VAX 9000 Family
SCU Technical Description

Order Number EK-KA90J-TD-001

digital equipment corporation
maynard, massachusetts

First Edition, May 1990

The postpaid Reader's Comment Card included in this document requests the user's critical evaluation to assist in preparing future documentation.

FCC NOTICE: The equipment described in this manual generates, uses, and may emit radio frequency energy. The equipment has been type tested and found to comply with the limits for a Class A computing device pursuant to Subpart J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such radio frequency interference when operated in a commercial environment. Operation of this equipment in a residential area may cause interference, in which case the user at his own expense may be required to take measures to correct the interference.

The following are trademarks of Digital Equipment Corporation:

BI	KDM	RSTS	VAX FORTRAN
CI	KLESI	RSX	VAX MACRO
DEC	MASSBUS	RT	VAXBI
DECmate	MicroVAX	RV20	VAXcluster
DECUS	NI	RV64	VAXELN
DECwriter	PDP	TA	VMS
DHB32	P/OS	TK	VT
DIBOL	Professional	ULTRIX	Work Processor
DRB32	RA	UNIBUS	XMI
EDT	Rainbow	VAX	
KDB50	RD	VAX C	digital™

® IBM is a registered trademark of International Business Machines Corporation.

® Intel is a registered trademark of Intel Corporation.

™ Hubbell is a trademark of Harvey Hubbel, Inc.

® Motorola is a registered trademark of Motorola, Inc.

This document was prepared and published by Educational Services Development and Publishing, Digital Equipment Corporation.

DIGITAL INTERNAL USE ONLY

Contents

3 JBox Cache Consistency

4 Micromachine Control

5 Array Control Unit and Main Memory Unit

6 I/O Control Unit

7 Error Detection and Reporting

Index

Examples

Figures

Tables

About This Manual

This manual describes the operation of the system control unit (SCU). It is a reference document for Customer Services personnel as well as a training resource for Educational Services.

Intended Audience

The content, scope, and level of detail in this manual assumes that the reader:

- Is familiar with the VAX architecture and VMS operating system at the user level
- Has experience maintaining midrange and large VAX systems

Manual Structure

The manual has seven chapters.

- Chapter 1, General Description, provides an introduction to and general description of the system control unit (SCU). This chapter includes an overview of the physical organization and briefly describes the logic implementation.
- Chapter 2, JBox Port Arbitration, provides a physical and functional description of the JBox. This chapter describes the data, address, and control paths for the access control unit (ACU), I/O control unit (ICU), and MBox interfaces.
- Chapter 3, JBox Cache Consistency, defines cache consistency concepts and describes how the SCU resolves cache conflicts. This chapter describes the contents of the global tag STRAMs and how and when the STRAMs are accessed.
- Chapter 4, Micromachine Control, describes the SCU control store. This chapter describes how the microcode is loaded into the control store and how the control store is addressed. Chapter 4 also describes the microword fields.
- Chapter 5, Array Control Unit and Main Memory Unit, describes the ACU interfaces to main memory. This chapter describes how the ACU controls the main memory unit's data, address, and DRAMs during normal and test operations.

  Chapter 5 includes the MMU logic, memory modes of operation, dynamic RAM organization, and memory interleaving. This chapter also describes the refresh operation, memory system initialization, and test descriptions.
- Chapter 6, I/O Control Unit, describes how the ICU interfaces to the XJA. This chapter describes how XJA and SPU requests are handled and how the ICU controls data and address paths to and from the receive and transmit buffers.
- Chapter 7, Error Detection and Reporting, describes the error detection and reporting. This chapter identifies fatal errors and how they are handled.

- The index contains entries, subentries, and cross-references that support the reader of the technical description. Figures, tables, and text are indexed. Figure references are listed with the letter *f* appended to the page number. Table references are listed with the letter *t* appended to the page number.

1
General Description

This chapter provides a general physical and functional description of the system control unit (SCU). It identifies the MCUs, MCAs, and STRAMs in the three major units of the SCU: the JBox, the array control unit (ACU), and the I/O control unit (ICU). The chapter also describes how the units support CPU, main memory, and I/O requests.

1.1 Overview

In the VAX 9000 family computer system (Figure 1–1), the system control unit (SCU) connects several major subsystems (Figure 1–2). These subsystems are the CPU, service processor unit (SPU), I/O, and memory. If the CPU, the SPU, or an I/O device needs data from main memory or needs to write to main memory, it sends a request to the SCU. The SCU contains the ACU that interfaces to memory.

If the CPU or the SPU needs to read I/O data or needs to write to an I/O register, it sends a request to the SCU. The SCU contains the ICU that interfaces to I/O. If the CPU needs to read data in an SPU register or needs to write data into an SPU register, it sends a request to the SCU. The SCU uses the ICU to handle these requests.

The SCU has a cache consistency unit that contains the global tag STRAMs. Each CPU cache set, 0 and 1, has a corresponding section in the global tag STRAMs. Each cache block within the cache set can have read, written full, written partial, or invalid status. Each cache block can have lock status.

If a CPU sends a request to the SCU, it is referred to as a requester. If a requester needs refill data that is written full or written partial in another CPU cache, it doesn't have to know that another CPU has the data. The requester sends a read refill request to the SCU, and the cache consistency unit updates the global tag STRAMs in the JBox. The cache consistency unit then gets the data from the data cache STRAMs of the other CPU, returns the data to the requester, and writes the cache block into main memory.

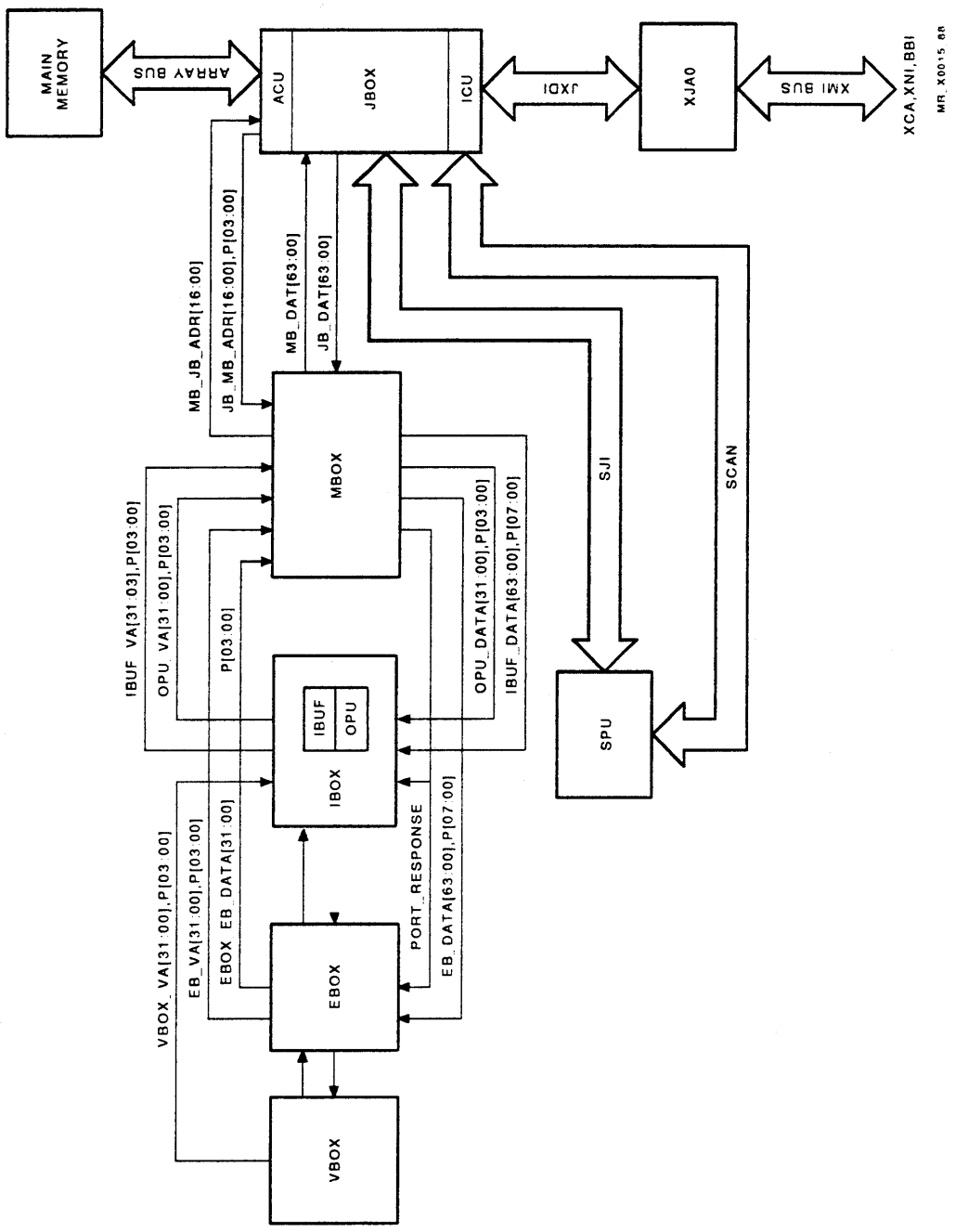


Figure 1–1 System Block Diagram

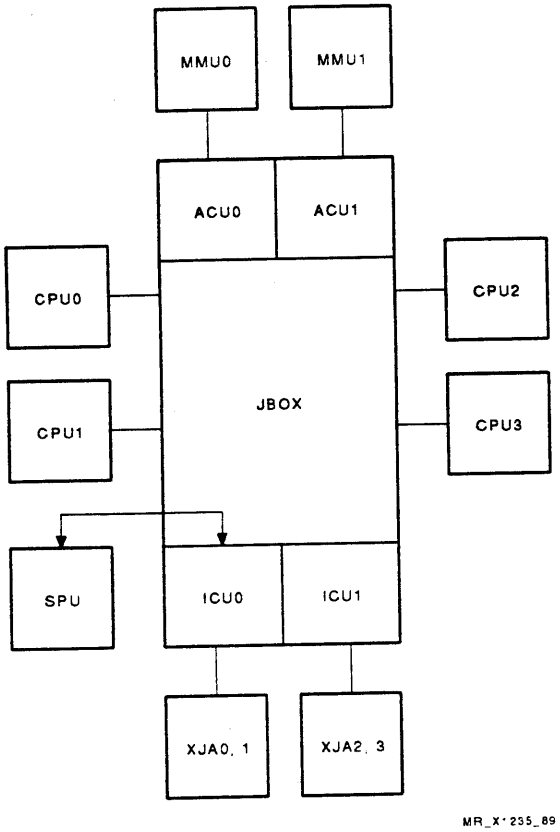


Figure 1–2 SCU Ports

1.1.1 SCU Logical Units

The SCU contains three logical units: the JBox, ACU, and ICU. Figure 1–3 is the SCU block diagram.

1.1.1.1 JBox

The JBox logic is in the CCU, DAX, DBX, and tag MCUs. It contains the SCU control in the CCU MCU, the data switch in the DAX and DBX MCUs, and the address receive latches in the tag MCUs. JBox logic interfaces to the MBox, memory, ICU, and SPU. The JBox arbitrates requests from the memory subsystem, I/O subsystem, and up to four CPUs. It contains the cache consistency unit that ensures cache consistency for the CPU write back cache, and checks for cache blocks that require invalidating as a result of I/O writes to memory. The cache consistency unit contains the global tag STRAMs and the MTCH and ADRX MCAs.

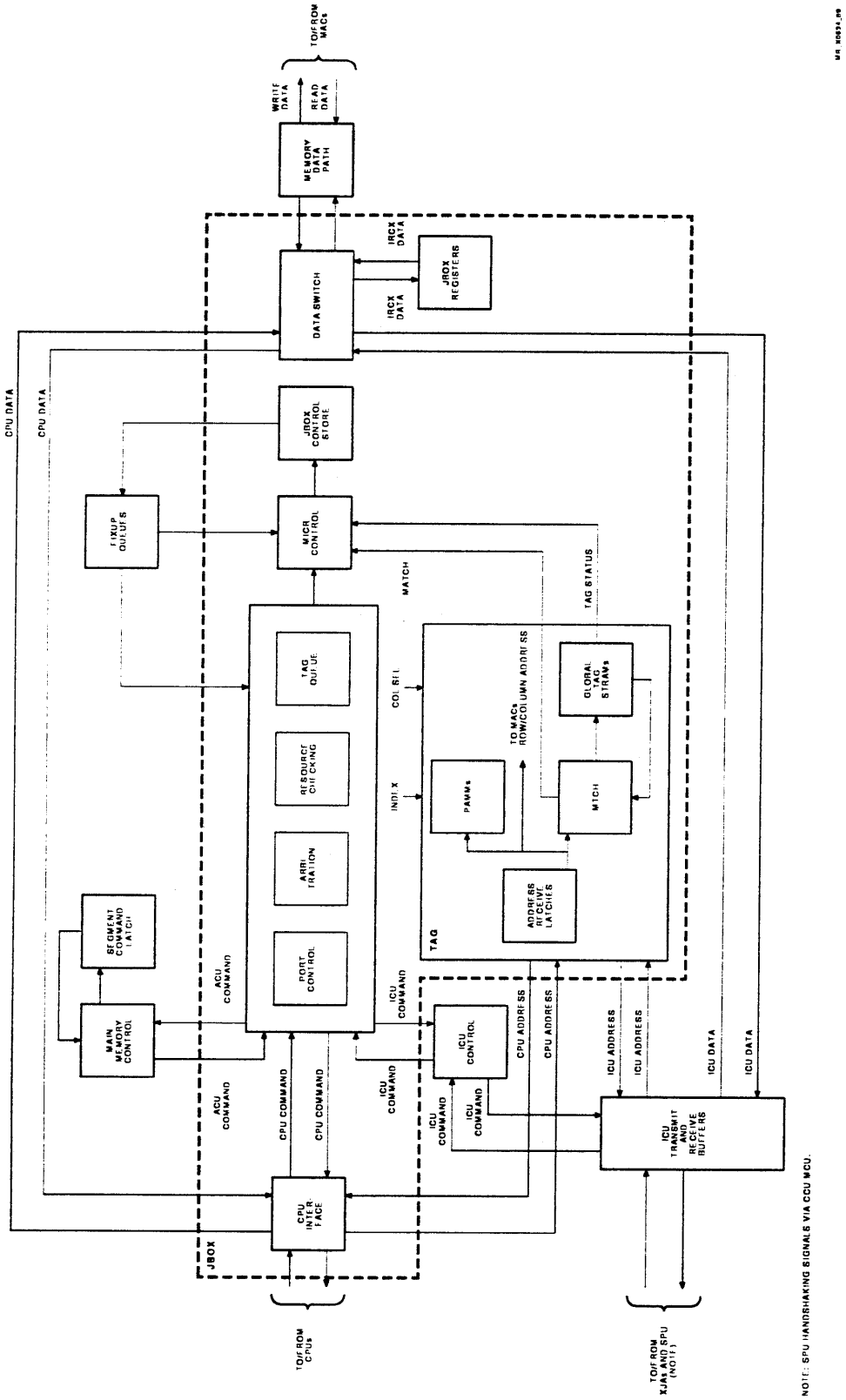


Figure 1–3 SCU Block Diagram

The JBox contains the micromachine that consists of the control store STRAMs and the MICR MCA. The control store STRAMs hold the JBox microcode, and the MICR MCA contains the microcontrol logic. Most of the logic within SCU is controlled either directly or indirectly by bits within the microword. The MICR MCA contains three fixup queues that handle cache inconsistencies, nonexistent memory errors, and lock busy and lock deny requests.

