WCU coolant pressure and flow rate; fault may initiate an ASD
- WCU cooling air temperature
- WCU pump status; fault may initiate an ASD
- WCU cooling fan rotation; fault may initiate an ASD

The PCS monitors the following optional UPC status signals: AC LO, BUS LO, UPC OK, AND UPC INPUT VOLTAGE OK. The PCS also monitors several state sensors. In all cases, changes in status and sensor states will cause an exception to be issued and may initiate an ASD.

- Input phase loss

- Input phase rotation reversed
- Overcurrent
- Thermal warning
- Thermal fault

The PCS performs self tests during power up to verify PCS integrity, and will indicate the failure of any module which fails these tests. It will run and maintain the power and environmental subsystems without SPU intervention. However, the PCS will not initiate a system power up unless commanded by the SPU.

The PCS controls the diagnostic display located on the Operator Control Panel (OCP). The display will indicate the cause of an emergency shutdown. On startup, the last error written to the status/fault display will be read by the PEM be available to the SPU for entry into the error log before the PEM overwrites display content. The PCS also supports an EMERGENCY OFF button located on the rear of the first XMI cabinet. Depressing the button will trip the main system circuit breaker, disconnecting the utility input power.

6.4.1 Power And Environmental Monitor

The main function of the PEM is to monitor and control the power system and system environment, and to provide a communications path between the PCS and the SPU. Power system control is performed through commands from the SPU and internal PEM software.

The PEM will report out-of-limit conditions or status changes to the SPU on an exception basis. The SPU can read the state of the system at any time through commands to the PEM. The PEM also controls the RICBUS power system communications bus.

The PEM resides in the SPU BI backplane, and consists of a custom 8051-driven module for control of the RICBUS, system interface logic, and interfacing logic for the II32 bus. The PEM does not communicate over the BI bus, but connects to the II32 bus of the SPM through 30 of the 180 BI user pins. Interface signals to the power system is accommodated by the other 150 BI user pins.

6.4.2 Signal Interface Panel

The Signal Interface Panel (SIP) is used for both AQUARIUS and ARIDUS systems. It acts a focal point for all signal cables coming from various parts of the power system. All signal lines are concentrated into one cable assembly that attaches to the user pins on the BI backplane for interface to the PEM.

The SIP provides the logic required to support BBU operation, DEC Total Off, SPU power supply control, and SPU power down. The SIP connectors concentrate the following system components:

- PEM Module
- OCP
- SPU power supplies
- Bias power supplies
- CPU and I/O RICs
- Master Clock Module
- BBUs

- Power Interface Panel
- System Total Off Shunt Switch

6.4.3 RICBUS Configuration

The RICBUS provides communications between the PEM and RICs. However, there is little intra-communication between RICs. It includes a single wire serial data bus which uses a CSMA/CD scheme (Carrier Sense Multiple Access with Collision Detection) referred to as XXNET. Each node (i.e., PEM or RICs) on the network uses a loopback scheme to simultaneously receive its own transmissions to ensure that the data was not corrupted by a collision. The XXNET protocol has provisions for recovering from a data collision.

The multiconductor RICBUS provides a serial data communication rate of 19.2K baud. The 160 KHz power system clock, generated in the PEM, is distributed by the RICBUS as a two-wire differential signal using twisted pairs.

Although logically a single bus, the RICBUS physically consists of three cables originating at the SIP and servicing separate cabinets or groups of cabinets. Some signals will be common to all cables, others appear only on some of the cables. However, no single cable is required to carry all signals. This partitioning of signals is required to allow one dual CPU in a quad CPU configuration to be powered off while the other dual continues operating.

Communication over the RICBUS is through one of two forms:

- commands and command responses between the PEM and a RIC, or
- exception messages which are sent asynchronously by a RIC to the PEM.

Commands to a RIC can be initiated either by the PEM firmware or SPU commands to the PEM.

To decouple noise and eliminate ground loops, each RIC is optically isolated from the RICBUS. This requires that the RICBUS include power and ground, derived from the PEM, to operate the RIC receivers, transmitters, and optical couplers. The PEM I/O power supply provides +5 vdc (+/- 5%) and return over two wires (i.e., two 5 vdc lines and two returns). The power line is designated: COM +5V; the return line is designated: COM RTN.

As specified in Table 6-3, the cable is designated according to the *side* of the system it services. A side is defined by the source of the 280 vdc power for the regulators.

Table 6–3 RIC Cable Designations

SIDE	SOURCE	RICS
Side A	from UPC 1	RICs: 12,22,51,52,61,71
Side B	ORed from UPC 1/UPC 2	RICs: 11,21,31,41
Side C	from UPC 2	RICs: 13,23,53,54,62, 72

The multidrop serial data link requires two lines, a data line and a reference line. The data line is driven by open collector devices. The PEM provides a pullup resistor for the data line, and a dc voltage for the reference line. The serial data lines are described in Table 6-4.

Table 6–4 Data Line Descriptions

SIGNAL NAME	SOURCE	DESTINATION	DESCRIPTION
COM DATA	PEM and RICs	ALL RICS	A logic 0 is represented by < 6 Volts; a logic 1 is represented by > 6 volts.
COM REF	PEM	All RICs	This line is the reference level for comparison with COM DATA. It is generated by dividing the PEM +12 vdc in half.

Table 6-5 lists the source and destination for the remaining signal lines, as well as a brief description of the signals functions. Note that the signal destinations are included in the function description column.

Table 6–5 RICBUS Signal Descriptions

SIGNAL	SOURCE	CABLE	DESCRIPTION
COM +5V	PEM	A,B,C	Data line power to all RICs
COM RTN	PEM	A,B,C	Data line power return for all RICs
COM DATA	PEM,All RICs	A,B,C	Serial data line to PEM and all RICs
COM REF	PEM	A,B,C	Data reference line to PEM and all RICs
COM A GRP LO L	PEM	A	Causes DC LO for side A, RICs 12, 22
COM B GRP LO L	PEM	B	Causes DC LO for entire system, RICs 11, 21, 31
COM C GRP LO L	PEM	C	Causes DC LO for side C, RICs13, 23
COM BBU GRP LO L	RIC 41	B	Causes DC LO for entire system, inhibits BBU operation
COM A TOTAL OFF L	PEM	A	Removes DC power to side A, RICs 21, 22, 51, 52 61, 71
COM B TOTAL OFF L	PEM	B	Removes DC power to system, RICs 11, 21, 31, 41
COM C TOTAL OFF L	PEM	C	Removes DC power to side C, RICs 13, 23, 53, 54, 62, 72
COM OFF ALERT L	PEM	A,B,C	Indicates TOTAL OFF condition, inhibits RICBUS traffic, all RICs

Table 6–5 (Cont.) RICBUS Signal Descriptions

SIGNAL	SOURCE	CABLE	DESCRIPTION
COM A BUS LO L	RIC 51 or 71	A	Causes DC LO for side A & BI/XMI CPU's, RIC 52, PEM
COM C BUS LO L	RIC 53 or 72	C	Causes DC LO for side C & BI/XMI CPU's, PEM, RIC 52
COM A ACLO L	RIC 51 or 71	A	Causes Latched ACLO for side A, PEM
COM C ACLO L	RIC 53 or 72	C	Causes Latched AC LO for side C, PEM
COM A LAT ACLO L	PEM	A	Latched ACLO for BI/XMI CPU's, RICs 51, 52
COM C LAT ACLO L	PEM	C	Latched ACLO for BI/XMI CPU's, RICs 53, 54
COM X CLK H & L	PEM	A,C	33KHz clock, dual differential for I/O power supplies, RICs 51,52, 53, 54
COM CLK H & L	PEM	A,B,C	160KHz clock, dual differential for H7380 power supplies, RICs 11, 12, 13, 21, 22, 23, 31, 41

6.4.4 Regulator Intelligence Card

Most direct interfacing with the power and environment subsystems is performed through the RICs. Each RIC receives a unique identity code from its particular backplane slot indicating its assigned functions. The identity code also establishes its address on the serial communication bus.

The RICs interface to the following power and environmental components:

- AC Front End (i.e., PCU and AC-DC Converter, or UPC)
- Cooling cabinet
- CPU Regulator Group (type H7380)
- Console Regulators (types H7214/H7215)
- XMI Regulators (type H7214 and H7215)
- Crowbar Module
- cabinet environment

If a RIC is connected to a group of H7380 regulators it will:

- monitor MOD (MODULE) OK and Over Temperature (OT) signals
- enable/disable the power supplies
- provide a clock
- measure and adjust the output voltage

o provide the reference and feedback amplifier for the Current Mode Control loop.

If a RIC is connected to I/O regulators it will monitor MOD OK, and OT, enable/disable the supplies, and provide a clock.

An I/O RIC monitors all pump and fluid status signals, and measures fluid temperature of the WCU. The UPC RIC monitors the AC LO and BUS LO status signals as well as status signals related to input voltage, output current, phase faults, and thermal faults. In addition, RICs in each cabinet are responsible for monitoring the fan status signals and measuring cabinet air temperature.

6.4.4.1 RIC Identification

Each RIC is identified with an 8-bit address. The high-order four bits designate the RIC type (i.e., which portion of the power system it interfaces to), and the lower four bits identifies the number of the RIC of that particular type. Some addresses are reserved for special use, and one type is set aside for addressing the PEM. Table 6-6 specifies the RIC address and type.

Table 6–6 RIC Addresses and Types

HEX ADDRESS	TYPE
0 0	PEM
1 X	-5.2V regulator, H7380
2 X	-3.4V regulator, H7380
3 X	+5.0V regulator, H7380
4 X	+5.0V BBU regulator, H7380
5 X	I/O regulator, H7214/H7215

Table 6-7 defines the assignment of RICs in a fully configured Aquarius system (i.e., Quad CPU).

Table 6–7 RIC Assignments in Quad Configuration

MODULE NAME	BUS TYPE	BUS	REGULATOR	CABINET
RIC 11	-5.2V	B	H7380	SCU/MEM
RIC 21	-3.4V	C	H7380	SCU/MEM
RIC 41	+5 BBU	A	H7380	SCU/MEM
RIC 31	+5.0V	D	H7380	CPU 1/MEM
RIC 12	-5.2V	J	H7380	CPU 1
RIC 22	-3.4V	K	H7380	CPU 1
RIC 13	-5.2V	M	H7380	CPU 2
RIC 23	-3.4V	N	H7380	CPU 2
RIC 51	I/O	X1	H7214/7215	I/O1
RIC 52	I/O	X2	H7214/7215	I/O2
RIC 53				ARIDUS
RIC 54				ARIDUS
RIC 14				ARIDUS
RIC 24				ARIDUS
RIC 32				ARIDUS
RIC 42				ARIDUS

I/O RICs 51 and 52 monitor WCU 0 and 1, UPC 0 and 1, bias, and air flow.

An I/O RIC monitors several UPC status signals. In addition, a control line is required to trip the breaker in the UPC. The UPC AC LO and BUS LO signals are transmitted to the RIC, which in turn relays the signals to the PEM. The signals pass through an optical isolation stage on the RIC before being transmitted to the PEM.

Table 6-8 describes the status signals which are monitored by the RIC, and the control signals which are transmitted to the UPC by the RIC to initiate control functions.

Table 6–8 UPC Status and Control Signals

SIGNAL NAME	DESCRIPTION
STATUS SIGNALS	
IOUT OK L	When asserted (L) this signal indicates that the UPC output current is less than TBD% of rated current. When deasserted (H) it indicates that the output current is greater than TBD% of rated current.
VIN OK L	When asserted (L) this signal indicates that the AC input voltage to the UPC is within specification. When deasserted (H) it indicates that there exists a condition of sustained undervoltage.
PHASE LOSS H	When asserted (H) this signal indicates that one or more of the three AC input phases has dropped below TBD vac. When deasserted (L) it indicates that all three phases are above TBD vac.
ROT OK L	When asserted (L) this signal indicates that the AC line phase rotation is in the proper sequence. When negated (H) it indicates that the phase rotation is reversed.
THERM WARN H	When asserted (H) this signal indicates a thermal warning condition in the UPC.
THERM FAULT H	When asserted (H) this signal indicates a thermal fault condition in the UPC.
CONTROL SIGNALS	
UPC ENABLE L	When asserted (L) this signal enables the UPC for normal operation. When deasserted (H) UPC operation is disabled.
UPC AC OFF	When asserted this signal trips the AC input circuit breaker to the UPC. To restore UPC power, the breaker is manually reset.

6.4.4.2 H7380 Regulator Interface

Each group of regulators having a common (shared) output will be assigned a single RIC which will measure, monitor, and control the state of the group. If a Crowbar Module is associated with the group, the RIC will also monitor its status. Group state signals are defined such that:

- status signals are monitored by the RIC
- control signals are generated by the RIC to initiate regulator control functions
- measurement signals are analog signals measured by the RIC A/D converter.

The RIC resides in a separate enclosure, communicating with the regulators through a backplane. Power for the RIC will be supplied by the Bias Supply. The Bias Supply will also provide start-up power for the regulators in the group, as well as delivering power to the Crowbar Module.

6.4.4.3 XMI Regulator Interface

The I/O card cages consist of the console BI or XMI backplanes, and are powered by a combination of H7214 and H7215 power modules. A pair of these modules constitute one I/O Power Supply. One pair of power modules is capable of powering one 14-slot XMI backplane or the 6-slot console BI backplane.

An I/O cabinet can have a maximum of four I/O Power Supplies. A single RIC is capable of monitoring and controlling the four I/O supplies. Power for the RIC is supplied through a separate Bias Supply. Table 6-9 describes the status and control interface for each power supply.

Table 6–9 I/O Power Supply Interface

SIGNAL NAME	DESCRIPTION
STATUS - H7214	
MOD OK L	This signal is generated by the regulator and monitored by the RIC. When negated (L) this signal indicates that the output voltage is not within the specified range.
STATUS - H7215	
MOD OK L	This signal is generated by the regulator and monitored by the RIC. When negated this signal indicates that the output voltage is not within the specified range.
OT SWITCH	This signal is generated by the regulator and monitored by the RIC. It is the diode-isolated output of an op amp. When pulled low it indicates that either the +12 vdc rectifier or the -5 vdc rectifier has reached an over temperature condition.
CONTROL	The following two control signal descriptions apply to both the H7214 and H7215 regulators. Both signals are generated by the RIC to control the regulators.
ENABLE L	When asserted (L) this signal commands the regulator to turn off, and resets it to the ready state.
REG CLOCK	This signal is a pulse train of 5 v amplitude at a frequency of 33 KHz. It is used to synchronize the regulators.

6.4.4.4 Crowbar Module Interface

The Crowbar Module will detect an overvoltage condition at the output of a CPU regulator group and will place a short across the output bus. This feature is only required for power busses in the CPU and SCU cabinets. The crowbar signals are handled by the same RIC that controls the associated group of regulators.

Table 6-10 describes the status and control signals. Status signals are monitored by the RIC, while the control signal is generated by the RIC.

Table 6–10 Crowbar Interface Description

SIGNAL NAME	DESCRIPTION
STATUS SIGNALS	
CROWBAR READY H	When asserted (H) this signal indicates that the Crowbar module is functional and that an overvoltage condition has not occurred. When negated (L) it indicates that the crowbar module is not functional or that the crowbar has fired as a result of an overvoltage condition on the output bus.
CROWBAR FIRED H	When asserted (H) This signal indicates that the crowbar has fired as a result of an overvoltage condition on the output bus, or as a result of the FIRE CROWBAR H control signal being asserted. Once the crowbar has fired, it may only be reset by removing power to the crowbar circuit.
CONTROL SIGNAL	
FIRE CROWBAR H	When asserted (H) this signal causes the Crowbar module to fire and short out the DC output bus. Once the crowbar has fired, it may only be reset by removing power to the crowbar circuit.

6.4.4.5 Cabinet Environment Interface

A designated RIC will monitor the environmental state of each cabinet in the system. The designated RIC will monitor the rotation of each fan, and measure the inlet air temperature of the cabinet, and the discharge air temperature of each fan.

The RIC is capable of initiating an Automatic Shut Down (ASD) sequence if it determines that the temperature is outside the acceptable limits, or if TBD fans are not operating. The ASD sequence concludes with the RIC asserting the Total Off Bus which in turn trips the main power system circuit breaker.

Table 6-11 describes the system cabinet environment interface signals, including the WCU cabinet.

Table 6–11 WCU Interface Description

SIGNAL NAME	DESCRIPTION
FAN OK L	When asserted (L) this signal indicates that the fan is rotating TBD1% faster than the design speed. When negated (H) it indicates the fan is rotating TBD2% slower than design speed. TBD1 is greater than TBD2 to introduce hysteresis in the signal.
Cabinet Air	Inlet air temperature and fan discharge air temperature are measured by a negative temperature coefficient thermistor. All thermistors associated with a single RIC have one terminal tied to a common reference voltage (or circuit ground). The other terminals are separately connected to the RIC. The overall measurement accuracy is +/- 0.3C over the range 0C to 50C.
TOTAL OFF BUS	This signal allows the RIC to trip the main system circuit breaker (which resides in the PCU) if it detects a temperature fault or fan failure. To restore power after a total off, the breaker must be manually reset.
Fluid Temperature	The temperature of the cooling fluid is measured in the WCU through immersion-type thermistors. The fluid is sensed at the inlet and outlet of the heat exchanger.
LEVEL OK L	When asserted (L) this signal indicates that the expansion tank water level is normal. When negated (H) it indicates that the height of the water in the expansion tank is low.
PUMP A ON L	When asserted (L) this signal indicates that Pump A is running. When negated (H) it indicates that Pump A is not running.
PUMP B ON L	This signal has the same monitoring function as PUMP A ON.
PUMPS OK L	When asserted (L) this signal indicates that there are no pump failures. When negated (H) it indicates that one or both pumps failed. Identifying the failed pump requires monitoring the states of PUMP A ON and PUMP B ON.
FLOW OK L	When asserted (L) this signal indicates that the fluid flow rate is normal. When negated (H) it indicates that the fluid flow rate is abnormal.
PRESS1 OK L	When asserted (L) this signal indicates that the system pressure is normal. When negated (H) it indicates that the system pressure is lower than TBD. The sensor is located on the upstream side of the filter.
PRESS2 OK L	When asserted (L) this signal indicates that the system pressure is normal. When negated (H) it indicates that the system pressure is lower than TBD. The sensor is located on the downstream side of the filter.

6.4.5 Operator Control Panel

The Operator Control Panel (OCP) is used on both the AQUARIUS and ARIDUS systems. As shown in Figure 6-4, the OCP will control and/or display the following functions.

o Power On/Off

o System Restart

o SPU Access

o System Fault Codes

- CPU and SPU status

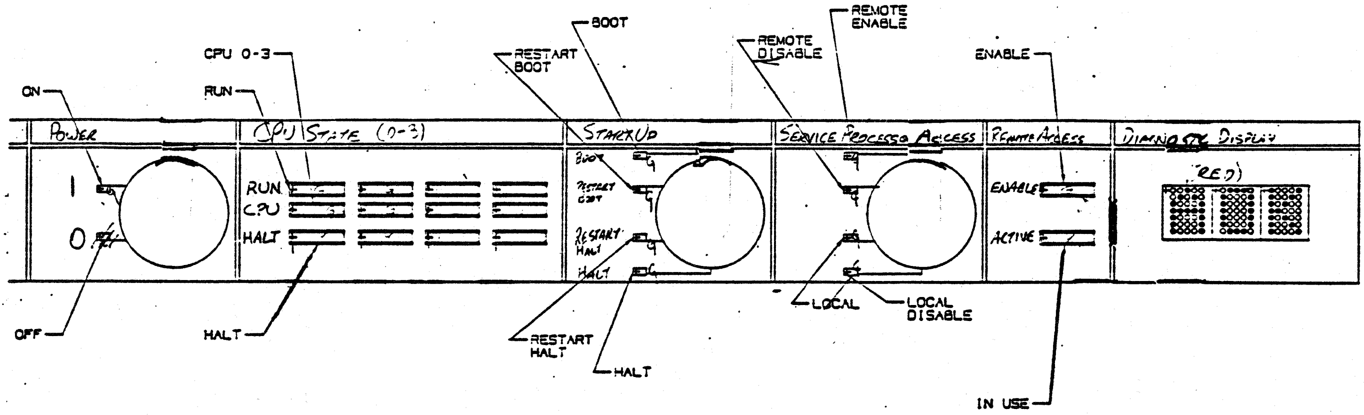


Figure 6–4 Preliminary OCP Layout

6.4.5.1 OCP Keyswitches

The three OCP keyswitches control the following functions:

- Power On/Off
- System Startup Action
- Service Processor Access

The keyswitch functions are described in the following subsections. Included in the descriptions are the BBU Test Switch and System Total Off Shunt Switch. Although not physically part of the OCP, these switches are either logically part of, or monitored by the OCP.

6.4.5.2 POWER Keyswitch

The POWER ON/OFF keyswitch is a 3-pole, 2-position rotary switch which controls system power. The first pole controls the Power Control Bus to apply power to the system. The second pole controls the Battery Back Up (BBU) Interlock. The third pole is used on the OCP to allow the PEM module to monitor the keyswitch position.

6.4.5.3 SYSTEM STARTUP Keyswitch

The STARTUP keyswitch controls the startup action taken by the system when power is applied to the system. The keyswitch is a single pole, 4-position rotary switch which is monitored by the PEM. The keyswitch positions are described below.

BOOT: On power on, the system will attempt a BOOT operation. If the operation fails, the system will enter console IO mode.

RESTART/BOOT: On power on, the system will attempt a RESTART operation. If the RESTART fails, the system will attempt a BOOT operation. If the BOOT operation fails the system will enter console IO mode.

RESTART/HALT: On power on, the system will attempt a RESTART operation. If the RESTART fails the system will enter console IO mode.

HALT: On power on, the system will initialize the processor set and enter console IO mode.

6.4.5.4 Service Processor Access Keyswitch

The SPU keyswitch controls the state of the Console Terminal Port on the system. The keyswitch is a single pole, 4-position rotary switch which is monitored by the PEM. The keyswitch positions are described below:

LOCAL DISABLED: In LOCAL DISABLED the Console Terminal Port operates similar to a VMS terminal passing CNTRL-P as a normal character. This mode allows protection of the console terminal. The Remote Terminal is disabled in this mode.

LOCAL: In LOCAL the Console Terminal can enter console IO mode through CNTRL-P when in program IO mode. The Remote Terminal is disabled in this mode.

REMOTE DISABLED: In REMOTE DISABLED the Remote Terminal can access the operating system only. Access to console IO mode is not allowed, and the Local Terminal is enabled.

REMOTE: Allows the Local and Remote Terminals to act as full-function console terminals.

6.4.5.5 Battery Backup Unit Test Switch

Although the BBU Test Switch is mounted inside of the front door of the XMI 1 Cabinet it is Logically part of the OCP. The purpose of the BBU Test Switch is to provide Field Service with a method of testing the BBU's of the system.

The BBU Test Switch latches a bit on the OCP. Once this BBU Test Switch bit is latched the OCP has this information available to the PEM. This bit can then be reset by the PEM.

6.4.5.6 System Total Off Shunt Switch

The System Total Off Shunt Switch is mounted on the rear of the first XMI cabinet. It is tied to the AC front end by way of the SIP. The OCP monitors this Switch and latches a Total Off bit on the OCP. The OCP then has this information available to the PEM. This bit can then be reset by the PEM.

6.4.5.7 Status LEDs

The OCP has a number of LEDs which display the status of the CPUs, system power, and Remote Terminal Access. The LED functions are described in the following subsections.

Power Status: The Power On/Off keyswitch has a LED which indicates that +5 vdc is applied to the OCP.
CPU Status: There are four sets of three indicators each to display status of the CPUs. Each set of LEDs consist of a Legend, Halt, and Run LED.

o Legend: On system startup the SPU will determine the number of CPUs in the system (i.e., one to four). The SPU will notify the PEM of the number of CPUs, which in turn will illuminate the appropriate CPU Legends .

o Halt: During system startup and following CPUs have been initialization, the appropriate Halt LEDs will be illuminated.

o RUN: Once system startup has been completed and the CPUs are executing macro code, the appropriate Run LEDs will be illuminated.

6.4.5.8 Remote Access Status

The SPU Access keyswitch has two associated LEDs which display status of the Remote Terminal Port. The LED functions are described below.

ENABLE: This LED is illuminated when the SPU Access keyswitch is set to the REMOTE DISABLE position.

ACTIVE: This LED is illuminated when the SPU detects a carrier on the Remote Terminal Port.

6.4.5.9 Diagnostic Display

The diagnostic Display (DD) is a set of three LEDs which displays shutdown and fault information. The DD is powered by a TOY power supply. The TOY is part of an H7231 BBU located in the first XMI cabinet. With power applied to the BBU the TOY draws no power from the batteries. Without power the TOY draws its power from the batteries. Each DD character is addressable by the PEM. Data is written to the DD in ASCII format and is saved in three registers which are also powered by the TOY. Since these registers are battery backed up, the PEM can read them at system startup to store the last shutdown code for entry into the error log.

The DD displays a code whenever the system is shutdown due to a power failure, or when the keyswitch is set to OFF. During PEM initialization the PEM writes each selftest number on the DD before the selftest is run. If the selftest fails then the test number will remain on the DD providing callout of the failure.

6.4.6 PCS Diagnostic Features

The PCS diagnostics consists of startup selftests (SST) and ROM Based Diagnostics (RBD) which are run under the SPU RBD Diagnostic Supervisor. The SSTs will verify the integrity of the PCS. The RBDs will aid in diagnosing and locating faulty PCS modules.

Each RIC has a LED indicator on the OCP. The LED states are specified in Table 6-12.

Table 6–12 LED States

LED STATE	STATE DEFINITION
OFF	This is the initial state, if off for more than 5 seconds after power up the RIC is considered faulty. This occurs when a RIC fails selftest and cannot communicate with the PEM.
Blink at 10 hz rate	The RIC detected a selftest fault and was able to communicate with the PEM, in some cases the fault may be ignored or correctable. In all cases the RIC's status should be available on the Service Processor console terminal.
Blink at a 1 hz rate	The RIC passed selftests but hasn't successfully communicated with the PEM. If this condition persists for more than 10 seconds then there is a communications problem between the RIC and the PEM or the PEM is dead.
ON	The RIC has passed selftests and has communicated with the PEM. This is the normal state of the RIC LED indicator. The RIC has passed all selftests and is able to communicate with the PEM.

Each RIC contains a diagnostic summary register. The register is available to the PEM through the RICBUS. This register contains the pass/fail result of the SSTs. In some cases it is possible for a RIC that fails a SST to communicate the SST to the PEM.

The PEM has two yellow LEDs, one is located on the top of the module and the other is on the front of the module. These LEDs are turned on by the PEM when it completes it's selftests. If these LEDs are not on within 10 seconds after power up the PEM is considered faulty.

The PEM has a diagnostic summary register that can be read directly by the SPU.

6.4.6.1 Startup Selftests

The SSTs consist of a series of tests which in general test the PEM PROMs and RAMs, clock, and RIC Bus loopbacks. In addition, the following RIC components are tested: PROMs and RAMs, clock, and D/A and A/D converters.

6.4.6.2 Rom Based Diagnostics

At this point the RBDs are not completely defined. A later PIP revision will include the detail definitions. The purpose of the RBDs are to provide the user with the capability of exercising the PCS through the PCU.

6.4.7 Power Up Sequence

The following list specifies the PCS power up sequence.

1. The user sets the OCP Switch B to ON. This action applies AC power to the WCU and to the UPC or AC Front End. The WCU will start its power up sequence, and should be fully operational in less than 10 seconds. The AC Front End will power up the high voltage DC bus to 280 volts nominal.
2. The regulator powering the SPU and PEM automatically turns on when the high voltage DC bus is on. At the same time, all Bias Supplies in the system turn on and provide power to the RICs as well as start-up power to the main power supplies. All power supplies under control of the RICs remain off.
3. The PEM and all RICs initiate the powerup selftests which execute in less than 10 seconds. On successful selftest completion each RIC enables it's status LED. A RIC will *not* enable any regulator until the selftest and environmental status has been retrieved by the PEM, and the RIC has been commanded to enable its regulator group.
4. The PEM will read the selftest status, version number, status code and environmental data from each of the RICs. Selftest results and the other data is stored in the PEM until status is requested be the SPU.
5. The PEM waits for the SPU to complete its self test. If the test fails, the PEM will indicate the failure on the OCP and the powerup sequence halts. The PEM will not enable any regulators unless commanded to do so by the SPU.
6. The SPU reads software version numbers, environmental data, and the result of the PCS self tests.
7. The SPU then initiates appropriate action based on PCS selftest failures or environmental faults. These actions may include writing to the local error log and notifying the operator through the CTY.
8. The SPU checks the version numbers returned by the PEM and updates the firmware of the PEM or any RIC if the firmware is out of date.
9. If required, the SPU will download any site-specific parameter limits to the PCS through a command to PEM.

10. The SPU commands the PEM to enable all regulators in the proper sequence, applying power to all logic, memory, and I/O. In turn the PEM reports the powerup command result (success or failure).

NOTE
The PEM will not allow power up into a fault condition.

11. On completion of power up,the PEM enables its keep alive task. That is, it begins to continuously monitor the power system and environment, reporting any limit violations or status changes to the SPU

6.4.8 Hardware Conventions

This subsection describes applicable design conventions.

6.4.8.1 Module Identification

When using letters to identify consecutive devices, the standard Digital alphabet is used. In this abbreviated alphabet, the following letters are omitted because they can easily be mistaken for other letters or numbers: G I O Q. Thus, the following is a consecutive sequence of letters:

A B C D E F H J K L M N P R S T U V W X Y Z

A letter (or letters) is used to identify a group of regulators. A letter (or letters) followed by a slash (/) and a number refers to a particular module within a group. For example, the second module in group B would be identified as B/2. In general, the letter(s) which identifies a RIC is the same as the letter(s) which identifies its associated group.

6.4.8.2 Isolated Logic Interface

This convention is followed wherever logic signals are transferred from one system to another with a requirement for electrical isolation. The isolation consist of an optical isolator which may reside on the transmitter or receiver side.

Two signal wires are required for an isolated signal. When the isolation is on the Transmitter side, the signal wire connects the collector of the Transmitter output device to the input (and pullup) of the Receiver. The reference wire connects the emitter of the Transmitter output device to ground on the side of the Receiver. See Figure 6-5.

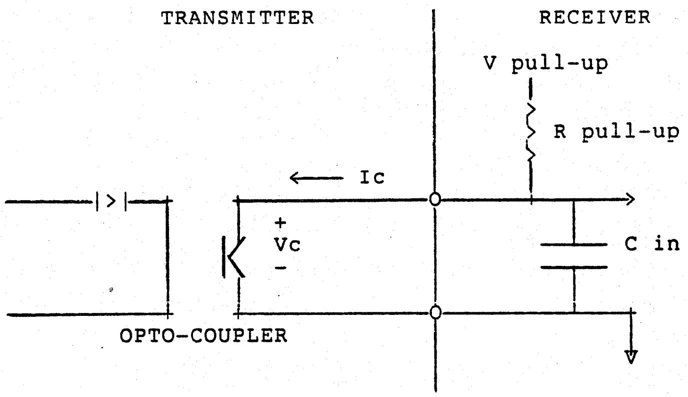


Figure 6–5 Transmitter-Side Isolation

When the isolation is on the Receiver side, the signal wire connects the collector of the Transmitter output device to the cathode of the diode in the Receiver opto isolator. The reference wire connects +12V or +15V from the Transmitter to a load resistor at the Receiver which is in series with the opto isolator diode. See Figure 6-6.

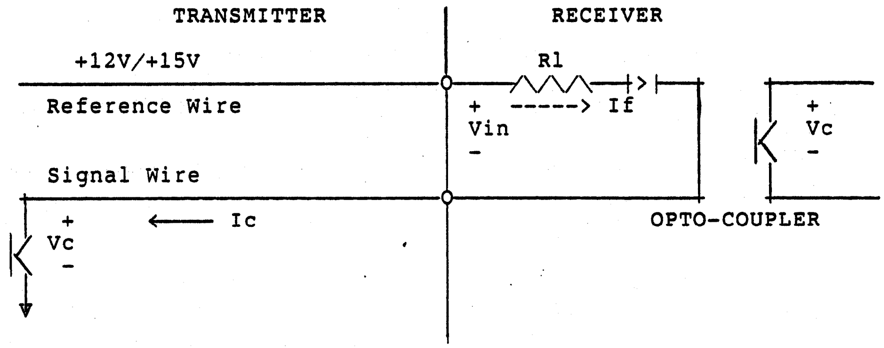


Figure 6–6 Receiver-Side Isolation

Whenever possible, the sense of the signal is such that a disconnected status signal will be interpreted as a NOT OK condition, and a disconnected control line will disable the subsystem it controls.