The JBox connects nine ports, using address receive latches and a data switch, and allows simultaneous transactions. The JBox latches a request, sends it to arbitration, and determines which SCU resources are needed to complete the request. The JBox compares available resources with required resources, placing the request either in a tag queue for execution, or on a reserve list until resources become available. The SCU resources include source and destination data paths, command buffers, and address receive latches.

When the JBox receives a request, it transfers control to a port controller that monitors the status of the request from load, arbitration, and retire. The JBox also latches the physical address in address latches and holds onto this address until the request is retired. The address latches are available to the microcode, memory, and the JBox itself.

1.1.1.2 Array Control Unit

The ACU logic is located in the DAX and DBX MCUs. It provides command, data, and address control, and it also maintains the status interface to the JBox. ACU sends DRAM control signals (RAS, CAS, and WE) to the memory modules, and supports built-in self-test (BIST) operations. ACU uses the main memory control and DRAM control located in the DBX MCA to control the write buffers, the read buffers, and the read bus on the memory modules. It also provides the SCU memory data path in the DAX and DBX MCUs that is used to receive and send DRAM data to the JBox data switch.

1.1.1.3 I/O Control Unit

The ICU logic is located in the DAX and DBX MCUs. It provides command, data, and address control, and it also maintains the status interface to the JBox. ICU supports the SPU and the XJAs. It contains the receive buffers and transmit buffers in the DAX and DBX MCUs that receive and send SPU and XJA commands to and from the JBox. ICU uses I/O control in the DAX MCU to control loading and unloading of the receive and transmit buffers in the DAX and DBX MCUs. (CCU interfaces to SPU and sends the command, address, and data to ICU.)

ICU provides the interface between the JBox (CCU MCU) and the interrupt arbiter and JBox registers on the DAX MCU. The JBox can read the contents of three registers and send write data to the registers by interfacing with ICU.

When the ICU receives a request, it loads a receive buffer with the command, address, and data. When the JBox signals the ICU to unload the buffer, ICU sends the command to the JBox, the address to the address receive latches, and the data to the JBox data switch. When the ICU sends a request, it loads a transmit buffer with the command, address, and data from the JBox.

1.2 Physical Organization

SCU logic resides on a module that is slightly smaller than the CPU module. Figure 1–4 shows the SCU planar module.

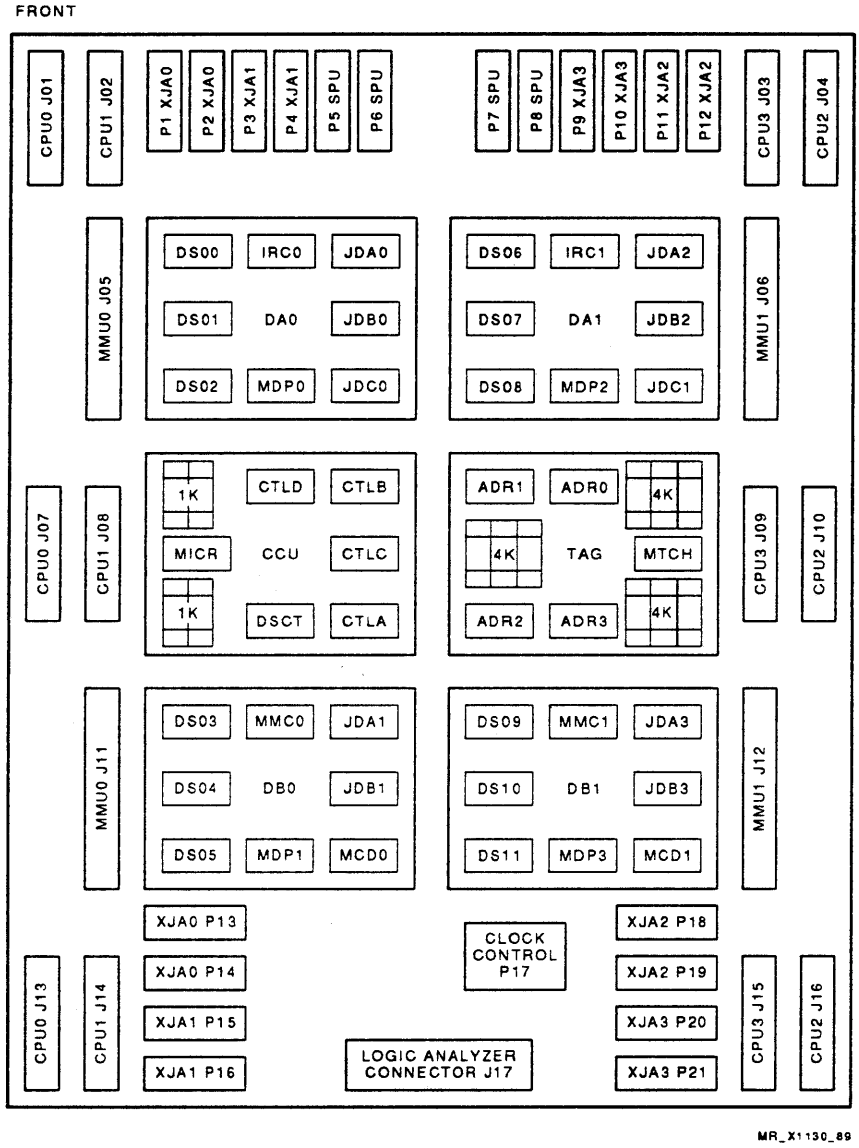


Figure 1–4 SCU Planar Module

SCU can have up to six MCUs for logic. The remaining space on the module accommodates the connectors that carry signals to and from the SCU, CPUs, SPU, I/O, and main memory unit (MMU).

SCU requires four MCUs (DA0, CCU, tag, and DB0) for the basic configuration (Figure 1–5). This configuration supports two CPUs (CPU0 and CPU1), one ACU (ACU0), and one ICU (ICU0). ACU0 supports one main memory unit (MMU0) that consists of four memory array cards. ICU0 supports two XMI-to-JBox adapters (XJA0 and XJA1).

With two additional MCUs, DA1 and DB1, the SCU can support two additional CPUs (CPU2 and CPU3), an additional ACU (ACU1), and an additional ICU (ICU1). ACU1 supports MMU1. ICU1 supports XJA2 and XJA3.

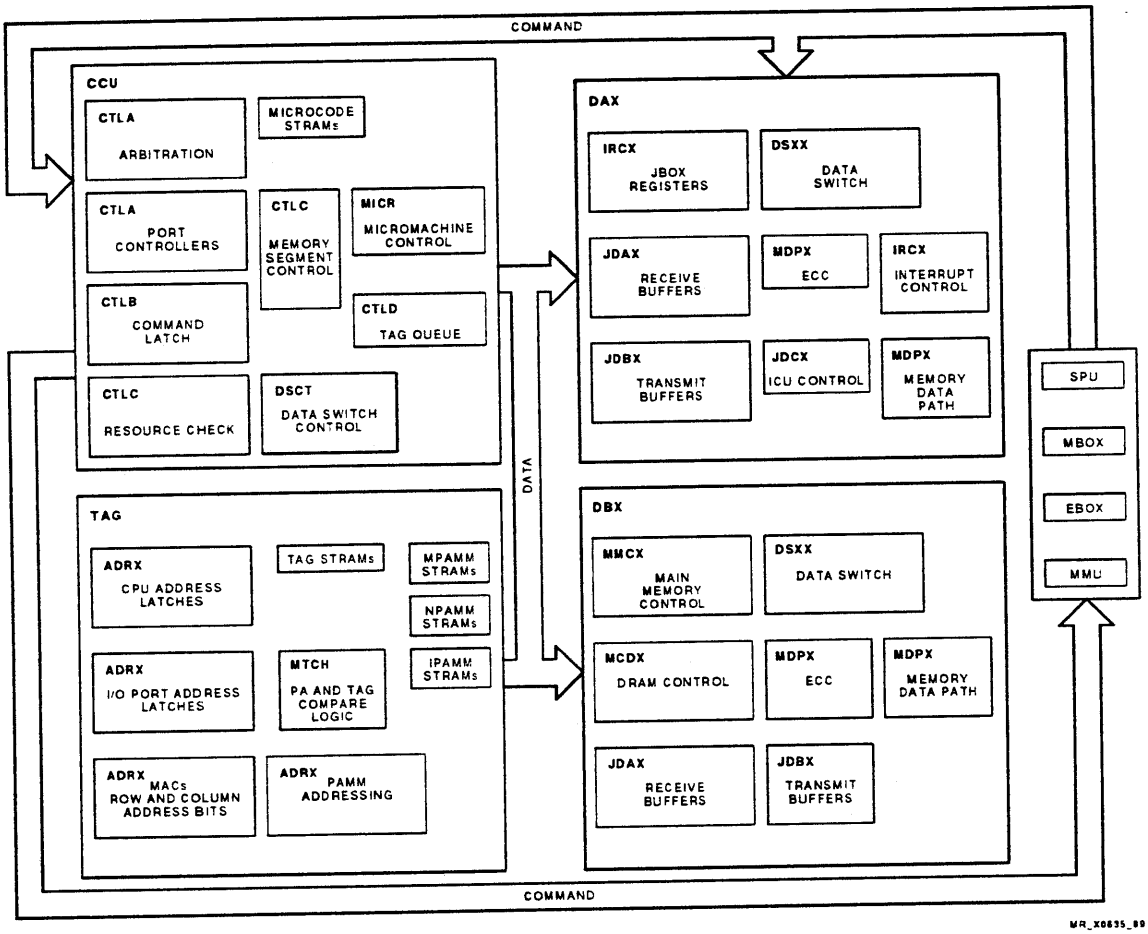


Figure 1–5 SCU MCUs

The SCU MCUs (Table 1–1) are as follows:

- **DA0 and DA1** — These MCUs control and monitor the I/O control units and contain byte-slices of the data switch, the memory data path, and the I/O transmit and receive buffers. DA0 contains the MDP0 (memory data path), JDC0 (I/O control), IRC0 (JBox interrupt arbiter and registers), JDA0 (I/O receive buffer), JDB0 (I/O transmit buffer), and DS00, DS01, and DS02 (data switch) MCAs. DA1 contains the MDP2 (memory data path), JDC1 (I/O control), IRC1 (JBox interrupt arbiter and registers), JDA2 (I/O receive buffer), JDB2 (I/O transmit buffer), and DS06, DS07, and DS08 (data switch) MCAs.

  These MCUs control the following data and address slices:

  Bytes 0 through 3 of the data to the DSXX MCAs
  Two of the four address bytes to the tag MCU
  One of the two nibbles of the SPU interface
  One of the two bytes of the JXDI interface

- **DB0 and DB1** — These MCUs control and monitor the memory array cards and contain byte-slices of the data switch, the memory data path, and the I/O transmit and receive buffers. DB0 contains the MDP1 (memory data path), MCD0 and MMC0 (memory control), JDA1 (I/O receive buffer), JDB1 (I/O transmit buffer), and DS03, DS04, and DS05 (data switch) MCAs. DB1 contains the MDP3 (memory data path), MCD1 and MMC1 (memory control), JDA3 (I/O receive buffer), JDB3 (I/O transmit buffer), and DS09, DS10, and DS11 (data switch) MCAs.

  Each MCU controls the following slices of address, data, and control lines:

  Bytes 4 through 7 of the data to the DSXX MCAs
  Two of the four address bytes to the tag MCU
  One of the two nibbles of the SPU interface
  One of the two bytes of the JXDI interface

- **CCU** — This MCU contains the JBox control logic in the CTLA, CTLB, CTLC, and CTLD MCAs. These MCAs control the command, data, and address paths. CCU contains the micromachine control (MICR MCA), which sends microaddresses to the JBox control store STRAMs.

- **Tag** — This MCU contains the ADR0, ADR1, ADR2, ADR3, and MTCH MCAs, and the physical address memory mapping (PAMM) STRAMs and global tag STRAMs. This MCU receives and transmits addresses to and from the ports and controls the global tag STRAMs. It compares the port addresses to the addresses in the global tag STRAMs, determining if an address match has occurred. The ADR0, ADR1, ADR2, and ADR3 MCAs contain the I/O and CPU address receive latches, and send physical address bits to memory, MTCH MCA, CPU, ICU, and the PAMMs.

Table 1–1 SCU MCUs

MCU	MCA
CCU	CTLA, CTLB, CTLC, CTLD, MICR, DSCT
DA0	JDA0, JDB0, JDC0, IRC0, MDP0, DS00, DS01,DS02
DA1	JDA2, JDB2, JDC1, IRC1, MDP2, DS06, DS07, DS08
DB0	JDA1, JDB1, MMC0, MCD0, MDP1, DS03, DS04, DS05
DB1	JDA3, JDB3, MMC1, MCD1, MDP3, DS09, DS10, DS11
Tag	ADR0, ADR1, ADR2, ADR3, MTCH

1.2.1 CCU MCU

The CCU MCU contains six MCAs, fifteen 1K × 4-bit control store STRAMs, and three 1K × 4-bit history buffer STRAMs. The MCAs are as follows:

- **CTLA** — Receives requests (load command) for data movement, contains the port arbitration logic, and generates an index that points to a command (in CTLB) and an address (in tag).
- **CTLB** — Receives and stores 20 port commands and then sends the commands to other ports.
- **CTLC** — Sends commands to DSCT (data switch controller), cache consistency information to a queue (in CTLD), and commands to ports.
- **CTLD** — Contains the tag queue. The microcode accesses the queue and uses the entries to form microaddresses.
- **DSCT** — Controls the DSXX that is a block multiplexer (64 bytes).
- **MICR** — Controls the microcode.
- **Microcode STRAMs**

1.2.2 DAX MCUs

There are two DAX MCUs: DA0 and DA1. Each MCU contains eight MCAs. The MCAs on DA0 are as follows:

- **JDA0** — Receives one byte of the interface from each of the two XJAs and one nibble of the SPU interface. JDA0 sends half of the address bits to the tag MCU, buffers and sends the data from the XJAs (using a 4-byte wide path) to the group of DSXX MCAs, and sends commands from the I/O to CCU.
- **JDB0** — Similar to the JDAX; deals with signals in the reverse direction.
- **JDC0** — Controls the operation of the JDAX and JDBX and monitors their errors. JDC0 coordinates the handshaking signals to and from the JXDI and to and from the CCU. It also sources the clocks that are transmitted to the XJAs.
- **DS00, DS01, and DS02** — Provide crossbar capability for four bytes of data. They support CPU0, CPU1, ACU0, and ICU0 (XJA0 and XJA1). For register reads and writes, they have a 4-byte wide path to and from the IRC.
- **IRC0** — Contains the SCU registers and interfaces to the crossbar. IRC0 contains the interrupt logic for I/O to CPU interrupts and inter-CPU interrupts. IRC0 also handles the handshaking signals for the SPU interface.
- **MDP0** — Provides a 4-byte wide path between the data crossbar and the memory array cards and handles ECC and read-modify-write operations. MDP0 provides check bit generation for write data, detects and corrects single-bit errors, detects double-bit errors, and generates data patterns during BISTs. It also contains the byte merge logic.

The DA1 MCU contains MCAs that are similar to the MCAs on DA0.

1.2.3 DBX MCUs

The SCU can have two DBX MCUs: DB0 and DB1. Each contains eight MCAs. The MCAs on DB0 are as follows:

- **JDA1** — Receives one byte of the interface from each of the two XJAs and one nibble of the SPU interface. JDA1 sends half of the address bits to the tag MCU, buffers and sends the data from the XJAs (using a 4-byte wide path) to the group of DSXX MCAs, and sends commands from the I/O to CCU.
- **JDB1** — Similar to the JDA1; deals with signals in the reverse direction.
- **MCD0** — Controls the dynamic RAMs (DRAMs) on the memory modules. This MCA receives and decodes the segment, and the command for the segment, from the MMCX MCA. To read the data from memory and write the data into memory, the MCD0 MCA sends RAS, CAS, and write enable control signals. MCD0 sends read bus control signals, such as bypass, to control the flow of data on the read bus in the memory module.
- **DS03, DS04, and DS05** — Provide crossbar capability for four bytes of data. They support CPU0, CPU1, ACU0, and ICU0 (XJA0 and XJA1). For register reads and writes, they have a 4-byte wide path to and from the IRCX MCA, which contains the JBox registers.
- **MMC0** — Provides the control signals to the memory array cards and receives status from them. MMC0 provides the command, control, and status interface to the JBox, the data path control and DRAM control commands to the MCD, and the error detection on all MMC control lines. MMC0 also supports BIST operations.
- **MDP1** — Provides a 4-byte wide path between the data crossbar and the memory array cards and handles ECC and read-modify-write operations. MDP1 provides check bit generation for write data, detects and corrects single-bit errors, detects double-bit errors, and generates data patterns during BIST. It also contains the byte merge logic.

The DB1 MCU contains MCAs that are similar to the MCAs on DB0.

1.2.4 Tag MCU

The tag MCU contains five MCAs and twenty-four 4K × 4-bit global tag STRAMs and 1K × 4-bit PAMM STRAMs. They are as follows:

- **MTCH** — Drives the addresses and data to the tag STRAMs. MTCH receives addresses from the global tag STRAMs and matches these addresses with the addresses received from the ports. This MCA sends address match signals to the MICR MCA.
- **ADR0, ADR1, ADR2, and ADR3** — Receives one-quarter of the address field from the four CPU ports and two I/O ports. These MCAs also transmit one-quarter of the address field to the same ports, and 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.
- **Tag STRAMs** — Each CPU cache set, 0 and 1, has a corresponding 1K section in the SCU global tag STRAMs. The global tag STRAMs consist of twenty-four 4K × 4-bit STRAMs and contain 16K locations. Two 1K sections of the 4K STRAMs are used for cache set 0 and 1. The remaining 2K sections are used for interlock status reservations for CPUs and I/O devices.
- **PAMM STRAMs** — The JBox control logic receives memory mapping information from three types of physical address memory mapping STRAMs:
  - **MPAMM**. Memory physical address memory mapping; points to a unit, segment, and bank of memory.
  - **IPAMM**. I/O physical address memory mapping; points to an adapter, ICU0 or ICU1, or an I/O register.
  - **NPAMM**. Nonexistent physical address memory mapping; generates the nonexistent memory (NXM) bit.

2
JBox Port Arbitration

The system control unit (SCU) contains three logical units: the JBox, the array control unit (ACU), and the I/O control unit (ICU). This chapter discusses the JBox in four major sections, as follows:

- JBox control
- JBox data paths
- JBox address paths
- Port interfaces for the CPU (MBox), memory, I/O, and service processor unit (SPU)

2.1 Overview

The JBox (Figure 2–1) arbitrates requests from the memory subsystem port, I/O subsystem port, and up to four CPU ports. The JBox controls the data and address paths, contains the cache consistency unit that ensures cache consistency for the CPU write back cache, and checks for cache blocks that require invalidation as a result of I/O writes to memory.

The JBox connects the following nine ports using address receive latches and a data switch, allowing simultaneous transactions:

- **One to four CPUs** — Each CPU is partitioned into four units: MBox, IBox, EBox, and VBox. The JBox interfaces with the CPU through the MBox and connects the CPU with main memory, I/O, console, and other CPUs. The JBox provides a high transfer rate for blocks of data between main memory and the CPU cache data STRAMs.
- **One to two ACUs** — Each ACU controls the main memory units (MMUs), and provides dynamic timing signals to the memory array cards (MACs) and daughter array cards (DACs). Each ACU controls one MMU. The JBox interfaces with ACU to send and receive memory commands and to address the DRAMs. (The JBox addresses the DRAMs under the control of the main memory control logic in ACU.)
- **One to two ICUs** — Each ICU handles one or two XMI buses. The JBox interfaces with ICU to send and receive I/O commands from the XMI bus to JBox adapter (XJA) and SPU.
- **One SPU** — SPU performs console functions and handles errors. The JBox interfaces with SPU to send and receive console commands. The JBox sends the SPU commands to ICU and receives SPU commands from ICU.

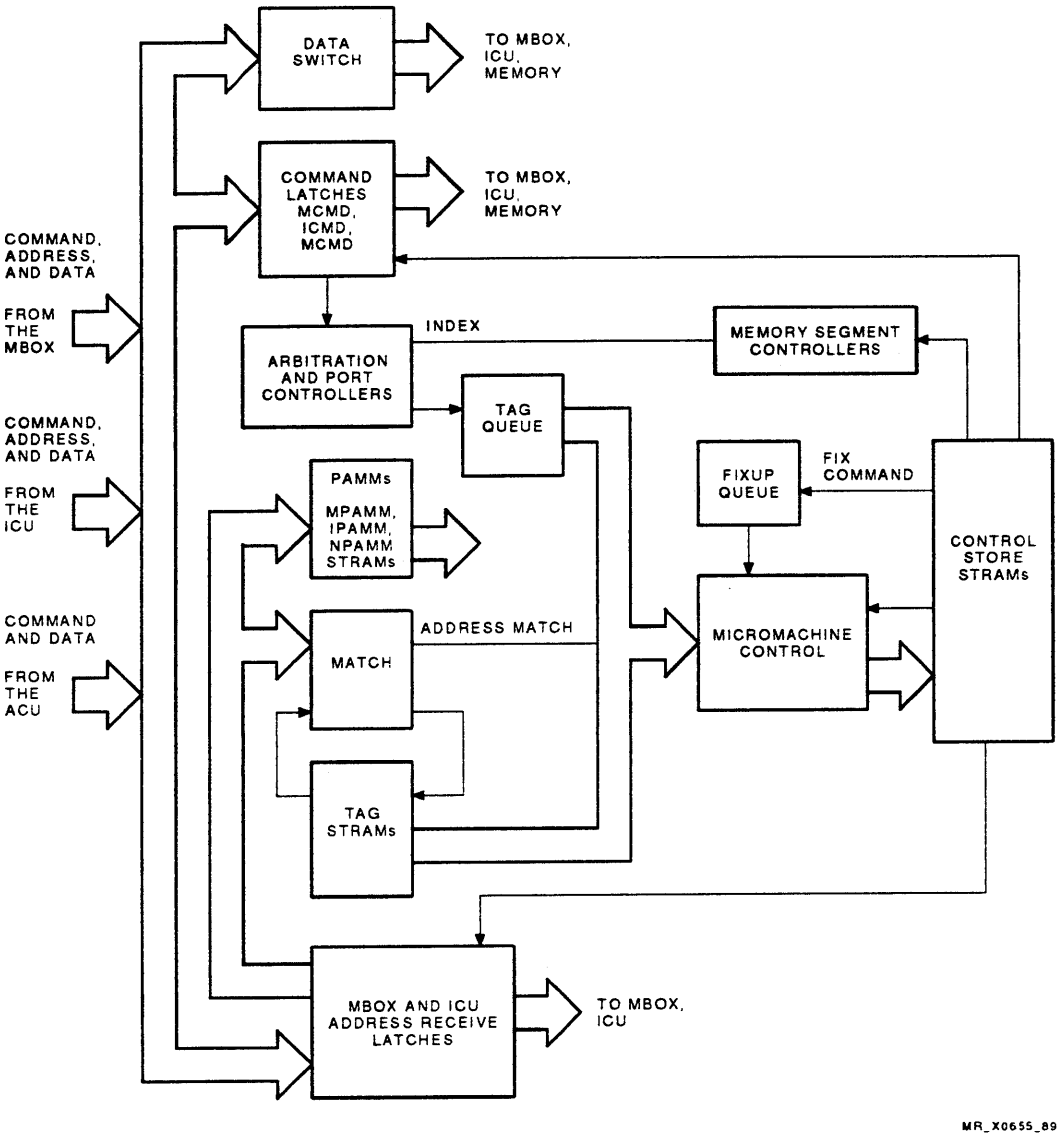


Figure 2–1 JBox Block Diagram

Table 2–1 lists the MCAs that comprise the JBox MCUs.

Table 2–1 JBox MCUs

MCU	MCA
CCU	CTLA, CTLB, CTLC, CTLD, DSCT, MICR
DAX	IRC0, IRC1, DS00, DS01, DS02, DS06, DS07, DS08
DBX	DS03, DS04, DS05, DS09, DS10, DS11
Tag	ADR0, ADR1, ADR2, ADR3, MTCH

2.2 JBox Control

The JBox control logic is located in the CCU MCU, which contains the CTLA, CTLB, CTLC, and CTLD MCAs. These MCAs control the command, data, and address paths. The CCU MCU contains the cache consistency unit and manages data to and from the ports.

2.2.1 CTLA MCA

The CTLA MCA receives requests for data movement, contains the port arbitration logic, and generates an index that points to a command in CTLB and an address in ADRX. Figure 2–2 shows the CTLA MCA. This MCA contains the following:

❶ Port input command latches for CPU, ACU, and ICU command fields. CPU sends load command and buffer available fields. ACU sends DBX fatal, load command, and buffer available fields. ICU sends DAX fatal error, load command, and buffer available fields.

❷ Port state controllers for four CPUs, two ACUs, and two ICUs.

❸ Arbitration vector decoder for 8 ports (20 requests).

❹ Arbitration index encoder for 8 ports (20 command buffers).

❺ CTLB and ADRX control encoder, which generates control lines that are sent to the CTLB and ADRX MCAs.

❻ Port output command latches for CPU, ACU, and ICU command fields. CTLA sends the CPU buffer available and fatal error fields, the ACU buffer available field, and the ICU the buffer available field.

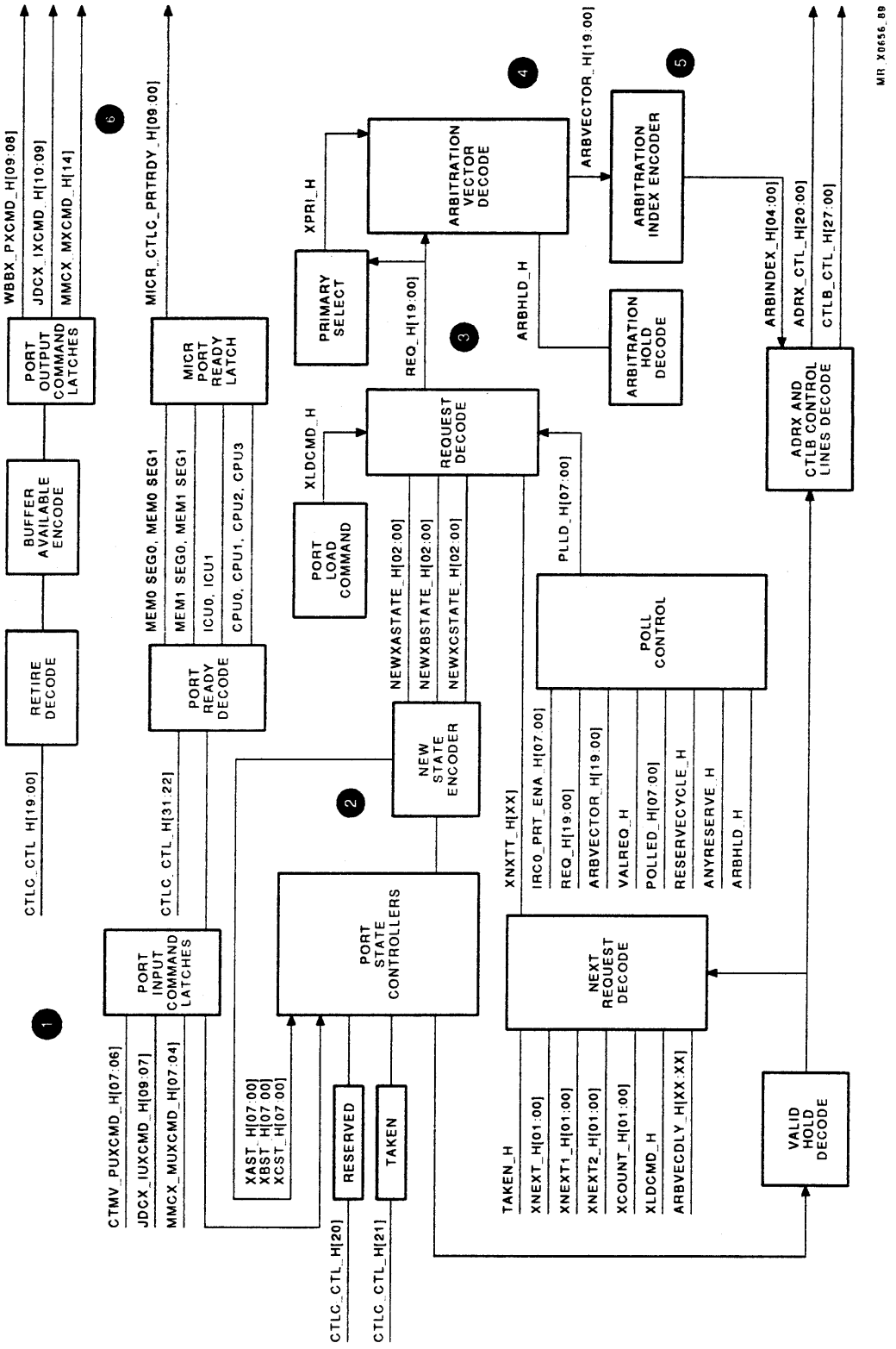


Figure 2–2 CTLA Block Diagram

2.2.1.1 Port State Controllers

Each port sends requests to the JBox, which loads the requests into command buffers. Each port state controller monitors the states of two or three requests. ICU and ACU can have two requests per port state controller. CPU can have three requests per port state controller. Figures 2–3, 2–4, and 2–5 show the inputs and outputs of the CPU, ACU, and ICU port state controllers, respectively. Each CPU, ICU, and ACU port has a state controller. The JBox has four CPU state controllers. Each controller contains three request state machines. Four CPU controllers contain 12 request state machines. Table 2–2 lists each port and its corresponding number.