6.4.9 PCS Software Overview

The PCS software consists of two software control programs: PEM and RIC. Each control program contains code to control its module and all related diagnostics. The programs reside in the PEM and RIC EPROMs and EEPROMs and are not interchangeable (i.e. RIC software will not run on the PEM; PEM software will not run on the RIC).

The PCS has two operational modes: command and exception. When in command mode, the PCS can be considered a polled peripheral. That is, the SPU polls the PEM and each RIC for the state of the power and environmental subsystems by sending commands to the PEM. Polling is inefficient and requires a large amount of SPU resources.

When in exception mode, the PCS can be viewed as a peripheral running in interrupt mode. Each RIC and the PEM run a series of software procedures known as exception mode tasks. When a RIC or the PEM detects a change in the state of a monitored parameter it sends an exception message. The exception message from a RIC is passed over the RICBUS to the PEM. When the PEM detects an exception condition it passes the exception message directly to the SPU.

The following paragraph describes an example of an exception mode task executed by a RIC. In the example, the task measures the coolant temperature and compares it against specific limits. The task determines whether the coolant temperature is out of limits.

The software compares the current state against the last reported state. If the current state and previous state are the same, the RIC takes no action. If the state has changed, the RIC sends an exception message to the SPU through the PEM. The RIC may also take additional action such as starting or canceling an ASD depending on the new state.

6.4.9.1 PCS Self Initialization

During power up the RIC reads its unique ID code from its ID register and then runs its selftests. If the selftests pass, the RIC uses its ID code as an index into a database located in EEPROM, and extracts default parameter limits and register values which relate to the RICs physical location.

The parameter limits and register values are then loaded into the RIC's RAM where the default values can be changed, if necessary, by the SPU through commands to the PEM. Thus, under normal circumstances each RIC is self initialized and the SPU is not required to write parameter limits and register values to each RIC in the system during startup.

6.4.9.2 PCS Initiated Shutdown

The PCS monitors and measures a large number of system parameters. In general, any fault that could present a safety hazard or damage equipment will cause an ASD. Other faults such as voltage out of limit or yellow zone violations will generate exception messages which are informational and are provided to warn of a potential problem.

There are two types of faults that cause the PCS to shut off the system: hardware detected, and software detected faults.

A hardware detected fault is hardwired to the TOTAL OFF BUS wire. An example of a hardware fault is where the OT SWITCH inputs on a RIC (which are connected to a thermal switch located on the regulator) close. In this case, TOTAL OFF is asserted with no intervention from the RIC microprocessor. This causes the main input circuit breakers to open and turn off AC power to the system.

The RIC sends a message to the PEM that contains the RIC ID and an error code that describes the fault. The PEM reads the message and extracts the error code and RIC ID and writes these values to the OCP.

An example of a software detected fault would be when a thermistor transitions into a red zone fault. The RIC that detected the fault starts a timer and sends an exception message to the PEM (if exceptions are enabled). If the timer expires before the fault has cleared, the RIC sends a message that contains the RIC ID and the fault code to the PEM and then asserts the TOTAL OFF line.

Since the RIC has time to send the message before it asserts the TOTAL OFF line, the timing is less critical than for a hardware detected fault. A timer is associated with each software fault that can initiate an ASD. The timer is loaded with a default value during initialization. The timer value can be changed at anytime by the SPU through commands to the PEM.

In a quad CPU system, fault conditions can occur such that TOTAL OFF causes only part of the system to power down. In this case the PEM will write the error code to both the SPU and the OCP status/fault display.

7
SPU and Scan Subsystem Overviews

7.1 Chapter Objective

This chapter provides an overview of the Service Processor Unit (SPU) sub-system and the Aquarius scan system. It discusses four major subjects.

- Service Processor Unit
- Scan Concepts
- Aquarius Scan System
- Scan Control Module

7.2 Service Processor Unit

This section identifies and briefly describes all the major components within the Service Processor Unit (SPU). It treats the SPU as an integral sub-system within the Aquarius computer system.

7.2.1 Purpose

The SPU is a uVAX driven, BI-based subsystem which provides service and maintenance support for the Aquarius computer system. It serves two major functions:

- operator console and initialization controller used to start-up, shut-down, and monitor system operation
- maintenance processor to test, diagnose, and isolate system hardware faults.

The SPU takes an active role in handling, reporting, and recovering from most errors detected by the Aquarius system and provides the following major functionality.

- Supports Uni/Dual/Quad processor Aquarius systems.
- Power system monitoring and control.
- Environmental monitoring (temperature, air-flow, etc.)
- System clock control (start, stop, change frequency).
- Independent local (CTY) and remote (RTY) user access.
- Standard VAX/VMS file system support.
- Aquarius error detection/correction
  - Correct Control Store RAM errors
  - Log main memory single bit errors

  - o Maintain system snapshot log files
  - o Support symptom directed diagnosis
  - o Aquarius Processor error recovery
- o Aquarius system test, diagnosis, and fault isolation using
  - o SPU-based diagnostics that use the scan system
  - o Aquarius-based VAX macrodiagnostics
  - o Console command sets for manual diagnosis

7.2.2 Physical Description

The SPU is mounted in the front of the system cabinet which contains CPU0 and consists of:

- o A BI card cage containing a backplane and five modules:

  SPM - Service Processor Module
  SCM - Scan Control Module
  PEM - Power and Environmental Module
  KFBTA - Disk Control Module
  DEBNK - Tape/Network Control Module

- o RD54 Disk Unit
- o TK50 Tape Unit
- o Two Power Supply Units
- o Connecting Cables

7.2.3 SPU System Block Diagram

The SPU Block Diagram shown in Figure 7–1 shows how all the major hardware components are connected including the major interfaces between the SPU and the Aquarius computer system.

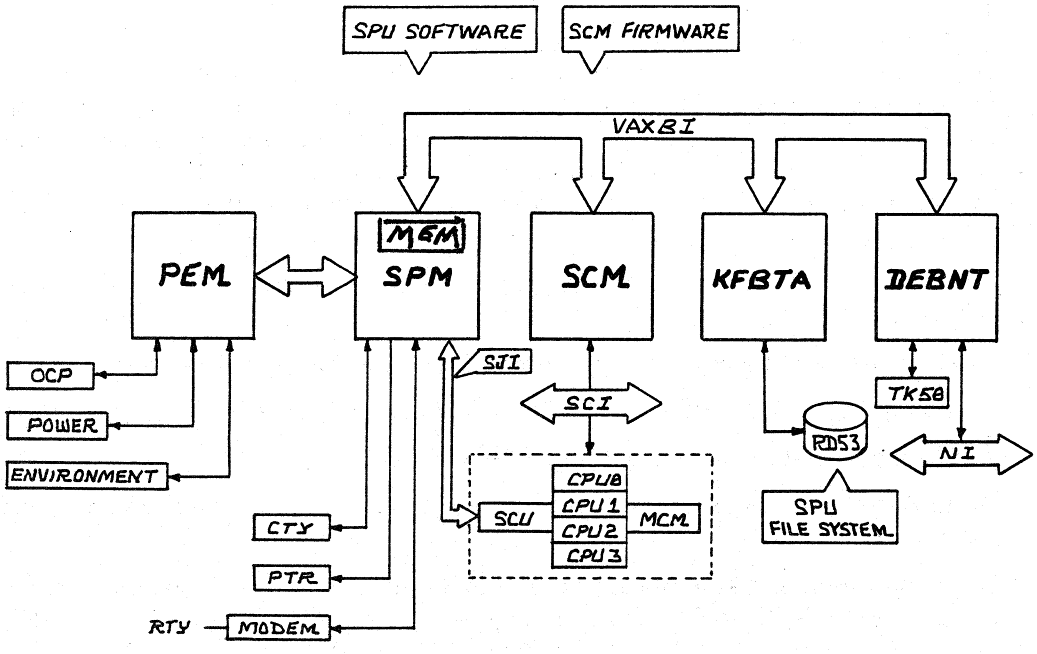


Figure 7-1 SPU System Block Diagram

7.2.3.1 Service Processor Module (SPM)

The SPM is a special purpose uVAX driven controller that contains all the hardware components to execute the SPU software. It is the primary processing element within the SPU sub-system and contains a 16MB daughter board to provide main memory for the SPU. It supports the following interfaces:

- o BI interface to communicate with the SCM, KFBTA, and DEBNT modules.
- o PEM interface to communicate with the PEM module. The SPM uses this interface to sequence turning on the power supplies during system initialization and to monitor the power/environmental sub-system once the system is up and running.
- o SJI (SPU to JBox Interface) to communicate with the SCU (System Control Unit). This interface is interrupt driven and consists of the standard RXCS, TXCS, RXDB, and TXDB registers. In addition, it includes a DMA facility that contains command, length, address, and status registers used to:
  - a. load the primary bootstrap (VMB) into main memory.
  - b. transfer error information from the SPU to the operating system.
  - c. transfer software updates from the operating system to the SPU.
  - d. transfer machine check stack frame information to main memory following a successful error recovery.
- o Three serial ports to communicate with:
  - a. CTY - Local terminal
  - b. PTR - Local logging printer

c. RTY - Full modem support to remote terminal

7.2.3.2 Scan Control Module (SCM)

The SCM is a special purpose, uVAX driven BI adapter, that contains the hardware components and firmware to support the Aquarius scan system. It connects to the SPM via the BI and the scan system via the SCI (SCan Interconnect). It contains a 128KB PROM to store the firmware and self-tests along with a 512KB RAM for local storage. The SCM can select up to six SCI ports to access state elements in the SCU, MCM, and up to four CPUs.

During system initialization, the SCM and its resident firmware use the scan system interface to:

o determine system configuration (number of CPUs, MCU revisions, etc).

o load all required SCU/CPU STRAMs from microcode files in the SPU file system.

o initialize all state elements within the SCU/CPUs.

o initialize the MCM (Master Clock Module) to select the frequency and start the system clocks to all units.

After initialization and system start-up, the SCM monitors system operation using the scan system. All SCU/CPU/MCM errors are signaled to the SCM via an ATTENTION interrupt on the SCI. The SCM responds by using the scan system to:

o determine which unit needs attention.

o which MCU on the unit caused the attention.

o the type of error (MCA, temperature, timing).

o retrieve all related error information.

The firmware then signals the SPU software to service the error based on the information retrieved from the scan system.

7.2.3.3 Power/Environmental Module (PEM)

The PEM is a microprocessor driven controller that contains all the hardware and firmware components to monitor and control the power/environmental subsystems and the OCP (Operator's Control Panel). It mounts in the SPU's BI backplane and communicates directly with the SPM via a separate interface.

7.2.3.4 KFBTA Disk Controller Module

The KFBTA disk controller provides access to the RD53 which contains the SPU file system. It is a standard BI adapter consisting of a 78032 uVAX chip, BI chip set (BCI3/BIIC), and a National DP844 disk controller which supports the standard MSCP disk interface protocol. Implemented as a single-host BI storage systems port, it supports Standard Communication Architecture (SCA). All file transfers to and from the disk are controlled by the SPM and occur over the BI. Refer to the KFBTA Technical Manual (EK-KFBTA-TM-001) for more detailed information on the KFBTA module.

7.2.3.5 DEBNT Network/Tape Controller Module

The DEBNT module provides access to the TK50 tape unit and the Ethernet (NI). It is a standard BI adapter which contains both network and tape interfaces. It consists of a 70832 uVAX chip, BI chip set (BCI3/BIIC), and

o an AMD AM7990 LANCE network controller chip to provide access to the Ethernet.

> **NOTE**
> **The network interface will be used only for prototype debug of the Aquarius system.**

o an Intel 80186 processor and NEC7201 chip to provide the access to the TK50

Refer to the DEBNA/DEBNK Technical Manual (EK-DEBNX-TM-001) for more detailed information on the DEBNK module.

7.2.4 SPU Software

The SPU operating system is a dedicated VAXELN application that resides on the SPM module. It is executed under control of the VAXELN kernel and deals solely with controlling and monitoring the operation of the Aquarius computer system. VAXELN is a real time real memory operating system that provides a memory resident kernel to be used by a dedicated application. It offers VAX/VMS compatible file service and Phase IV DECNET end node network service in addition to the system services required for real time multitasking applications. VAXELN also provides support for VAX/VMS C programs, FORTRAN (ELN V2.3), VAXELN Pascal, and VAX macro.

The kernel provides memory management (VAX P0/P1 mapping) without paging or swapping. The absence of page files and swap files increases system reliability and performance by removing the dependency on disk drives.

The majority of the SPU software is written in VAX C with the remainder written in VAX Macro and VAXELN Pascal. Using VAXELN makes it possible to include service processor functions in the operating system itself rather than executing them as application programs. This allows more consistent functionality across the various console modes of operation.

VAXELN, as used in the SPU, is not intended to be a general-purpose operating system for developing customer applications. Figure 7–2 shows the relationship between the major elements within the SPU software image.

```
------------------------------------------------------
|                   APPLICATION CODE                 |
------------------------------------------------------
| Runtime   | Device  | File    | Network | Debugger |
| Libraries | Drivers | Servers | Servers |          |
------------------------------------------------------
|                     VAXELN KERNEL                  |
------------------------------------------------------
|                     SPU HARDWARE                   |
------------------------------------------------------
```

Figure 7–2 VAXELN System Elements

7.2.5 SCM Firmware

The SCM firmware resides in ROM on the SCM module and permits direct control of the Aquarius scan system via the SCI. The firmware provides the following functions:

- Communicate with the SPU software using VAX BI Port Protocol (BVP).
- Perform normal system monitoring for running CPUs while concurrently allowing scan operations on non-running CPUs.
- Provide high-level functions that allow the SPM to continue processing while major translation and scan operations are being performed on the SCM module. This allows time for the SPM to prepare data for additional transfers to create an overlapped scan I/O environment.
- Provide high-level functions used by the SPM to test and diagnose a faulty machine or unit.
- Provide some degree of service for CPU/SCU unit attention requests (errors), particularly those which can be identified and corrected quickly. Attention requests which cannot be handled by the firmware are passed to the SPM.
- Provide full ROM based self-tests to verify the operation of the SCM module during system power-up.
- Provide support for MCU testers used during manufacturing and MCU repair.

7.3 Scan Concepts

Since Aquarius is the first computer system designed by Digital to implement scan, it's important to discuss what scan is and how it works before describing how it is implemented in Aquarius. This section provides a brief tutorial on the principles of scan. It is not intended to be a complete technical analysis of scan technology or the problems associated with testing digital systems. A brief overview of scan pattern diagnostic testing is also included to describe how scan is used to test and isolate CPU logic faults.

7.3.1 Basic Model

Most digital systems can be represented by the simple model shown in Figure 7-3.

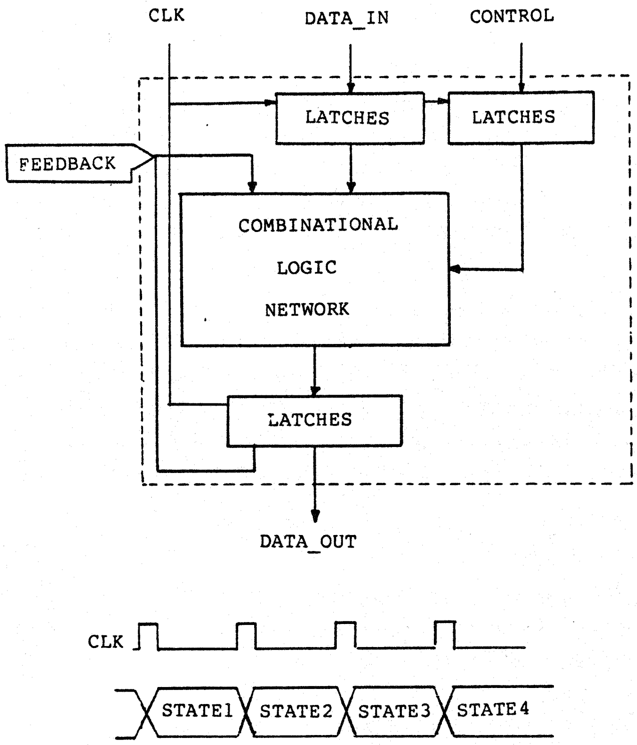


Figure 7-3 Model of a Simple Digital System

Regardless of size and complexity, most digital systems consist of combinational logic networks and bistable latch elements. The combinational logic network may contain thousands of logic gates (AND, OR, and NOT gates) that perform all the required decision making functions. The latches surround the combinational logic and serve as memory elements to temporarily store the:

- o Input data
- o Output data
- o Control information

During any interval of time, as defined by the frequency of the system clock, the state of the system is defined by the state of all the latches. At the occurrence of each CLK pulse, the state of the system normally changes. The four CLK pulses shown in Figure 7-3 would sequence the machine through four unique states, STATE1 through STATE4. At each CLK the next state of the system is determined by four factors:

1. The state of the input latches (DATA_IN)
2. The state of the control latches (CONTROL)
3. The state of the output latches (DATA_OUT)
4. The structure of the combinational logic network

It is important to note the feedback path in Figure 7-3 which implies that the next state of the machine is influenced by the current state of the machine. This feedback is characteristic of any sequential machine and complicates the testing problem.

7.3.2 The Testing Problem

Now let's discuss how the system shown in Figure 7–3 might be tested. Normally, the test programmer only has direct access to the DATA_IN, CONTROL, and DATA_OUT ports. The system is tested by applying binary test patterns to the DATA_IN and CONTROL ports followed by retrieving and checking the result at the DATA_OUT port. If the system produces a faulty output, what caused it? Was it one of the latches or was it within the combinational logic network? Obviously, if the latches are suspect, the test program can't isolate the fault to specific circuit elements within the combinational logic network. About all that might be done is to display the input, output, and expected patterns and leave it up to manual analysis and probing to isolate the failure. This requires an enormous capacity to understand the design of the machine in order to be able to efficiently test and repair it.

Since effective testing must isolate as well as detect problems. The first step in overcoming the testing problems inherent in the model shown in Figure 7–3 is to incorporate some mechanism which allows the test programmer direct access to the latches. This permits testing the latches first before using them to test the combinational logic network. Look at Figure 7–3 again. There's simply no way to test the operation of the latches independent from the combinational logic. This is what *scan* is all about.

7.3.3 Principles Of Scan

Scan is a technique for designing testable logic. It considers the problems of logic testing and fault isolation during the design stage. Basically it modifies the design of the system to add mechanisms which permit direct access and control of the individual latch elements by the test program. Figure 7–4 shows the same model previously discussed that has been modified to incorporate scan.

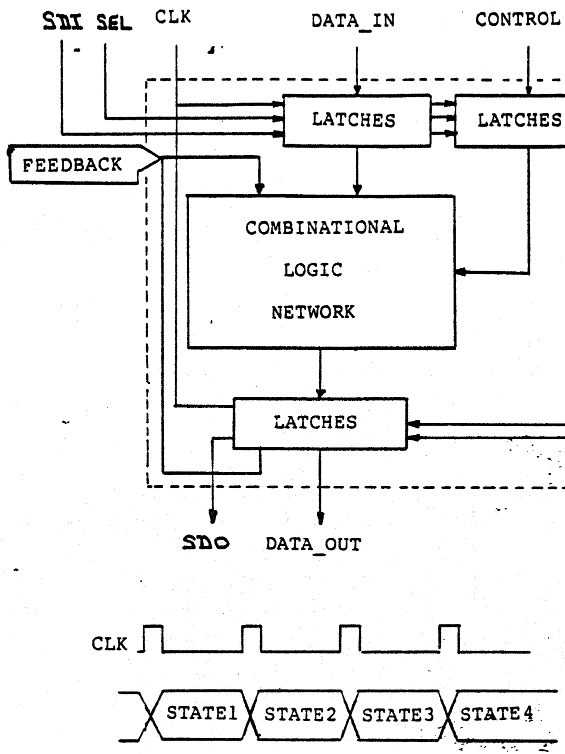


Figure 7-4 Model of a Simple Digital System Using Scan

All the latches are modified to function like parallel-load serial-shift registers connected end to end. This allows reconfiguring the latches into a single giant serial shift register for test purposes. Three additional signals are added to control the latches.

- SEL - modifies operation of the latches to disconnect the normal system data and control inputs and enables the latch to function as a shifter.
- SDI - Serial Data In provides a diagnostic path to shift binary test patterns into the system to establish a known state.
- SDO - Serial Data Out provides a path to shift patterns out of the system to test the results.

7.3.4 Testing With Scan

Assume that the CLK, SEL, SDI, and SDO signals are connected to a special diagnostic test machine. The basic test strategy consists of the following steps.

- Use scan mode (SEL = 1) to test the connectivity of the scan path and the operation of all the scan latches in the path by shifting test patterns in at SDI and out of SDO.
- Determine the optimum set of test patterns required to test the combinational logic network.
- Apply each of the test patterns as follows:

  a. Use the scan mode (SEL = 1) to scan in the test pattern

b. Use the non-scan mode (SEL = 0) along with a system clock to load the output from the combinational logic network into the latches.

c. Use the scan mode (SEL = 1) to scan-out the result pattern and compare it to an expected result pattern.

This is obviously an over simplification of scan technology but it should suffice to explain the principles.

Scan latches can also be used to access RAM structures. Figure 7-5 is a simplified model that shows how scan is used to test memories.

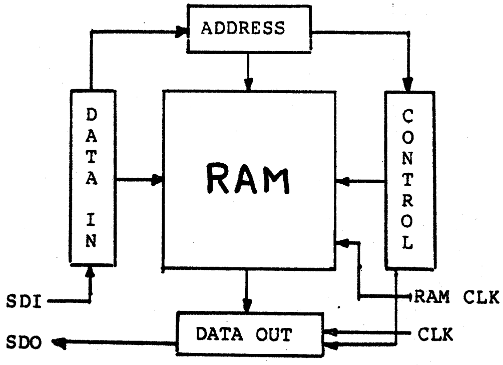


Figure 7-5 Testing Memories With Scan

Normally, to read or write a RAM requires latches surrounding the RAM that:

- Hold the address to be read or written.
- Hold the data to be written.
- Hold the data read out.
- Hold control information to enable the RAM and specify read or write.

Figure 7-5 shows how all these latches are connected to form a serial scan path. SDI and SDO are connected to a programmable tester.

Writing new data into a RAM location requires the following sequence.

1. Select scan mode and scan-in the address to be read, the data to be written, and the control bits to enable writing the RAM. For writing the RAM the contents of the DATA OUT latch is not important.
2. Deselect scan mode and generate a RAM clock to write the data.

To read the contents of a memory location requires the following sequence:

1. Select scan mode and scan-in the address to be written and the control bits to enable reading the RAM. For reading the RAM the contents of the DATA IN and DATA OUT latches is not important at this stage.
2. Deselect scan mode and generate a system clock (CLK) to latch the data from the selected location into the DATA OUT latch.
3. Select scan mode and scan-out the DATA OUT latch to retrieve the contents of the selected location.

Besides memory testing, the scan system can also be used to initialize all the required RAMs at system start-up. In the case of RAMs used as control stores, scan can be used to intervene and correct for intermittent control store parity errors detected while the system is running. Scan can also be used to shift in a reset pattern to initialize all the scan latches in the system, which eliminates the need for separate reset logic.

Figure 7-6 shows another representation of the scan model to show how the state of the system, as specified by the state of all the scan latches, changes at each SYS_CLK.

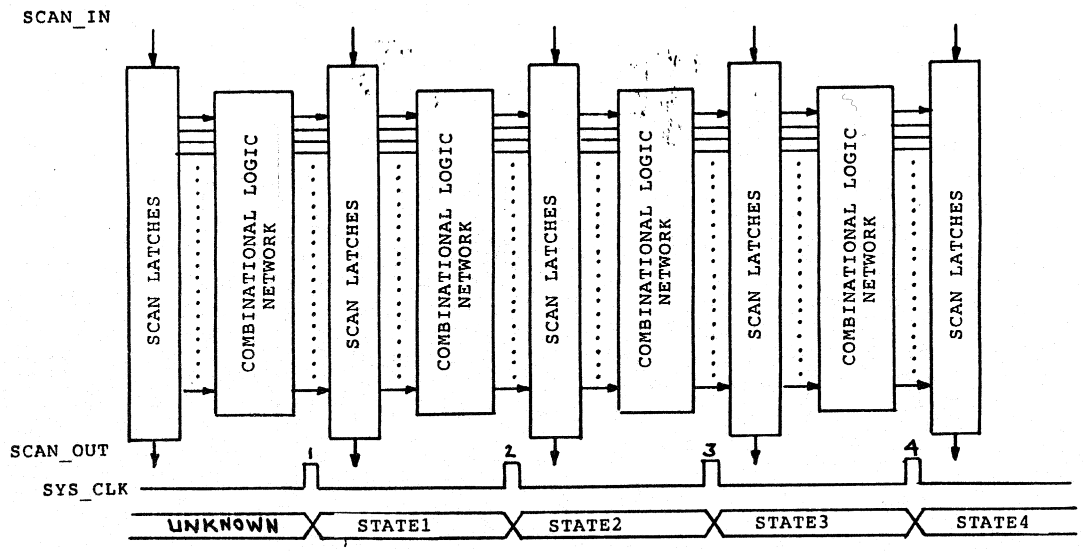


Figure 7-6 Modified Scan Model

Initially, the system is in some unknown state. Data is scanned into the latches to initialize the system followed by the generation of SYS_CLK 1. This places the system in STATE 1. The contents of the latches are then scanned out and verified. The same procedure (scan-in, SYS_CLK, scan-out) is repeated to test the operation of the system for STATE 2, 3, and 4.

Another, more complex test, might initialize the scan latches to a known state, burst the clock four ticks, and then scan-out to verify that the system ends up with its normal output at STATE 4. This is a type of dynamic test that verifies the response of the system at the normal system clock speed.

7.3.5 Scan Latches

As discussed, the key to scan technology is implementing all the latches in the system, critical to system test and fault isolation, as scan latches. This section describes the operation of some typical scan latches used in Aquarius. In effect, the latches really consist of a pair of latches, called A and B to provide a mechanism for raceless shifting. Figure 7-7 shows a schematic of a two-port scan latch along with its macro symbol representation.

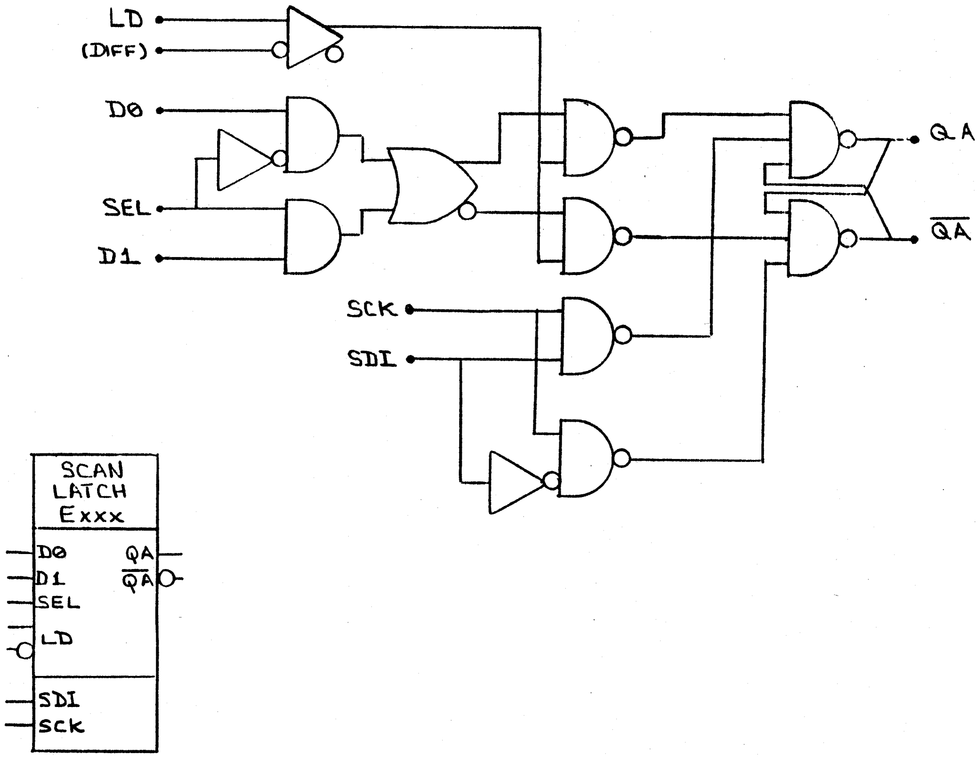


Figure 7-7 Basic Scan Latch

Refer to Table 7-1 for a description of the input-output signals for the scan latch.

Table 7-1 Basic Scan Latch Signal Descriptions

Signal Name	Description
LD	Differential system clock input. When asserted the output assumes the state of the D0 or D1 system data inputs
SEL	Input select signal. When asserted D1 is selected and when negated D0 is selected.
D0	System data input
D1	System data input
SCK	Scan clock input. When asserted, the output assumes the state of the SDI input.

NOTE
If SCK and LD are asserted simultaneously the state of the latch is unpredictable.

SDI	Scan Data Input. Serial data to be shifted in is applied here.
QA	Scan latch output (1)
-QA	Scan latch output (0)

Figure 7–8 shows the actual implementation of a scan latch pair consisting of an A and B latch.

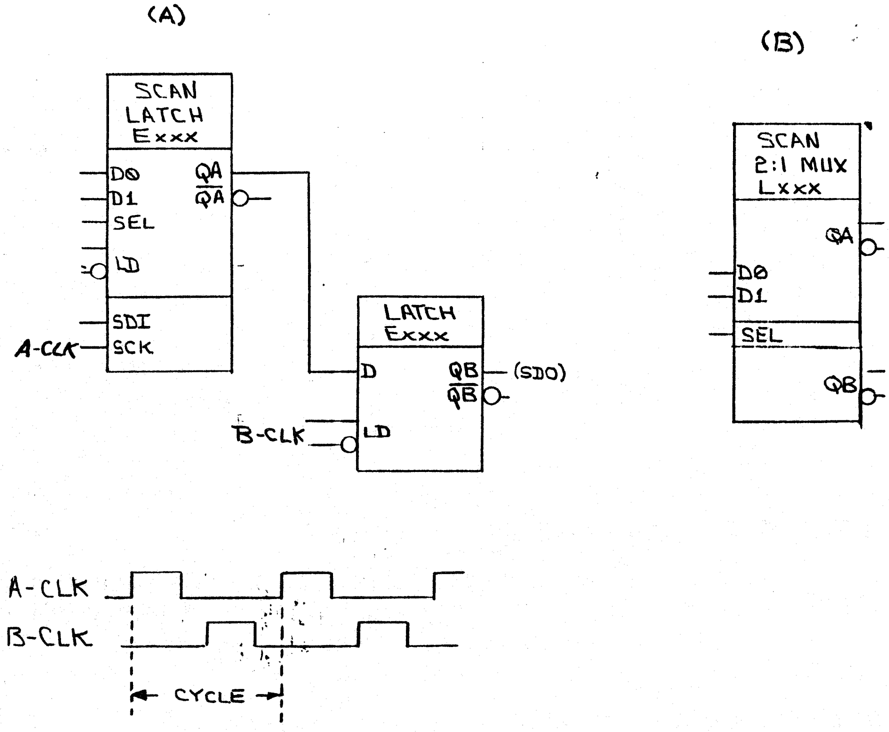


Figure 7–8 Scan Latch Pair

The macro body symbology is shown in (A) and the RTL body symbology is shown in (B). Also included is a timing diagram showing the relationship between the A and B scan clocks. A complete cycle consists of an A-CLK followed by a B-CLK. The A-latch is loaded from the SDI input at A-CLK time and its QA output is transferred into the B-latch at B-CLK time. At the end of any machine cycle (A-CLK followed by B-CLK) both latches are in the same state. The SDO (Scan Data Output) is derived from the QB output of the B-latch. The system designer may use either the QA or QB outputs ,or both, as system data outputs.

It is important to note that the A-CLK is only generated when in scan mode, but the B-CLK must be generated both in scan mode and non-scan mode in order to transfer the A-latch to the B-latch. Note also that asserting both the A and B CLKs simultaneously places the latch pair in a pass-thru state for diagnostic testing of the SDI-SDO chain.

Figure 7–9 shows an implementation of the scan latch pair where the output of the B-latch is fed back to the D1 input of the A-latch. When the HOLD signal is asserted, the latch recirculates and holds its current state.