Each ACU port state controller contains two request state machines. SCU can have two ACU controllers.

Each ICU port state controller contains two request state machines. Two ICU port state controllers contain four request state machines.

Table 2–2 Port Numbers

Port	Number
CPU0	0
CPU1	1
ICU0	2
CPU2	3
CPU3	4
ICU1	5
MEM0	6
MEM1	7

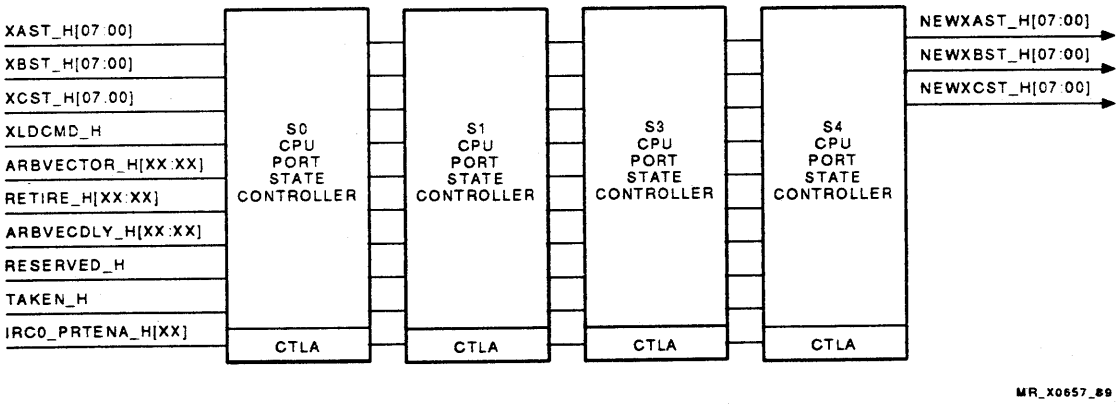


Figure 2–3 CPU Port State Controllers Inputs and Outputs

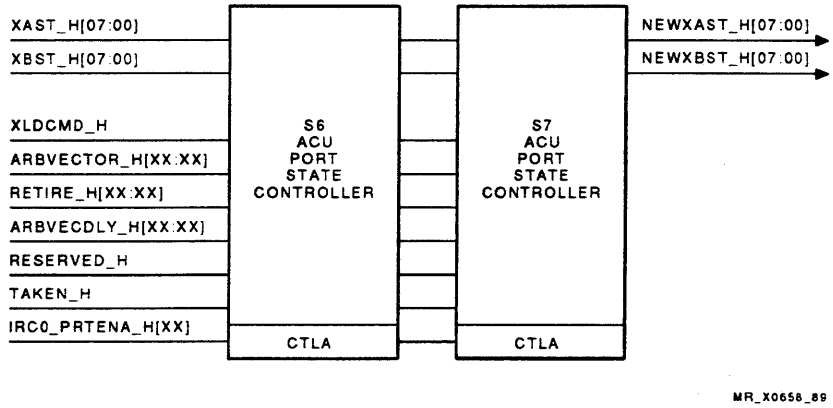


Figure 2–4 ACU Port State Controllers Inputs and Outputs

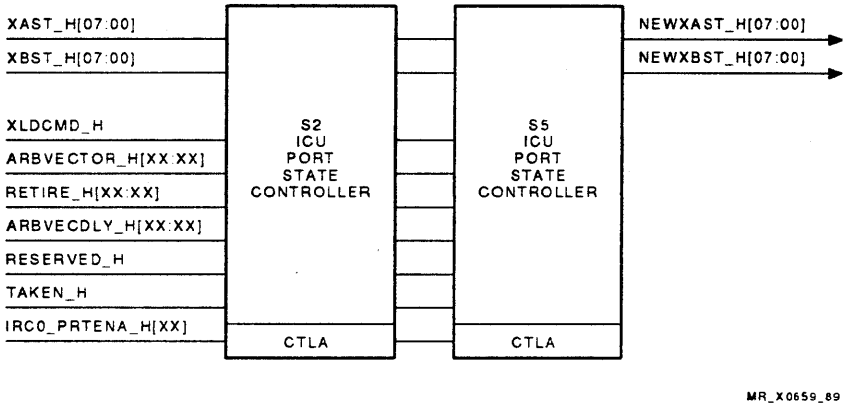


Figure 2–5 ICU Port State Controllers Inputs and Outputs

Figure 2–6 shows how a port state controller receives status from CTLA and CTLC.

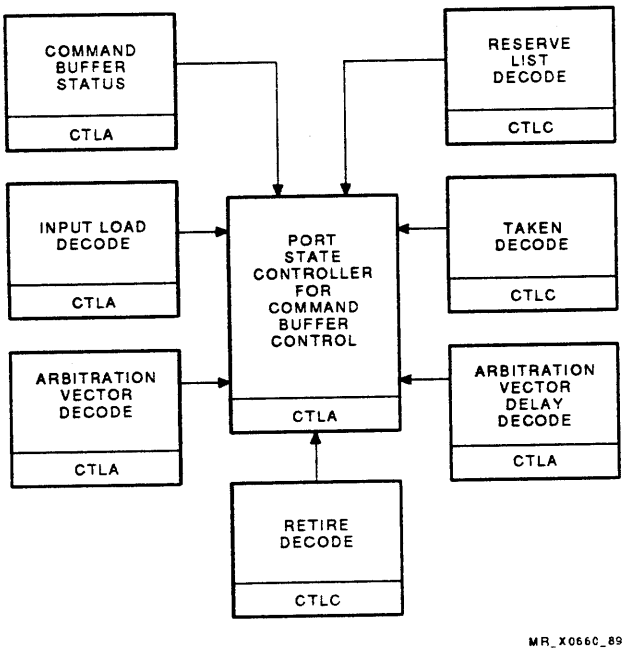


Figure 2–6 Port State Controller

2.2.1.2 Pipeline Stages

In the JBox pipeline, three requests can be in progress at a given time. Figure 2–7 shows the pipeline stages. The following describes the pipeline stages:

- **Load** — The port is notified that a command buffer is available. The port sends a load command to the JBox when sending a valid command.
- **Arbitration** — The JBox arbitrates requests from the CPU, ACU, and ICU ports. If the port is the next port, the JBox assigns an arbitration vector and an arbitration index to the request. The index identifies the port and the command buffer associated with the request that won arbitration.

  If necessary resources are not available when a request is made, the request is placed on a reserved list, where it is arbitrated later by the JBox.
- **PAMM/command** — After generating an arbitration index, the JBox control logic sends control lines to CTLB and ADRX. CTLB latches the port's command and ADRX latches the port's physical address associated with the request.
- **Resource check** — The JBox performs a resource check for each request. Resources include a data path (source and destination), address paths, and a microcode tag queue.

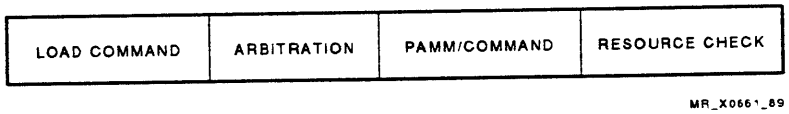


Figure 2–7 Pipeline Stages

2.2.1.3 Monitoring the States of a Request

A request can be in only one of seven states. Figure 2–8 shows the seven request states.

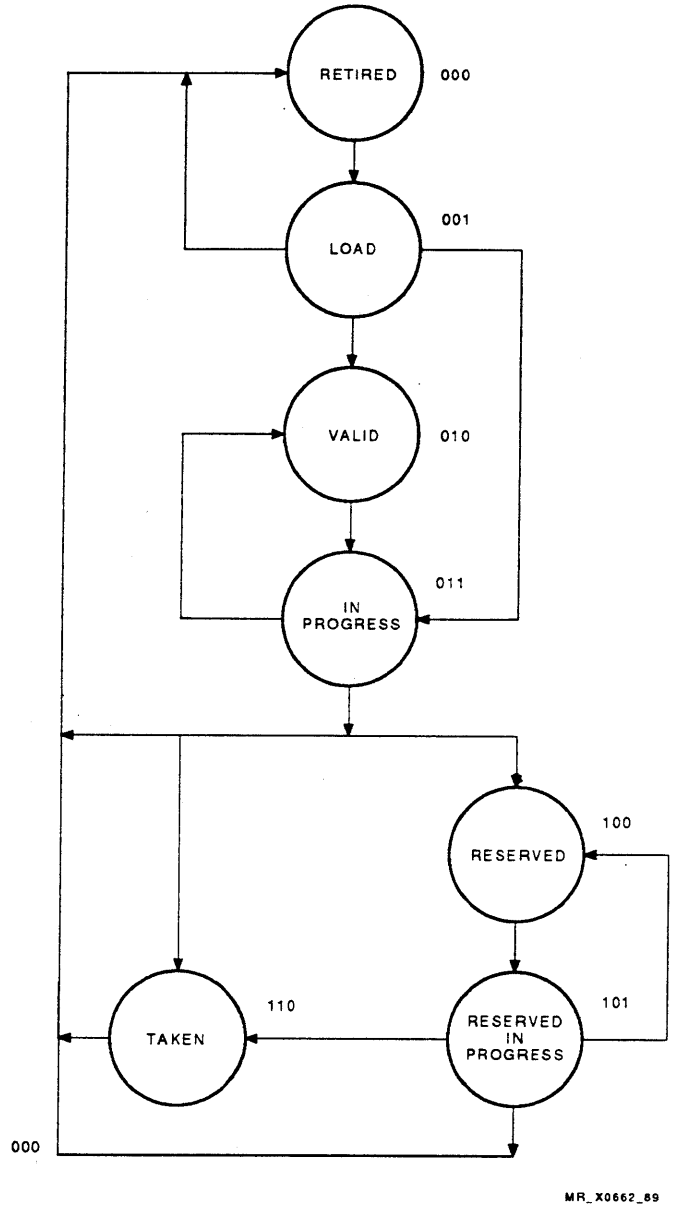


Figure 2–8 Request States

Table 2–3 lists the state requests and their descriptions.

Table 2–3 State Requests

Machine State	State	Description
1	Load	Request is waiting to load its buffers.
2	Valid	Request is loaded but not started into pipeline.
3	In Progress	Request has been started into pipeline.
4	Reserved	Resources are reserved for this request.
5	Reserved in progress	Request has been started into pipeline.
6	Taken	Request has been removed from the list of requests to be started into pipeline.
0	Retired	Request buffers are available.

Depending on arbitration, polling, and available resources, each request moves from the first state (load) to the last state (retire).

Each request in the command buffer is controlled by a state machine. Figure 2–9 shows a CPU port state controller monitoring the state of three requests in CPU command buffers A, B, and C. Each port controller can have only one command buffer in the load state. For a CPU having three command buffers, only one command buffer can have a request in the load state. However, all three command buffers can simultaneously retire requests. The MBox can send up to three commands before having to wait for a command buffer.

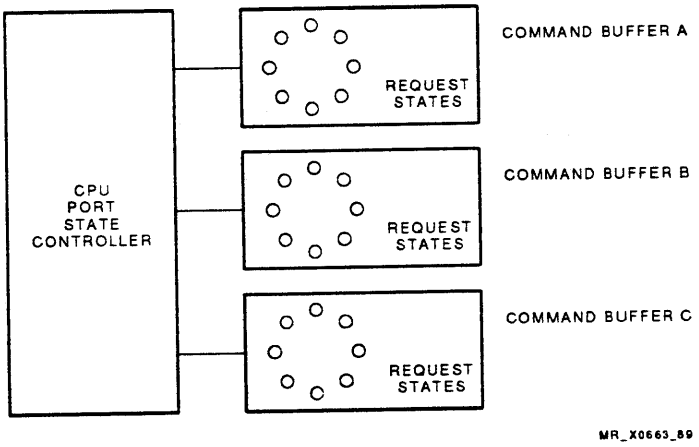


Figure 2–9 CPU State Machines

Figure 2–10 shows an ACU port state controller monitoring the state of two requests in command buffers A and B.

Figure 2–11 shows an ICU port state controller monitoring the state of two requests in command buffers A and B.