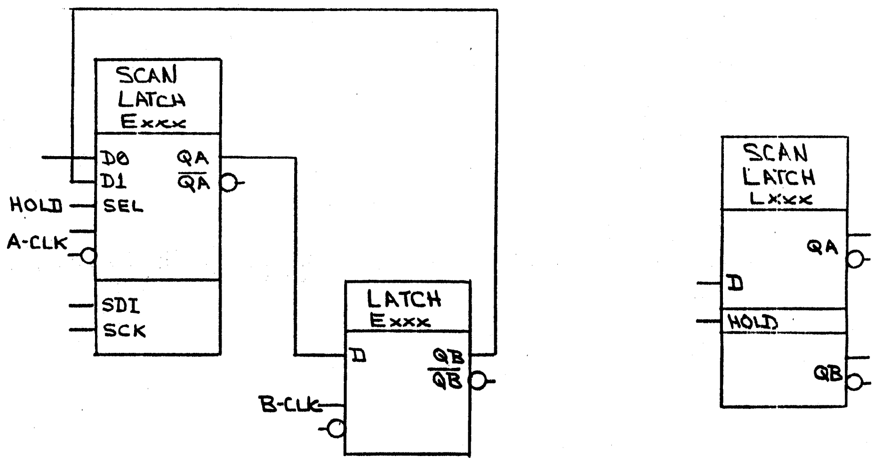


Figure 7–9 Scan Latch With Feedback

The HOLD signal can be used to "freeze" all scan latches required to analyze intermittent errors detected by the hardware. It is asserted by the error detection logic and remains asserted until cleared by scanning in a reset pattern. This prevents the state of the machine from changing until the scan system intervenes to capture the current state.

One final version of the scan latch pair is shown in Figure 7–10.

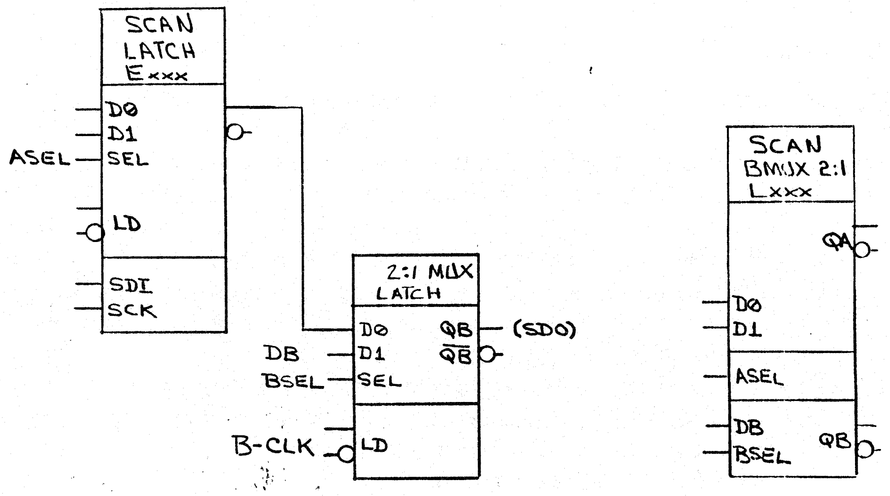


Figure 7–10 Scan Latch With 2:1 B-Latch

Here the B-latch has a 2:1 multiplexer at its input which permits loading it from a source other than the output of the A-latch. If BSEL is asserted followed by a B-CLK, the state of DB is loaded into the B-latch. All other scan latch pairs simply copy A into B with no change in state. This is used to sample the output of RAMs without generating an A-CLK which would change the state of all A-latches in the scan ring.

7.3.6 Scan Pattern Diagnostics

This section describes the fundamentals of the scan pattern diagnostics used to test and isolate hardware faults within the Aquarius CPU and SCU. It begins with a brief discussion of the various types of diagnostics used to test previous members of the VAX family. It emphasizes the differences between traditional approaches to CPU testing and fault isolation and the scan pattern diagnostic technique used in Aquarius. It then proceeds to describe how scan patterns are used to detect and isolate faults to the field replaceable unit (FRU). For Aquarius, the FRU is the Multiple Chip Unit (MCU). Both static and dynamic testing are discussed using simplified examples to show how the technique works.

7.3.6.1 Types Of CPU Diagnostics

There are several possible ways to classify diagnostics. The following paragraphs discuss two ways. First, they are classified according to where the actual test code resides followed by whether the tests are functional or non-functional.

All members of the VAX family of CPUs (excluding uVAX) are supported by two types of diagnostics, microdiagnostics and macrodiagnostics. Microdiagnostics are not implemented for uVAX based systems.

Macrodiagnostics

Macrodiagnostics reside in the system's main memory and provide functional tests of the CPU, that is they verify that the CPU can successfully execute the VAX architecture. When a macrodiagnostic fails, it generally indicates that a fault exists in the CPU hardware. Sometimes the macrodiagnostics successfully call out the most likely FRU that is causing the fault. Quite often though, this type of diagnostic only indicates what test failed, what the result was (ACTUAL), what the result should have been (SHOULD BE), and the difference between the two. The Field Service Engineers must be proficient in analyzing the program listings to extract more information to perform manual fault data analysis. They also need a detailed understanding of the internal workings of the CPU's hardware. To meet the customer's demand for increased system availability, the service engineer often resorts to massive module replacement, a costly process, to repair the machine based on the fault information displayed by the macrodiagnostics. Microdiagnostics are an attempt to provide better fault resolution.

Microdiagnostics

Microdiagnostics generally reside in the CPU's main control store in the EBox and run under the control of a separate diagnostic processor, which also has access to a limited set of visibility points in the CPU's hardware logic. The number of visibility points varies from one machine to another with most points being read-only, that is, the diagnostic processor has limited direct control over initializing individual CPU state elements. The VAX 8700, for example, has just over 150 visibility points, while the VAX 86xx provides over 3000 visibility points. In the the Aquarius CPU, scan provides access to over 20,000 internal machine state elements for both reading and writing.

Since microdiagnostics can be coded to provide more direct control of the logic elements in the CPU and the diagnostic processor can examine key state elements, it is possible to achieve better fault isolation than macrodiagnostics. Again though, if the microdiagnostics fail to isolate the fault, the service engineer must perform manual isolation that requires an intimate knowledge of both the hardware logic and the internals of the microdiagnostic program. Microdiagnostics must also be hand coded which requires the programmer to have an enormous capacity for understanding the machine. This fact often results in increased costs and schedule slips during development.

Both types of diagnostics, micro and macro, often fail to provide the desired fault resolution because of fault propagation. Propagation causes the fault to spread across module boundaries because both types of diagnostics require executing many machine cycles before they can check the results. For example, a typical macro test may execute several instructions, each requiring one or several CPU cycles, before the results are analyzed. A micro test suffers from the same problem, it must execute several micro instructions before it analyzes the result. Obviously, the way to solve this problem is to have some technique that can stimulate the machine and examine the results after just one machine cycle. Also more direct control over most of the internal logic elements must be provided. Scan design, coupled with scan pattern diagnostics residing in an independent diagnostic processor, provide this solution for Aquarius.

Another way to compare diagnostic testing is to classify them as functional or non-functional.

Functional Tests

Functional tests stimulate the CPU logic by executing the actual instructions the machine is designed to perform. Macrodiagnostics are classified as functional tests. VMS is the ultimate functional test, but unfortunately it does not provide ultimate fault isolation.

Non-functional Tests

Non-functional tests stimulate the CPU logic by executing microcoded instruction sequences that shuffle data through the internal address, data, and control paths. The actual microsequences generated are not the same as those generated by the macrodiagnostics. If properly designed, they come close, but not necessarily exact. This explains why we need both micro and macro level tests.

Scan pattern diagnostics are truly non-functional. They are designed to test the physical structure of the hardware logic. They really don't care if the machine was designed to execute VAX instructions, PDP 11 instructions, or even PDP8 instructions. The criteria is the physical design and structure of the hardware logic, not the system architecture. Perhaps a better term to describe scan pattern testing is structural rather than non-functional. What we call them is not important, the important point is that they are very different from micro and macro diagnostics.

7.3.6.2 General Theory

This section discusses the basic idea behind scan pattern diagnosis and fault detection to show how it is possible to use the Aquarius scan system to isolate faults to a replaceable unit (RU). The patterns are initially designed to be used on an MCU tester to test and isolate faults to an MCA or group of logic elements within the MCA. The patterns used for system test in the field are modified to provide MCU isolation. The following paragraphs will use the generic term RU which could mean MCU, MCA, or circuit.

7.3.6.3 Fault Detection and Isolation With Scan

Figure 7–11 provides a graphic aid to explain the principles of fault detection and isolation using scan pattern diagnosis. Each column represents the state of the CPU after any one of six possible clocks (SYS_CLK). The small boxes labeled RU represent all the combinational logic and non-scan latches, while the long rectangles represent all the scan latches. At each SYS_CLK the system data at the output of the RUs are loaded into the scan latches causing a change in machine state. The inputs to the combinational logic networks (RUs) come from the same scan latches. This represents the inherent feedback that exists in any digital system.

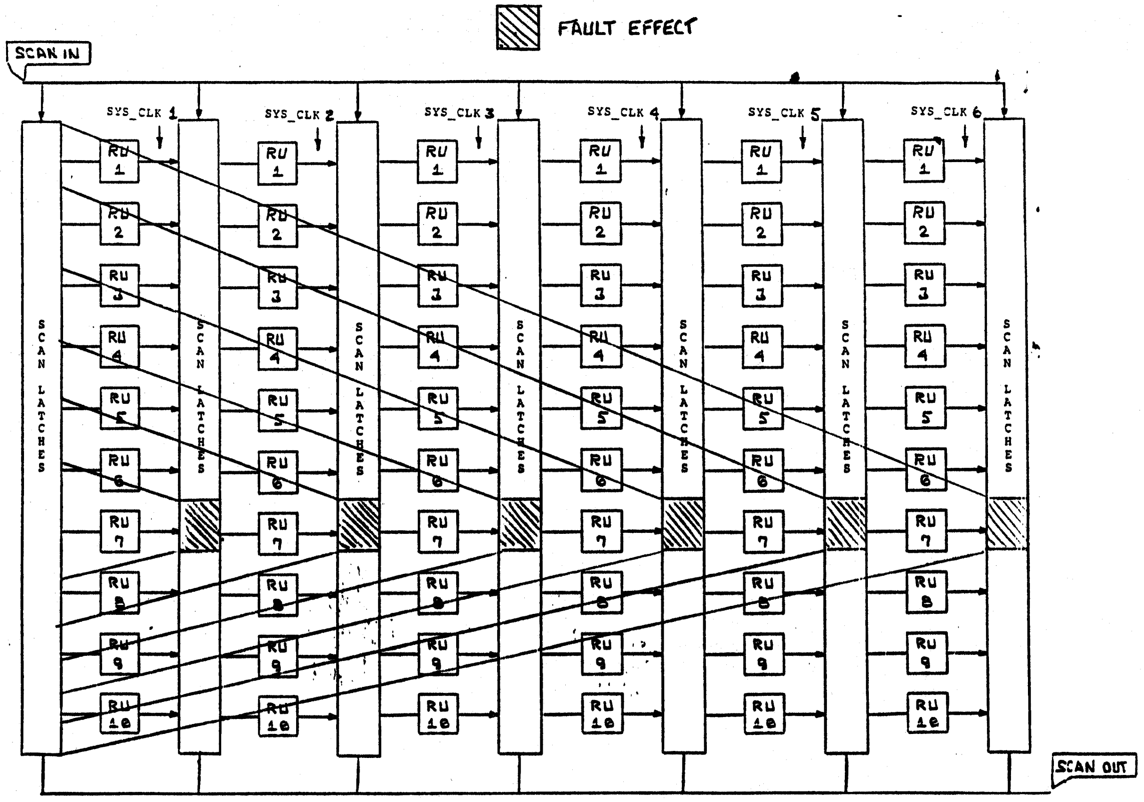


Figure 7–11 Fault Detection/Isolation Model

The basic sequence of events during scan pattern testing is as follows.

1. Stop the system clock
2. Scan in a test pattern to place the CPU in a known state (over 20,000 bits).
3. Generate a single SYS_CLK to load the scan latches with system data from the RUs.
4. Scan out the result pattern (all 20,000 latches).

   NOTE
   This 20,000 figure is simply an estimate. Aquarius test patterns may contain more or less than this.

5. Compare the result pattern with an expected pattern to determine any deviations.
6. The deviation corresponds to one or more bits that represents the fault effect (FE)
7. Relate the FE to a set of possible failing RUs and save the set.

This set of failing RUs is shown as a triangle in the figure. This is called a physical cone. For each FE there exists a physical cone that contains all the possibilities. Each pattern is associated with a fault effect and a cone. Look at Figure 7–11 again. Note how six FEs and six cones are shown. The area of the cone increases adding more RUs as the number of system clocks are increased before scanning out the result. If six SYS_CLKs were generated before scan out, the cone contains all 10 RUs. Not very good isolation. As the number of clocks is reduced, the area of uncertainty decreases, containing less RUs. If only one clock were issued, the cone contains only RU7 and part of RU6, which is much better isolation. This clearly demonstrates the idea of fault propagation discussed previously in micro and macro diagnostic techniques.

Figure 7-12 shows how sequencing and relating the results of three scan patterns can resolve the fault to a single RU. The technique used involves what is called cone intersection. It works like this.

1. FE1 represents the results of pattern 1 clocked at SYS_CLK 1 and is associated with a physical cone containing RU5, 6, 7, and 8.
2. FE2 represents the results of pattern 2 clocked at SYS_CLK 2 and is associated with a physical cone containing RU3, 4, 5, and 6.
3. FE3 represents the results of pattern 3 clocked at SYS_CLK 3 and is associated with a physical cone containing RU6, 7, 8, and 9.
4. The three cones are compared to arrive at an intersection that includes only RU6 which must contain the fault.

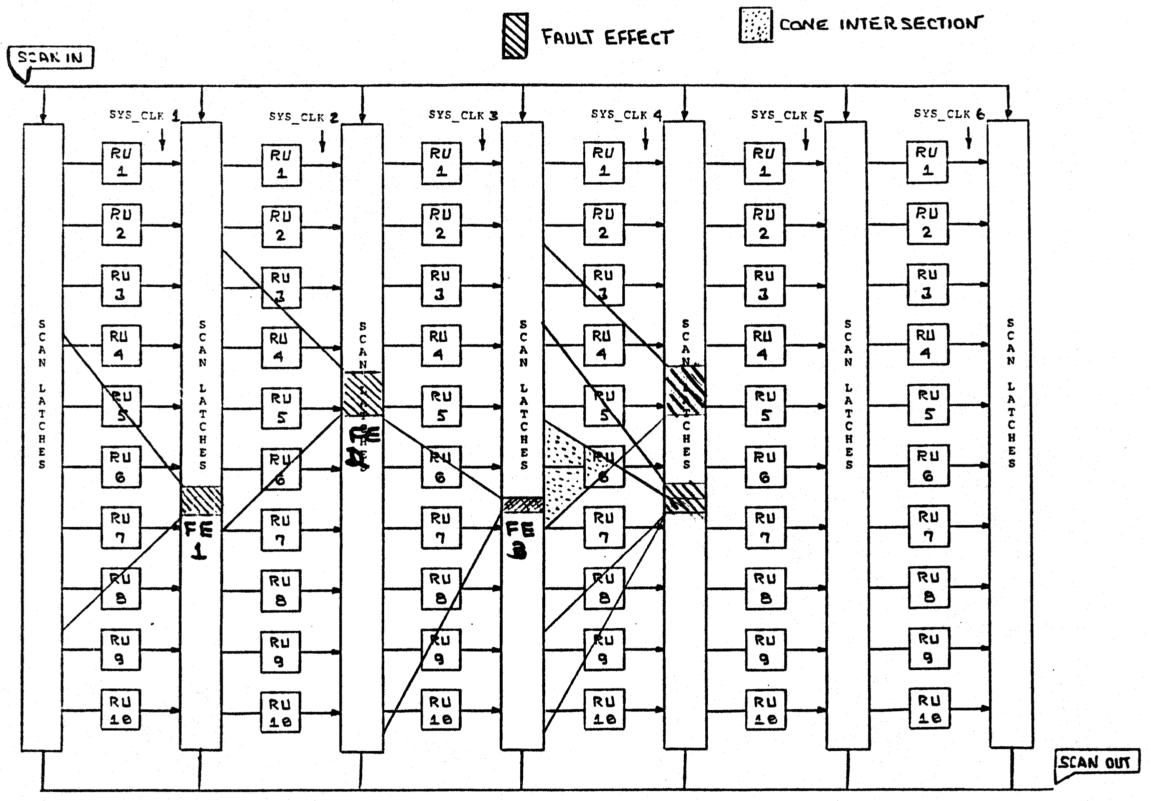


Figure 7-12 Physical Cone Intersection

7.3.6.4 Pattern Generation

During the design of the Aquarius system the test patterns, FEs, and physical cones are derived from the design data base and logic simulation techniques. An Automated Test Pattern Generation process (ATPG) is used to generate the actual test patterns and associated cones. It is beyond the scope of this discussion to describe the details of ATPG and is really not needed at the user level. The patterns and cones produced by the ATPG process are designed to execute on a special MCU tester and can resolve faults to the MCA and circuit level.

These patterns are post-processed to generate the MCU level isolation patterns required by field service at the system level. These patterns are stored on the Service Processor Unit local disk and used by the scan pattern diagnostic control and analysis software contained within the Aquarius SPU.

7.3.6.5 Static Testing

The goal of static testing is to generate the optimum number of scan patterns to detect and isolate all stuck-at-1 (SA1) and stuck-at-0 (SA0) faults which could occur in the CPU logic. The faults could be caused by either a defective logic element or an open or short in the connections between the elements. Figure 7-13 shows a simplified example of how scan pattern diagnostics perform this function.

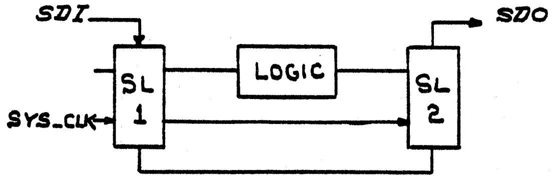


Figure 7-13 Static Testing - Single Clock

To test the logic between SL1 and SL2 requires four patterns.

Scan in 00 - SYS_CLK - check result for 0 in SL2
Scan in 01 - SYS_CLK - check result for 0 in SL2
Scan in 10 - SYS_CLK - check result for 1 in SL2
Scan in 11 - SYS_CLK - check result for 1 in SL2

Obviously, the actual Aquarius scan patterns are much more complex but must achieve the same results.

Figure 7-14 shows another simple example that includes a non-scan latch between SL1 and SL2. Testing this configuration requires two ticks of the system clock to get the data from SL1 into SL2, but the basic principle is the same.

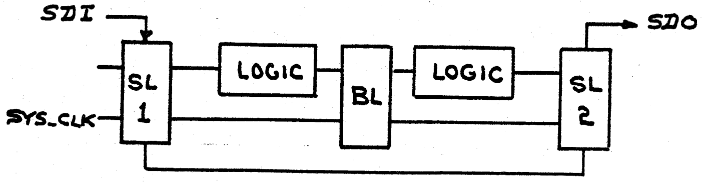


Figure 7-14 Static Testing - Multiple Clocks

7.3.6.6 Dynamic Testing

Static SA1 and SA0 faults can normally be detected with single cycle/multiple pattern tests. Timing problems caused by logic elements that are "slow to rise" or "slow to fall" don't manifest themselves in static testing because the CPU logic has sufficient time to respond correctly before the SCM scans the result data back into the SPU. Timing problems caused by circuit delays may not be detected by this type of test. Different patterns must be generated to detect logic faults caused by timing problems. To overcome this problem requires bursting the system clock two or more ticks. Figure 7-15 illustrates this technique.

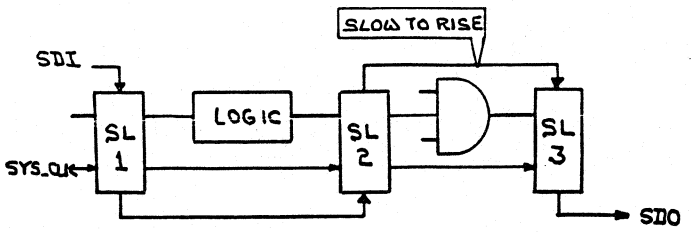


Figure 7-15 Dynamic Testing

Assume that the AND gate between SL2 and SL3 is "slow to rise". To detect this fault, a pattern has to be scanned in that will condition SL1=1, SL2=0, and SL3=0. After scanning in the pattern, SYS_CLK is bursted to generate two successive clocks at the normal system clock rate. This requires the AND gate to respond fast enough to ensure the "1" in SL1 ends up in SL3. If the "slow to rise" fault manifests itself a "0" is found in SL3 after scanning out the result.

7.3.6.7 Multiple Adjacent Faults

All of the techniques just discussed work well for single faults and certain multiple fault conditions. There is, however, a special case of multiple faults that leads to incorrect RU callout. Figure 7-16 illustrates an example of this special case called multiple adjacent faults.

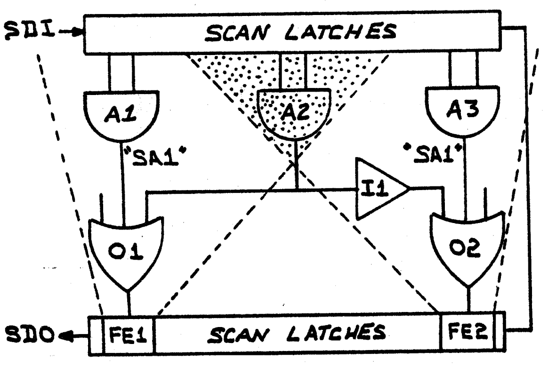


Figure 7-16 Multiple Adjacent Fault Isolation

Assume a bolt of lightning has just struck the Aquarius CPU causing both A1 and A3 to fail high (Stuck At 1). Scan pattern diagnostics are run and two intersecting cones are found.

- o Cone one includes RUs A2, A3, I1, and O2
- o Cone two includes RUs A1, A2, and O1

RU A2 is located at the intersection of cone one and cone two. The program displays the result which directs the service engineer to replace A2. Guess what happens when the service engineer replaces RU A2 and reruns the tests. Right! The program calls out A2 again. This is an isolated, but possible situation, which can occur and requires some type of manual analysis to isolate the multiple faults at A1 and A3. Careful logic partitioning and scan latch placement, along with proper pattern generation helps to avoid ,but can't prevent, multiple adjacent faults from displaying misleading RU callout.

7.4 Scan System Overview

7.4.1 Scan System Block Diagram

The Aquarius scan system shown in Figure 7-17 consists of a combination of software, firmware, and hardware that controls access to the scan latches contained in each of the four CPUs, the SCU, and the MCM. The following sections identify and describe all the major components.

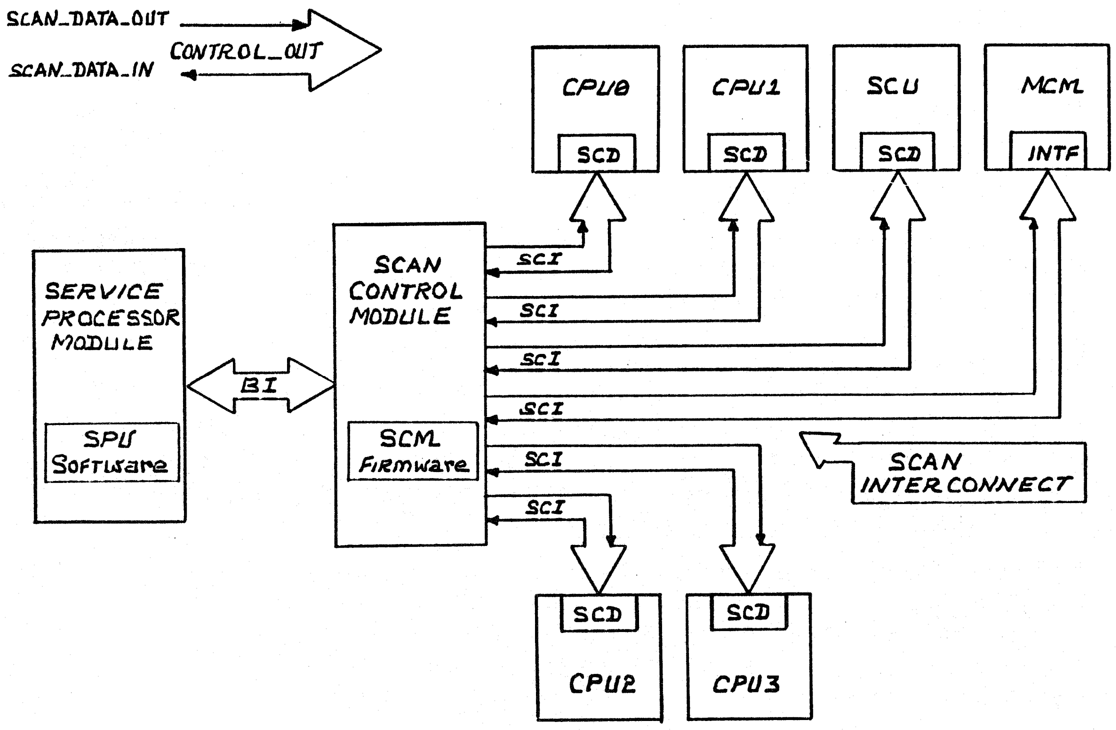


Figure 7-17 Scan System Overview

7.4.1.1 SPU Software

The SPU software resides on the Service Processor Module and contains all the programs and data files required to access the scan system during system start-up, initialization, error handling and recovery, and system testing and diagnosis. It is also the primary user interface to the scan system. The SPU software communicates with the SCM firmware via the BI using command and response queues stored in the SPM's main memory.

7.4.1.2 SCM Firmware

The SCM firmware resides in a ROM on the Scan Control Module (SCM) and provides independent control of the scan system latches in the CPUs, SCU, and the MCM via the SCan Interconnect (SCI). It uses signal and STRAM descriptor tables stored in SCM local RAM to access the scan latches. The local SCM RAM is also used to buffer scan ring information to and from the scan latches.

7.4.1.3 Scan Control Module

The Scan Control Module is an Aquarius-specific BI adapter that contains all the hardware to provide access to the scan system latches in the CPUs, SCU, and MCM. It is driven by a uVAX chip under the immediate control of the resident SCM firmware. The heart of the SCM is the Scan Control Chip (SCC), a custom gate array, designed to perform the following functions:

- o Read scan latch rings into SCM local RAM
- o Write scan latch rings from SCM local RAM
- o Compare ring patterns read from the scan latches with expected patterns stored in SCM local RAM

- Compare ring patterns read from two or more CPUs (XOR testing) and store results in SCM local RAM
- Load CPU/SCU STRAMs during system initialization
- Control operation of the Master Clock Module (MCM) (start, stop, change frequency, burst, etc..)
- Respond to attention signal interrupts (CPU/SCU errors) generated by scan system

Scan system operations in the SCM are overlapped with operations occurring in the SPM. For example the SCM can be retrieving and formatting scan ring information from the CPU while the SPM is processing information retrieved by a previous SCM operation.

7.4.1.4 SCI

The SCM connects to the scan latch rings in the CPUs, SCU, and MCM via the SCan Interconnect (SCI). There are six SCI cables connected to the SCM module, one each for the four CPUs, the SCU, and the MCM. Five of the six cables (CPUs and SCU) are identical and consist of 30 lines (13 differential signal pairs, 3 grounds, and 1 reserved). The MCM SCI cable differs in that it only carries 6 differential signal pairs.

During normal operation each SCI port is independently selected by the SCM with only one port selected at any one time. The exception to this is the four CPU ports. During system initialization, it is possible to scan out the same ring information to two or more CPUs simultaneously if there are no hardware revision conflicts between CPUs. During CPU XOR testing two or more CPUs can be scanned at the same time.

7.4.1.5 SCD

The SCan Distribution MCA (SCD) provides a standard interface in each of the four CPUs and the SCU. It distributes the SCI signals to the MCUs on the CPU and SCU planar modules. In the CPU the SCD is located in the EBox. It is a logical MCA contained within the RLOG MCA in the INT MCU. A similar interface is located on the (TBD) MCU in the SCU. The MCM contains an equivalent but simpler interface.

7.4.2 SCI Signal Descriptions

The SCan Interconnect (SCI) consists of 13 signals that initiate and control scan operations between the SCM and the scan latches located in each CPU, the SCU, and the MCM. Table 7–2 lists all the signals by name along with a brief description of their functions.

Table 7-2 SCI Signal Descriptions

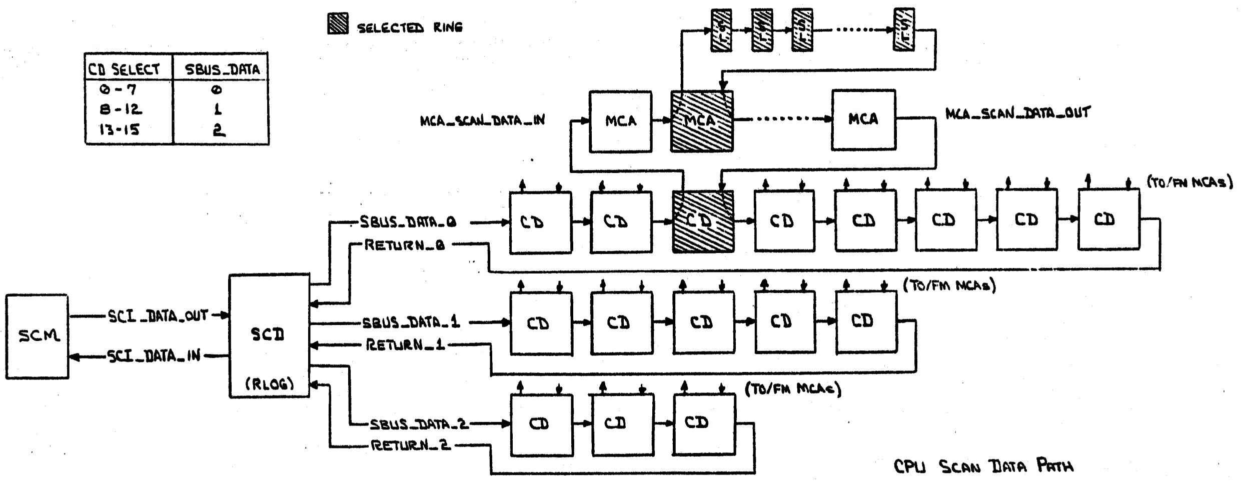
Signal Name	Description
SELECT[3:0]	Serves two functions: 1. Provides the address of a scan ring to select for reading (scan in) or writing (scan out). When CDS is asserted, SELECT[3:0] specifies the MCU address which is latched in the SCD and decoded to generate the required CD select signal. When CDS is negated, SELECT[3:0] specifies the address of an MCA scan ring or an internal CD ring. 2. Provides an enable mask for selecting STRAM clocks during the STRAM LOAD function.
FCT[1:0]	Specifies one of four possible CD functions. 00 - NOP/ATTN (DATA_OUT=ATTN if CD not selected) 01 - Scan shift operation 10 - LOAD 11 - STRAM LOAD
A_CLK	Loads the A phase latches in the scan ring.
B_CLK	Loads the B phase latches in the scan ring.
DATA_IN	Sends scan data from the SCD to the SCM.
DATA_OUT	Returns scan data from the SCM to the SCD.
BDCST	Enables all CD select lines during a broadcast function. It also enables SCD diagnostic functions.
CDS	Enables latching the CD select ID (MCU address) present on the SELECT[3:0] lines. When asserted, all CD select lines are disabled to enable all the CDs to drive the attention line. CDS also enables the SCD loopback function for testing scan signal connectivity between the SCM and the SCD. During loopback all the SCI signals (except CDS) are ORed on to the DATA_OUT line.
BYPASS	Used to enable bypassing the A/B scan latch pair in the DATA_OUT return path reducing the time to shift each bit from the SCD to the SCM. These latches are required for timing at the 100ns/bit shift rate. The bypass function allows the interface to run at slower rates without the extra bit delay. BYPASS also permits the SCM to sample the attention signal (on the DATA_OUT line) without issuing scan clocks.

7.4.3 CPU Scan Paths

The following two sections describe the scan signal distribution within the CPU. The signal distribution is identical for all four CPUs. For clarity, the scan signal distribution is described in two parts, first the data path followed by the control path.

7.4.3.1 CPU Scan Data Path

Figure 7–18 shows how the SCI data lines are distributed from the SCM to the scan latch rings within the MCAs. Each CPU contains 16 MCUs with each MCU containing a single CD chip. The SCD receives the scan data from the SCM on the SCI_DATA_IN line and returns scan ring data to the SCM on the SCI_DATA_OUT line. For ease of explanation, only the scan data signals are discussed in this section. The next section describes the control signal distribution.



CD SELECT	SBUS_DATA
0 - 7	0
8 - 12	1
13-15	2

Figure 7–18 Scan Signal Data Path

First, note that the 16 CDs within the CPU are connected to form three major CD loops. The SCD drives a single-ended bus called the SBUS that distributes the SCI_DATA_OUT line to one of the three major loops depending upon which CD is selected.

- o Loop 0 contains eight CDs. SBUS_DATA_0 provides the scan data from the SCD to the first CD in the loop while RETURN_0 provides the return path from the last CD in the loop to the SCD, for CD_SELECT<7:0>.
- o Loop 1 contains five CDs. SBUS_DATA_1 provides the scan data from the SCD to the first CD in the loop while RETURN_1 provides the return path from the last CD in the loop to the SCD, for CD_SELECT<12:8>.
- o Loop 2 contains three CDs. SBUS_DATA_2 provides the scan data from the SCD to the first CD in the loop while RETURN_2 provides the return path from the last CD in the loop to the SCD, for CD_SELECT<15:13>.

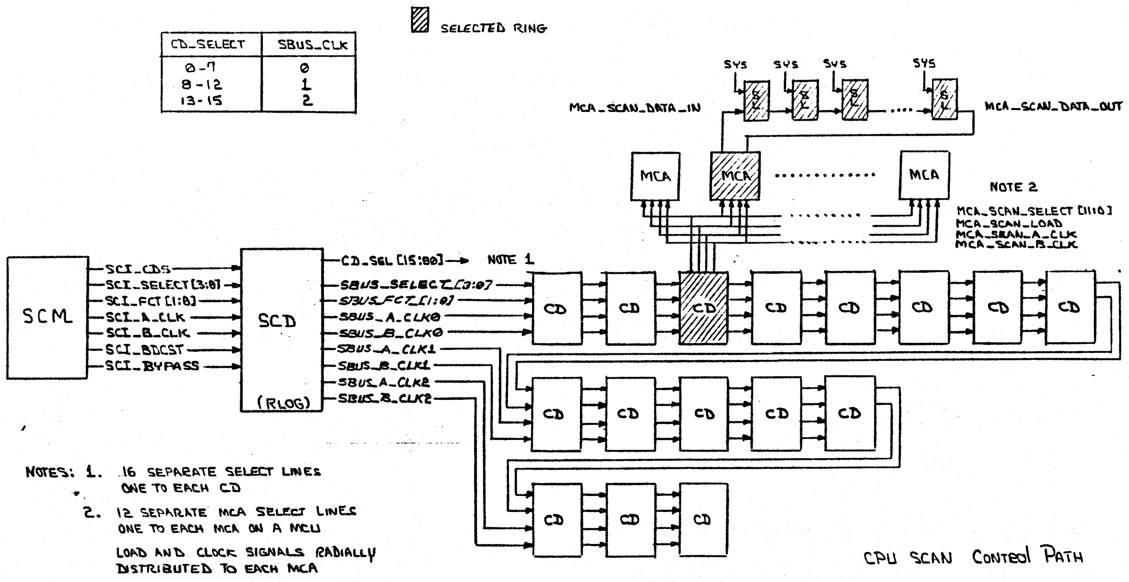
For all three loops the output from each CD (except for the first and last) drives the input to the next CD in the loop. The first CD receives its input from the SCD MCA and the last returns its output to the SCD MCA. Only the selected CD routes the data path through the MCAs. Unselected CDs simply connect their input to their output.

The selected CD routes the scan data at its input to the first of up to eight MCAs located on the same MCU. Like the CDs, the MCAs are also connected in serial loops with the output of one connected to the input of the next. Each MCA receives the scan data on MCA_SCAN_DATA_IN and provides an output on MCA_SCAN_DATA_OUT. The first MCA receives its input from the CD on MCA_SCAN_DATA_IN and the last MCA returns its output to the CD on MCA_SCAN_DATA_OUT.

The selected MCA routes the scan data at its input serially through all the scan scan latches contained on that MCA. Deselected MCAs simply connect their input to their output bypassing all the scan latches.

7.4.3.2 CPU Scan Control Path

Figure 7-19 shows how the SCI control signals are distributed from the SCM to the scan latch rings within the MCAs. The SCD MCA (contained within the RLOG MCA) in the EBox latches and distributes the SCI control signals to the three major CD loops via the single-ended SBUS.



CD_SELECT	SBUS_CLK
0-7	0
8-12	1
13-15	2

NOTES: 1. 16 SEPARATE SELECT LINES ONE TO EACH CD

2. 12 SEPARATE MCA SELECT LINES ONE TO EACH MCA ON A MCU

LOAD AND CLOCK SIGNALS RADIALLY DISTRIBUTED TO EACH MCA

Figure 7-19 Scan Signal Control Path

The SCI_SELECT<3:0> and SCI_FCT<1:0> signals are distributed to all 16 CDs in series via SBUS_SELECT<3:0> and SBUS_FCT<1:0>. The SELECT and FCT signals from one CD are regenerated and used to drive the next CD.

The SCI_A_CLK and SCI_B_CLK signals are distributed to each of the three major CD loops via separate SBUS signals.

- o SBUS_A_CLK0 and SBUS_B_CLK0 provide the clock signals for loop 0 and are serially connected through the eight CDs.
- o SBUS_A_CLK1 and SBUS_B_CLK1 provide the clock signals for loop 1 and are serially connected through the five CDs.
- o SBUS_A_CLK2 and SBUS_B_CLK2 provide the clock signals for loop 2 and are serially connected through the three CDs.

When SCI_CDS is asserted, the SCD MCA latches SCI_SELECT<3:0> to store the address of the CD to be selected. The SCD MCA then decodes this address and asserts one of 16 select lines, CD_SEL<15:00>. When SCI_CDS is negated, SBUS_SELECT<3:0> specifies the address of one of the MCAs connected to the selected CD. This CD decodes SBUS_SELECT<3:0> and generates one of twelve MCA select signals, MCA_SCAN_SELECT<11:0> which are radially distributed to each MCA.

In addition to the MCA_SCAN_SELECT lines the CD chip also radially distributes the following signals to each of its MCAs.

MCA_SCAN_LOAD - specifies FCT<1:0>=LOAD
MCA_SCAN_A_CLK - scan A clock for shifting scan latches
MCA_SCAN_B_CLK - scan B clock for shifting scan latches

Figure 7-20 relates the physical layout of the CPU planar module to the actual MCU addresses (CD select). The address is shown in parenthesis.

OPU (5)	XBR (6)	VIC (7)	VML (13)
VAP (4)	DTB (10)	CTL (11)	VRG (14)
DTA (3)	DST (9)	INT (8)	UCS (12)
CTU (2)	MUL (1)	FAD (0)	VAD (15)

```
EBox - CTL,DST,INT,UCS,MUL,FAD
IBox - OPU,XBR,VIC
MBox - VAP,DTB,DTA,CTU
VBox - VML,VRG,VAD
```

Figure 7-20 MCU Address Assignments

Table 7-3 summarizes the SBUS signals and Table 7-4 summarizes the MCA signals.

Table 7-3 SBUS Signal Descriptions

Signal Name	Description
SBUS_ SELECT<3:0>	Specify scan ring address or STRAM clock group. Table x summarizes the encoding of this field.
SBUS_FCT<1:0>	Single-ended copy of SCI_FCT<1:0> that specifies one of four functions, NOP/ATTN, SCAN, LOAD, or STRAM LOAD.
SBUS_A_CLK0	Single-ended copy of SCI_A_CLK that provides A phase scan clock to major loop 0.
SBUS_A_CLK0	Single-ended copy of SCI_A_CLK that provides A phase scan clock to major loop 0.
SBUS_A_CLK0	Single-ended copy of SCI_A_CLK that provides A phase scan clock to major loop 0.
SBUS_A_CLK0	Single-ended copy of SCI_A_CLK that provides A phase scan clock to major loop 0.
SBUS_A_CLK0	Single-ended copy of SCI_A_CLK that provides A phase scan clock to major loop 0.
SBUS_A_CLK0	Single-ended copy of SCI_A_CLK that provides A phase scan clock to major loop 0.
SBUS_DATA_0	Scan data input to first CD chip from SCD.

Table 7–3 (Cont.) SBUS Signal Descriptions

Signal Name	Description
RETURN_0	Scan data output from last CD chip in major loop 0.
SBUS_DATA_1	Scan data input to first CD chip from SCD.
RETURN_1	Scan data output from last CD chip in major loop 1.
SBUS_DATA_2	Scan data input to first CD chip from SCD.
RETURN_2	Scan data output from last CD chip in major loop 2.

Table 7–4 MCA Scan Signal Descriptions

Signal Name	Description
MCA_SCAN_ SELECT<11:00>	Twelve select lines decoded from SBUS_SELECT<3:0>. Used to select one MCA on the selected MCU.
MCA_SCAN_LOAD	Indicates that the SBUS_FCT<1:0> lines specified a LOAD function.
MCA_SCAN_A_CLK	Scan A phase clock. This clock is gated with the MCA_SCAN_SELECT signal. MCA_SCAN_B_CLK is not gated and always provides a B phase clock to allow B latches which receive data from other MCUs to correctly reflect the state of the A phase driving latch.
MCA_SCAN_B_CLK	Scan A phase clock.
MCA_SCAN_DATA_IN	Scan data input line that receives data from the previous MCA or CD.
MCA_SCAN_DATA_OUT	Scan data output line that transmits data to the next MCA or CD.

7.4.4 CD Functions

The following paragraphs describe the five major functions performed by the CD during scan system operations. The first four are directly specified by the encoding of the SCI_ FCT<1:0> lines.

7.4.4.1 SCAN Function

The SCAN function is the typical operation performed by the scan system. It consists of selecting a CD, selecting a ring and shifting data thru the ring. The CD is selected by the CD_SEL_x line decoded from the latched SCI_SELECT<3:0> field of the SCI in the SCD MCA. The SBUS_FCT<1:0> lines are set to SCAN from the SCI_FCT<1:0> lines. The SBUS_SELECT<3:0> lines address the desired ring and are driven from the SCI_ SELECT<3:0> lines. For scan write operations, SBUS_DATA is driven with the first (last latch) data from the SCI_DATA_IN line. The SBUS_CLK_A clock is generated followed by SBUS_CLK_A clock. These lines are also driven from the SCI. This DATA and CLOCK sequence is repeated until the entire ring is written. SCI_DATA_OUT is ignored. For scan read operations, SCI_DATA_IN is driven by SCI_DATA_OUT (data is looped at the scan controller (SCM)) and the clock sequence is repeated until the entire ring has been rotated. For write operations within a ring, a read is performed for the leading n bits followed by a write operation of the desired length and finally a read operation for the remainder of the ring. This allows fields to be modified within rings. For obvious reasons, fields can not be read from within rings.

7.4.4.2 LOAD Function

The LOAD operation is used to parallel load a scan ring from the contents of the system data presented to the latch inputs. This function is used for two purposes. The first is loading STRAM data which has been driven out of a STRAM via the READ STRAM sequence. This allows the data out lines of a STRAM to be read without affecting rings not directly involved in the STRAM outputs. This limits the extent of the data which must be saved for STRAM READ operations and will decrease the time required for read operations. This time is most critical during error recovery sequences.

The second purpose of the LOAD function is during "B phase Latch Hold Testing", outlined in Chapter 11 of the Aquarius Notebook. The LOAD function is used to load the A phase latches directly, thus eliminating the need for a BLOCK B phase system clock signal proposed for that function. This allows a more direct testing scheme. Note that all CDs can be selected to perform a load function and select 15 (broadcast select) can be used to load ALL A phase latches simultaneously.

7.4.4.3 STRAM LOAD Function

The STRAM LOAD operation allows a group of STRAM clocks to be issued by driving them from SCI_CLK_A. This allows STRAM contents to be modified without cycling the system clocks. This feature will be employed for console EXAMINE, DEPOSIT, LOAD and UNLOAD commands as well as by error handling software. The STRAM clock group will be selected by the SCI_SELECT<3:0> lines and will be issued on the following CLK A. The clocks will be directly driven by CLK A and will not require blocking.

7.4.4.4 NOP Function

The scan system will be left in the NOP condition when no accesses are required. This will reduce the probability of erroneous operations due to environmental noise. The NOP function disables the scan distribution output drivers in each CD to each MCA. The SBUS remains active and is used to report exception conditions.

7.4.4.5 ATTN Function

Exceptions are reported by the CD via an ATTN function. This function is enabled when the CD is in the NOP state and is NOT selected. The ATTN function ORs various exception signals onto the SBUS_DATA line to be driven on the SBUS_OUT lines. The SBUS_DATA line is returned to the SCD and is driven onto the SCI_DATA_OUT line to the SCM. The SCM will determine the source of the ATTN by selecting each CD and testing the ATTN line. When the CD driving the ATTN line is selected, the SCI_DATA_OUT line will deassert.

It should be noted that more than one CD can drive the ATTN line at any given time. This would defeat the above test. CD ring 14 will be used to verify that the correct CD has been selected by testing the LAT_ATTN bit in the CD ring. This ring can be rotated with system clocks running. In the event that the ATTN line is clear the SCM will then read ring 14 from each CD to determine the real ATTN CD. This scheme will reduce the amount scan shifting as well as the time required to service an exception. It provides a useful diagnostic function to test the ATTN line as well. The following list summarizes the exceptions reported via the ATTN line.

Temperature violation in CD
Clock phase detector
External error line asserted

7.4.5 MCM Interface

The Aquarius Master Clock Module (MCM) is controlled and initialized by the Service Processor via the SCM. The interface between the SCM and MCM consists of a 20 bit shift register (located in the MCM), 4 control lines, and 2 synchronizer lines as shown in Figure 7-21. The SCM writes the shift register with register select information along with data to be written and asserts the MCM_TRANSFER line. The MCM reads/writes the selected register and loops the data back to the shift register and toggles the MCM_TRANSFER_ACK line. This line clears MCM_TRANSFER and completes the transaction. The MCM deasserts the MCM_TRANSFER_ACK line when the MCM_TRANSFER line deasserts. Currently five registers are defined in the MCM and are described in the Aquarius Clock System documentation.

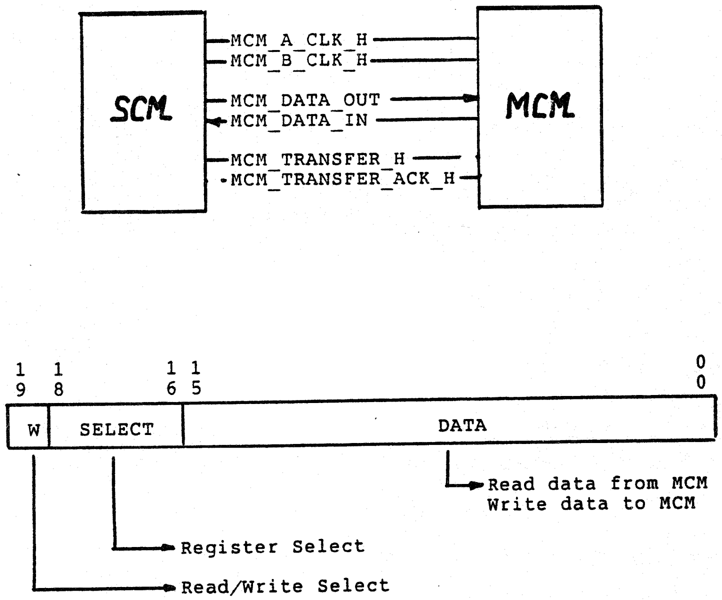


Figure 7–21 MCM Interface

7.4.5.1 MCM Control Lines

The MCM Interface uses a modified SCI interface consisting of six differential lines (12 wires). These lines address a scan shift register in the MCM and provide the interface between the SCM and the MCM. The following paragraphs describe these control lines.

MCM_CLK_A This line is the A phase shift clock. This line clocks the scan shift register when the MCM_TRANSFER line is not asserted. Clocking the shift path with MCM_TRANSFER asserted causes UNPREDICTABLE results.

MCM_CLK_B This line is the B phase shift clock.

MCM_DATA_OUT This line provides data from the MCM to the SCM and is synchronized with MCM_CLK_B.

MCM_DATA_IN This line provides data from the SCM to the MCM and is synchronized with MCM_CLK_B.

MCM_TRANSFER This line is used to signal the MCM that the scan shift register is valid. The MCM executes the indicated register access and asserts MCM_TRANSFER_ACK.

NOTE
The MCM must wait for the deassertion of MCM_TRANSFER after asserting MCM_TRANSFER_ACK. Failure to do so will result in the MCM repeatedly executing the current command. This may result in UNPREDICTABLE operation.

MCM_TRANSFER_ACK This line is used by the MCM to clear the MCM_TRANSFER line and terminate a command transaction. The assertion of this line indicates that the shift path is available to the SCM and may be read or loaded with a new command.

7.4.5.2 MCM Functions

The MCM interface is based on the 20 bit shift path shown in Figure 7-21. MCM_DATA_IN connects to the MSB (bit 19) of the shift path and MCM_DATA_OUT connect to the LSB (bit 0) of the shift path.

MCM Read Function The SCM tests the value of MCM_TRANSFER and shifts the desired register ID to the SELECT field of the MCM Shift Register with the W bit clear. The SCM then asserts MCM_TRANSFER. The MCM loads data from the register selected by the SELECT field of the shift path into the DATA field and issues MCM_TRANSFER_ACK. The SCM shifts out the register to complete the read.

MCM Write Function The SCM tests the value of MCM_TRANSFER and shifts the desired register ID to the SELECT field of the MCM Shift Register, the desired data to the DATA field, and then asserts the W bit. The SCM then asserts MCM_TRANSFER. The MCM writes the register selected by the SELECT field with the data in the DATA field and loops the written data back to the shift register. The MCM then issues MCM_TRANSFER_ACK. The SCM optionally shifts out the register to compare the loopback data.

7.4.5.3 MCM Interconnect

The interconnect to the MCM is a [TBD] pin flat twisted ribbon cable. The cable connects to the BI transition header in the Service Processor and to the [TBD] connector on the MCM.

7.5 Scan Control Module

7.5.1 SCM Block Diagram

Figure 7-22 shows how all the major components on the SCM are connected. The following list briefly describes the function of each component.

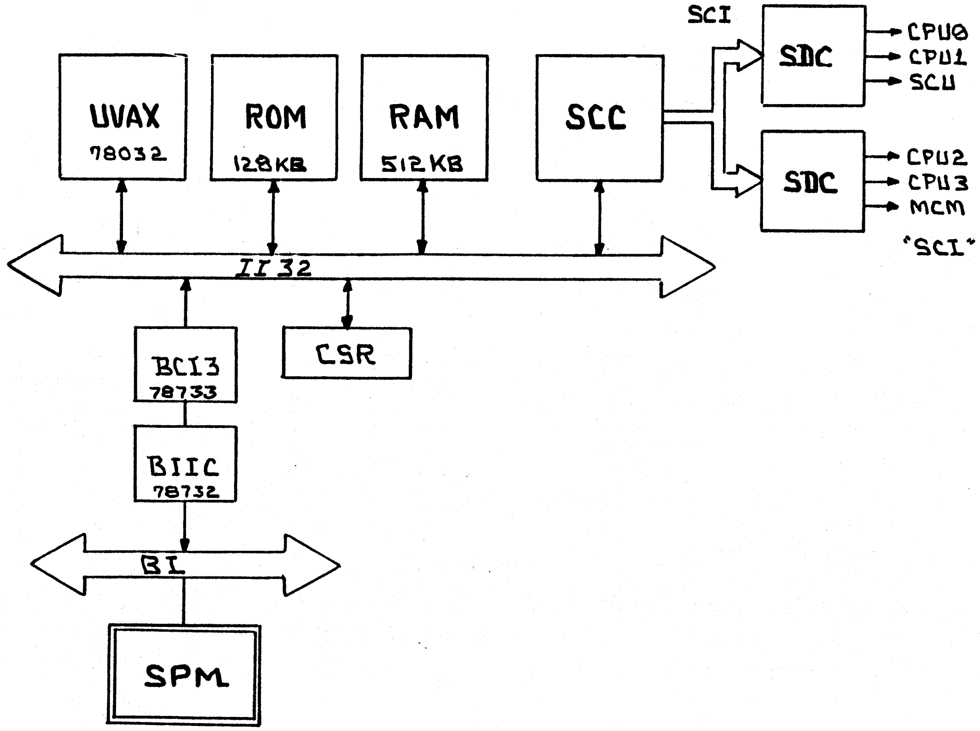


Figure 7–22 SCM Block Diagram

- o SCM - The SCM is driven by a 78032 uVAX chip connected to the II32 bus. It provides the uVAX subset of the VAX instruction set at 90% of the performance of a VAX 11/780 CPU. The uVAX also provides full VAX memory management features.
- o ROM - The ROM provides 128KB storage that contains the SCM self-tests and firmware. It connects to the uVAX via the II32 bus. During power-up, the self-tests automatically run to verify operation of the hardware components on the SCM module.
- o RAM - The RAM provides 512KB of storage used to buffer all information transfers between the scan latch rings in the CPUs, SCU, and MCM. During system initialization, the SPU software loads signal and STRAM definition tables into RAM to provide logical access to signals and STRAM structures. Additional data structures are initialized in the RAM to define the structure of the scan system.
- o SCC - The SCC (Scan Control Chip) is a custom gate array that contains all the logic to control the scan latch rings via the SCI. It connects to the uVAX and RAM via the II32 bus and supports DMA transfers between RAM and its internal data registers.
- o SDC - The SDC (Scan Distribution Chip) is a custom gate array that connects the SCI output from the SCC chip to up to three SCI cables. It translates the single ended TTL SCI signals from the SCC to differential STECL SCI signals at its output. There are two SDC chips. One drives the SCI to CPU0, CPU1, and the SCU, while the other drives CPU2, CPU3, and the MCM. One of the SDC chips also contains the clock control and generation logic for the MCM and SCI scan A and B clocks.
- o CSR - SCM module Control and Status Register that provides visibility to module and BI status bits.

- BCI3 - The BCI3 (VAXBI 78733), contained in a 132-pin ceramic pin grid array (PGA) package, is used to connect the II32 bus (Integrated Circuit Interconnect) of the uVAX 78032 chip to the VAXBI bus through the BIIC. The BCI3 handles both low-level interface functions of the II32 bus and high-level protocol translation functions necessary for interbus communication. It assists in managing the exchange of data and provides transparent translations of read and write transactions, error conditions, and interrupt requests.
- BIIC - The (BIIC) is contained on a 133-pin ZMOS integrated circuit and serves as the primary interface between the VAXBI and the SPM module. It performs bus transactions, address decoding and matching, and controls the interface signals to the BCI3 chip.
- SPM - The Service Processor Module provides the hardware to store and execute the SPU software. It contains a uVAX, 16MB of RAM storage, an interface to the SCU, and local and remote user interfaces. It connects to the SCM via the BI.

7.5.2 SCC Operations

The heart of the SCM module is the SCC chip, which controls all scan system operations as directed by the SCM firmware. The following paragraphs describe most of these operations using simplified block diagrams. The basic scan operations discussed are:

1. Ring Read
2. Ring Write
3. Ring Read-write-read
4. SPE - Scan Pattern Execute
5. SPV - Scan Pattern Verify
6. CPU XOR Testing
7. Attention Interrupt Handling

Since all of these scan operations require DMA transfers between RAM and the SCC chip, the SCC's DMA control logic and register data paths are described first.

7.5.2.1 SCC DMA Control/Data Paths

The SCC chip contains the control logic to support a four channel DMA facility. Each channel is associated with a data register and an address offset register. Three of the channels support reading RAM while the fourth supports writing RAM. Figure 7–23 shows how all the registers used by the DMA facility are connected.

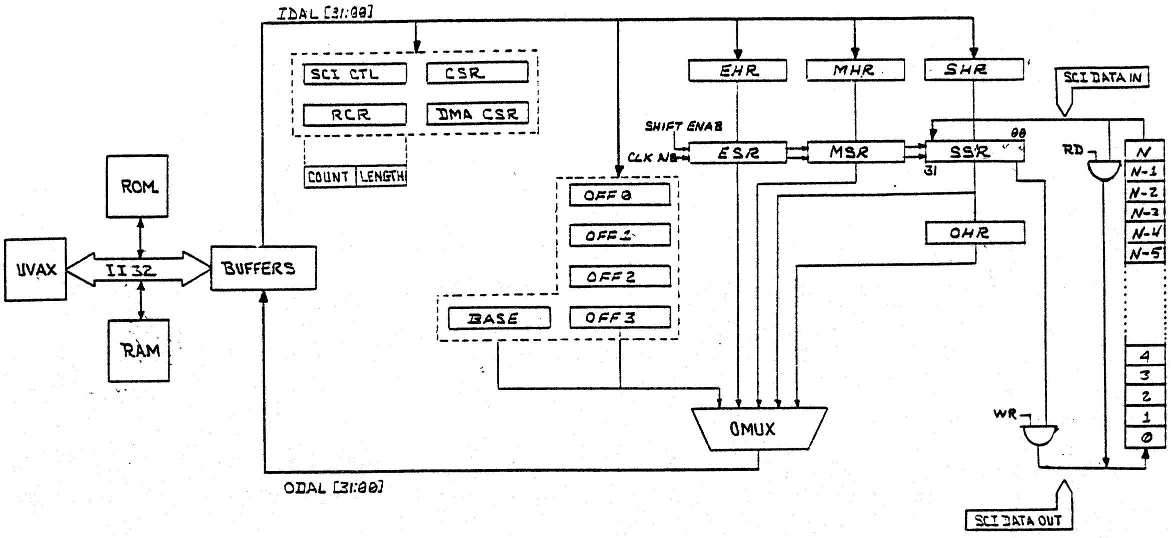


Figure 7-23 SCC Data path Block Diagram System

The four DMA channels are listed and described below.

1. Channel 0 - This channel is used to write the contents of the Output Hold Register (OHR) to the specified RAM ring buffer. The OHR is loaded from the SSR and contains either scan ring data read from the scan system or the result of a pattern compare operation. The RAM address is specified by the concatenation of the BASE and OFF0 address registers. This address is always on a longword boundary and the offset register is updated by +4 after each transfer. This channel is enabled by setting the OUT VALID bit (bit 0) in the DMA CSR.

2. Channel 1 - This channel is used to read the specified RAM buffer and load the data into the Scan Hold Register (SHR). The SHR is then loaded into the Scan Shift Register from where it is shifted out to the selected scan latch ring. The RAM address is specified by the concatenation of the BASE and OFF1 address registers. This address is always on a longword boundary and the offset register is updated by +4 after each transfer. This channel is enabled by setting the IN VALID bit (bit 1) in the DMA CSR.

3. Channel 2 - This channel is used to read the specified RAM buffer and load the data into the Mask Hold Register (MHR). The MHR is then loaded into the Mask Shift Register (MSR) from where it is shifted out to the scan pattern compare logic to enable Expected/Ring Data comparison. The mask bit must be set to enable comparison checking. The RAM address is specified by the concatenation of the BASE and OFF2 address registers. This address is always on a longword boundary and the offset register is updated by +4 after each transfer. This channel is enabled by setting the MASK VALID bit (bit 2) in the DMA CSR.

4. Channel 3 - This channel is used to read the specified RAM buffer and load the data into the Expected Hold Register (EHR). The EHR is then loaded into the Expected Shift Register (ESR) from where it is shifted out to the scan pattern compare logic that compares each bit with the corresponding scan ring bit. The corresponding bit in the MSR must be set to enable the comparison. The RAM address is specified by the concatenation of the BASE and OFF3 address registers. This address is always on a

longword boundary and the offset register is updated by +4 after each transfer. This channel is enabled by setting the IN VALID bit (bit 3) in the DMA CSR.

Setting the DMA bit in the SCC CSR (bit 14) enables the DMA control logic when the GO bit is set in the SCC CSR (bit 15) to initiate a scan operation. If the DMA bit is not set (STEP MODE) when the GO bit is set, the firmware must write the EHR, MHR, and SHR and read OHR to perform the required RAM transfers for each 32-bit step. STEP MODE is normally only used for hard core diagnostics that validate the operation of the scan system.

The DMA control logic maintains a set of logic flags that monitor the state of all the data path registers during active scan operations. Empty flags are set whenever an input hold register is transferred to its corresponding shift register (EHR to ESR, MHR to MSR, and SHR to SSR). If the operation needs more data from the ring buffers in RAM, the empty flag initiates a DMA read request to get the next longword of data to load into the hold registers that are empty.

A full flag is used to monitor the state of the OHR. It sets when the contents of the SSR are transferred to the OHR to initiate a DMA write request to transfer the contents of the OHR to its associated ring buffer in RAM.

The VALID bits in the DMA CSR (bits [3:0]) are the key to activating the DMA channels. These bits along with the WRITE bit in the SCC CSR determine the actual scan operation. Table 7-5 summarizes the relationship between these bits and the resulting scan operation.

Table 7-5 DMA CSR Valid Bits and Scan Operations

DMA CSR [3:0]	W	Operation
0001	0	Ring read
0010	1	Ring write
1000	0	Unmasked Scan Pattern Verify - no result transferred to RAM
1100	0	Masked Scan Pattern Verify - no result transferred to RAM
1001	0	Unmasked Scan Pattern Verify - result transferred to RAM
1101	0	Masked Scan Pattern Verify - result transferred to RAM
1010	1	Unmasked Scan Pattern Execute - no result transferred to RAM
1110	1	Masked Scan Pattern Execute - no result transferred to RAM
1011	1	Unmasked Scan Pattern Execute - result transferred to RAM
1111	1	Masked Scan Pattern Execute - result transferred to RAM

One significant aspect of the DMA control in the SCC is that it is separate, although dependent upon, the shift control logic that controls the actual shifting of the data in the ESR, MSR, and SSR. The design of the data path allows simultaneous DMA and scan shift/compare operations. While one 32-bit longword is being shifted in/out, the next longwords required are being transferred from RAM into the hold registers.

Since the DMA must service requests on four channels, it is possible that more than one request can exist simultaneously. A priority mechanism is included to arbitrate multiple requests as follows.

- Channel 0 - HIGHEST
- Channel 1 -
- Channel 2 -
- Channel 3 - LOWEST

For any DMA request, the SCC must arbitrate for control of the II32 bus to become bus master before making the transfer to or from RAM. In the case of multiple requests, the DMA control logic supports a "burst" mode that allows transferring up to four longwords once the SCC becomes bus master.

7.5.2.2 Scan Operation Setup

Before executing any scan system operation, the firmware must initialize the required registers contained within the SCC chip. The following outlines the common setup procedures. All steps except the last can be performed in any sequence. The GO bit is what starts the operation; it must be the final step.

1. Set up the SCI Control Register to select the port, ring, and function.
2. Set up the Ring Control Register to specify the ring length and zero the count.
3. Set up the address base register (BASE) to define the base address of all the ring buffers in RAM.
4. Set up all the required address offset registers (OFF[3:0]) to define the location of each ring buffer relative to the base address.
5. Set up the DMA CSR to enable all DMA channels required by the operation (EXP, MASK, IN, and OUT valid bits).
6. Set up the SCC CSR to specify DMA, WRITE (if required), and DONE IE (enable interrupts).
7. Finally, set the GO bit in the SCC CSR to start the scan operation and hope for the best.

7.5.2.3 Ring Read

Ring read is used to transfer the contents of the selected scan latch ring in the CPU into the SCM's local RAM. Figure 7-24 is a simplified block diagram that summarizes the primitive ring read operation.

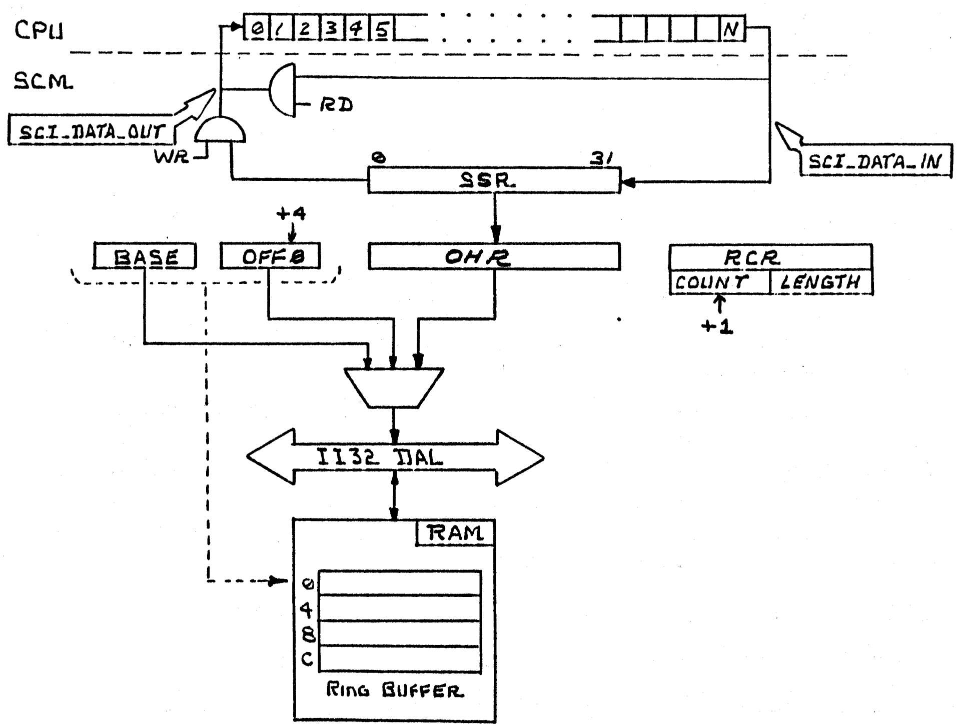


Figure 7–24 Ring Read Operation

Assume that the ring shown in the diagram contains 128 scan latches (N = 127). The SCC chip uses five registers to make the transfer.