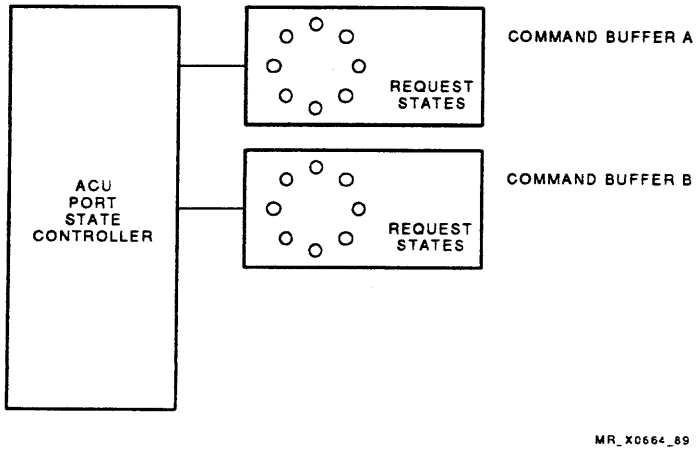


Figure 2–10 ACU State Machines

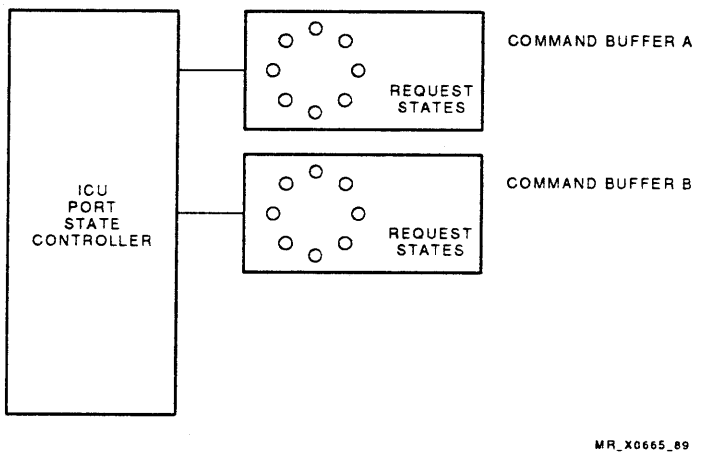


Figure 2–11 ICU State Machines

If more than one command buffer is available, the port state controller decides which command buffer to load when the port sends a new request. The port state controller selects the CPU command buffer with the lowest number. For example, if all three buffers were available, the port state controller would select the buffers in the following order:

Command buffer A = 0
Command buffer B = 1
Command buffer C = 2

If two buffers were available, the port state controller would select the buffers in the following order:

Command buffer A = 0
Command buffer B = 1

Figure 2–12 shows the loading of command buffer A. The port state controller performs the following activities:

❶ Determines that command buffers A and C are available and selects command buffer A.

❷ Loads command buffer A with the command.

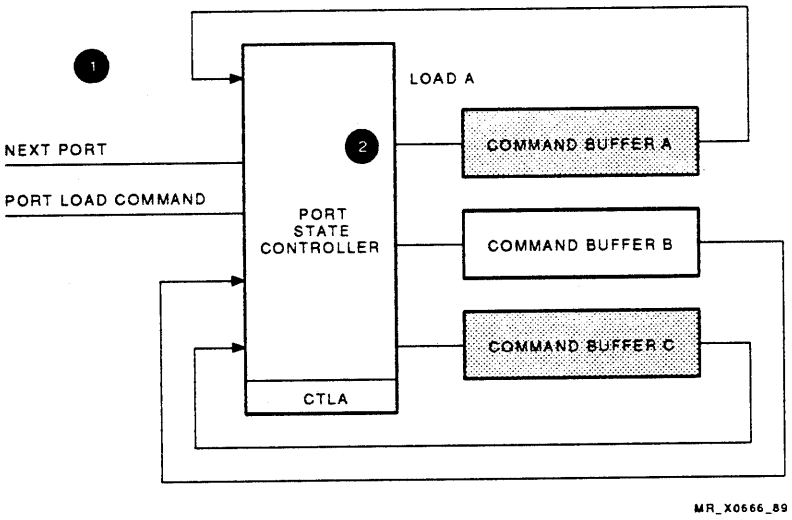


Figure 2–12 Loading Command Buffer A

Figure 2–13 shows the loading of command buffer B.

Command buffers can retire requests simultaneously. For example, the CPU port state controller can retire requests in command buffers A, B, and C simultaneously. The CPU port state controller can retire command buffers out of order as well. The port state controller performs the following activities:

❶ Determines that command buffers B and C are available and selects command buffer B.

❷ Loads command buffer B with the command.

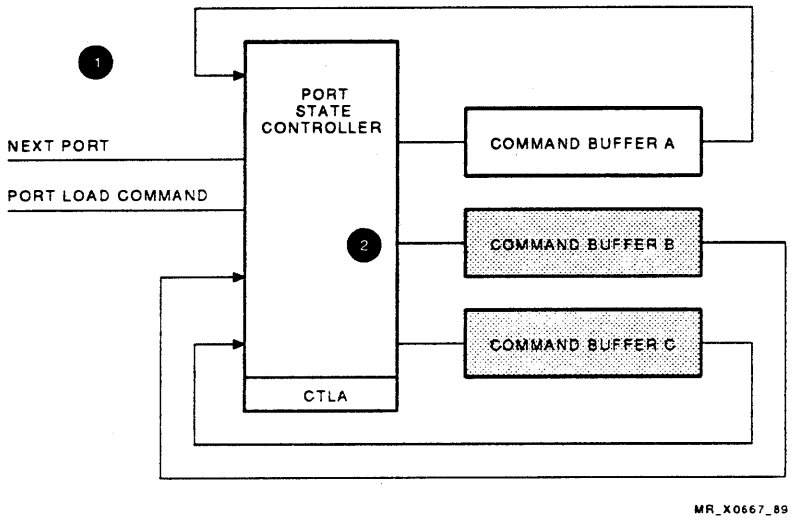


Figure 2–13 Loading Command Buffer B

2.2.1.4 Load Command — Pipeline Stage

Figure 2–14 shows the latching of the port command and address. The sequence is as follows:

❶ The CTLA port state controller receives a load command.

❷ CTLA sends CTLA_CTLB_CTL_H[27:00] and parity to CTLB. CTLB latches the command and uses the index field of the control lines to determine the command that wins arbitration.

❸ CTLA sends CTLA_ADRX_CTL_H[20:00] and parity to ADRX. ADRX latches the port address in one of the 16 receive latches. The 4 CPUs have 12 address receive latches and the 2 ICUs have 4 address receive latches for a total of 16. The JBox does not have to latch the address for the 4 ACU requests.

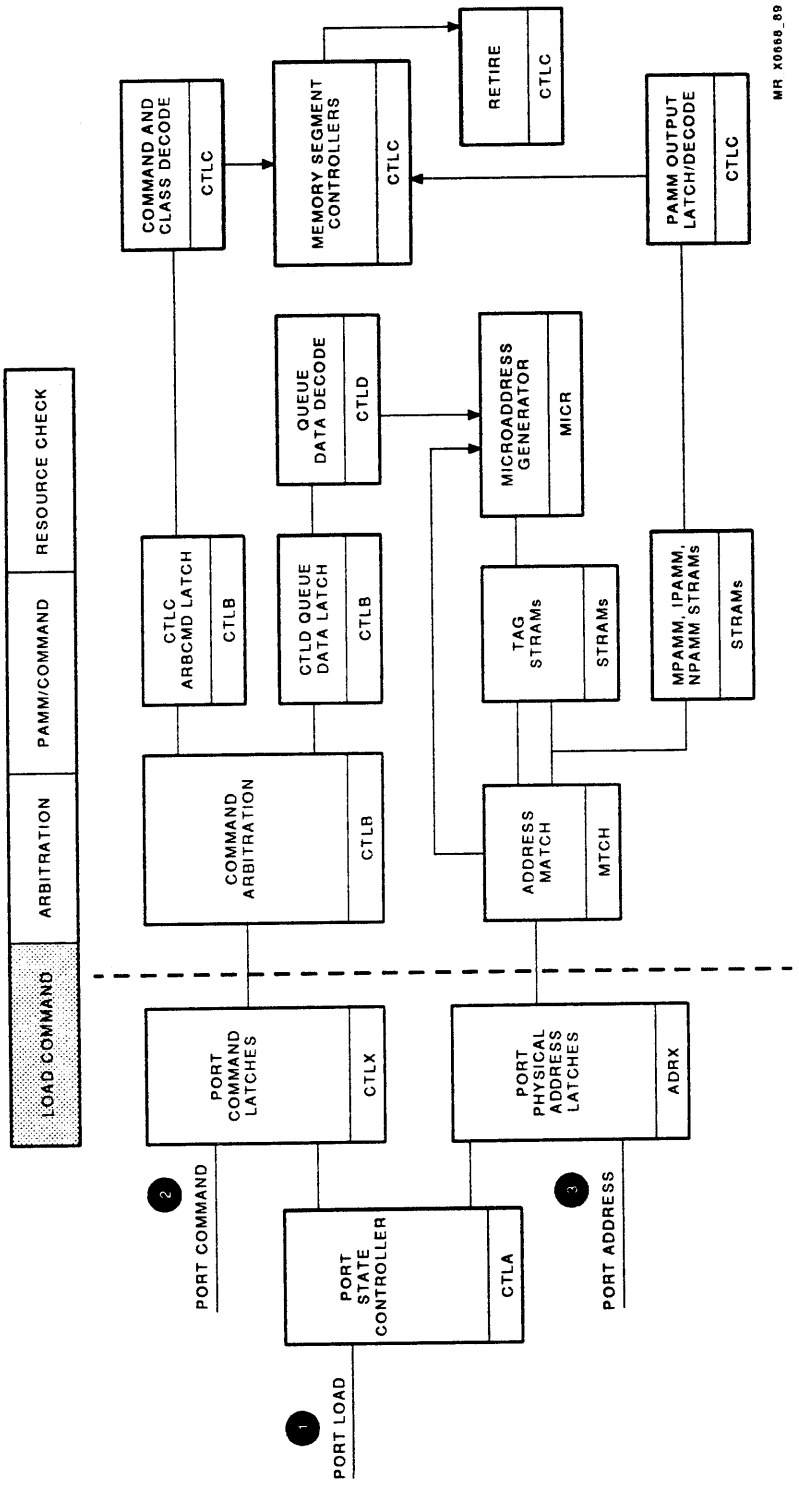


Figure 2–14 Latching the Port Command and Address

2.2.1.5 Arbitration

As shown in Figure 2–15, CTLA contains the following arbitration logic:

❶ Polling

❷ Port state controller

❸ Arbitration vector and index generator

The JBox uses the following logic to arbitrate port requests:

- Polling logic
- Valid request decode logic
- Next request logic
- Priority logic
- Arbitration vector and index logic

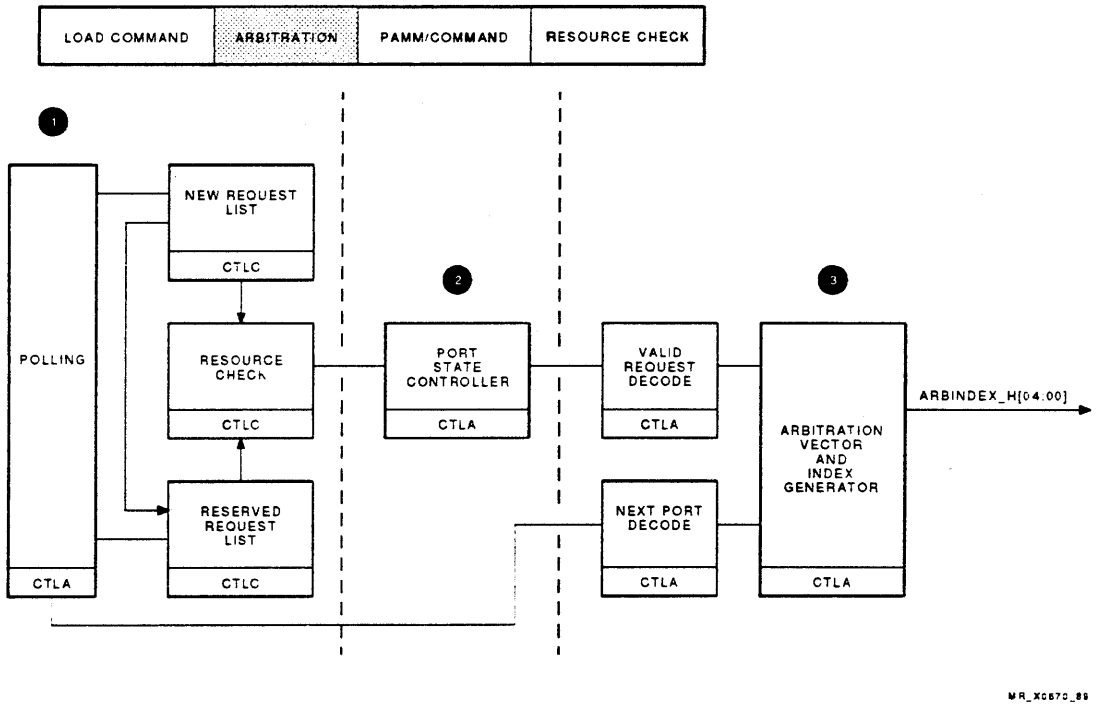


Figure 2–15 Arbitrating Requests

2.2.1.5.1 Polling Logic

The JBox uses the polling logic to poll and examine two lists for requests, the new request list and the reserved request list. The JBox adds requests from eight ports to the new request list. The JBox polls the new request list and performs a resource check. If the resources are not available, the JBox denies the request and moves the request to the reserved request list (only on the first polling sequence).

If resources are available, the JBox places the request in the tag queue. The micromachine control logic removes the entry, using the information to form the microaddress that is sent to the control store for the microword needed to handle the request. Figure 2–16 shows the new request and reserved request lists.

The JBox alternates between the two request lists. First, JBox removes the port requests and continues to poll the new request list until the last request is examined. Then the JBox polls the reserved requests until the last request on that list is examined. The JBox does the following:

- Does not process the last request of the new request list and the last request of the reserved request list back-to-back.
- Cannot add another request to the reserved request list for a port until the reservation list for all ports are empty.

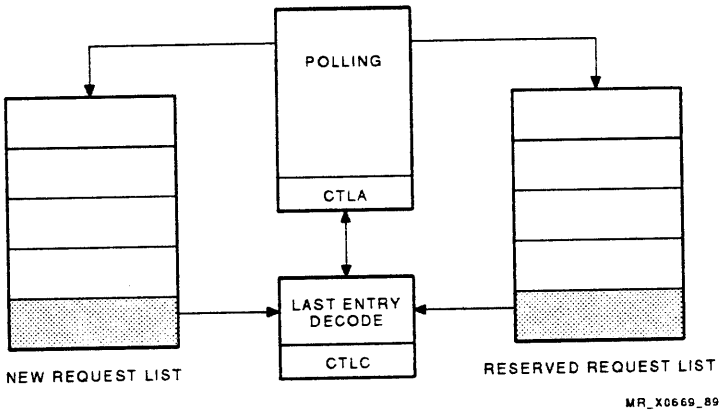


Figure 2–16 Polling the New Request List and the Reserved Request List

2.2.1.5.2 Valid Request Decode Logic

The JBox uses the valid request decode logic to indicate that a new request is in the load or valid state (machine states 1 or 2) or that a reserved request is in the reserved state (machine state 4). The JBox arbitrates requests from the new request list or the reserved request list.