1. SHR - a 32-bit shift register that is loaded serially from the scan ring data. The data is shifted in the MSB.
2. OHR - a 32 bit parallel-load register that is loaded with the data accumulated in the SHR.
3. BASE - Loaded by the firmware during setup to point to the ring buffer area in SCM RAM.
4. OFF0 - Loaded by the firmware during setup and concatenated with the BASE register to point to a specific ring buffer in SCM RAM. It always points to a longword boundary and is updated by +4 after each transfer.
5. RCR - Ring Control Register that is loaded by the firmware during setup to specify the ring length (in this case, 128) and zero the count. Incremented for each bit shifted into the SSR.

Once the desired ring has been selected and the BASE, OFF0, and RCR registers are initialized, the firmware sets a GO bit to start the transfer. Here's how it works.

1. The firmware sets the GO bit.
2. The SCC control logic enables generation of scan A and B clocks to shift the SSR and the scan ring latches in the CPU. The bits enter the SSR at the MSB via SCI_DATA_IN. With RD = 1 (a read), SCI_DATA_IN is looped back out on SCI_DATA_OUT to be shifted back into the scan latch ring in the CPU. The COUNT field in the RCR is incremented for each shift. When the COUNT field equals the LENGTH field, the

SCC control turns off the scan A and B clocks to the SCI to stop shifting the scan latch ring in the CPU. If the ring length is not a multiple of 32, the SSR is zero-filled in the high order bits.

3. When the SSR is full (32 shifts), the contents of the SSR are transferred to the OHR and a DMA write request is initiated by the SCC. When the SCC becomes master of the II32 bus it transfers the contents of the OHR to RAM using the contents of BASE,OFF as an address. After the transfer, OFF0 is updated by +4. If the COUNT field equals the LENGTH field, the SCC control logic sets the DONE bit to request an interrupt to the firmware which signals completion of the transfer.

It is important to note that the DMA write operation is overlapped with the shifting of the scan ring. The example just described requires generating 128 scan A and B clocks to complete the transfer. At a normal clock rate of 100ns this would take 12.8 usec. Once the ring data is in the SCM's local RAM it can be retrieved by the SPM via the BI.

7.5.2.4 Ring Write

Ring write is used to transfer the contents of a ring buffer in the SCM's local RAM into the selected CPU scan latch ring. Figure 7-25 is a simplified block diagram that summarizes the primitive ring write operation.

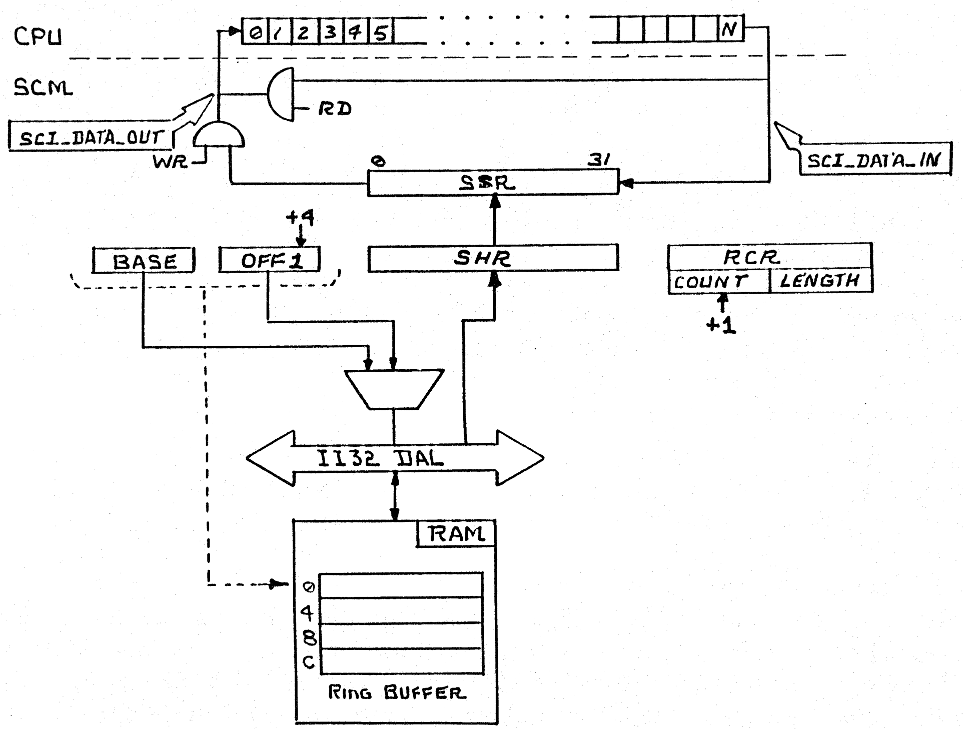


Figure 7-25 Ring Write Operation

Assume that the ring shown in the diagram contains 128 scan latches (N=127). The SCC chip uses five registers to make the transfer.

1. SHR - a 32-bit shift register that is parallel loaded from the SHR register and serially shifted out to the CPU scan ring latches.
2. SHR - a 32 bit parallel-load register that is loaded with the data from the RAM ring buffer via the II32 bus.

3. BASE - Loaded by the firmware during setup to point to the ring buffer area in SCM RAM.
4. OFF1 - Loaded by the firmware during setup and concatenated with the BASE register to point to a specific ring buffer in SCM RAM. It always points to a longword boundary and is updated by +4 after each transfer.
5. RCR - Ring Control Register that is loaded by the firmware during setup to specify the ring length (in this case, 128) and zero the count. Incremented for each bit shifted into the SSR.

Once the desired ring has been selected and the BASE, OFF1, and RCR registers are initialized, the firmware sets a GO bit to start the transfer. Here's how it works.

1. The firmware sets the GO bit.
2. If the SHR is empty and the LENGTH field is greater than the COUNT field in the RCR, the SCC control logic requests control of the II32 bus to perform a DMA read. When the SCC becomes bus master, it transfers the first longword from the ring buffer in SCM local RAM to the SHR using the contents of BASE,OFF1 as the address. OFF1 is then updated by +4 to prepare to get the next longword.
3. When the SCC control logic senses that the SSR is empty and the SHR is full, it transfers the contents of the SHR to the SSR and enables the scan A and B clocks to shift the data from the SSR into the CPU scan ring latches via SCI_DATA_OUT. COUNT is incremented for each shift. At the same time data is leaving the LSB of the SSR, data from the scan latch ring is entering the MSB via SCI_DATA_IN. For vanilla flavored scan ring write, these bits are overwritten with the next longword to be shifted. It is possible to perform overlapped write/read, which will be described later.
4. As soon as the SCC control senses that the SHR is empty and if the COUNT field is not equal to the LENGTH field, it requests control of the II32 bus to perform a DMA read. After becoming bus master, it transfers the next longword from the ring buffer in RAM to the the SHR using the contents of BASE,OFF1 as the address. OFF1 is updated by +4 after each transfer. When the COUNT field equals the LENGTH field, the SCC control logic waits for the last bit to be shifted out of the SSR and then sets DONE to request an interrupt to the firmware.

It is important to note that the DMA write operation is overlapped with the shifting of the scan ring. The example just described requires generating 128 scan A and B clocks to complete the transfer. At a normal clock rate of 100ns this would take 12.8 usec.

7.5.2.5 Ring Read-Write-Read

At times the firmware needs to modify a field within the CPU scan latch ring. This is achieved by combining the primitive ring read and ring write operations. Suppose that the firmware wants to modify 32 consecutive bits in the scan ring starting at position 32. To perform this operation, the firmware would execute the following sequence.

1. Execute a ring read with LENGTH=64
2. Execute a ring write with LENGTH=32
3. Execute a ring read with LENGTH=32

It is important to note that all ring accesses spin the entire ring to maintain a known machine state. Any errors that abort a ring read or write before completion, require the firmware to re-initialize the selected ring or rings. Broadcast mode enables concatenation of multiple rings.

7.5.2.6 Scan Pattern Execute

Scan Pattern Execute (SPE), the bread and butter scan operation, combines and overlaps three primitive operations.

1. Writes the scan ring latches in the CPU.
2. Reads the scan ring latches in the CPU.
3. Compares the scan ring data read from the CPU with the expected ring data stored in RAM (bit for bit) and stores the result of the comparison back to RAM.

SPE is the primary scan operation used by the scan pattern diagnostics to detect and isolate hardware faults to the failing MCU. The design of the SCC chip permits pattern N to be scanned out at the same time pattern N-1 is being scanned in and compared. An even higher level of simultaneity also exists in that the SPU software can be analyzing the results of pattern N-2 at the same time. Figure 7-26 summarizes the SPE operation.

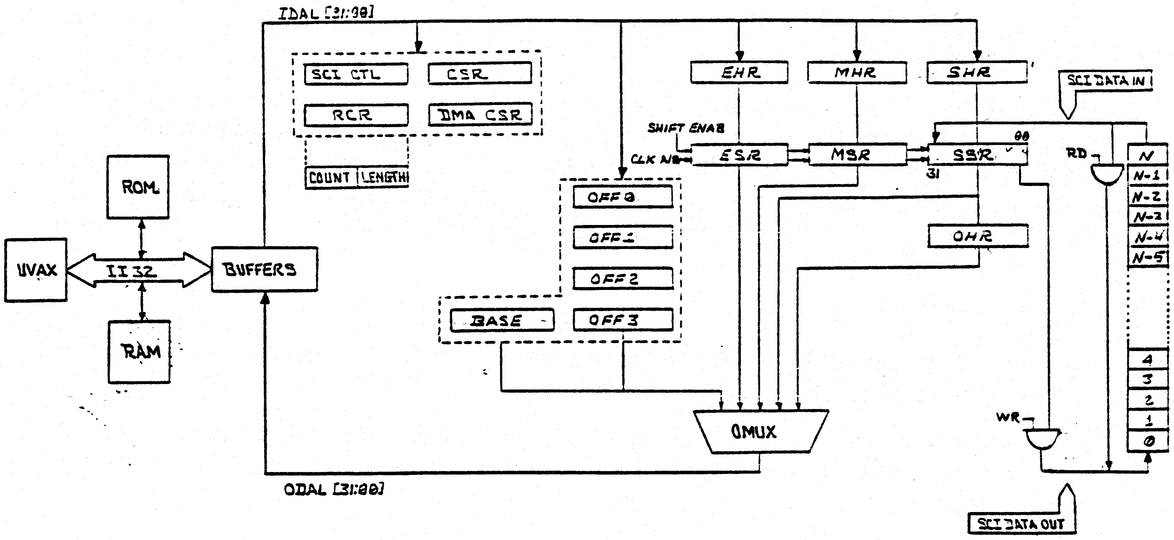


Figure 7-26 Scan Pattern Execute

SPE can be masked or unmasked. The masked SPE operation is the normal case and the one described here. The only difference is that unmasked forces comparison of all bits, while masked permits selecting which bits in the ring are to be compared.

An SPE operation requires that all of the address offset registers be initialized to point to four unique ring buffers in RAM when concatenated with the base address loaded into the BASE register.

1. OFF0 - points to the ring buffer to receive the results of the comparison.
2. OFF1 - points to the ring buffer containing the pattern to be written.
3. OFF2 - points to the ring buffer that contains the mask pattern.
4. OFF3 - points to the ring buffer that contains the expected pattern.

All four VALID bits in the DMA CSR must be set for a SPE operation, which uses all four DMA channels.

The sequence of events required to execute the SPE operation is summarized in Figure 7–27 and described below.

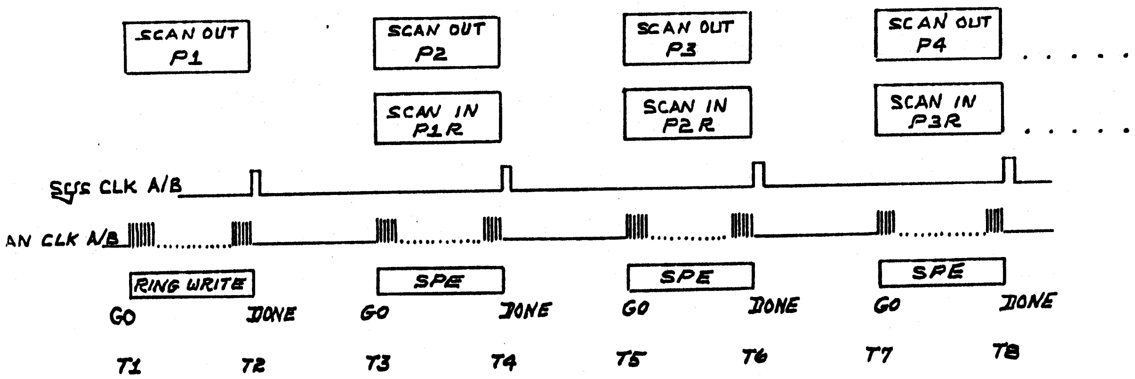


Figure 7–27 SPE Timing

The first pattern (P1) is scanned out using the primitive ring-write operation. GO is set by the firmware at T1 and the SCC sets DONE at T2 to signal completion. From T1 to T2, SCAN CLK A/B are enabled to shift all the bits in P1 into the selected CPU scan latch ring. The number of clock pairs is determined by the ring length. During this interval, the SCC DMA control logic initiates DMA read requests to retrieve the ring data from the RAM ring buffer containing data for P1.

Between T2 and T3 the firmware:

1. Generates a single SYS CLK A/B to load system data into the scan latch ring in the CPU which is the result for pattern 1 (P1R).
2. Sets up the address offset registers to point to the RAM ring buffers containing the expected and mask data for pattern P1 and the ring buffer to receive the results of the comparison for P1.

   Sets up the address offset register for DMA channel 1 to point to the next test pattern (P2) to be scanned out.
3. Sets GO to start the first SPE which starts at T3. From T3 to T4 the SCC
   a. Scans out P2 using DMA channel 1.
   b. Scans in P1R and performs the pattern compare using the expected and mask patterns using DMA channel 2 and 3.
   c. Transfers the comparison result pattern to the specified RAM ring buffer using DMA channel 0.
4. The SCC sets DONE at T4 to signal completion of the first SPE operation.

For each test the same sequence is repeated until all test patterns P1, P2, P3, P4, ... Pn are executed. Each SPE begins with GO and ends with DONE. After each DONE interrupt, the firmware must clock the results and set up for the next pattern before setting GO again. The results of the ring comparison must also be transferred from the RAM ring buffer to the SPM via the BI for analysis by the SPU software.

Figure 7–28 shows a simplified diagram of the pattern compare logic. The result patterns scanned in appears on one or more of the following inputs.

MCM_DATA_IN
CPU_DATA_IN_3
CPU_DATA_IN_2
SCU_DATA_IN
CPU_DATA_IN_1
CPU_DATA_IN_0

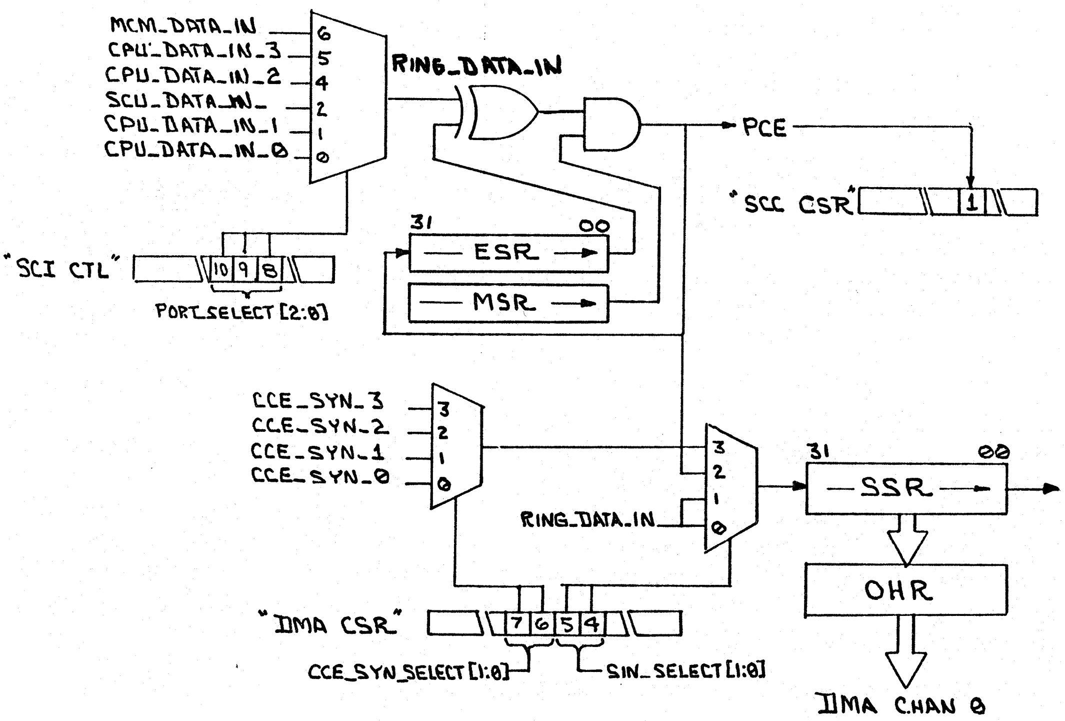


Figure 7–28 Pattern Compare Logic

The six scan data inputs are shown entering a multiplexer in the SCC. The firmware sets up the PORT SELECT field in the SCI CTL register to select which scan input controls RING_DATA_IN. RING_DATA_IN is XORed with the LSB of the ESR (expected), bit by bit, and the result of the comparison is enabled, bit by bit, by the LSB of the MSR (mask). If any enabled bit fails to compare, the error signal PCE (Pattern Compare Error) is asserted to generate a result error pattern. The state of PCE is shifted back into the LSB of the ESR register so that after any 32-bit segment of the scan pattern, the ESR contains the result error pattern for that 32 bit segment. PCE is also shifted back into the MSB of the SSR register, if selected by bits in the DMA CSR,

Figure 7–28 shows the path that selects shifting PCE back into the SSR. The output from a 4x1 multiplexer, controlled by SSR_DATA_SEL[1:0] field (bits [5:4]) in the DMA CSR. The firmware must set SSR_DATA_SEL[1:0]=2 to shift the state of PCE into the SSR. Codes of 0 or 1 select the actual ring data being scanned in on RING_DATA_IN while a code of 3 selects the output from a second multiplexer controlled by CCE_SYN_SEL[1:0] which is described in Section 7.5.2.8. Once the error pattern is in the SSR, it can be loaded into the OHR to be transferred to RAM via DMA channel 0.

7.5.2.7 Scan Pattern Verify

The Scan Pattern Verify (SPV) operation is the same as the SPE operation except that no change occurs in the current state of the selected CPU scan latch ring. It verifies that the last data pattern scanned out compares correctly when scanned in. No new data is written to the selected scan ring. No system clocks are generated to change the state of the scan latch ring.

The SPV uses the same registers as the SPE with the following differences.

1. No system clock is generated after scanning out the pattern to set up the selected ring.
2. The OFF1 and SHR registers are not required since no new scan data is written. The OUT VALID bit in the DMA CSR is cleared to disable DMA channel 1.
3. The OFF3 register is initialized to point to the ring buffer that contains the ring pattern previously written since SPV is verifying the data just written.
4. The OFF2, MHR, and MSR registers are not used since all bits must be verified. The MASK VALID bit in the DMA CSR is cleared to disable DMA channel 2.
5. The results may or may not be transferred to RAM. OFF0 must be initialized to save the result pattern in RAM. The OUT VALID bit in the DMA CSR is cleared to disable DMA channel 0.

7.5.2.8 CPU XOR

In addition to the normal scan pattern testing with the SPE and SPV operations, the SCC chip also provides the logic facilities to compare scan patterns from one CPU called the *PRIMARY* to one or more of the other three CPUs in a quad processor Aquarius system. This CPU XOR feature allows the SCM to scan out the same test pattern to one or more CPUs and then scan in and compare the results from two, three, or four CPUs simultaneously. Basically, it permits comparing a known good CPU against one that is suspected to be faulty. The XOR operation may be masked or unmasked.

Figure 7–29 shows a simplified diagram of the logic in the SCC that supports the XOR function.

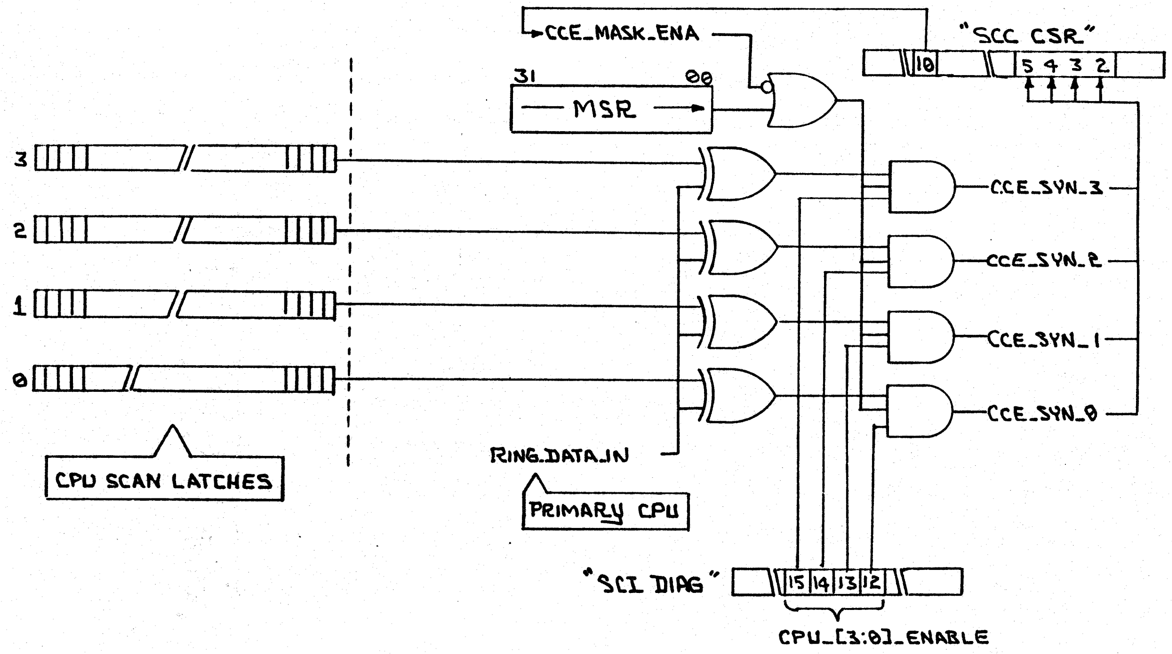


Figure 7–29 CPU XOR Testing

The scan latch data from the four CPUs enters the SCC on CPU_DATA_IN[3:0] where it is compared, bit for bit, with the scan data from the primary CPU on RING_DATA_IN. The primary CPU is specified by the PORT_SEL[2:0] field in the SCI control register as shown in Figure 7-28. The result of the comparison is then enabled by two additional conditions.

1. The CPU_ENABLE[3:0] field in the SCI DIAG register (bits [15:12]).
2. The MSR00 bit if a masked compare is specified by setting the CCE MASK ENABLE bit (bit 10) in the SCC CSR register.

The final result generates four possible compare error signals, CCE_SYN[3:0]. These error syndrome signals set a corresponding error latch in the SCC CSR, CCE[3:0] (bits [5:2]). One of the four results may be shifted into the SSR and transferred to a result ring buffer in RAM via DMA channel 0. Figure 7-28 shows how bits in the DMA CSR are used to make the selection. To enable the DMA transfer, the firmware must initialize the BASE and OFF1 address registers and also set the OUT VALID bit in the DMA CSR register.

7.5.2.9 Attention Handling

When the Aquarius system is up and running the operating system and user applications, the SCM monitors the SCI for attention interrupt requests from each of the six SCI ports, CPUs, SCU, and MCM. This section describes the conditions that generate attention requests, how the are signaled, and how the SCM firmware responds to the requests. Refer to Figure 7-30 for an overview of attention interrupt handling.

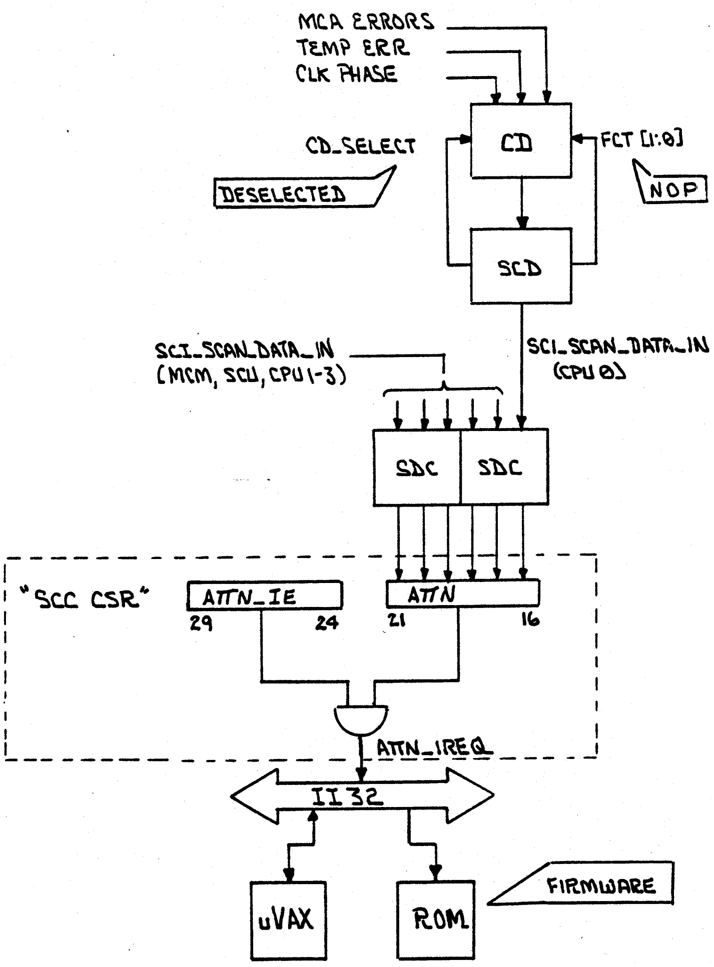


Figure 7-30 Attention Handling Overview

Logic located in the CD chip on each MCU monitors CPU operation and detects the following error conditions.

- Errors detected by the MCAs (parity errors, etc.)
- Overtemperature conditions sensed by the CD.
- Clock phase errors monitored by the CD.

When the scan system is idle, all CDs are deselected and the the SBUS_FCT[1:0] lines specify a NOP function. This condition enables the error logic within each CD to drive its SBUS_DATA_OUT line. The SCC chip monitors the SCI_DATA_IN lines from all four CPUs, and the SCU and MCM via the SCD, SCI lines, and SDC chips.

Assertion of any scan data in line sets a corresponding latch in the SCC to signal the occurrence of an error. ATTN[5:0] in the SCC CSR (bits [21:16]) reflect the state of these latches. If the corresponding ATTN_IE[5:0] latch in the SCC CSR (bits [29:24]) is set, the SCC generates an interrupt request to the firmware via the II32 bus.

Once interrupted, the firmware must,

1. Determine which unit caused the interrupt.
2. Which MCU within that unit sensed the error.

3. What type of error (MCA, Temperature, or Timing).
4. Use the scan system to retrieve all the scan latch rings needed to analyze the cause of the error.

To simplify the explanation, the following discussion assumes a single error.

First, by reading the SCC CSR, the firmware can determine which unit (CPU0-3, SCU, or MCM). Next it can select all MCUs in that unit and monitor the state of the scan data in signal to determine which MCU. Since the attention logic in the CD responds to errors only when deselected, selecting the faulty MCU causes the data in line to deassert. Once the faulty MCU is determined, the firmware can use the scan system to select that MCU and scan in all CD and MCA rings for subsequent analysis by the SPU software.

8
System Initialization

8.1 Chapter Objective

The chapter objective is to introduce and provide specifications, and a functional overview of the AQUARIUS:

- power subsystem
- Power Control Subsystem (PCS)
- Operator Control Panel (OCP)

8.2 Power On Initialization

Power On Initialization begins when the OCP POWER switch is set to ON. Power is then applied to the AC Front End (or the optional UPC) and WCU.

The flow chart of Figure 1 summarizes the Power On Initialization sequence. The flow operation block numbers correspond to the following description numbers.

8.2.1 WCU Power On

1. Ac Power Applied To The WCU

   The OCP POWER switch applies ac power to the WCU
2. WCU Starts Power Up Sequence.
3. WCU Fully Operational

   WCU becomes fully operational in less than 10 seconds.

8.2.2 Ac Front End/UPC Power On

1. Power Applied To Ac Front End/UPC

   The OCP POWER switch applies ac power to the Ac Front End (or UPC).

2. High Voltage Dc Bus Powered Up To 280 VDC

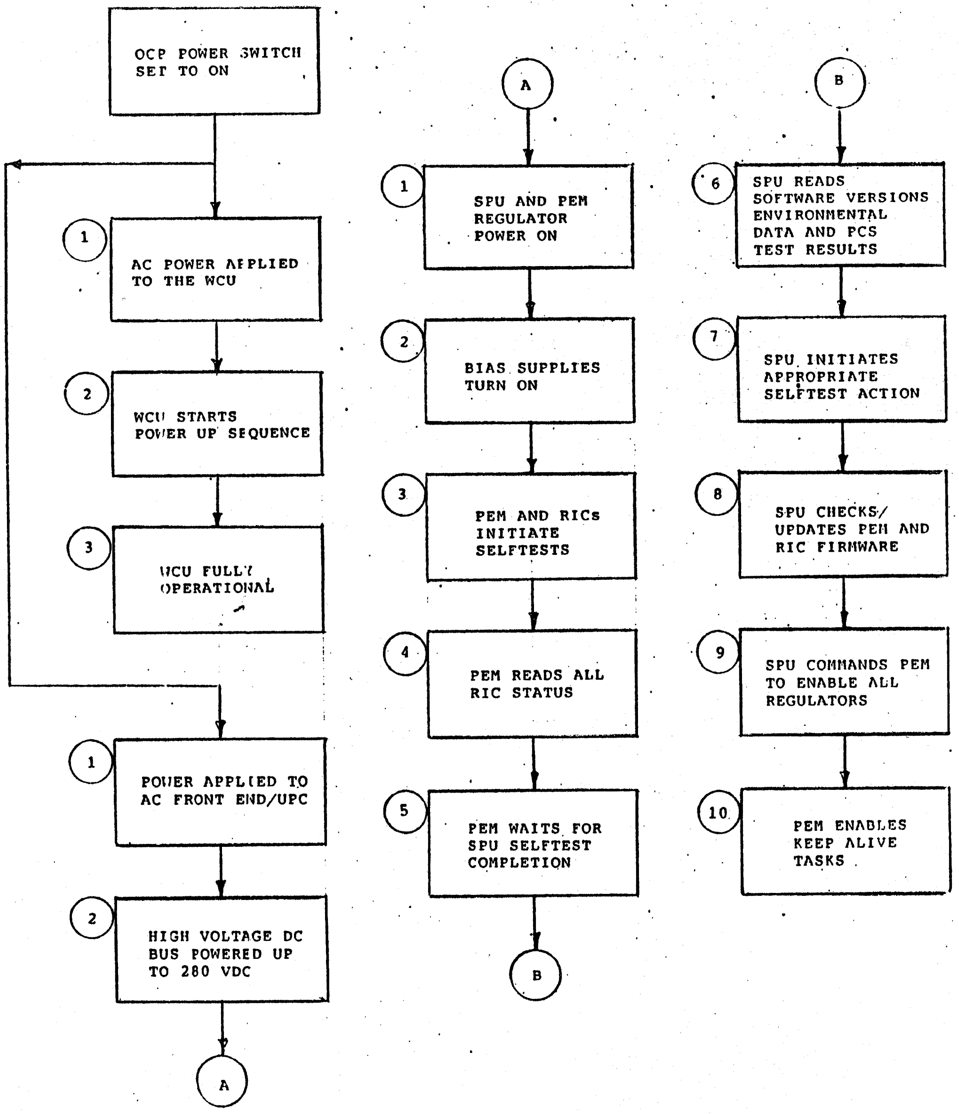

OCP POWER SWITCH SET TO ON
1
AC POWER APPLIED TO THE WCU
2
WCU STARTS POWER UP SEQUENCE
3
WCU FULLY OPERATIONAL
1
POWER APPLIED TO AC FRONT END/UPC
2
HIGH VOLTAGE DC BUS POWERED UP TO 280 VDC
A
A
1
SPU AND PEM REGULATOR POWER ON
2
BIAS SUPPLIES TURN ON
3
PEM AND RICs INITIATE SELFTESTS
4
PEM READS ALL RIC STATUS
5
PEM WAITS FOR SPU SELFTEST COMPLETION
B
B
6
SPU READS SOFTWARE VERSIONS ENVIRONMENTAL DATA AND PCS TEST RESULTS
7
SPU INITIATES APPROPRIATE SELFTEST ACTION
8
SPU CHECKS/ UPDATES PEM AND RIC FIRMWARE
9
SPU COMMANDS PEM TO ENABLE ALL REGULATORS
10
PEM ENABLES KEEP ALIVE TASKS

Figure 1 Power On Initialization

8.2.3 SPU and PEM Power On

1. SPU And PEM Regulator Power On

   The regulator powering the SPU and PEM automatically turns on when the high voltage DC bus is on.

2. Bias Supplies Turn On

   At the same time, all Bias Supplies in the system turn on and provide power to the RICs as well as start-up power to the main power supplies. All power supplies under control of the RICs remain off.

3. PEM And RICs Initiate Selftests

   The PEM and RIC selftests execute in less than 10 seconds. On successful selftest completion each RIC enables it's status LED.

   A RIC will *not* enable any regulator until the selftest and environmental status has been retrieved by the PEM, and the RIC has been commanded to enable its regulator group.

4. PEM Reads All RIC Status

   The PEM will read the selftest status, version number, OCP Diagnostic Display status code and environmental data from each of the RICs.

   Selftest results and the other data is stored in the PEM until status is requested be the SPU.

5. PEM Waits For SPU Selftest Completion

   The PEM waits for the SPU to complete its self test. If the test fails, the PEM will indicate the failure on the OCP Diagnostic Display and the powerup sequence halts.

   The PEM will not enable any regulators unless commanded to do so by the SPU.

6. SPU Reads Software Versions, Environmental Data, and PCS Test Results

   The SPU reads software version numbers, environmental data, and the result of the PCS self tests.

7. SPU Initiates Appropriate Selftest Status Action

   The SPU initiates appropriate action based on PCS selftest failures or environmental faults. These actions may include writing to the local error log and notifying the operator through the CTY.

8. SPU Checks/Updates PEM and RIC Firmware

   The SPU checks the version numbers returned by the PEM and updates the firmware of the PEM or any RIC if the firmware is out of date.

   If required, the SPU will download any site-specific parameter limits to the PCS through a command to PEM.

9. SPU Commands PEM To Enable All Regulators

   The SPU commands the PEM to enable all regulators in the proper sequence, applying power to all logic, memory, and I/O. In turn the PEM reports the powerup command result (success or failure).

   NOTE
   The PEM will not allow power up into a fault condition.

10. PEM Enables Keep Alive Tasks

    On completion of power up,the PEM enables its keep alive task. That is, it begins to continuously monitor the power system and environment, reporting any limit violations or status changes to the SPU.

8.3 Service Processor Initialization

SPU initialization boot straps the SPU software and configures it based on the system configuration. The initialization is controlled by the SPM firmware.

The flow chart of Figure 8-2 summarizes the SPU initialization sequence. The flow operation block numbers correspond to the following description numbers.

1. SPU Boot Code Locates SYSBOOT.EXE

   The execution of the SPU self test leaves the SPM initialized with the available memory map in TBD GPRs to be passed to SPU_VMB. SPU_VMB resides in the SPM ROM and is responsible for initializing the KFBTA controller and locating SYSBOOT.EXE (VAXELN image) on the RD53.

   Error reporting will switch to the SPU at this point as the terminal line is known good. No further error indicators will be displayed on the OCP.

   Failures at this step will be displayed on the SPU console terminal and the SPU will remain in the BOOT ROM. No further action will be taken without manual intervention.

2. SPU Boots System Image

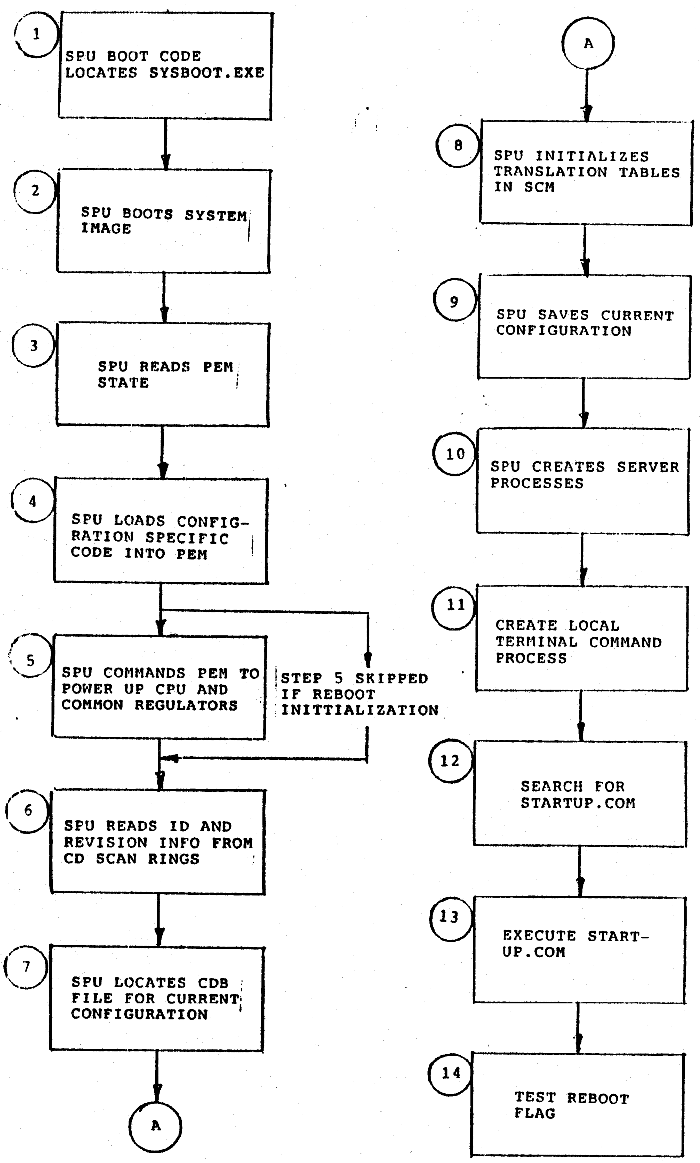

1
SPU BOOT CODE LOCATES SYSBOOT.EXE
2
SPU BOOTS SYSTEM IMAGE
3
SPU READS PEM STATE
4
SPU LOADS CONFIG-RATION SPECIFIC CODE INTO PEM
5
SPU COMMANDS PEM TO POWER UP CPU AND COMMON REGULATORS
STEP 5 SKIPPED IF REBOOT INITTIALIZATION
6
SPU READS ID AND REVISION INFO FROM CD SCAN RINGS
7
SPU LOCATES CDB FILE FOR CURRENT CONFIGURATION
A
A
8
SPU INITIALIZES TRANSLATION TABLES IN SCM
9
SPU SAVES CURRENT CONFIGURATION
10
SPU CREATES SERVER PROCESSES
11
CREATE LOCAL TERMINAL COMMAND PROCESS
12
SEARCH FOR STARTUP.COM
13
EXECUTE START-UP.COM
14
TEST REBOOT FLAG

Figure 8–2 SPU Initialization Sequence

VMB will load SYSBOOT.EXE and begin execution. At this point VAXELN begins kernel initialization. Currently KERNEL initialization takes less than 15 seconds (includes driver initialization). VAXELN will load auxiliary images as required and will begin executing the SPU initialization image.

Failures at this point will bugcheck to the SPU BOOT ROM. Error information is displayed on the SPU console terminal and no further action is taken without manual intervention.

3. SPU Reads PEM State

The SPU uses the state of the PCS to determine if an SPU REBOOT has just occurred or if this is a cold start. If the power system is off (not including BBU) the SPU concludes it is a cold start. Otherwise a REBOOT initialization is performed.

The basic difference is that a REBOOT initialization will not affect the current state of the CPU(s). The state will be sensed and restored to the SPU. A cold start will initialize the CPU(s). State read includes the OCP switch positions.

4. SPU Loads Configuration Specific Code Into PEM

   To reduce configuration-specific PROMS for the PEM, the PEM supports a writeable control RAM. This RAM contains the code which controls configuration-specific functions. The RAM is also used as a patch for loading corrections and updates to the PROM.

5. SPU Commands PEM To Power Up CPU And Common Regulators

   The following describes the action taken if the SPU determines a cold start is required.

   The PEM is requested to enable all regulators. This applies power to the CPU and SCU logic assemblies. This power on is required to check the configuration of the logic MCUs and verify the revisions via scan rings.

   This step is skipped on a REBOOT initialization.

6. SPU Reads ID And Revision Information From CD Scan Rings

   The SPU requests the SCM to sense the ID and revision information from the CD chips located on each MCU. The scan rings in this chip are fixed and may be read with the system clocks running. This is required to correctly determine the revision and configuration present on a REBOOT. (Alternately this information may be saved on the disk for future REBOOTs and eliminate this step).

7. SPU Locates The CDB File For The Current Configuration(s).

   The Configuration Data Base (CDB) file describes the scan rings in a processor for a given revision. The file contains register, signal, and STRAM configuration data and allows the initialization and control of the processor(s) to be data driven. This file is required for further scan access.

   If the correct file revision can not be located the initialization may stop. The processor(s) that do not have CDB files will be removed from the VAXSET and initialization will continue on the remaining processor(s). If no processors remain the initialization will stop and the SPU will enter CIO mode (if enabled).

   Once in CIO mode no further action is taken without manual intervention. The removed processors will remain off line until the correct CDB is installed. They may then be initialized and brought on line where they will become available to enter the VAXSET.

8. SPU Initializes Translation Tables In SCM

   Part of the CDB file contains the scan descriptor tables used by the SCM. These tables are initialized by the SPM with data read from the CDB. Two tables exist in the SCM and therefore two configurations will be supported in hardware.

   If more than two configurations exist in one system the SPM must assist the SCM in translation. The tables may be swapped during execution.

9. SPU Saves The Current Configuration

   The SPU will save the configuration information on the RD53 disk for future initializations.

10. SPU Creates Server Processes

    The server processes are started for remote access, power system, and error handling.

11. Create Local Terminal Command Process

    The local terminal port (OPA0)is connected to a command interpreter to begin execution of STARTUP.COM. The monitors (processor servers) will be installed from the startup command file (STARTUP.COM).

12. Search For STARTUP.COM

    The file STARTUP.COM is located (if present) on the RD53 in the top level [TBD] directory and the commands are executed. When this command file finishes, the command process enters CIO mode or PIO mode based on the commands in the file and the OCP switches. If the file is not found the initialization continues.

13. Execute STARTUP.COM

14. Test Reboot Flag

    If the Reboot flag is set the initialization is complete. The SPU will determine the previous SPU state through TBD. The SPU will then either enter CIO mode or PIO mode attached to the primary processor.

    If the flag is clear this is a cold start and initialization continues with the processor initialization. Processor initialization may be controlled by the command file INIT.COM

8.4 Processor Initialization

Processor Initialization is the process of configuring the system hardware, loading the control stores, scanning in the initial state vector, configuring main memory and resetting the I/O subsystem. The processor is then in the initialized state and is ready to bootstrap the operating system.

The flow chart of Figure 8-3 summarizes the processor initialization sequence. The flow operation block numbers correspond to the following description numbers.

1. Initialize The MCM, ICD And CDC

   The SPU uses scan to initialize the CDC and MCM. The MCM initialization consists of setting the clock frequency and starting the oscillators. CDC initialization consists of loading STRAM clock group selects and clearing the status registers.

2. Save Clock Select Information

3. Scan In Reset Vector

   The processor reset consists of loading all latches with a predetermined pattern (vector) through the scan system. The vector is extracted from design data (parameters) in the Engineering CAE system.

4. Load Control Stores And Control RAMS

   Control Stores and RAMS will be loaded with the contents of predefined files on the RD53. The files are precompiled and bit-mapped (if required). The SCM controls location access. STRAM loading is accomplished by loading access registers and then stepping the STRAM clocks to load the new data.

5. Save RAM Image Files

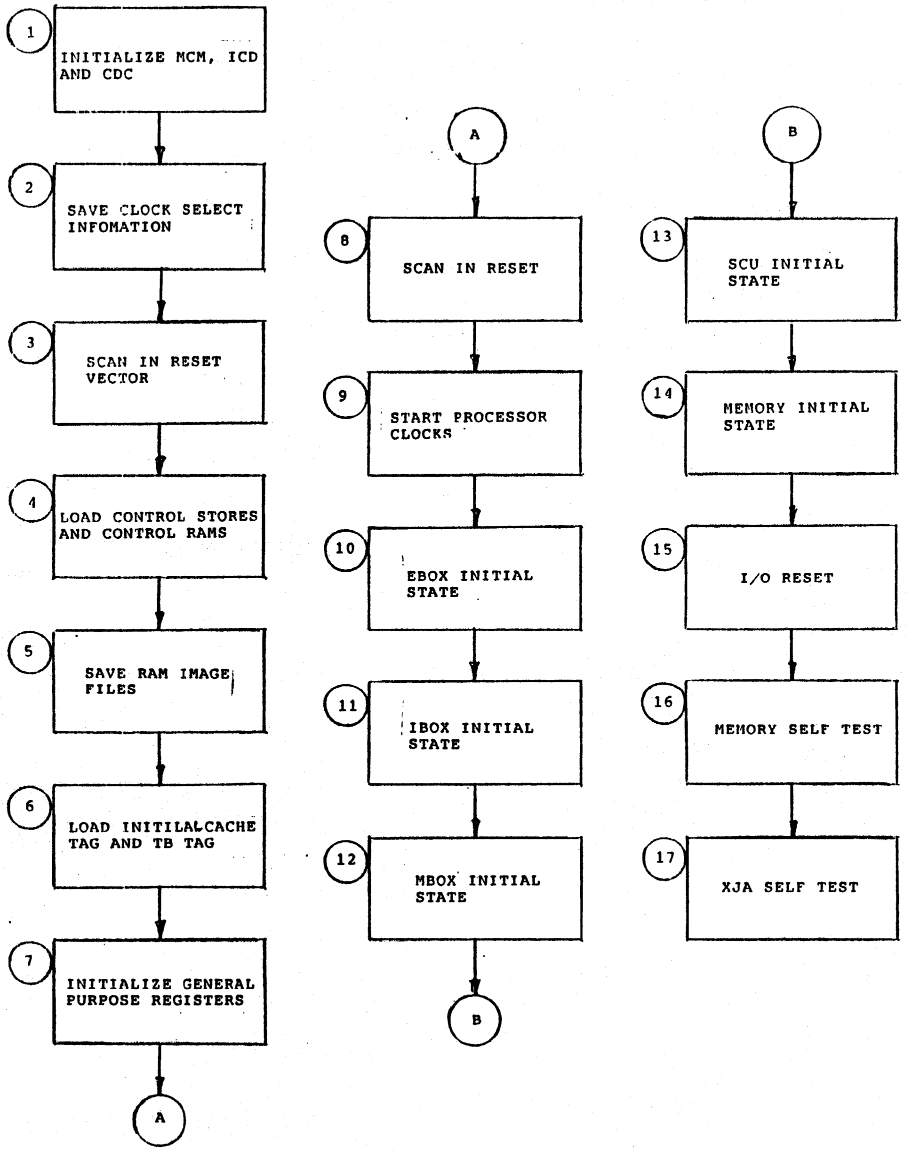


Figure 8–3 Processor Initialization Sequence

All loaded RAMs will have the contents saved in image files on the RD53 disk for error recovery. If the contents of the STRAM are modified by a LOAD command (partial or full) or by a DEPOSIT command, the image file will be updated. This will allow error recovery in all processor modes including diagnostics.

6. Load Initial Cache TAG And TB TAG

   The cache TAG and TB TAG will be cleared to indicate good parity with nothing valid.

7. Initialize General Purpose Registers

   All GPRs will be cleared and all IPRs will be initialized if not already initialized through the scan reset pattern.

8. Scan In Reset

The reset pattern will be applied again to complete the Reset sequence. The processor state is now initialized.

9. Start Processor Clocks

   The processor clocks will be started which will cause each functional box to enter the idle state.

10. EBox Initial State

    The EBox will be executing in a tight loop following Reset waiting to be dispatched to a FLUSH instruction. The FLUSH will take the address from a [TBS] scratch location. The PSL will have been previously deposited by the SPU. The EBox will issue an UNSUSPEND to the IBox through the FLUSH. The tight loop will be the same loop as is entered on a HALT instruction.

11. IBox Initial State

    The IBox will be in SUSPENDED state waiting for a FLUSH instruction from the EBox.

12. MBox Initial State

    The MBox will be looping on PORT requests from the IBox, EBox and SCU. Since the IBox is suspended, the EBox is halted and the SCU and MBox are idle.

13. SCU Initial State

    The SCU will determine if self test has been requested by the SPU. A bit in the SCU will be set by the SPU to indicate the memory is to run self test. The SCU will co-execute the memory self test (if requested) and then enter the idle loop waiting for PORT requests. If the self test is not requested (Power Fail Recovery) the SCU will enter the idle loop.

14. Memory Initial State

    The memory subsystem must power up without disturbing memory contents if Battery Backup is active. The memory subsystem will enter the idle loop waiting for PORT requests. If Self Test is requested the memory subsystem will co-execute with the JBox. Any errors will be reported to the SPU and the test will suspend.

15. I/O Reset

    The XJA (interface to the XMI) is reset from the SPM. The initial state of the XJA is (TBD). The assertion of XJA_RESET will cause an XMI RESET sequence which will cause XBIs (XMI to BI adapter) to cause a BI RESET sequence and so on. This is the action of the UNJAM command.

    If a RESTART is to be attempted, the following two initialization steps are skipped. This is to avoid disturbing memory contents.

16. Memory Self Test

    The memory self test consists of five passes through each cell in the memory. The first two passes consist of writing and reading random patterns. The second two passes consist of writing and reading the compliments of the first pattern and the last test clears all memory. Self Test must provide an indication of failed cells (pages or segments) to create the initial memory bitmap.

17. XJA Self Test

    The XJA Self Test will execute in first 64K bytes of main memory. The test contents are TBD. (TBD). The test is responsible for initializing each XJA, and locating and initializing all XBIs.

The XBI diagnostic will be executed main memory. The primary processor will execute the tests. The diagnostic will be passed a configuration mask indicating which XJAs are to be tested and initialized.

8.5 System Initialization

System initialization is the process of starting the operating system on an initialized processor set. The set of processors in the initialized state are referred to as the VAXSET.

The flow chart of Figure 8-4 summarizes the system initialization sequence. The flow chart operation box numbers correspond to the following description numbers.

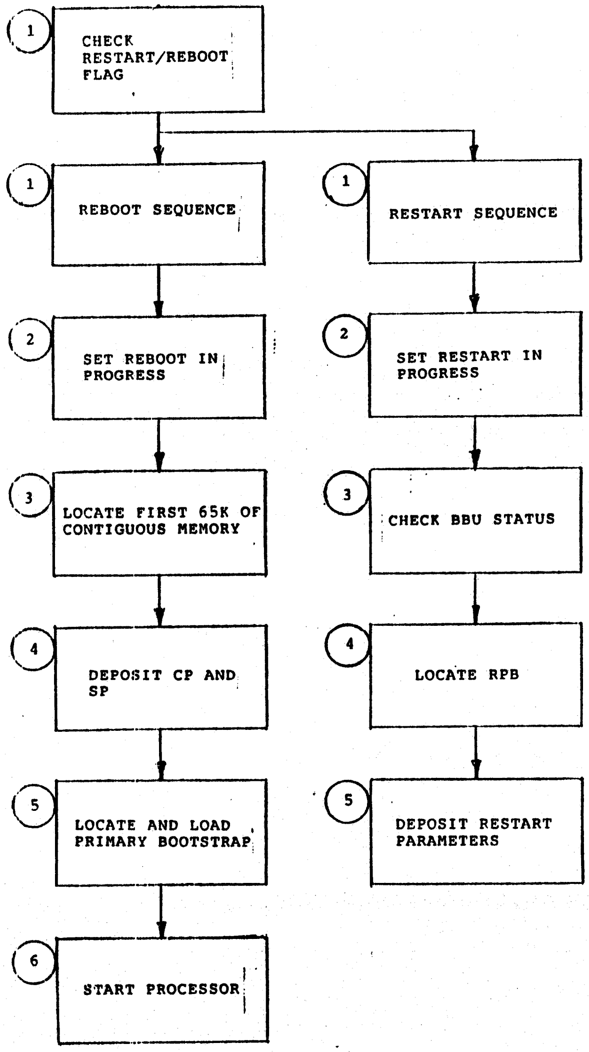

1 CHECK RESTART/REBOOT FLAG
1 REBOOT SEQUENCE
1 RESTART SEQUENCE
2 SET REBOOT IN PROGRESS
2 SET RESTART IN PROGRESS
3 LOCATE FIRST 65K OF CONTIGUOUS MEMORY
3 CHECK BBU STATUS
4 DEPOSIT CP AND SP
4 LOCATE RPB
5 LOCATE AND LOAD PRIMARY BOOTSTRAP
5 DEPOSIT RESTART PARAMETERS
6 START PROCESSOR

Figure 8–4 System Initialization Sequence

1. Check RESTART/REBOOT Flag

   The RESTART/REBOOT switch position determines if a RESTART attempt is to be made. If the switch is in the RESTART position a RESTART is attempted. If the RESTART fails, the REBOOT sequence is attempted. If the REBOOT fails the SPU enters CIO mode.

If the switch is set to HALT the SPU enters CIO mode. Once in CIO mode no further action is taken without manual intervention.

The following subsections describe the REBOOT and RESTART sequences. In both sequences the processor designated as the primary processor is bootstrapped. Each auxiliary processor waits in the initialized state until a request is received to start it.

8.5.1 Reboot Sequence

1. REBOOT Sequence

   The REBOOT_IN_PROGRESS flag is tested to determine if multiple attempts to reboot the processor have failed. If this flag is set the SPU enters CIO mode as described above. If the flag is clear the boot attempt continues.

2. Set REBOOT_IN_PROGRESS

   The REBOOT flag is set to inhibit future attempts to boot until the operating system has started initialization. This prevents errors from producing an infinite REBOOT attempt. The operating system will send a request to clear the flag when it starts initialization.

3. Find 64K Of Memory

   The SPU will search the memory bit map returned by the memory self test to locate the first contiguous 64K byte block of memory. This block will be used to load VMB. The bit map will be passed to VMB so that retesting main memory will not occur.

4. Deposit The PC And SP

   The PC is loaded with the address of the good memory and the SP is loaded with the PC + 200 (Hex).

5. Locate And Load Primary Bootstrap

   The image VMB.EXE will be loaded into main memory from the SPU RD53. If the RD53 is not operational, the TK50 will be used. VMB will be passed the main memory bit map so that memory testing will not be required of VMB.

6. Start The Processor

   The processor is started at the current PC. This will be the entry point to VMB. The primary CPU is now in the running state. The remaining members of the VAXSET will be started when a request is received from VMS. The start address is passed at that time.

8.5.2 Restart Sequence

1. RESTART Sequence

   The RESTART_IN_PROGRESS flag is tested to see if multiple attempts to RESTART the processor have failed. If this flag is set the SPU attempts a REBOOT as described above. If the flag is clear the RESTART attempt continues.

2. Set RESTART_IN_PROGRESS

The RESTART flag is set to inhibit future attempts to RESTART until the operating system has started initialization. This prevents errors from producing an infinite RESTART attempt. The operating system will send a request to clear the flag when it starts initialization.

3. Check BBU Status

   The BBU status is tested to determine if the memory contents have been preserved. This avoids a long search for the RPB if the memory is dead. If the BBU status indicates that it was OFF when the power was restored, a REBOOT is attempted.

4. Locate The RPB

   The RPB is located through the SPM to SCU interface (SJA on the SPM). The first longword of every page is checked to determine if it contains it's address. If so, the second longword is tested to see if it is a valid address and is non zero.

   If this succeeds the first 31 longwords at this address are checksummed and compared against the third longword of the page. If this succeeds the RPB is found. The LSB of the fourth longword is tested (RESTART_IN_PROGRESS) and if clear the RESTART continues. If any of the above tests fail the REBOOT sequence is entered.

5. Deposit Restart Parameters

   The PC is loaded with the address at the second longword of the RPB. The AP is loaded with the HALT code (reason for restart) and the SP is loaded with the address of the RPB + 200 (hex). Note that each member of the VAXSET has the same register content.

Each member of the VAXSET is started at the restart PC. The VAXSET is now in the run state. Note that if an exception occurs before each processor modifies it's SP, the stack will be corrupted.

9 System Interrupts and Exceptions

9.1 Chapter Objective

The chapter objective is to introduce and provide an overview of the system interrupts and exceptions. The chapter content is based on the Interrupt and Exception Engineering Specification.

9.2 Interrupt and Exception Handling

The EBox handles interrupts and exceptions. The beginning and end of the EBox pipeline are where the microcode begins to handle these conditions.

The IBox handles memory faults, reserved opcodes, and reserved addressing mode faults. Memory management faults in the IBox can be Istream and operand references.

The MBox contains registers for memory management handling. The MBox handles the occurrence of a TB miss without EBox intervention. This allows the EBox to handle only exceptions which change the instruction flow and cause an IBox flush.

The SCU contains registers for hardware interrupts. The System Control Block (SCB) offset for pending interrupts is maintained in the SCU or the interrupting adapter. The I/O adapters are accessed through physical addresses in the SCU.

9.2.1 Interrupt Types and Sequences

Interrupts are specified as two types: hardware generated interrupts and software generated interrupts.

Software generated interrupts are handled entirely by the EBox microcode and hardware. The interrupts are held in the same CPU where generated.

Hardware generated interrupts are generated in the SCU, which dispatches a particular EBox to respond to the interrupt. The interrupts are assigned to a single CPU or in rotating fashion.

The EBox responds when its IPL is lowered to the appropriate level. The microcode will read an SCU register to determine the correct SCB vector. The SCB vector then provides the correct address of the interrupt service routine. The MBox and IBox do not get involved in interrupt servicing.

Interrupts are processed at the end of an instruction or under microcode control. Those interrupts requiring microcode control are processed at certain well defined points in the microflow.

Interrupts between instructions are generated by the hardware. These instructions divert the instruction flow to the interrupt microtrap address.

Interrupts under microflow are checked by microbranching. Interrupts occurring in instructions are directed into the handler by the microflow.

Figure 9-1 summarizes the basic interrupt sequence for hardware generated interrupts.

The following subsections describe the remaining interrupt types. Note that the IPL level and SCB offsets are expressed in hexadecimal (i.e., h = hex).

9.2.1.1 CPU Power Fail Interrupt

When the SPU detects a power failure condition it will cause a Power Fail Interrupt. The interrupt is generated at IPL level 1Eh. The SCB offset vector is 0Ch. The microcode will follow the standard interrupt procedure.

9.2.1.2 Interval Timer Interrupt

Each CPU has its own interval counter contained in the EBox. The interval timer is incremented at 1 microsecond intervals. The three registers used to control the interval timer are described in the following paragraphs.

The Interval Count Register (ICR) is a read only register. It is loaded from the Next Interval Count Register (NICR) whenever it overflows. When the register overflows an interrupt is generated if it is enabled. Processor initialization leaves ICR unpredictable.

The Next Interval Count Register (NICR) is a write only register and contains the value to be loaded into ICR. ICR is loaded only when it overflows. Processor initialization leaves NICR unpredictable. The Interval Clock Control Status Register (ICCS) contains the control and status information for the clock. Figure 9-2 and Table 9-1 describe the ICCS register.

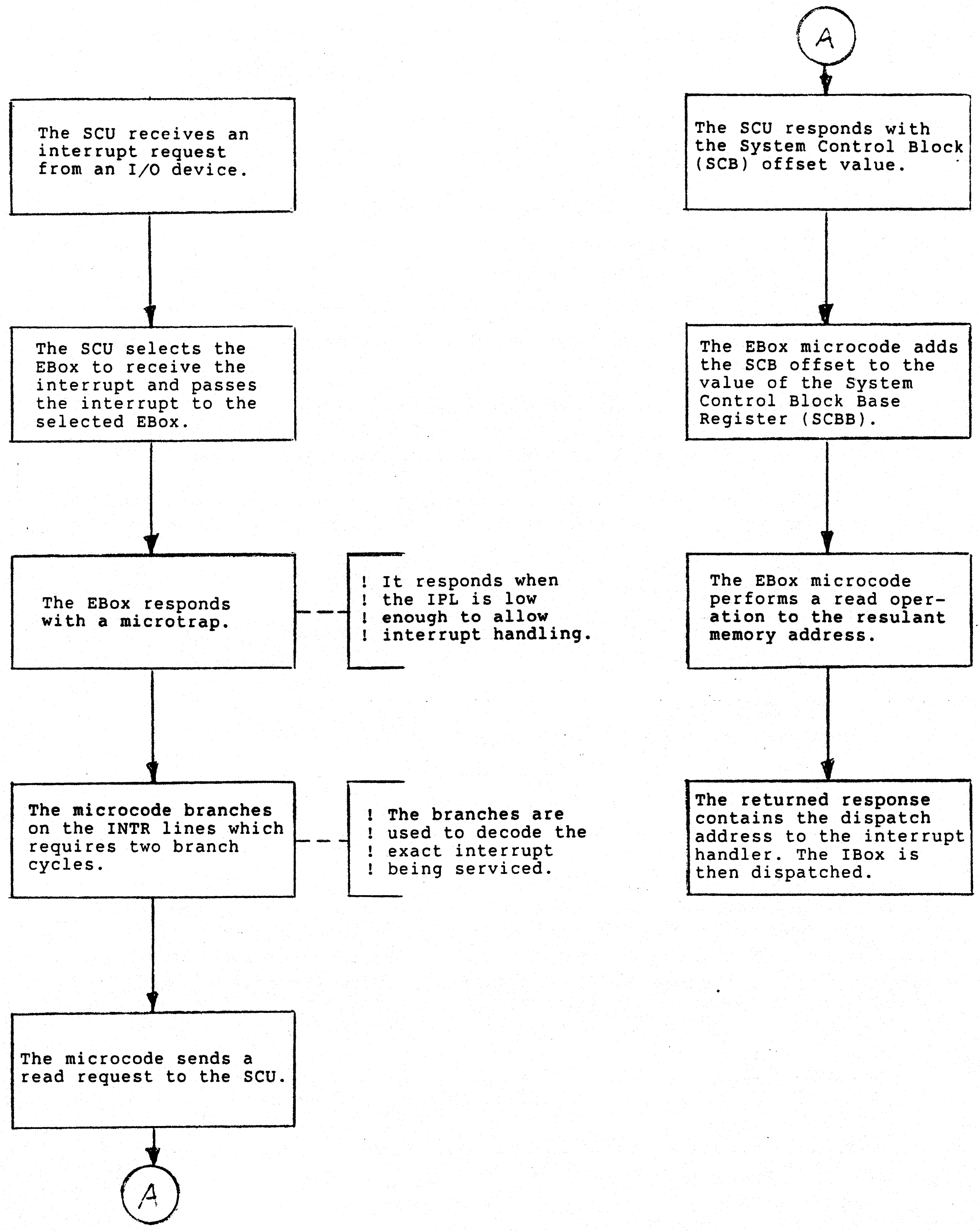

The SCU receives an interrupt request from an I/O device.
The SCU selects the EBox to receive the interrupt and passes the interrupt to the selected EBox.
The EBox responds with a microtrap.
! It responds when ! the IPL is low ! enough to allow ! interrupt handling.
The microcode branches on the INTR lines which requires two branch cycles.
! The branches are ! used to decode the ! exact interrupt ! being serviced.
The microcode sends a read request to the SCU.
A
A
The SCU responds with the System Control Block (SCB) offset value.
The EBox microcode adds the SCB offset to the value of the System Control Block Base Register (SCBB).
The EBox microcode performs a read operation to the resulant memory address.
The returned response contains the dispatch address to the interrupt handler. The IBox is then dispatched.