2.2.1.5.3 Next Request Logic

The JBox uses the next request logic to identify the next request to win arbitration. The next request logic enables a valid request to be sent for arbitration. Figure 2–17 shows (❶) the CPU3 port state controller and the request bits REQ_H[13:11] associated with CPU3's three command buffers. Valid request decode examines the current state of each command buffer for reserve status (4CST_H[04] - 4 = CPU3, CST = current state, [04] = reserved state). As shown (❷), the output of the valid request decode logic can generate any request bit for CPU3 to generate 0PRI_H (priority level = 0).

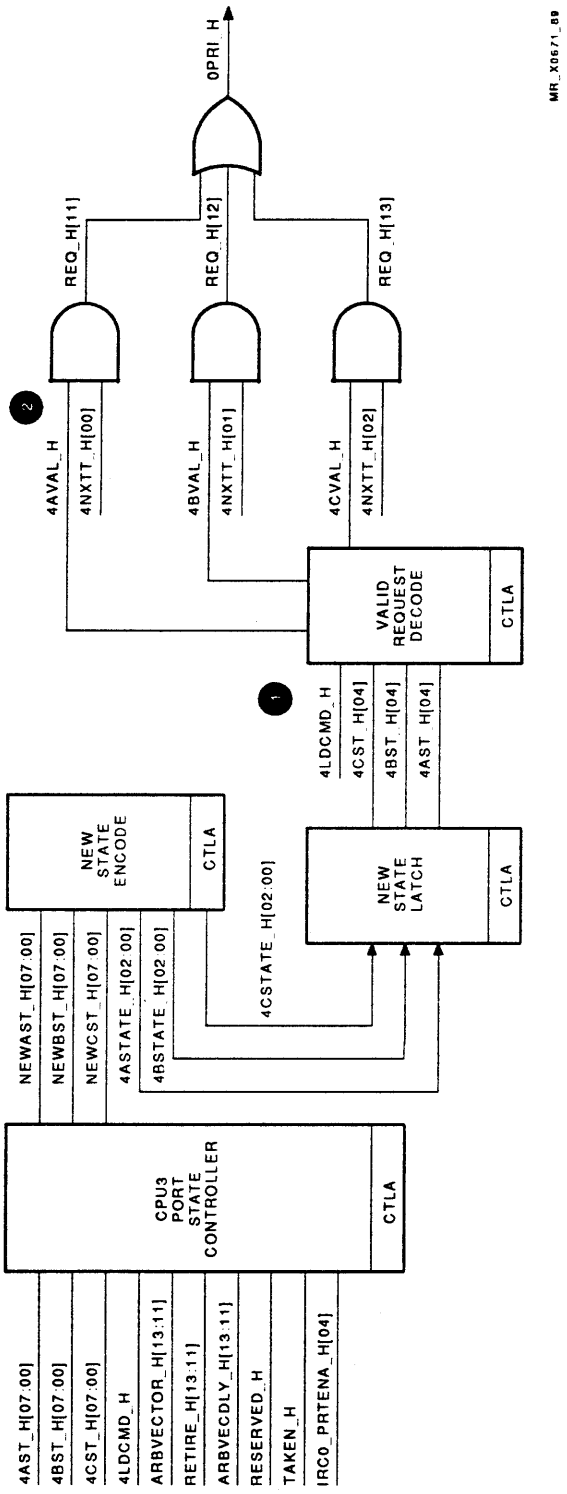


Figure 2–17 Generating the Request Bits

2.2.1.5.4 Priority Logic

Priority levels range from 7 (highest) to 0 (lowest). Table 2–4 lists the ports and priority levels.

The ports' request decode logic determines the priority level.

Table 2–4 Port Priority Levels

Port	Priority
ACU0	7
ACU1	6
ICU0	5
ICU1	4
CPU0	3
CPU1	2
CPU2	1
CPU3	0

2.2.1.5.5 Arbitration Vector and Index Logic

The arbitration vector decode logic receives inputs from priority logic and valid request logic, generating ARBVECTOR_H[19:00] as shown in Figure 2–18. The JBox sends ARBVECTOR_H[19:00] to the priority encoder and generates ARBINDEX_H[04:00]. The arbitration index identifies the port and command buffer that has won arbitration and whose request is in the pipeline. (In this example, CPU3 command buffers A, B, and C have index values B, C, and D[hex].)

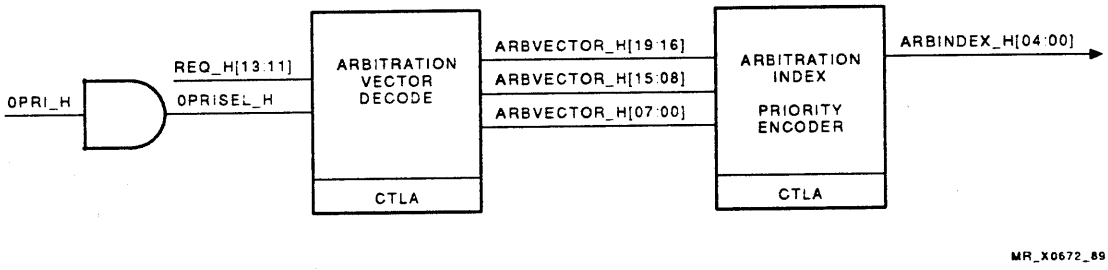


Figure 2–18 Generating the Arbitration Vector and Index

2.2.1.6 Arbitration Index

The arbitration index is a unique value that identifies a port and command buffer associated with a request that has won arbitration. The JBox sends the arbitration index to ACU. However, while the index value remains unchanged, the index name changes as follows:

- **Arbitration index** — This name identifies the request that has won port arbitration.
- **Primary index** — The memory segment controllers latch the arbitration index as the primary index.

- **ECC index** — The ACU latches the primary index for ECC as the ECC index.
- **Return index** — The JBox sends the arbitration index with a memory read command to ACU. If the memory segment is available, ACU sends the index to the ADRX MCAs. The index selects the buffer holding the address bits for the request. The JBox uses the bits to generate row and column address bits to the main memory array for the read data. ACU sends back a return data read command and the arbitration index as a return index to the JBox.

Table 2–5 lists each port and command buffer and the corresponding index value.

The JBox stores the addresses for 12 CPU requests and 4 ICU requests. The JBox uses 4 index bits to address 1 of 16 locations. Memory requests have index bit [05] set to 1 and do not need to have the addresses stored.

Table 2–5 Index

Port and Command Buffer	Index Value
CPU0 A	0
CPU0 B	1
CPU0 C	2
CPU1 A	3
CPU1 B	4
CPU1 C	5
ICU0 A	6
ICU0 B	7
CPU2 A	8
CPU2 B	9
CPU2 C	A
CPU3 A	B
CPU3 B	C
CPU3 C	D
ICU1 A	E
ICU1 B	F
MEM0 A	10
MEM0 B	11
MEM1 A	12
MEM1 B	13

2.2.1.7 PAMM STRAMs

The JBox control logic receives memory mapping information from three types of STRAMs, as follows:

- Memory physical address memory mapping (MPAMM)
- I/O physical address memory mapping (IPAMM)
- Nonexistent physical address memory mapping (NPAMM)

SPU issues an initialize memory command and determines the memory configuration from the self-test results. SPU configures the results and loads MPAMM and NPAMM. The following list summarizes the steps required for SPU to load the MPAMM and NPAMM:

1. Initialize the control stores and control STRAMs.
2. Start system clocks, with the JBox beginning in the idle loop. The memory subsystem begins executing the autosize routine and tests main memory (built-in self-test).
3. Read the results from the memory control registers through the logical interface. The JBox PAMM data is constructed from these results.
4. Load MPAMM and NPAMM using the PAMSCAN_DATA_A_H[07:00] to the ADRX MCAs.

SPU issues an initialize I/O command to reset all XJAs and scan the XMI interface for I/O adapters. SPU configures the results and loads IPAMM.

2.2.1.8 MPAMM

Figure 2–19 shows the MPAMM format. ADRX latches the port's physical address and addresses MPAMM. The output of MPAMM points to a physical unit, segment, and bank of memory.

MPAMM maps every memory address into a specific physical memory bank. For example, address 1000 can reside in any one of the eight memory banks. The JBox sends address 1000 to MPAMM, which decodes the address and generates a code identifying the memory bank.

The MPAMMs allow the following:

- Reconfigurations to accommodate different chip sizes for upgrades or for MMU having different chip sizes.
- Patches around bad or suspect areas in memory, accomplished either manually or by means of software intervention.

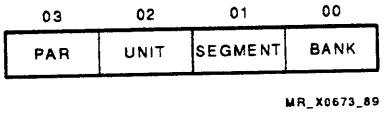


Figure 2–19 MPAMM Format

2.2.1.9 IPAMM

Figure 2–20 shows the I/O PAMM (IPAMM) address allocation.

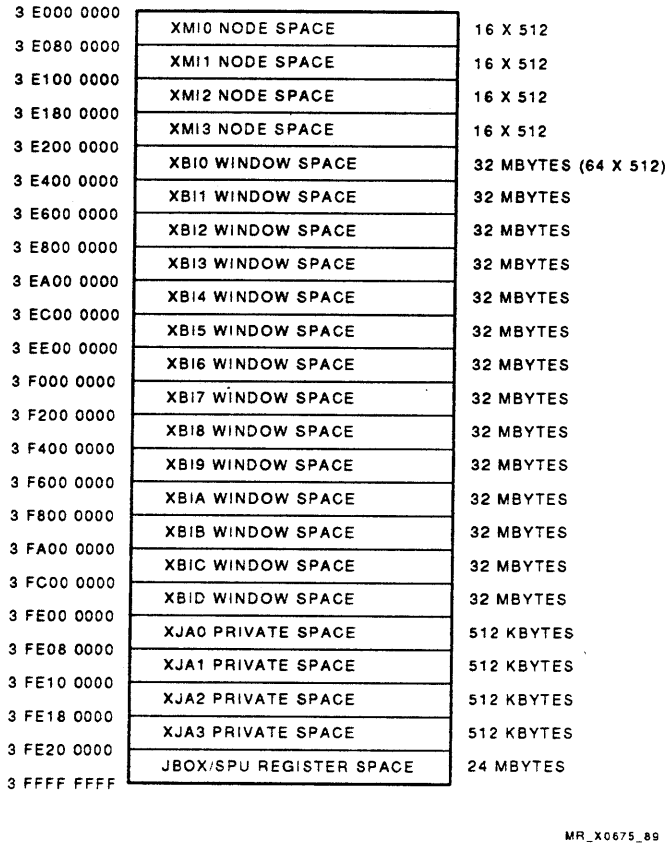


Figure 2–20 IPAMM Address Allocation

The exact physical location of a device at an XMI node address is defined by the contents of the I/O PAMM STRAM. IPAMM has 1K locations.

The operating system determines the exact mapping by reading every XMI node device register starting with the first XMI bus, node 0. Each IPAMM location defines where a 512-Kbyte section of I/O space resides.

The I/O mapping PAMM STRAMs decode PA [28:19] of the address field to determine whether the I/O address points to an adapter, ICU0, or ICU1. The IPAMMs determine whether the address is a JBox or an SPU register. Figure 2–21 shows the IPAMM format at each location.

The starting addresses defined by the contents of the IPAMM STRAMs are as follows:

XJA0 = 3 E000 0000
XJA1 = 3 E080 0000
XJA2 = 3 E100 0000
XJA3 = 3 E180 0000
IRCX = 3 E200 0000
SPU = 3 E300 0000

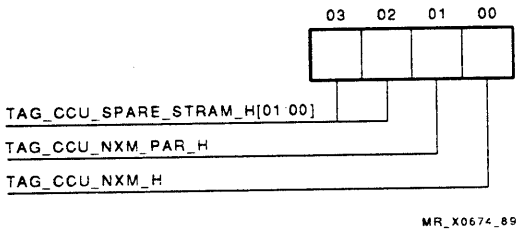


Figure 2–21 IPAMM Format

2.2.1.10 NPAMM

NPAMM outputs the nonexistent memory (NXM) bit. This bit is sent to the CTLD MCA and inserted in the tag queue entry associated with the request. This bit forces the MICR MCA to form a microaddress to send to a microword in the control store that handles a nonexistent memory address and to notify the requesting port that this condition has occurred.

2.2.1.11 CTLA MCA — Inputs and Outputs

Table 2–6 lists the inputs and outputs that CTLA receives from and sends to the MCAs and ports.

Table 2–6 CTLA MCA Inputs and Outputs

MCAs or Ports	Inputs	Outputs
CPU	Buffer available Load command Parity	Buffer available Fatal error Parity
ICU	Buffer available Load command DAX fatal error Parity	Buffer available Parity
ACU	Buffer available 0 and 1 Load command DBX fatal error Parity	Buffer available Parity
CTLB	PPARIN[07:00] PPAROUT[02:00] PARERR_L	Hold command [19:00] Arbitration index Valid request Valid reserve Last Parity
CTLC	Retire [19:00] Reserved Taken Load command [09:00] PARERR_L PPARIN[01:00] PPAROUT[07:00] Parity	Port ready [09:00] Parity
ADRX	–	Hold address [15:00] Arbitration hold PAMM index Parity
CDXX	–	Set attention
IRCX	Port enables [07:00] Parity	–
MICR	PARERR_L	Port ready [09:00] Parity
Control store	Arbitration hold Parity	–

2.2.2 CTLB MCA

The CTLB MCA receives and stores 20 port commands. CTLB also sends the commands to the ports. Figure 2–22 shows the CTLB MCA. This MCA includes the following:

❶ Port input command latches for CPU, ACU, and ICU command fields. CTLB receives the CPU data ready, which cache set, and command fields. CTLB receives ACU segment number and memory command fields, as well as ICU ID, data size, and command fields.

❷ Command latch that latches the port commands that win arbitration. CTLB latches and decodes input commands received from, and determines the corresponding output commands to send to, the ports. For example, if a CPU sends read refill or write refill to the JBox, CTLB latches and decodes the refill command and sends a read command to memory. Another example, if a CPU sends I/O register read to the JBox, CTLB latches and decodes the I/O command and sends a CPU read command to I/O.

❸ CTLC and CTLD control encoder that generates control lines for the CTLC and CTLD MCAs. CTLC decodes the control lines to determine the type of command and to check available resources. CTLD decodes the control lines to determine tag queue data to send to the MICR MCA. MICR uses the queue data to form the microaddress for the microcode.

❹ MICR command decoder that decodes control lines sent from the MICR MCA.

❺ MRMXX control decoder that decodes the fields of the microword.

❻ Port output command latches for CPU, ACU, and ICU command fields. CTLB sends which cache set and command fields to CPU. CTLB sends segment number, index, length, and command fields to ACU, as well as ID, destination value, and command fields to ICU.

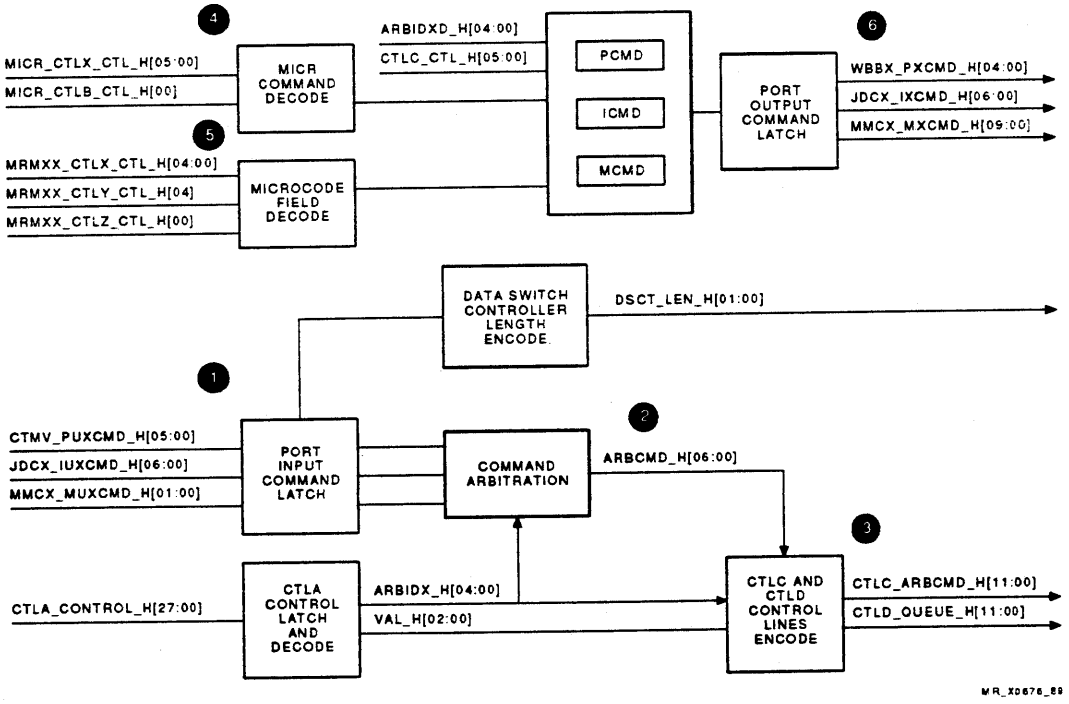


Figure 2–22 CTLB Block Diagram

2.2.2.1 PAMM/CMD Stage

CTLB receives the command field and arbitration index and determines the CTLD queue data. CTLB sends VAL_H[02:00], ARBIDX_H[04:00], and ARBCMD_H[03:00] to CTLC. CTLC performs a resource check, builds the reservation list, and decodes the command and class. Figure 2–23 shows the ADRX and CTLB control lines encoded during the PAMM/CMD pipeline stage.

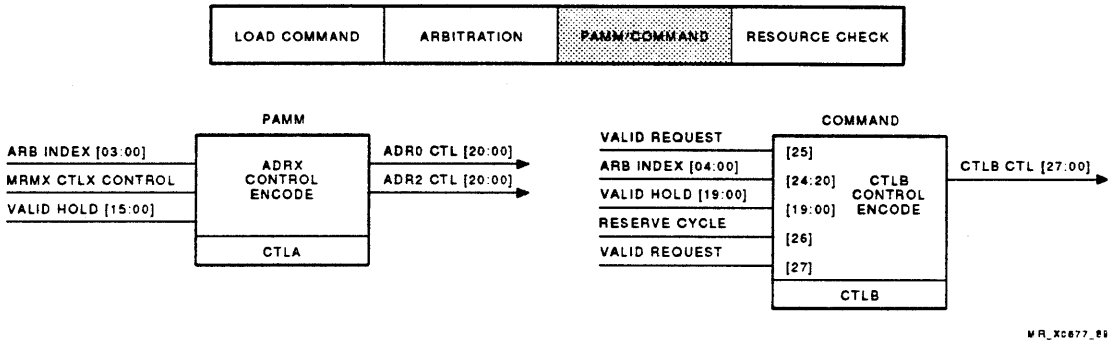


Figure 2–23 PAMM/CMD Pipeline Stage

2.2.2.2 CTLB MCA — Inputs and Outputs

Table 2–7 list the inputs and outputs that CTLB receives from and sends to the MCAs and ports.

Table 2–7 CTLB MCA Inputs and Outputs

MCAs or Ports	Inputs	Outputs
CPU	Command Which set Data ready	Command Which set
ICU	Command Length ID	Command Destination ID
ACU	Command Memory segment	Command Length Index Memory segment
CTLA	Hold command [19:00] Arbitration index Valid request Valid reserve Last Parity	PPARIN[07:00] PPAROUT[02:00] PARERR_L
CTLC	Original index Memory segment Memory bank Parity	Command Arbitration index Valid request Valid reserve Last Parity
CTLD	–	Command Arbitration index Length Which set Parity
DSCT	–	Length Parity
MICR	Index value Memory segment Memory bank Which set Parity	Data ready [03:00] Parity
Control store	Arbitration hold Command Parity	–

2.2.3 CTLC MCA

The CTLC MCA sends commands to the data switch controller, cache consistency information to a tag queue in CTLD, and cache consistency commands to ports. Figure 2–24 is a block diagram of the CTLC MCA. This MCA includes the following:

❶ Port input command latch for ACU, which sends read OK fields for each memory segment.

❷ Reservation register that holds reserved resources waiting to become available.

❸ Resource check logic that compares required resources with available resources to determine whether a request can be taken.

❹ Memory segment controllers that monitor the states for each memory segment.

❺ CTLB, CTLD, ADRX, and DSCT control encoder that generates control lines to the CTLB, CTLD, ADRX, and DSCT MCAs.

❻ CTLA control encoder that generates control lines to the CTLA MCA. These control lines indicate the status of a request as reserved, taken, or retired.

❼ Port output command latches for sending commands to the CPUs, ICUs, and ACUs.

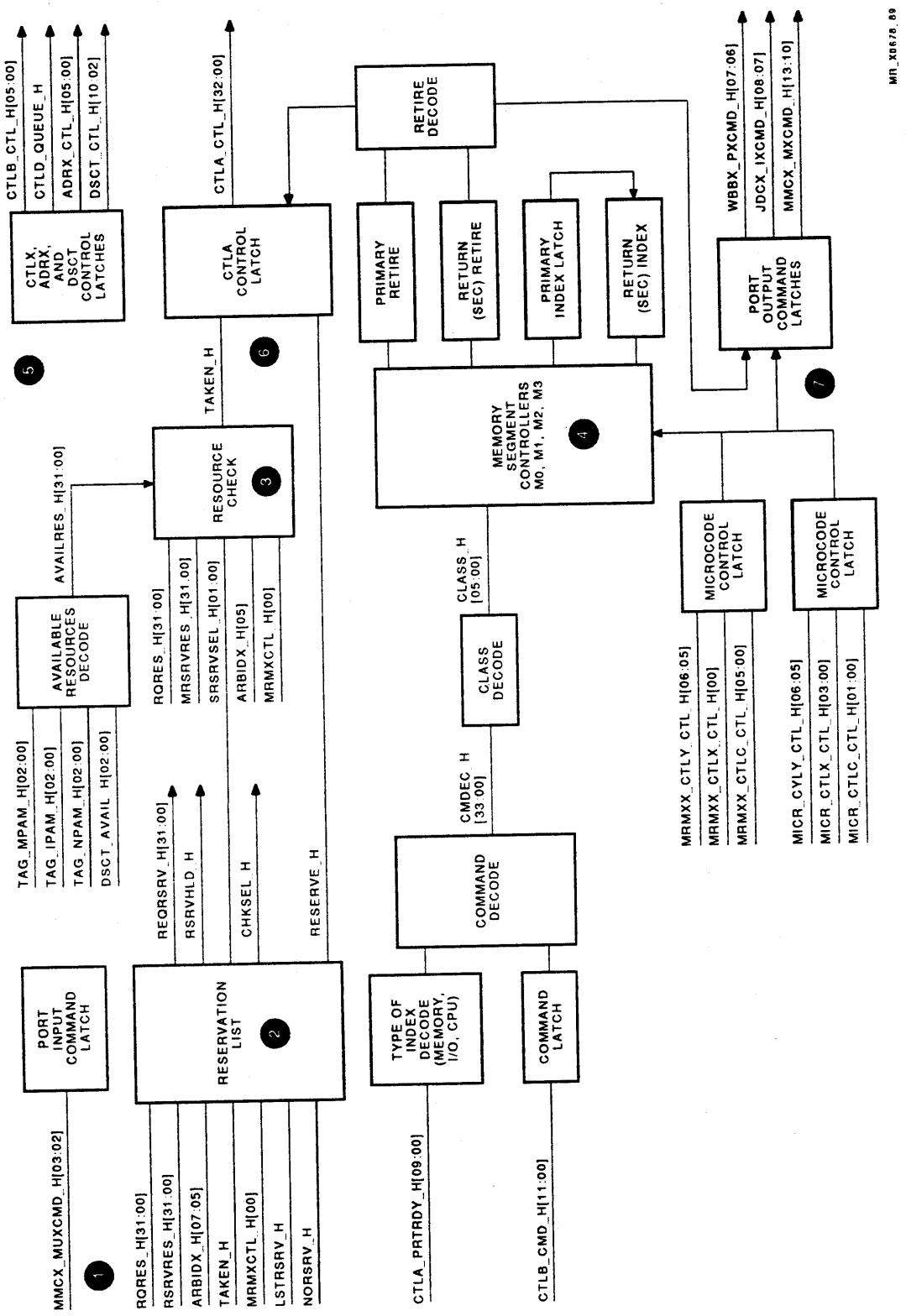


Figure 2–24 CTLC Block Diagram

2.2.3.1 Resource Check Stage

The JBox uses command buffers, data paths (source and destination), address paths, and memory segments as resources. Table 2–8 lists the resources. At any given time, the status of each resource is in one of the following categories:

- **Required** — Needed to complete the command
- **Available** — Not being used
- **Reserved** — Allocated for commands that previously have won arbitration but were unable to be executed because required resources were not available

CTLC decodes the command and performs a resource check to determine whether the resources needed by the request are available. Figure 2–25 shows the inputs to the resource check logic. Usually, if the resources are not available, the request cannot be taken and the JBox moves it onto the reservation list.

The startup logic and microcode share the resources. ARBHOLD is a signal that determines whether the startup logic or microcode has control of the resources. The microcode generates ARBHOLD when it needs the resources. The CTLC resource check logic compares the available resources with the resources requested for the new or reserved requests.

The resource check logic generates TAKEN_H and sends it to the CTLA state controllers for each request taken.

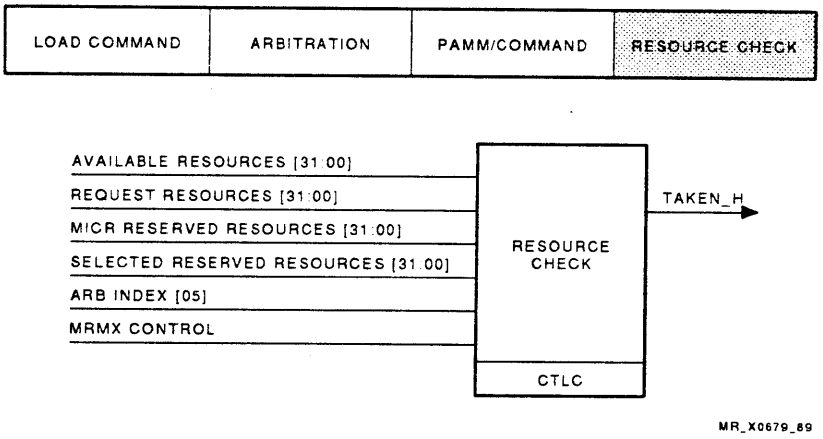


Figure 2–25 Resource Checking

Table 2–8 Resource List for Arbitration

Number	Resource
00	CPU0 command buffer
01	CPU1 command buffer
02	CPU2 command buffer
03	CPU3 command buffer
04	MEM0 SEG0 command buffer
05	MEM0 SEG1 command buffer
06	MEM1 SEG0 command buffer
07	MEM1 SEG1 command buffer
08	IO0 command buffer
09	IO1 command buffer
10	CPU0 data source
11	CPU1 data source
12	CPU2 data source
13	CPU3 data source
14	MEM0 data source
15	MEM1 data source
16	IO0 data source
17	IO1 data source
18	IRC data source
19	CPU0 data destination
20	CPU1 data destination
21	CPU2 data destination
22	CPU3 data destination
23	MEM0 data destination
24	MEM1 data destination
25	IO0 data destination
26	IO1 data destination
27	IRC data destination
28	From DX0 to DX1 cross
29	From DX1 to DX0 cross
30	Address out source 0
31	Address out source 1

Table 2–9 lists the resources needed by SCU to complete memory, CPU, and I/O requests.