Figure 9-1 Interrupt Sequence Summary

31	30 8	7	6	5	4	3 1	0
ERR	MBZ	INT	IE	SGL	XFR	MBZ	RUN

Figure 9–2 ICCS Register Format

Table 9–1 ICCS Register Field Descriptons

BIT	NAME	FUNCTION
0	RUN	When set the ICR increments. When clear ICR does not run. Reset by processor initialization.
4	XFR	Transfer. Write only. When set NICR is transferred to ICR.
5	SGL	Single Step. If run is cleared it causes ICR to increment by one. Write only.
6	IE	Interrupt Enable. When set an interrupt request is generated every time ICR overflows. When clear no interrupt is generated. Processor initialization clears this bit.
7	INT	Interrupt. Set by hardware whenever ICR overflows. If Interrupt Enable is set then an interrupt is also generated. Writing a 1 to this bit clears the interrupt.
31	ERR	Error. Whenever ICR overflows and Interrupt is set then Error is set. Writing a 1 to this will clear the bit.

9.2.2 Software Requested Interrupts

Software requested interrupts are handled entirely in the EBox, and are assigned IPL levels between 01h and 0Fh. The following subsections describe the registers associated with software interrupts.

9.2.2.1 Software Interrupt Summary Register(SISR)

The SISR register contains 15 interrupt levels, and is accessed by MxPR instructions. The SISR contains ones in the bit positions corresponding to the levels at which software interrupts are pending. When the processor initiates a software interrupt the corresponding bit in the SISR is cleared. Figure 9-3 describes the register format.

31 16	15 1	0
MBZ	Pending Software Interrupts F E D C B A 9 8 7 6 5 4 3 2 1	MBZ

Figure 9–3 SISR Register Format

9.2.2.2 Software Interrupt Request Register (SIRR)

The SIRR register does not physically exist in hardware. The microcode will convert writes to this register to a bit position in the SISR. The EBox hardware will then generate an interrupt if the current IPL is low enough. Figure 9-4 describes the register format.

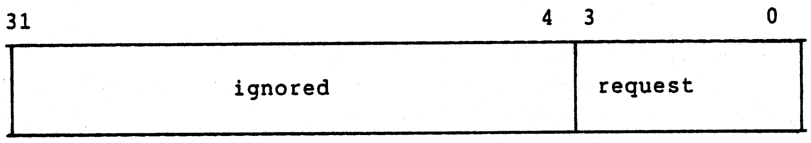


Figure 9–4 SIRR Register Format

9.2.2.3 Asynchronous System Traps (ASTLVL)

The ASTLVL register is checked only at REI for valid pending software traps. If a valid AST is pending, bit 2 of the SISR register is set. The EBox hardware will then generate the interrupt at the correct time. Figure 9-5 and Table 9-2 describe the register format.

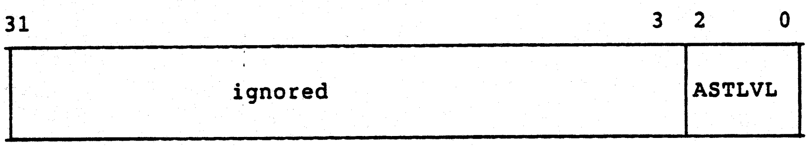


Figure 9–5 ASTLVL Register Format

Table 9–2 ASTLVL Register Field Descriptions

VALUE	DEFINITION
0	AST pending for access mode 0 - Kernel
1	AST pending for access mode 1 - Executive
2	AST pending for access mode 2 - Supervisor
3	AST pending for access mode 3 - User
4	No pending AST

9.2.3 Exception Handling

Table 9-3 describes the exceptions generated on the system, which is the standard VAX exception list. The location where each exception is generated is also shown in table. All arithmetic exceptions use the same SCB vector. A code is pushed on the stack to identify the particular exception. The following subsections describe the exception types.

Table 9–3 System Exceptions

EXCEPTION TYPE	LOCATION	GENERATED BY	TYPE	SCB
INTEGER OVERFLOW	EBOX	INT,MUL,DIV,FL	TRAP	34
INTEGER DIV BY 0	EBOX	DIV	TRAP	34
FLOATING OVERFLOW	EBOX	FLOAT,MUL,DIV	FAULT	34
FLOATING DIV BY 0	EBOX	DIV	FAULT	34
FLOATING UNDERFLOW	EBOX	FLOAT,MUL,DIV	FAULT	34
DECIMAL DIV BY 0	EBOX	UCODE	TRAP	34
DECIMAL OVERFLOW	EBOX	UCODE	TRAP	34
SUBSCRIPT RANGE	EBOX	UCODE	TRAP	34
RESERVED OPERAND	EBOX	FLOAT,MUL,DIV,UCODE	FAULT	18
RESERVED ADR MODE	IBOX	OPU	FAULT	1C
ACCESS VIOLATION	E,IBX	IBUF,OPU,EBOX	FAULT	20
TRANS. NOT VALID	E,IBX	IBUF,OPU,EBOX	FAULT	24
TRACE TRAP	EBOX	RETIRE	TRAP	28
IS NOT VALID	EBOX	UCODE	HALT	NONE
KERNEL SP NOT VALID	EBOX	UCODE	ABORT	08
MACHINE CHECK	EBOX	UCODE	ABORT	04
SUBSET EMULATION	EBOX	UCODE	TRAP	C8
SUSPENDED EMULATION	EBOX	UCODE	FAULT	

9.2.3.1 EBox Exceptions

The EBox will dispatch the microcode to service all exceptions. Some exceptions are generated in the IBox. Those exceptions must be passed to the EBox for the exception to be executed in the order of the instruction stream.

The EBox is not required to save the state of the pipeline. All exceptions cause the EBox to dispatch the IBox for a new Istream. TB miss processing is done entirely by the MBox. It is not a microtrap in the EBox.

9.2.3.2 Machine Check Exceptions

Machine Checks are handled by pushing a serial number, the faulting PC, and the PSL on the stack. The console will contain a record of all errors which occur in the system. The serial number pushed on the stack will provide the index into this database.

9.2.3.3 Kernel Stack Not Valid Abort

This indicates that the Kernel stack was not valid while pushing data onto the Kernel stack during the initiation of an exception. The IPL is raised to 1Fh. The PSL and PC of the original exception are saved on the interrupt stack. If the exception vector <1:0> is not 1, the processor will use the interrupt stack.

9.2.3.4 Arithmetic Exceptions

All arithmetic exceptions use the same SCB vector, 34h. A different code is pushed on the stack for the different conditions. Table 9-4 lists the exceptions and exception codes. Note that there are both faults and traps in the arithmetic exception group.

Table 9–4 Arithmetic Exception Codes

TYPE	CODE	EXCEPTION TYPE
Trap	1	Integer Overflow
Trap	2	Integer Divide by Zero
Trap	4	Decimal Divide by Zero
Trap	6	Decimal Overflow
Trap	7	Subscript Range
Fault	8	Floating Overflow
Fault	9	Floating Divide by Zero
Fault	A	Floating Underflow

9.2.3.5 Memory Management Faults

There are two principle memory management exceptions:

- o Translation Not Valid Faults
- o Access Control Violation Fault.

Both exceptions are reported to the microcode with a microtrap vector. The microcode reads an MBox register to determine which exception is being generated. The MBox returns a code which provides all the information required to service the exceptions. Figure 9-6 and Table 9-5 provide the register format and field definitions. The microcode will push this content on the stack together with the faulting address. Memory management faults may be generated by the I and E boxes.

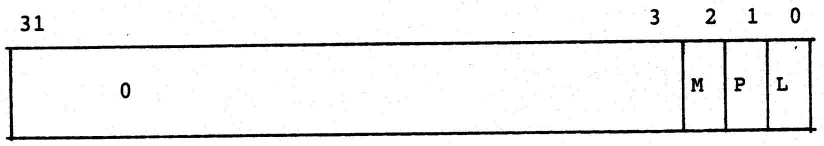


Figure 9–6 Memory Problem Register Format

Table 9–5 Memory Problem Register Field Descriptions

NAME	BIT	DEFINITION
M	2	Modify. This bit indicates the intended access was write or modify.
P	1	PTE reference. Set to 1 to indicate the fault occurred on a reference to the process page table.
L	0	Indicates if set that the Access Control Violation was the result of a length violation rather than a protection violation. Always zero for a translation not valid fault.

9.2.3.6 Privileged Instruction Exceptions

Some instructions may only be executed in Kernel mode. The microcode will check the current mode before attempting to execute them. If the mode is not Kernel a privileged instruction exception is generated. The SCB offset is 10h. The Kernel mode instructions are:

- HALT
- MTPR
- MFPR
- LDPCTX
- SVPCTX

9.2.3.7 Emulation Exceptions

The system does not implement the full VAX instruction set. When certain instructions in the Character String Group are encountered an emulation exception is taken. These exceptions allow the use of First Part Done (FPD) in the emulation code. This allows the instruction to be treated as though it were implemented in hardware. There are two exceptions:

- Subset Emulation Trap
- Suspended Emulation Fault.

The trap is taken when FPD is not set. The exception is used when FPD is set. The SCB offset for Subset Emulation Trap is C8h. The SCB offset for the Suspended Emulation is CCh. These exceptions do not affect the current mode. If the IS bit is set, the processor will halt.

9.2.3.8 CHMx Exceptions

To be supplied.

9.2.3.9 Vector Instruction Exceptions

The VBox processes Vector class instructions. Since vector instructions are not retired in the order of their issue (as are the scalar instructions) the PC pushed onto the stack does not indicate the instruction causing the fault. The exceptions generated in the VBox are:

- Floating Underflow
- Floating Divide by Zero
- Floating Reserved Operand
- Floating Overflow

- Integer Divide by Zero
- Integer Overflow
- Machine Check

9.2.4 Hardware Generated Interrupt Overview

The SCU arbitrates interrupts from I/O devices and other CPU's. The Central System Interrupt Arbiter (CSIA) is located in the SCU and contains registers for sending interrupts to specific CPUs. The CSIA will deliver the following interrupts to the selected EBox:

- Console TRX at IPL level 14h.
- Console TTX at IPL level 14h
- Console storage receive at IPL level 17h
- Console storage transmit at IPL level 17h
- Power fail at IPL level 1Eh
- JBox memory errors at IPL level 1Dh
- XJA fatal errors at IPL level 1Dh
- CPU interprocessor interrupts at IPL level 14h
- XJA interrupts at IPL levels 14,15,16,17h

9.2.4.1 Interrupt Control Register

The Interrupt Control Register (INTRCTRL) register is located in I/O space at location 3E20 0000h, and is initialized to zero. Figure 9-7 and Table 9-6 provide the register format and field definitions.

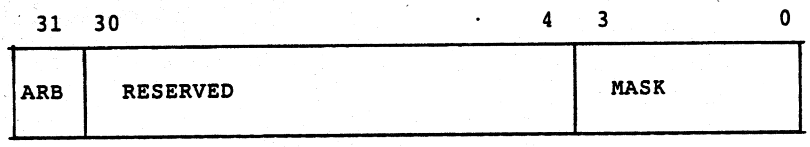


Figure 9–7 Interrupt Control Register Format

Table 9–6 Interrupt Control Register Field Definitions

FIELD	DEFINITION
ARBITRATION	If set this bit sets SCU JBox arbitration mode to round robin. If clear the SCU will broadcast interrupts to the CPU's enabled by the mask bits.
MASK	This field enables the broadcast of interrupts to a specific CPU. If a bit is set it allows the interrupt broadcast. The field encoding is specified below: 0000 - No interrupts xxx1 - CPU 0 receives interrupts xx1x - CPU 1 receives interrupts x1xx - CPU 2 receives interrupts 1xxx - CPU 3 receives interrupts

9.2.4.2 Interprocessor Interrupt Control Register

The content of the Interprocessor Interrupt Control Register (IPINTRCTRL) is CPU-specific, and is used to direct an interrupt from one processor to another. The register address is 3e200004h in I/O space. Figure 9-8 and Table 9-7 provide the register format and field definitions.

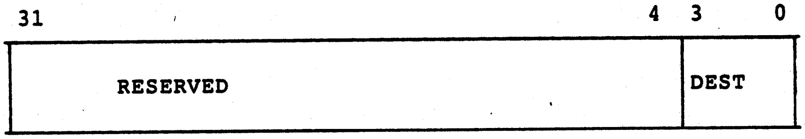


Figure 9–8 Interprocessor Interrupt Control Register Format

Table 9–7 Interprocessor Interrupt Control Field Definitions

FIELD	DEFINITION
Destination	The field indicates which CPU receives the interrupt. When the interrupt is delivered the bit are cleared. The field encoding is specified below: 0000 - No interrupts xxx1 - CPU 0 receives interrupts xx1x - CPU 1 receives interrupts x1xx - CPU 2 receives interrupts 1xxx - CPU 3 receives interrupts

9.2.4.3 XMI Device Vectors

The vectors returned by native XMI devices (including XBI vectors returned for XBI error interrupts) in response to XMI IDENT transactions (generated in response to a read from an XJA SCB Offset Register), for which an XMI node has generated the interrupt request, are described in Figure 9-9 and Table 9-8.

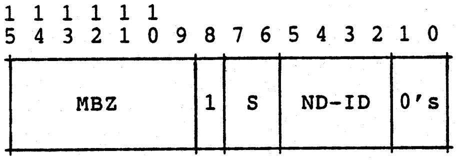


Figure 9–9 XMI Device Vector Format

Table 9–8 XMI Device Vector Field Definitions

FIELD	DEFINITION
Node ID	The XMI node ID of the interrupting node 0-15
S	the interrupt vector number which can be one of four possible interrupt vectors per node
1	Initialized to one to insure that the device vector resides above the processor area of the SCB - first 256 bytes of the SCB
MBZ	Must be zero to specify the first page of vector offsets in the SCB

VMS will assign vector values to each of four possible vector registers that may exist on XMI devices that are capable of generating interrupt requests during initialization.

The EBox servicing a given XMI interrupt will append the XJA number (0 - 3) to bits <10:09> of the returned XMI vector only if bits <15:09> of the returned vector are zero.

9.2.4.4 BI Device Vectors

The second type of interrupt vector is the format returned by BI nodes. The major difference between this vector format and the XMI format is that bits <15:09> are non-zero, and are assigned a value by the operating system software during initialization. This offset value, contained in the BVOR (Vector Offset Register) on the XBIB will be concatenated with the Vector value returned by a BI node, bits <08:02>, providing that bits <13:09> of the BI vector are equal to zero (non-offsettable bus adapter).

This new value will be returned to the XJA during an XMI IDENT cycle (generated in response to a read from an XJA SCB Offset Register), for which a BI node has generated the interrupt request. If bits <13:0> of the BI vector are nonzero, the vector will not be concatenated with the VOR register, and will be passed to the XJA unchanged. The assumption is that a vector with bits <13:09> being nonzero, is generated by a BI node with an offsettable bus.

BI device initiated interrupts will return vectors described in Figure 9-10 and Table 9-9 in response to an XMI IDENT transaction.

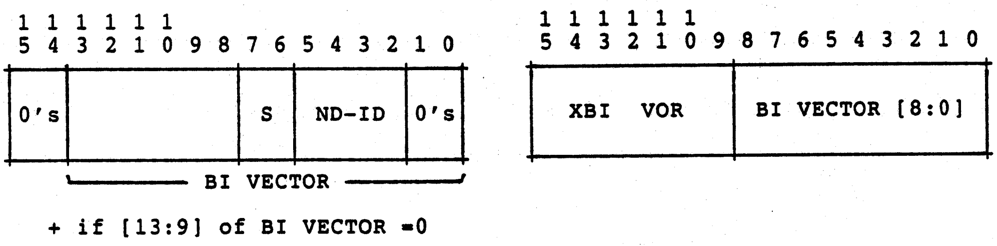


Figure 9-10 BI Device Interrupt Vector Format

Table 9-9 Interrupt Vector Field Definitions

FIELD	DEFINITION
Node ID	The BI node ID of the interrupting node 0-15.
S	The interrupt vector number which can be one of four possible interrupt vectors per node.
VOR	Must be a non-zero software assignable offset value to be used to index into the SCB with a unique vector for multiple BI devices. Note that the VOR bits [15:09] may be supplied by the XBI as noted above.

The VOR register is necessary since an XMI is capable of supporting multiple XBI nodes, where the same device may exist on multiple BI's. Since some BI nodes may have fixed vectors that are unchangeable by software, the VOR is used as a means of insuring that multiple BI devices with fixed vectors have a unique entry point into the SCB.

VMS will assign unique XBI VOR values that do not conflict with the native XMI device SCB pages. That is, the value of all XBI VOR's must be different and must be of a value greater than the number of XJA's present in a given system minus 1.

The EBox servicing a given XMI interrupt will use, unmodified, a returned XMI vector only if bits <15:09> of the returned vector are nonzero.

9.2.4.5 Offsettable Bus Vectors

The third type of interrupt vector is one that is returned by offsettable devices. Several examples of this type are the DWBUA (BI to UNIBUS Adapter), and the BI-LESI (BI to Low End Storage Interconnect). All of these devices are characterized by the fact that the other bus can support devices that can generate interrupts, and these requests must be differentiated from other Vectors such as those generated by BI devices.

Figure 9-11 and Table 9-10 describe the implementation of the Unibus.

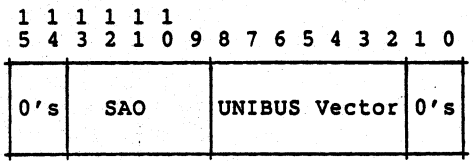


Figure 9-11 UNIBUS Interrupt Vector Format

Table 9–10 Interrupt Vector Field Definitions

FIELD	DEFINITION
Unibus Vector	The Unibus vector portion of this type of vector is an architectually fixed vector that is returned by Unibus devices. Bits <8:0> cannot be modified by software.
Software Assignable Offset	Must be a nonzero software assignable offset value to be used to index into the SCB with a unique vector - physically located and supplied by the DWBUA BI to Unibus adapter.

VMS will assign unique Software Assigned Offset (SAO) values that do not conflict with the native XMI device SCB pages or XBI VORs. That is, the value of all SAO's and XBI VOR's must be different and must be of a value greater than the number of XJA's present in a given system minus 1.

The EBox servicing a given XMI interrupt will use, unmodified, a returned XMI vector only if bits <15:09> of the returned vector are nonzero.

9.2.5 XJA INTERRUPTS

The XJA delivers two types of interrupts to the SCU CSIA: normal vectored interrupts at IPL14h to 17h, and normally fatal IPL1Dh interrupts.

9.2.5.1 XJA Vectored Interrupts

The XJA delivers normal vectored interrupts to the ICU through a INTR type JXDI transaction. The IPL is contained in bits <7:4> of the JXDI command word. The JDA MCA on the SDA MCU (part of the logical ICU) encodes bits <7:4> on the JDAIRCINTRH <1:0> wires. The IRC MCA implements the physical CSIA and takes these bits and delivers the requested interrupt to an EBox.

The EBox, on receiving the interrupt request from the CSIA will, as soon as its IPL is lower than the IPL of the requested interrupt, issue a read request to one of the four XJA SCB Offset registers corresponding to the four IPL levels. The data returned in response to this read request is the offset into the SCB where the EBox can find the vector that points to the interrupt service routine.

The source of the SCB offset returned by the XJA, is dependent on the source of the interrupt. Table 9-11 specifies the possible vectored interrupts, IPL levels and source of SCB offset.

Table 9–11 Vectored Interrupt Types and Offsets

INTERRUPT TYPE	IPL	SCB OFFSET SOURCE
XMI INTR	14h to 17h	XMI Ident
XMI IPINTR	14h	XJA supplied 80h
XJA Error/Stat	17h	XJA Configuration Register<15:2>

From the perspective of EBox ucode, all SCB offsets for XJA IPL14h to 17h interrupts are to be found in the four XJA SCB offset registers. The XJA will determine whether to issue an XMI IDENT and return the response, return 80h, or return the contents of XJA CNF <15:2> in response to a read request to one of the XJA SCB offset registers.

9.2.5.2 Fatal XJA Interrupts

The XJA will:

- on receipt of an XMI IVINTR transaction that specifies a Write Error Interrupt, or
- on the assertion of XMIFAULT

assert the JXDI signal XJAFATALERRH. The JDC MCA (part of the logical ICU) will pass this signal to the IRC MCA (CSIA) which will deliver the requested interrupt to an EBox. The SCB offset is implied to be 60h at which location in the SCB, should be a pointer to a fatal XJA error routine. The assertion of XJAFATALERR does not necessarily indicate the XJA is incapable of responding to further CPU requests.

9.2.6 System Halts

The following subsections describe the conditions which will cause the system to halt.

9.2.6.1 HALT Instruction

The halt instruction is a privileged instruction. It will halt the machine if the CUR MDE of the PSL is Kernel. All other modes generate a Reserved to Digital exception. This exception uses SCB offset of 10h.

9.2.6.2 Console Halt

If the console halt is generated the microcode is trapped into the the vector at microcode address 20Bh. The processor stops execution of macroinstructions after completion of the current macroinstruction.

9.2.6.3 Interrupt Stack Not Valid

An interrupt stack not valid halt results when the interrupt stack was not valid or a memory error occurred while the processor was pushing data onto the interrupt stack during the initiation of an exception or interrupt. The processor leaves the PC and the PSL in CPU registers for the console to use.

9.2.6.4 Double Error Halt

To be supplied.

9.2.6.5 Incorrect SCB Vector

This occurs when the returning SCB vector contains an incorrect value in the bottom lower two bits. The only allowed values at this time are 00 and 01. All other vectors result in a halt of the CPU.

9.2.6.6 CHMX Vector

This occurs when the CHMx instruction is executed with the IS bit set in the PSL. The CPU will halt before executing the instruction.

9.2.7 EBox Pipeline

The EBox pipeline is between three and five levels deep. The first stage is the lookup of a microword and check of the unit being used. This is called the issue stage. The next cycle is the actual execution of the instruction. In the Int unit, this is one cycle. The last stage is retirement of the instruction. The retire unit checks for exceptions and memory write faults before issuing instruction done. This indicates the instructions has successfully completed.

Any interrupts to be taken must be injected into the pipe at the first stage. This allows any arithmetic or memory management exceptions to be completed before the interrupt is serviced. The exceptions are arrived at either by microtrap or microcode branching. Microtraps are handled by shutting down instruction retire while doing a microcode lookup. The microcode will then clear the pipe stages and service the exception.

Some exceptions wait for the result queue to empty before allowing the microcode to proceed. Others, such as arithmetic exceptions, will occur at the end of the instruction and need to flush the pipeline before the microcode can proceed.

The EBox microcode branches on the INTR<4:0> bits to determine which Interrupt is to be serviced. Table 9-12 specifies the encoding of the bits for the microsequencer. The microcode will branch twice, on different bits, in the interrupt handler to get the correct interrupt. When the correct interrupt is found the microcode will dispatch the IBox to the correct address based on the SCB vector. The microcode will also push the correct data onto the stack.

It will then write into the Clear Int Register. This internal EBox register then clears the interrupt just processed. The microcode will then fork to the first instruction of the new routine. Any higher interrupts may then be serviced at this time.

Table 9-12 Interrupt Field Coding

INTERRUPT FIELD <4:0>	DEFINITION
0 0 0 0 0	No interrupt
0 0 0 0 1	Console TRX level 14h interrupt
0 0 0 1 0	Console TX level 14h interrupt
0 0 0 1 1	Console storage receive level 17h interrupt
0 0 1 0 0	Console storage transmit level 17h interrupt
0 0 1 0 1	Console powerfail level 1Eh interrupt
0 0 1 1 0	JBox Memory errors level 1Dh interrupt
0 0 1 1 1	Reserved - used by EBox for interval timer
0 1 0 x x	XJA xx fatal error level 1Eh interrupt
0 1 1 c c	CPU cc IPINTR level 14h interrupt
1 i i x x	XJA xx interrupt at IPL ii

where:

xx = 00—XJA0	ii = 00—14H
xx = 01—XJA1	ii = 01—15H
xx = 10—XJA2	ii = 10—16H
xx = 11—XJA3	ii = 11—17H

9.2.8 Microcode Microtrap Addresses

Table 9-13 contains the listing of the microtrap addresses for the microsequencer. The EBox dispatches the microcode to one of the addresses listed below. Some of these conditions will occur at the beginning of the EBox pipe; others occur at the end of the pipe.

Table 9–13 Micrcode Microtrap Addresses

ADDRESS	CONDITION
057	reserved opcode fork address
200	fork not valid
·ARITHMETIC TRAPS	
201	integer overflow
202	floating overflow
203	floating underflow
204	reserved operand fault
205	integer divide by zero trap
207	decimal overflow
MISCELLANEOUS TRAPS	
206	branch not correctly predicted
208	trace pending
209	interrupt
20A	FPD fault
20B	console halt
MEMORY MANAGEMENT	
20C	reserved addressing mode fault
20D	memory problem
STARTUP LOCATION	
20E	reset
20F	floating divide by zero fault
HARDWARE ERRORS	
210	EBox error
211	MBox error

Table 9–13 (Cont.) Micrcode Microtrap Addresses

ADDRESS	CONDITION
212	SCU error
213	IBox error
214-21F	currently unassigned

9.2.9 IBox Exceptions

The IBox decodes the following exceptions:

- o Reserved Opcode Fault
- o Reserved Addressing Mode Fault
- o Ibuffer Memory Problem
- o Operand Memory Problem

The Reserved Opcode Fault is passed to the EBox as any other fork address. The EBox forks to the exception location in a normal manner. The EBox microcode processes the fault. The SCB offset is 10h.

The Reserved Addressing Mode Fault is loaded into the queues with the pointer to the specifier. When the FRAM or microcode select the operand, the microsequencer directs the code to the handler location. Then it is handled as a normal microinstruction sequence. The functional units propagate their previous data to retire. The result queue is entirely empty before the handler routine retires any results. The SCB offset for Reserved Addressing Mode Faults is 1Ch.

The Ibuffer memory faults make there way to the EBox. The EBox provides the microaddress for the fault.

The read operand memory faults are sent by the MBox to the EBox. They are kept until the operand is selected. The Issue unit again sends a signal to Useq to signal the fault. The write faults are kept by the MBox until the EBox writes the data. The EBox must then flush the pipeline of further instructions. All memory problem exceptions use the standard SCB offsets:

- o 20h for Access Violations
- o 24h for Translation Not Valid

9.2.10 Processor Interrupt Registers

The EBox contains a number of registers used to service interrupts. The JBox also has SCB offset registers. The MBox holds memory management fault information.

9.2.11 EBox Registers

The EBox registers are either in Self Timed RAMs (STRAMs) or in the EBox hardware, with some having a copy stored in both places. The STRAM data is sent to the data path through the normal source methods. The hardware registers enter the data path through the Int unit shifter. The microcode can select the STRAM data on either source 1 or source 2. The hardware registers must be selected through the source 2 inputs. Some hardware registers are write only.

9.2.11.1 ASTLVL Register

The ASTLVL register is in the STRAM. It is checked by REI instruction for any pending software traps. The microcode in REI will set SISR <2> if there is a pending AST. This is an architecturally defined register.

9.2.11.2 SISR Register

The SISR register resides in the STRAM with a copy kept in the EBox hardware. The EBox checks this register against the current IPL to decide if an interrupt should be generated. The microcode will usually use the Int unit encoder circuit to set and clear bits in the SISR. The microcode must allow enough cycle after writing the register to allow the hardware to check the new value. This is an architecturally defined register.

9.2.11.3 SIRR Register

The SIRR an architecturally defined register; however, it is not a physical register. The microcode will make an entry into the SISR. This is done by using the encoder logic in the INTUNIT. The SIRR value is loaded into the encoder the SISR is then passed through it. The correct bit is set and the SISR is reloaded with the new value. The EBox hardware compares the IPL level against the value of the highest software interrupt. It then issues an interrupt when the IPL is low enough.

9.2.11.4 IPL Register

The Interrupt Priority Register (IPL) is a 5-bit wide architecturally defined register. It content is the same as the IPL field in the PSL. The IPL register allows setting the IPL field without writing the whole PSL. When the microcode writes this register it must allow enough cycles for the IPL to be set before setting instruction done. Otherwise an interrupt might be an instruction late.

9.2.11.5 SCBB Register

The System Control Block Base (SCBB) register is an architecturally defined register, and resides in the STRAM. It is accessed by the microcode to provide the base address to the SCB. The SCB provides information on handling exceptions and interrupts. It contains the address of the handling routine and the stack pointer to use while running.

9.2.11.6 Clear Interrupt Register

The Clear Interrupt (CLR INT) register is used to clear the Interrupt Pending bit. It also allows the clocking of the branch condition (INTR <4:0> bits) for the interrupts generated. The purpose of it is to ensure that the interrupt has reached the bottom of the pipeline and is being serviced. A exception which preempts it would not allow the bit to be reset. It is a write only register. Writing the value 01h will activate it. This is an implementation specific register.

9.2.11.7 Interrupt Vector Register

This register provides the address to the microcode for the correct XMI to read. This is an implementation specific register.

9.2.11.8 Interval Counter Control Register(ICCS)

The ICCS is an architecturally defined register, and is implemented in the EBox. It is used to control generation of interval timer interrupts.

9.2.12 SCU Registers

The SCU provides the address from the XMI of the interrupting device. The microcode will read these registers to compute the SCB offset. These are implementation specific registers. Given the EBox 5 bit wide INTR register, bits <3:2> determine the XJA, and bits <1:0> determine the IPL.

Table 9-14 lists the JBox registers, the priority level, and the SCB offset locations.

Table 9–14 JBox Registers and Locations

XJA NUMBER	IPL	SCB OFFSET
0	14	3E000040
0	15	3E000044
0	16	3E000048
0	17	3E00004C
1	14	3E080040
1	15	3E080044
1	16	3E080048
1	17	3E08004C
2	14	3E100040
2	15	3E100044
2	16	3E100048
2	17	3E10004C
3	14	3E180040
3	15	3E180044
3	16	3E180048
3	17	3E18004C

EBox microcode will first branch on bits <4:2> of the INTR field which will tell it if the SCB offset is implied or needs to be fetched from I/O space and if so which XJA to fetch it from. Microcode will then take a second branch on INTR bits <1:0> which will either further qualify the implied SCB offset interrupt or indicate at what IPL the given XJA interrupt request is for and consequently where the SCB location can be fetched from.

EBox microcode will, before the start of the execution of the first instruction of the interrupt routine, clear the corresponding bit in the EBox interrupt pending register. This is done by the write to the CLR INT register.

9.2.13 MBox Registers

The MBox contains two sets of memory fault handling registers. The Fault VA register is loaded with the address of the memory management fault. The Fault Parameter register is loaded with the type of fault.

There are two sets of these registers so that the EBox may make references which may override the references further up the pipeline. The MBox will save both sets of data. This way if the EBox makes a reference the IBox fault information will not be lost. These are implementation specific registers.

9.2.14 System Control Block

Table 9-15 specifies the vectors, vector names, type, and related notes.

Table 9–15 System Control Block Table

VECTOR	NAME	TYPE	NOTES
00	passive	int	
	release		
04	machine check	all	error information
08	Kernel Stack not valid	Abort	IPL is raised to 1F and IS is used
0C	Power Fail	Int	IPL is raised to 1E
10	reserved ins	fault	reserved or privileged opcodes
14	customer	fault	XFC instruction
18	reserved operand	fault	reserved operand
1C	reserved address	fault	
20	Access violation	fault	
24	TNV	fault	Translation Not Valid
28	Trace Pending	fault	Trace pending fault
2C	breakpoint	fault	breakpoint instruction
34	arithmetic	trp,flt	A type code is pushed on stack
40	CHMK	trap	

Table 9–15 (Cont.) System Control Block Table

VECTOR	NAME	TYPE	NOTES
44	CHME	trap	
48	CHMS	trap	
4C	CHMU	trap	
50	XJA error	fault	XJA fatal error, IPL is 1E
84	Soft lvl 1	int	IPL is 1
88	Soft lvl 2	interrupt	IPL is 2, used for AST
8C-BC	Soft lvls 3-f	interrupt	Vector corresponds to IPL
C0	interval timer	interrupt	IPL is 18
C8	sub emul	trap	FPD clear
CC	susp emul	fault	FPD set
F0	Con st rec	interrupt	IPL is 17
F4	Con st trx	interrupt	IPL is 17
F8	con ter rec	interrupt	IPL is 14
FC	con ter trx	interrupt	IPL is 14
100-1FC	adapters	interrupt	IPL is 14 to 17
200-5FC	devices	interrupt	IPL is 14 to 17

10
Diagnostic System Overview

10.1 Chapter Objective

The chapter objective is to provide a general introduction to the overall capabilities of the Diagnostic System. The content of this chapter is based on the Diagnostic System Plan.

10.2 Diagnostic System Summary

All standalone TTDs can be invoked by the SPU, which also has the capability of reporting status on the success or failure of the diagnostics. This will be accomplished by:

- o interfacing to built in self tests (BISTs) using the ROM Based Diagnostic (RBD) interface
- o loading and executing Level 4 diagnostics
- o loading and running the VDS

The system kernel will be tested at each subsystem level and also as a complete system.

There are four major schemes used for system diagnosis:

- o Initialization Testing
- o Standalone Diagnosis
- o User Mode Diagnosis
- o Symptom Directed Diagnosis

10.2.1 Initialization Testing

Initialization testing occurs when the system is powered on either as a result of a user action, or as a result of a restoration of power after a power failure. This testing detects non-functional devices and provides information to the CTY as to which devices failed.

Note that not all devices are critical to system availability. Non-critical devices will vary depending on system configuration. (Refer to Chapter 8, System Initialization for test descriptions and sequence.)

The following devices will be tested as a result of power-on initialization:

- o RICs (Regulator Intelligence Cards) (NOTE: RICs that have battery backup are not tested during power fail restart)
- o PEM (Power and Environmental Monitor) (NOTE: PEM is not tested during power fail restart)

- SPM (Service Processor Module)
- SCM (SCAN Control Module)
- MS820 (2Mb ECC Memory Module)
- AIO (KFBTA Disk Controller)
- AIE (DEBNT Tape/NI Controller)
- Clock system
- SCAN system
- CPUs and SCU via SCAN diagnostics
- Memory (NOTE: Not tested during power fail restart)
- XJA (JBox to XMI adapter) (NOTE: Not tested during power fail restart)
- XBI (XMI to BI adapter) (NOTE: Not tested during power fail restart)
- XCD (XMI to CI adapter)
- XXA (XMI to XI adapter)
- All controllers on the BI busses attached to the XBIs

10.2.2 Standalone Diagnosis

Standalone diagnostics are executed off line when VMS is not running. These diagnostics have complete control of devices in the system and provide the most complete testing of devices in a hardware system.

Standalone diagnostics verify device functionality, and isolate failing devices after a system has completed power on initialization testing. The diagnostics detect "stuck at" and dynamic fault conditions. They provide the capability to diagnose specific devices selected by the user or executed from scripts controlled by the SPU.

SPU ROM Based Diagnostics (RBD) tests cannot be controlled by scripts executed by the SPU. However, RBD tests for XMI devices and BI devices, connected through the XBI, can be controlled from scripts executed by the SPU.

The following diagnostics will execute in standalone mode:

- All RBD (ROM Based Diagnostics)
- Clock System Diagnostic
- SCAN System Diagnostic
- SCAN Pattern Based Diagnostics
- STRAM Data Cell Diagnostic
- Memory System Diagnostic
- VAX Instruction Exerciser Diagnostics
- EVKAA - VAX Macrohardcore (Level 4 - Halts on error)
- EVKAB - VAX Basic Instruction Exerciser (Level 2)
- EVKAC - VAX Floating Point Instruction Exerciser (Level 2)
- EVKAE - VAX Privileged Architecture Exerciser (Level 2)

- EVKA? - VAX Vector Instruction Exerciser (Level 2)

The CPU, SCU and adapter functional diagnostics include the following:

- E?KAX - AQUARIUS Kernel Specific Diagnostic
- E?KMP - AQUARIUS Multi Port SCU Diagnostic
- XJA Functional Diagnostic (Level 3)
- XBI Functional Diagnostic (Level 3)
- XCA Functional Diagnostic (Level 3)
- XXA Functional Diagnostic (Level 3)

10.2.3 User Mode Diagnosis

User mode diagnostics are executed under VMS. Normally, these types of diagnostics do not have complete control of devices unless VMS provides a mechanism for releasing the device for test without impacting the rest of the system.

Typical VAX user mode diagnostics test all device functions that can be executed in a VMS environment. These include processor instruction verification tests, I/O device function tests, and system exercisers.

User mode diagnostics support the standard diagnostics that execute on all VAX/VMS systems. These are the Level 2 and 2R diagnostics used for VAX instruction verification and I/O device tests. In addition the User Environment Test Package (UETP) is supported.

The requirement for high system availability requires that VMS be modified to allow additional diagnostic functionality in user mode. In particular, the system must have the capability of deselecting a faulty CPU or I/O channel without halting VMS. (That is, deselect the pair of CPUs in which one of the CPUs is faulty.) User mode diagnostic could then take advantage of this capability.

If a CPU is determined to be faulty and is deselected by VMS, it will then be possible to run the SCAN Pattern based diagnostics on that CPU from the SPU. If a fault is isolated, the faulty CPU - within a pair - can be powered off and the FRU replaced. After FRU replacement, the SCAN Pattern diagnostics can be executed again to verify the repair.

If an I/O channel is determined to be faulty and is deselected by VMS, then it will be possible to run RBD tests on devices connected to the deselected I/O channel. If a fault is isolated, the faulty I/O channel subsystem can be powered off and the faulty FRU replaced. After FRU replacement, the RBD tests can be executed to verify the repair.

Two devices in the I/O subsystem do not support RBD:

- XJA (SCU to XMI Adapter)
- XBI (XMI to BI Adapter).

To test the XJA or XBI in user mode, further modifications to VMS are required in addition to the ability to deselect an I/O channel. These modifications would allow a process running under VMS to access the XJA or XBI devices on a deselected channel for test purposes.

10.2.4 Symptom Directed Diagnosis

Symptom Directed Diagnosis (SDD) fault isolation is the ability to isolate intermittently occurring faults or hard faults by using machine state information captured at the time an error occurred. SDD is supported by the use of the error detection circuits in the hardware, and history buffers. It will be the primary strategy used by Field Service for fault detection and isolation in the CPUs and SCU.

The system will support two types of SDD fault isolation:

o a fault isolation matrix (also known as a fault dictionary) implemented by the SPU software

o a knowledge based error log analysis program (SPEAR) running under VMS.

10.3 Test Directed Diagnosis

10.3.1 Power Control Subsystem Diagnosis

The PCS (Power Control Subsystem) will use Built In Self Test (BIST) techniques for diagnosis. The PCS is comprised of a PEM (Power and Environmental Monitor) and several RICs (Regulator Intelligence Cards). The PEM and the RICs will have independent BISTs. Failures by the RIC BIST will be detected by the PEM module. The PEM will then display BIST error status on the Operator Control Panel, and report errors to the SPU.

The PEM supports RBDs (ROM Based Diagnostics) that can be invoked by a user from the SPU. The RBDs for the PCS cannot be executed in user mode since the CPUs will be powered on and off during execution of the PEM RBD tests. The CPUs must be powered off because the PEM and RICs serve a critical function in power and environment monitoring.

10.3.2 Service Processor Subsystem Diagnosis

The SPU subsystem is based on the BI bus. Modules interfacing to the BI contain BISTs. All BI modules in the SPU subsystem support RBDs. The TK50 tape drive and RD53 disk drive will also be tested with RBDs.

The SPU subsystem cannot be tested in user mode because of the critical role it plays in system functionality.

10.3.3 Clock System Diagnosis

The Clock system will be diagnosed by a diagnostic executed by the SPU. This diagnostic verifies the functionality of the Master Clock Module (MCM) and the distribution of clocks throughout the system. The clock diagnostic can only be executed in standalone mode.

10.3.4 SCAN Pattern Based Diagnostics

The CPUs (including the VBox), and SCU, contain SCAN latches. The latches allow use of program generated patterns for fault detection and isolation. The SCAN patterns will be generated by an automated test pattern generation process. A different set of SCAN patterns are required if a VBox is installed in the CPU.

SCAN pattern based diagnostic software will execute in the SPU. The SCAN pattern diagnostics can be executed in user mode if the target CPU is deselected from use by the operating system.

10.3.5 STRAM Data Cell Diagnostic

The purpose of the Self Timed RAM (STRAM) Data Cell diagnostic is to ensure that all STRAM data cells do not contain any "stuck at" faults. This test is required since the SCAN pattern based diagnostics do not test all STRAM data cells. The diagnostic will be executed in the SPU and consist of a series of DEPOSIT and EXAMINE functions to every STRAM location.

10.3.6 Memory Subsystem Diagnosis

The memory control logic resides on the SCU and implements SCAN latches. This logic will also be tested using SCAN pattern based diagnostics. The memory modules (Memory Array Cards - MAC, and Daughter Array Cards - DAC) will be tested by BISTs. The SPU will control initiation of the memory BISTs and provides the mechanism for reporting memory self test status. The self tests cannot be executed in user mode since it would destroy the memory contents.

In addition to the memory BISTs, a Level 3 SCU Multiport Functional diagnostic will verify operation of the various memory functions.

10.3.7 I/O Subsystem Diagnosis

The I/O subsystem consists of three levels of adapters. The first level is the XJA which provides an interface from the SCU to the XMI bus.The second level of adapters consists of several devices that interface the XMI bus to various interconnects including the BI. Most of the devices at this level support BISTs. The third level of adapters are devices that interface the BI bus to various interconnects. All devices at the third level support BISTs.

10.3.7.1 XJA Adapter Diagnosis

The XJA adapter has no BIST capability and will require macro level diagnostics. These diagnostics will verify the functionality of the XJA module, as well as the operation of the interface between the XJA and the IOC of the SCU. The XMI interface will also be validated by performing loopbacks of XMI transactions. The loopback features will allow isolation to a single module.

The XJA will be tested during system power on initialization by a Level 4 diagnostic loaded into main memory by the SPU. The diagnostic will set the Self Test Failed (STF) bit in the XJA when initiated. On successful completion the diagnostic will clear the STF bit.

The XJA will be tested by a Level 3 diagnostic executing under the VDS in standalone mode.

In user mode the XJA will be tested with a program executing as a process under VMS, which requires modifications to VMS. The implementation of this diagnostic is dependent on support from the VMS development group.

10.3.7.2 XBI Adapter Diagnosis

The XBI (XMI to BI adapter) hardware has no BIST capability and will require a macro coded diagnostic. This is an existing diagnostic and will be modified to execute on the system.

The XBI will be tested during system power on initialization by a Level 4 macro diagnostic loaded into main memory by the SPU. The diagnostic will set the STF bit in the XBI when started. On successful completion the diagnostic will clear the STF bit.

The XBI will be tested by a Level 3 diagnostic running under VDS in standalone mode.

In user mode the XBI will be tested with a program executing as a process under VMS, which requires modifications to VMS. The diagnostic implementation is dependent on support from the VMS development group.

10.3.7.3 XCA/XXA Adapter Diagnosis

The XCA (XMI to CI adapter) and XXA (XMI to XI) adapter are being designed with a high degree of hardware commonalty. The major differences between the devices is the CI/XI port logic and the module control microcode. The diagnostics for these modules will share many common test algorithms due to the hardware commonalty.

The XCA and XXA will be tested by BISTs during system power on initialization. The SPU will determine the result of the BIST.

In standalone mode the XCA and XXA will be tested by RBDs with the SPU providing the RBD interface software. A Level 3 functional diagnostic that executes under the VDS will also be provided.

In user mode the XCA and XXA can be tested by RBD (ROM Based Diagnostics) after the I/O channel is deselected by the operating system. Deselection of an I/O channel is not currently supported by VMS. The user mode diagnostic capability will only be available if VMS provides the deselection support.

10.3.8 Macro Diagnostics

The following diagnostics will be modified to execute on AQUARIUS and ARIDUS systems, and to maintain compatibility among the various VAX systems:

- Diagnostic Supervisor
- Diagnostic Autosizer
- CPU instruction verification
- Cross-product peripheral diagnostics

The system will implement a vector extension of the VAX instruction set. These vector instructions will be verified by a new VAX cluster series diagnostic. It will be a Level 2 diagnostic that executes under the VDS, and in standalone and user modes under VMS.

A functional diagnostic (E?KAX) will be provided to verify those functions in the system kernel that are not defined by the VAX architecture. The diagnostic will validate the unique CPU features. For example, it will test the error logic, internal processor registers, halt conditions, machine check logic, power fail recovery, interrupt and trap handling, and interfacing to the SPU. It will be a Level 3 diagnostic that executes the VDS in standalone mode.

A functional diagnostic will be developed to test the ability of the SCU to handle requests from multiple CPUs and multiple XJAs. The diagnostic will test the SCU based on the system configuration and will detect and isolate SCU port interaction problems. It will also verify all memory subsystem functions.

The SCU Multi Port Diagnostic will be a Level 3 diagnostic that executes under the VDS in standalone mode. The diagnostic will not execute in user mode.

10.4 Symptom Directed Diagnosis

Symptom Directed Diagnosis (SDD) will provide the ability to isolate intermittent and dynamic fault conditions without the need to execute traditional test directed diagnostics. The SDD technique is based on the high degree of error detection logic implemented in the kernel hardware.

A CAD tool will be used to produce a fault isolation matrix that will correlate each hardware error detector to the components and FRUs that could have caused the error to occur. The fault isolation matrix will be further refined to account for fault propagation and secondary error syndromes that will result in better fault isolation capability.

The fault isolation matrix will also contain information concerning:

o the relative failure rate for each component and FRU in a particular error domain

o the estimated probability that each component and FRU contributed to the error.

The fault isolation matrix will be used by error isolation software in the SPU, and by the SPEAR error analysis tool operating under VMS.

The fault isolation matrix implemented by the SPU will provide FRU and component level isolation information for each error latch in the CPUs and SCU. The SPU software implementing the fault matrix will store the isolation information in an error log file on the RD53 disk. The SPU will provide an error report generation utility that can be used to display the implicated FRU's and components for the errors contained in its error log.

The SPU will perform fault reporting, error log update, and fault recovery for all CPU and SCU hardware detected errors that occur while executing diagnostics out of main memory. The diagnostics will have the capability of disabling this feature by communicating with the SPU.

When the system is operating under the control of the operating system, the symptom directed fault isolation will be provided by the SPEAR (Software Program for Error Analysis and Repair) program. SPEAR will implement the fault isolation matrix but in addition it will use a rule based correlation analysis to provide better isolation than can be achieved by using the fault isolation matrix alone.

A set of system SNAPSHOT analysis rules will be developed for use by the SPEAR software. These rules will deal with the analysis of SNAPSHOT files that do not contain errors detected by hardware fault detection circuits (e.g., a Keep Alive Failure - KAF).

10.5 Remote Diagnosis

All standalone mode diagnostics previously described can be executed from the remote console port. However, protocol may not be supported when executing the SPU subsystem RBDs from the remote port.

Glossary

This glossary defines terms that describe the AQUARIUS system. It is a compilation of plan and specification glossaries, and terms defined in specifications.

In some cases, the glossary specifies a term's general functional location in the system. In other cases, a term is specific to a functional area (for example, the XMI bus).

System-Wide Terms

ACU — array control unit
The MCA III logic of the memory system. Consists of two memory data paths (MDPs), one memory control DRAM (MCD), and one main memory control (MMC).

ACU — array control unit
Part of the system control unit (SCU). Controls the main memory unit (MMU) and provides timing signals to the memory array cards (MACs) and daughter array cards (DACs). Consists of one main memory control MCA and two memory data path MCAs for a single-MMU system. A two-MMU system consists of two main memory control MCAs and four memory data path MCAs.

APG — address pattern generator
A linear shift feedback register in the DRAM control and address gate array that generates pseudo random address patterns for the built-in self-test (BIST).

ASD — automatic shutdown
A timed shutdown sequence that the power control subsystem (PCS) initiates.

ASMP — asymmetric multiprocessing
Occurs in a computer system with multiple CPUs where one CPU is the master and the other CPUs are the slaves.

availability
The fraction of time that the system is available for use. Calculated as MTBF/(MTBF + MDT) MDT = mean time down, where MTBF is the mean time between failure and MDT is the mean downtime.

BBU — battery backup unit
Provides power to memory for a minimum of 10 minutes during power failures.

BCAI — BCI adapter interface
A 133-pin ZMOS chip that functions as a buffer between user-designed processors, memories, and adapter modules and the VAXBI bus.

BCI3 — BCI to MicroVAX II bus interface
A 132-pin package that connects the integrated circuit interconnect bus of the MicroVAX processor to the VAXBI bus through the VAXBI BIIC.

BIIC — backplane interconnect interface chip
A 133-pin ZMOS chip that serves as the primary interface between the VAXBI bus and a master or slave port interface.

BIST — built-in self-test
A series of diagnostic functions that are designed into the hardware to provide go-no-go testing. Typically, the initial power-up of a device invokes this type of test.

bottom-up approach
A technique that divides the system into small functional units for individual testing and then builds them into higher level functional units for further testing.

building block
A technique that uses previously tested functional units (blocks) to test other functional units. Assumes that the lower level functional blocks do not contribute to failures in the higher level functional blocks.

BVP — BI VAX port
A standard software architecture that defines the data structures and protocols required to move information between a VAX host processor and an adapter via the VAXBI bus.

CAD — computer-aided design
An industry-wide term. Also refers to the High Performance Systems Computer Aided Design (CAD) group.

CCU — cache consistency unit
Part of the system control unit (SCU). Responsible for content consistency between multiple CPU caches. Maintains cache consistency with duplicate cache tag stores — one for each CPU.

CDB — configuration database
Scan access and configuration database.

CD chip — clock distribution chip
The clock distribution chip (CDC) is a custom VLSI chip in the center of a multiple chip unit (MCU). Distributes clock signals to all MCU devices, that is, macro cell arrays (MCAs) and self-timed RAMs (STRAMs).

CI bus
The computer interconnect bus.

CIO mode — console I/O mode
The state of operation in which the service processor unit (SPU) interprets characters entered at the console terminal as SPU commands. *See* PIO mode.

clock
A general term. May refer to a specific system clock, scan clock, or the entire clock system.

Corporate device
A cross-product-line device; a building block in a wide variety of systems.

CPU — central processing unit
May refer specifically to the EBox, IBox, or MBox.

crash
A customer-perceived failure, that is, the failure of a process to complete its intended function. May be due to hardware, software, administrative, or operator error.

CSIA — central system interrupt arbiter
Located in the system control unit (SCU). Arbitrates interrupt requests from I/O devices and other CPUs.

CSMA/CD — carrier sense multiple access with collision detection
Protocol used on the regulator intelligence card bus (RICBUS). Each node on the network uses a loopback scheme to receive its own transmissions simultaneously to ensure that the data was not corrupted by a collision. Also called *XXNET.*

DAC — daughter array card

Each DAC contains 160 dynamic RAMs, or 16 Mbytes of MOS memory.

DCA — DRAM control and address

An AMCC Q3500 gate array, located on the memory array card (MAC), that buffers the DRAM control signals from the main memory control (MMC) and latches the row and column address.

DDP — DRAM data path

An AMCC Q3500 gate array. Each memory array card (MAC) has four DDPs that buffer data going to or coming from the DRAMs.

DECSIM

A CAD software tool that allows simulation of hardware designs.

DPG — data pattern generator

A linear shift feedback register that produces pseudo random data patterns for the built-in self-test (BIST).

ECC — error correction code

Logic in the memory data path MCA that implements an ECC algorithm to detect double-bit errors and to correct single-bit errors in a 39-bit data field (32 bits data, 7 bits ECC).

EDFI — error detection and fault isolation

The ability to detect the occurrence of a fault in the hardware and to isolate that fault to a set of FRUs that are likely to have caused the fault. Error syndrome analysis provides fault isolation.

EEPROM

An electrically erasable programmable read-only memory

EPROM

An erasable programmable read-only memory

error

An invalid change in process state caused by a hardware or software failure.

ESD — Electronic Storage Development

Refers to the Electronic Storage Development group in Shrewsbury.

fault detection

The ability to detect that a fault has occurred in a hardware system.

fault isolation

The ability to determine the cause of a fault in a hardware system.

fault isolation matrix

A probability matrix that correlates each detected hardware error to the components and FRUs that could have caused the error. The HIDE CAD tool produces the matrix. Also called the *fault dictionary.*

The matrix is implemented in the service processor unit (SPU) and provides component and FRU isolation for each scan latch. The matrix also contains information about the relative failure rate for each component and FRU in the particular error domain. The matrix includes information about the estimated probability that each component and FRU contributed to the error.

FRU — field replaceable unit
The lowest level of system component that is economical to replace in the field.

functional diagnostics
Tests that are typically written in assembly language (or a higher level language) and are designed to verify that all device hardware functions perform correctly.

generic VAX diagnostics
Tests designed to verify that a system conforms to the VAX architecture or to verify the operation of Corporate devices connected to a VAX system.

HDSC — high density signal carrier
A multilayer substrate on which macro cell arrays (MCAs) and other integrated circuits are mounted. Contains the intercomponent connections. Also connects to the planar module.

hexword
A data type consisting of a contiguous 16-word string.

HIDE — hardware isolation domain emulator
A set of CAD tools that provide information about the hardware design's fault detection and isolation capabilities. Also produces the fault isolation matrix that the scan system uses. See fault isolation matrix.

HPS — high-performance system
Refers to the High Performance Systems group. The group may also be referred to as HPS/C, or High Performance Systems and Clusters.

ICU — I/O control unit
Part of the system control unit (SCU) that is the logical interface between the SCU and two XJAs via the JXDI. Also interfaces to the service processor unit (SPU) and implements the central system interrupt arbiter. Maximum of two ICUs per system.

intermittent failure
A momentary change in hardware state caused by a degrading component.

JXDI — ICU-to-XJA interface
A 12-foot cable that connects the I/O control unit (ICU), in the system control unit (SCU), to the individual XJA adapters.

kernel
A configuration that includes a service processor, a power system, a populated CPU planar module (which includes the EBox, IBox, MBox, and VBox), an SCU planar module, memory array modules, and a clock system.

lambda
Failure rate, usually expressed as failures per million hours or FITS — failures per 10^9 hours. Expresses the solid failure frequency of a component or component assembly.

loopback
The ability to route a signal so that one line transmits the signal and another receives it.

MAC — memory array card

Each MAC contains 32 Mbytes of on-board MOS memory (with 1 Mbit of DRAM) and two daughter array cards (DACs), which provide 16 Mbytes each, for a total of 64 Mbytes per MAC. Four MACs in a main memory unit (MMU) provide a minimum of 256 Mbytes of MOS memory.

macrodiagnostics

Diagnostics written in either an assembly or a high-level language. *See* functional diagnostics.

MBE — multiple-bit error

More than one error detected in a 39-bit data field by the ECC logic on the memory data path MCA.

MCA — macro cell array

The generic term for a VLSI device consisting of an array of cells that may be configured by a hardware designer to perform a specific function.

MCA III — macro cell array III

Third generation ECL gate arrays with an equivalent gate count of 10,000 gates. The arrays provide high speed logic gates in dense packaging to reduce interconnect delays.

MCD — memory control DRAM

A macro cell array (MCA), located on the system control unit (SCU), that contains the DRAM controller and self-test controller.

MCM — master clock module

The system clock module located in the system control unit (SCU) cabinet.

MCU — multiple chip unit

TBD Advanced multichip (MCA III and STRAM) packaging which incorporates a high density signal carrier (HDSC). The unit is an FRU for the SCU and CPU.

MDP — memory data path

A macro cell array (MCA), located in the array control unit (ACU) of the system control unit (SCU), that transfers one longword between the ACU and the MMU. Also contains the ECC logic for that longword.

MMC — main memory control

A macro cell array (MCA), located in the array control unit (ACU) of the system control unit (SCU), that provides control signals for the data path, address path, and DRAMs.

MMU — main memory unit

Aquarius MOS memory. Consists of four memory array cards (MACs) with each MAC carrying two daughter array cards (DACs). Each MMU (2 maximum) contains 256 Mbytes of MOS memory (1 Mbit of DRAM).

MTBF — mean time between failures

Predicts the frequency of solid part failures. A parts-count reliability figure computed as the reciprocal of the sum of the component failure rates.

nonrecoverable error

An error whose effects cannot be removed by restoring a previous state because either the failure recurs or the previous state is no longer available for use.

octaword
A data type consisting of a contiguous 16-byte string.

PCS — power control subsystem
An intelligent subsystem that monitors, measures, and controls the state of the power subsystem. Consists of regulator intelligence cards (*see* RIC), the power and environmental monitor (*see* PEM), and power converters.

PCU — power control unit
The module to which the ac line connects to distribute ac to the rest of the system. Includes the main system circuit breaker as well as various contactors that the the DEC power bus controls.

PEM — power and environmental monitor
The module that controls the regulator intelligence cards (RICs) in the power control subsystem (PCS) by communicating over the RICBUS.

PIO mode — program I/O mode
The state of operation in which the service processor unit (SPU) passes all user input to the VMS operating system.

planar module
A multilayer module on which multiple chip units (MCUs) are mounted and interconnected. The module mounts vertically in the system cabinet and can be air or water cooled.

QTA facility — quick turnaround facility
A laboratory-type facility that is equipped to produce multiple chip units (MCUs) in low volumes for prototype purposes.

quadword
A data type consisting of a contiguous 8-byte string.

RDS — read data substitute
An error code indicating that the accessed data contained an uncorrectable error.

recoverable error
An error caused by an intermittent error or transient condition. Its effects can be removed by restoring the system to a previous valid state.

RIC — regulator intelligence card
The module that provides the interface between the slave processor (on the RIC) and the power module it controls, which may be a power converter or utility port conditioner (UPC).

RICBUS — regulator intelligence card bus
The single-wire, multidrop, serial-communication bus that links the master processor (PEM) to the slave processors.

SBE — single-bit error
A 1-bit error detected by the ECC logic on the memory data path MCA.

SBus
The interface between the scan and clock distribution (SCD) logic and the CD chips on the CPU and SCU planar modules.

SCAN — Scan system
A design technique that partitions logic into small networks (rings) of combinational logic. A loading mechanism serially shifts patterns into the ring input. The technique serially shifts the logic results out of the ring(s) and compares them to a known good pattern related to each ring.

SCAT
A CAD tool that generates the optimal patterns used by the scan rings in testing the hardware.

SCC — scan control chip
A CMOS gate array, located on the scan control module (SCM), that controls all scan operations. Responsible for shifting the scan rings, DMA access to local memory, pattern comparison, and controlling the master clock module (MCM). Implemented in 1.5-micron channel technology from LSI Logic Corporation.

SCD — scan and clock distribution
TBD A part of the RLOG and TBD MCA logic that distributes scan signals throughout the planar module.

SCI — scan interconnect
The differential interface between the scan control module (SCM) and the scan and clock distribution (SCD) logic in a CPU or SCU MCA.

SCM — scan control module
A module in the VAXBI backplane of the service processor unit (SPU) that contains the scan logic.

SCU — system control unit
The planar module that contains the array control unit (ACU) and the I/O control unit (ICU).

SCI — scan control interconnect
Provides the scan interface to the CPU(s) and SCU.

SDC — scan distribution chip
A custom gate array on the scan control module (SCM) that distributes and receives scan control and data lines. The scan interconnect (SCI) starts at the SDC. Implemented as a Bipolar Q3500 gate array from AMCC.

SDD — symptom-directed diagnosis
The ability to isolate intermittent and hard failures to a failing device by analyzing machine state information retrieved at the time the error occurred. Supported with hardware error detection circuits, scan latches, and operation history buffers.

SJA — SPU-to-JBox adapter
Interfaces the service processor unit (SPU) to the JBox through RXCS, RXDB, TXCS, and TXDB VAX registers; and through DMA map registers.

SMP — symmetric multiprocessing
A computer system with multiple CPUs, where all CPUs are equal members and can perform all types of work.

SPEAR — Software Program for Error Analysis and Reporting
A software tool that runs as a process under the VMS operating system (or the ULTRIX operating system) to analyze error log entries and produce reports that are useful in isolating errors to an FRU.

SPM — service processor module
The host processor of the service processor unit (SPU). Contains the SJA, terminal ports, and processor.

SPU — service processor unit
A VAXBI-based, MicroVAX subsystem that provides the traditional console processor functions, including system initialization. Also acts as the diagnostic processor, controls the scan system rings, and retrieves symptom data used by the SDD analysis tools.

SST — startup self-tests
Tests that verify the integrity of all components in the power control subsystem (PCS).

standalone
A generic term that refers to a mode, condition, or device disconnected from the system or the VMS operating system. Also refers to diagnostic software that does not require VMS support for processing.

STRAM — self-timed RAM
Self-timed random access memory. Dynamic RAM array with internal latches that allow synchronous operation. Used in place of standard RAMs.

stuck at fault
The class of logic fault conditions that are manifested by a circuit input or output that is permanently held, or stuck, at a logic 1 or 0.

TDD — test-directed diagnosis
A diagnostic technique that attempts to detect fault conditions by applying a controlled stimulus to a circuit and then observing the output to verify that the circuit operates as expected.

transient error rate
The rate at which a component or component assembly produces transient or intermittent errors. Observed at 10 to 100 times lambda. Aquarius uses 15.

transient failure
Momentary change in hardware state caused by environmental or random events.

UETP — User Environment Test Package
A set of software routines that execute under the VMS operating system to simulate system loading conditions found in a customer operating environment.

UPC — utility port conditioner
A system that converts the ac line voltage to a regulated, high-voltage dc. Reduces harmonic line distortion and causes the power system to appear as a unity power-factor load. The high-voltage dc powers the DC/DC converters.

VAXBI bus

A low-cost, high-bandwidth, 32-bit synchronous bus that connects VAX processors to memories, I/O controllers, I/O adapters, and other VAX processors. Provides a large addressing range and high data integrity. Allows a single VAX processor to communicate with up to 16 nodes on the VAXBI bus.

VAXBI card cage

An enclosure containing a backplane that holds devices interconnected with a VAXBI bus.

VAXBI disk controller

A module that interfaces to the VAXBI bus and provides control functions to a disk subsystem.

VAXBI memory module

A module that interfaces to the VAXBI bus and provides storage for the MicroVAX processor.

VAXBI power controller

A module that interfaces to the VAXBI bus and provides control and monitoring functions for the power regulators and environmental monitors.

VAXBI scan controller

A two-module set that interfaces to the VAXBI bus and provides control of the scan paths in the CPU and SCUs.

VAXset software

A set of on-line processors including the SCU executing VMS or waiting to be started. A VAXset is loaded at initialization time and may be modified by command. By default the first processor is PRIMARY.

vector coprocessor

An optional processor that operates with a scalar processor to provide additional performance for vector operations.

XBI — XMI-to-BI adapter

Provides the control and data interface between the XMI bus and the VAXBI bus.

XCA — XMI-to-CI adapter

Provides the control and data interface between the XMI bus and the CI bus.

XCI — XMI corner interface

The bus that interfaces node-specific logic to the XMI corner that, in turn, interfaces to the XMI bus.

XCLOCK

A component in the XMI corner that receives the three sets of radially distributed XMI clocks and provides them to the node-specific logic. Also provides XL lines (control to the seven XLATCH components).

XI

The Corporate XI-2 communication interface that is currently under development as a replacement for the CI, NI, and SI buses.

XJA — XMI-to-SCU adapter
A module that resides in the XMI backplane and interfaces to the system control unit (SCU).

XL lines
Control lines from XCLOCK to the XLATCH(s) received from the XMI bus.

XLATCH
Seven latches that interface to the majority of the XMI signal lines.

XMI bus
Calypso memory interconnect used as the Aquarius I/O bus.

XMI adapter cage
An enclosure containing a backplane that holds devices interconnected with an XMI bus.

XXNET
The CSMA/CD (carrier sense multiple access with collision detection) protocol that is used by the power and environmental monitor (PEM) and the power control subsystem (PCS). *See* CSMA/CD.

XMI-Specific Terms

NOTE
The following XMI-specific terms will eventually be worked into the main glossary.

node
A hardware device that connects to the XMI backplane. The largest XMI subsystem supports 14 nodes.

transfer
The smallest quantum of work that occurs on the XMI bus. Typical examples are the command cycle of a read operation and the command and following data cycles of a write operation.

transaction
Composed of one or more transfer. *Transaction* is the name given to the logical task being performed (for example, read, write). In the case of a read operation, the transaction consists of a command transfer followed some time later by a return data transfer.

commander
The node that initiated the transaction in progress. In any write transaction, the commander is the node that requested the write. For read transactions, the commander is the node that requested the data.

The distinction of being the commander in a transaction holds for the duration of the transaction, although it might appear that the commander changes in some cases. For example, a commander initiates a read transaction. The responder (data source) initiates the return data transfer, but the node that requested the data is still the commander.

responder
The complement to the commander in a transaction.

transmitter
The node that is sourcing the information on the bus. For example, during a read transaction, the commander is the transmitter during the command transfer, and the receiver during the return data transfer.

receiver
The complement of the transmitter in a transfer. The receiver is the sink of data being moved during a transfer.

naturally aligned
TBD A data quantity whose address could be specified as an offset, from the beginning of memory, of an integral number of data elements of the same size. The lower order address bits of naturally aligned data items are characteristically zero. All XMI reads and writes transfer a naturally aligned block of data.

wraparound read
TBD An octaword or hexword read operation where read data is returned in a specific pattern in which the specifically addressed quadword is returned first independent of alignment. The remaining data in the naturally aligned data block containing the addressed quadword is returned in subsequent transfers in descending quadword address order.