Table 2–9 Resources Needed for Commands

Command	Resource Number[1]	
Memory Commands		
Memory read data return (CPU)	00, 01, 02, or 03	Primary index
	14, 15	Arbitration index
	19, 20, 21, or 22	Primary index
	28, 29	Data crossing
	30, 31	Address out
Memory read data return (I/O)	08 or 09	Primary index
	14, 15	Arbitration index
	25, 26, 27	Primary index
Memory ECC report	14, 15	Primary index
I/O Commands		
IRC return data (CPU)	00, 01, 02, or 03	Register index
	18	–
	19, 20, 21, or 22	Register index
	28, 29	Data crossing
	30, 31	–
IRC return data (I/O)	08 or 09	Register index
	18	–
	25 or 26	–
	28, 29	–
	30, 31	–
SPU write I/O register	08, 09	IPAMM
	16 or 17	Arbitration index
	25, 26, or 27	IPAMM
	28, 29	Data crossing
	30, 31	–
SPU read I/O register	08 or 09	IPAMM
I/O register response	None of the resources listed in Table 2–8.	–
Nonexistent device address	00, 01, 02, or 03	Register index
	30, 31	–
DMA write unlock	04, 05, 06, or 07	MPAMM
DMA write	04, 05, 06, or 07	MPAMM
SPU write unlock	04, 05, 06, or 07	MPAMM
Read I/O register return data (CPU)	00, 01, 02, or 03	Register index
	18, 17, 16	IPAMM
	19, 20, 21, or 22	Register index
	28, 29	Data switch crossing
	30, 31	–
DMA read lock	04, 05, 06, or 07	MPAMM
DMA read	04, 05, 06, or 07	MPAMM

[1]The resource numbers in this table correspond to the resource numbers in Table 2–8.

Table 2–9 (Cont.) Resources Needed for Commands

Command	Resource Number[1]	
CPU Commands		
Unlock	04, 05, 06, or 07	MPAMM
Write refill link lock	04, 05, 06, or 07	MPAMM
Cache block invalidate	04, 05, 06, or 07	MPAMM
Longword write update link	04, 05, 06, or 07	MPAMM
Write back linked	04, 05, 06, 07 10, 11, 12, or 13 23, 24 28, 29 30, 31	MPAMM Arbitration index MPAMM Data switch crossing Address
CPU write I/O register	08, 09 10, 11, 12, 13 25, 26, 27 28, 29 30, 31	IPAMM Arbitration index IPAMM Data switch crossing Address
CPU read I/O register	08, 09	IPAMM
Longword write update	04, 05, 06, or 07	MPAMM
Write back	04, 05, 06, or 07 10, 11, 12, or 13 23, 24 28, 29	MPAMM Arbitration index MPAMM Data switch crossing
Write refill unlock	04, 05, 06, or 07	MPAMM
Write refill lock	04, 05, 06, or 07	MPAMM
Write refill link	04, 05, 06, or 07	MPAMM
Read refill link	04, 05, 06, or 07	MPAMM
Write refill	04, 05, 06, or 07	MPAMM
Read refill	04, 05, 06, or 07	MPAMM

[1]The resource numbers in this table correspond to the resource numbers in Table 2–8.

2.2.3.2 Resources Required — Examples

This section provides examples to show what resources are needed for memory, CPU, and I/O requests. Refer to Tables 2–8 and 2–9.

2.2.3.2.1 Resources Needed for a Memory Request

For a memory read data return command from ACU in response to a CPU request, the resource checking logic checks for the following resources:

- CPUn output command buffer to send return data read or written to the CPU that requested the read data.
- MEMn data source lines to send the data to the data switch.
- CPUn data destination lines to receive the data from the data switch.
- Data switch crossing lines (DX1 to DX0 or DX0 to DX1, if applicable).
- Address out source to return the address with the data to the CPU.

2.2.3.2.2 Resources Needed for a CPU Request

For a CPU read refill link command, the resource checking logic checks for the MEMn SEGn command buffer in order to send ACU a read command. (This information is provided by the MPAMMs.) The CTLB command latch receives and decodes the read refill link command, loads the ACU output command buffer, and sends a read command to ACU. A read refill link command is followed by a write back command. The read refill link command is loaded into one command buffer, and the write back command is loaded into another command buffer. Although the MBox sends the commands separately, the JBox handles them as though they were linked. The JBox executes the read refill link command and then the write back command.

The MBox moves the cache block (with valid and written data) into the write back buffer to make the block available for refill data, which allows the JBox to return the read data to the MBox first. To avoid the error condition that would result if the MBox requested read data for an address that the JBox has marked as written full or partial, the MBox sends the refill command as refill link. Link indicates that the MBox knows that a valid and written block exists and must be written back to main memory. This command notifies the JBox that this is not an error condition.

The JBox processes the commands in order, ensuring that the commands are not interrupted through the JBox consistency logic.

For the write back command, the resource checking logic checks for the following resources:

- MEMn SEGn command buffer that sends the read command from the JBox command latch in CTLB.
- CPUn data source lines that the CPU uses to hold the write back data.
- MEMn data destination lines that send the data to memory from the data switch. (MPAMM identifies the memory unit, segment, and bank.)
- Address out source lines for the address that ACU uses to address the DRAMs.

2.2.3.2.3 Resources Needed for an SPU Write I/O Register Request

For an SPU write I/O register, the resource checking logic checks for the following resources:

- ICUn command buffer to receive the write I/O register command. (The IPAMM identifies the I/O device and determines which I/O command buffer is needed.)
- IOn data source lines to send the data to the data switch.
- IOn data destination lines to send the write data from the data switch.
- Data crossing lines (DX0 to DX1 or DX1 to DX0, if applicable). (IPAMM identifies the I/O device and determines which I/O command buffer is needed.)

2.2.3.3 Memory Segment Controllers

The main memory units have four memory segments that can be cycled independently:

MEM0 SEG0
MEM0 SEG1
MEM1 SEG0
MEM1 SEG1

Each segment has a memory segment controller. The memory segment controllers are shown in Figures 2–26 and 2–27.

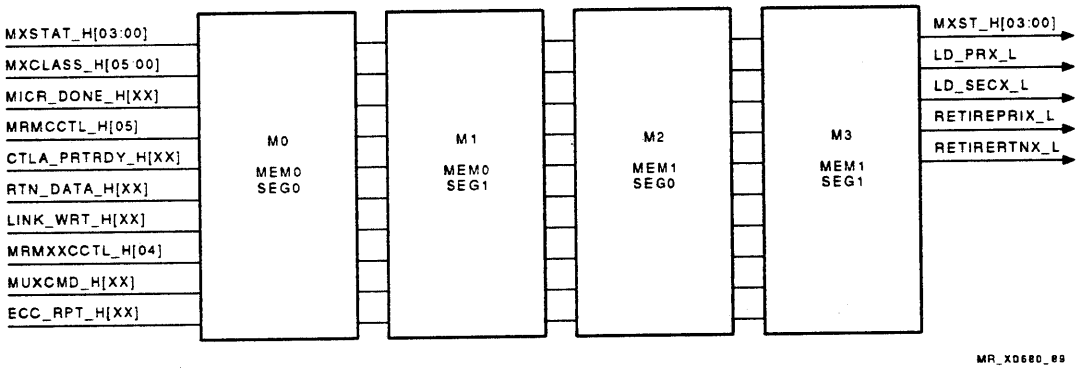


Figure 2–26 Memory Segment Controllers

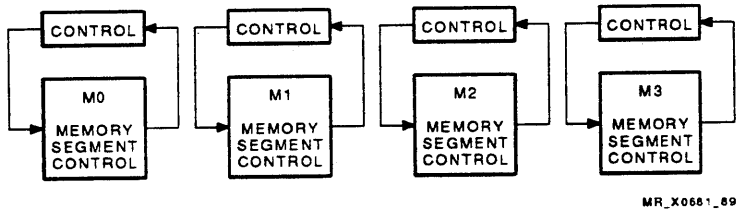


Figure 2–27 Controlling the Memory Segments

The segment controllers are bit-specific state controllers that receive and send a 4-bit state field. Table 2–10 lists the bits and their descriptions.

Table 2–10 Segment Controller Bit Descriptions

Bit	Name	Description
00	MICR	Memory segment controller is waiting for microcode MICR_DONE. When the memory segment controller receives microcode done, the address stored in that tag is released.
01	RTN	Memory segment controller is waiting for memory to send the return data request for that segment. This is for reads only.
02	PRTRDY	Memory segment controller is waiting for the port ready logic to send port ready for that segment. The memory segment has completed using the address in the tag.
03	LINK	Depends on interleave mode. The memory segment controller needs this to order the requests for link commands. The read is done before the write. The microcode handles the read portion first and the link bit indicates that the information must be held to handle the write request. The microcode knows there is a link request and that the command buffers for the CPU should have the read portion in one and the write portion in the other. This bit is reset when write is completed.

If two or more memory commands have an address in the same segment, the JBox handles the requests sequentially. While the JBox accesses a segment for a command, the segment is inaccessible to other commands.

The memory segment controllers receive inputs from the present state, the class decode, MICR, microcode, port ready for the four ports (4, 5, 6, 7), command decode, memory command field read OK, and the ECC report for that segment. The segment controllers control the primary and secondary index load and retire decode logic. CTLC determines the return index.

2.2.3.4 Command Class Types

Port commands fall into one of five command class types. Table 2–11 lists the classes of requests and their descriptions.

Table 2–11 Class Types

Class	Description
0	Wait for microcode
1	Memory read
2	CPU write to memory
3	DMA write to memory
4	Link, read refill
5	Longword write link

2.2.3.5 Retire Decode Stage

CTLC contains the retire logic shown in Figure 2–28. When the JBox retires a request, the address that is held in the ADRX address receive latches is no longer needed. The JBox can retire a request as a result of the following:

- **MICR_DONE_H** — The memory segment controller receives MICR_DONE_H, indicating that an index value can be retired. The memory segment controller keeps track of two index values, primary and secondary (ECC). The memory segment controller latches the primary index value, which the JBox sends to ACU. The JBox retires primary index values for write operations. The secondary index field corresponds to the index value ACU sends to the JBox when an ECC SBE or DBE is detected. The JBox retires secondary index values for ECC reports.
- **I/O register read and write operations** — The JBox can retire I/O register read and write operations immediately.
- **JBox encodes an output load command** — The JBox encodes an output load command and makes a command buffer available to the port.
- **TAKEN_H** — The resource check logic generates TAKEN_H.
- **Return data** — A port receives return data, a memory segment becomes available, or the microcode sends MICR_DONE_H.

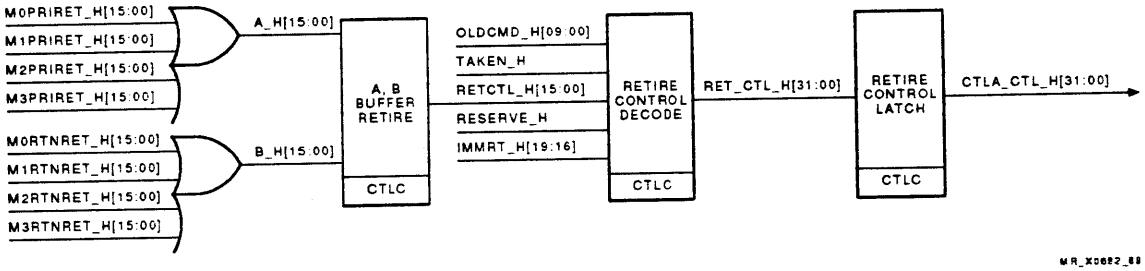


Figure 2–28 Retire Request Decode

2.2.3.6 PAMM Data Decode

ADRX addresses the PAMM STRAMs, which send the following data to CTLC:

- TAG_MPAM_H[02:00] and parity
- TAG_NPAM_H and parity
- TAG_IPAM_H[02:00] and parity

CTLC decodes the PAMM data as follows:

1. MPAM_H[02:01] is used for write refill linked with a write back command. MPAM_H[02:01] or IPAM_H[02:01] is selected to determine resources that are being used for the current request.
2. CTLC latches the MPAMM, NPAMM, and IPAMM data. The control logic selects IPAM_H[01:00] or MPAM_H[01:00] to determine CTLB_CTL_H[05:04], which is sent to the ICMD, PCMD, and MCMD command encoders. The output of the encoders contains the command for the ICU, CPU, or memory.
3. IPAM_H[02:00] determines which I/O device, IODEV_H[07:00], is involved. NPAM_H determines whether nonexistent memory was accessed.

2.2.3.7 CTLC MCA — Inputs and Outputs

Table 2–12 lists the inputs and outputs that CTLC receives from and sends to the MCAs and ports.

Table 2–12 CTLC MCA Inputs and Outputs

MCAs or Ports	Inputs	Outputs
CPU	–	Load command Send data
ICU	–	Load command Send data
ACU	Read OK	Load command Send data OK Abort
Tag	IPAMM [02:00] with parity NPAMM [02:00] with parity MPAMM [02:00] with parity	–
CTLA	Arbitration index Valid request Valid reserve Last Parity	Retire [19:00] Reserve Taken Load command [09:00] PPARIN[01:00] PPAROUT[07:00] PARERR_L Parity
CTLB	Command Arbitration index Valid request Valid reserve Last Parity	Original index Memory segment Memory bank Parity
CTLD	Tag queue available Parity	Valid tag entry Parity
ADRX	–	Address out Cycle out Parity
DSCT	–	Destination [03:00] Source [03:00] Valid Parity
MICR	Command buffer [09:00] MBox source Memory segment Memory bank Memory unit DSCT control ICU source Address out 0 and 1 Index Parity	–

Table 2–12 (Cont.) CTLC MCA Inputs and Outputs

MCAs or Ports	Inputs	Outputs
Control store	Stop arbitration Out cycle Send data I/O command Done Memory OK Memory abort Read abort Parity	–

2.2.4 CTLD MCA

The CTLD MCA contains the tag queue in which the microcode access obtains cache check information. Figure 2–29 is a block diagram of the CTLD MCA. This MCA includes the following logic:

❶ Tag queue that contains entries used by the micromachine control logic to form a microaddress to be sent to the control store. The tag queue generates the cycle count for tag cycles.

❷ Queue data decoder that latches and decodes the queue data from the CTLB MCA. CTLB sends command and index values that form the fields of the tag queue data.

❸ MICR queue data latch that latches the output of the tag queue and queue data decoder. The tag queue data is shown in Figure 2–30. Table 2–13 lists the fields.

❹ SPU interface for handshaking signals CTLD receives from and sends to SPU. CTLD sends the command, address, and data from SPU to ICU. ICU decodes the SPU commands and loads the SPU receive buffer with the command, address, and data.

To send the command, address, and data to SPU, ICU loads a transmit buffer. Under ICU control, the transmit buffer is unloaded, sending the command, address, and data to the CTLD MCA. CTLD sends handshaking signals to SPU.

❺ Microcode address generator that is used during system initialization. The address generator addresses the control store when the microcode is loaded.

❻ Microcode data and write enable logic that is used during system initialization. The enable logic allows data to be written to the control store.

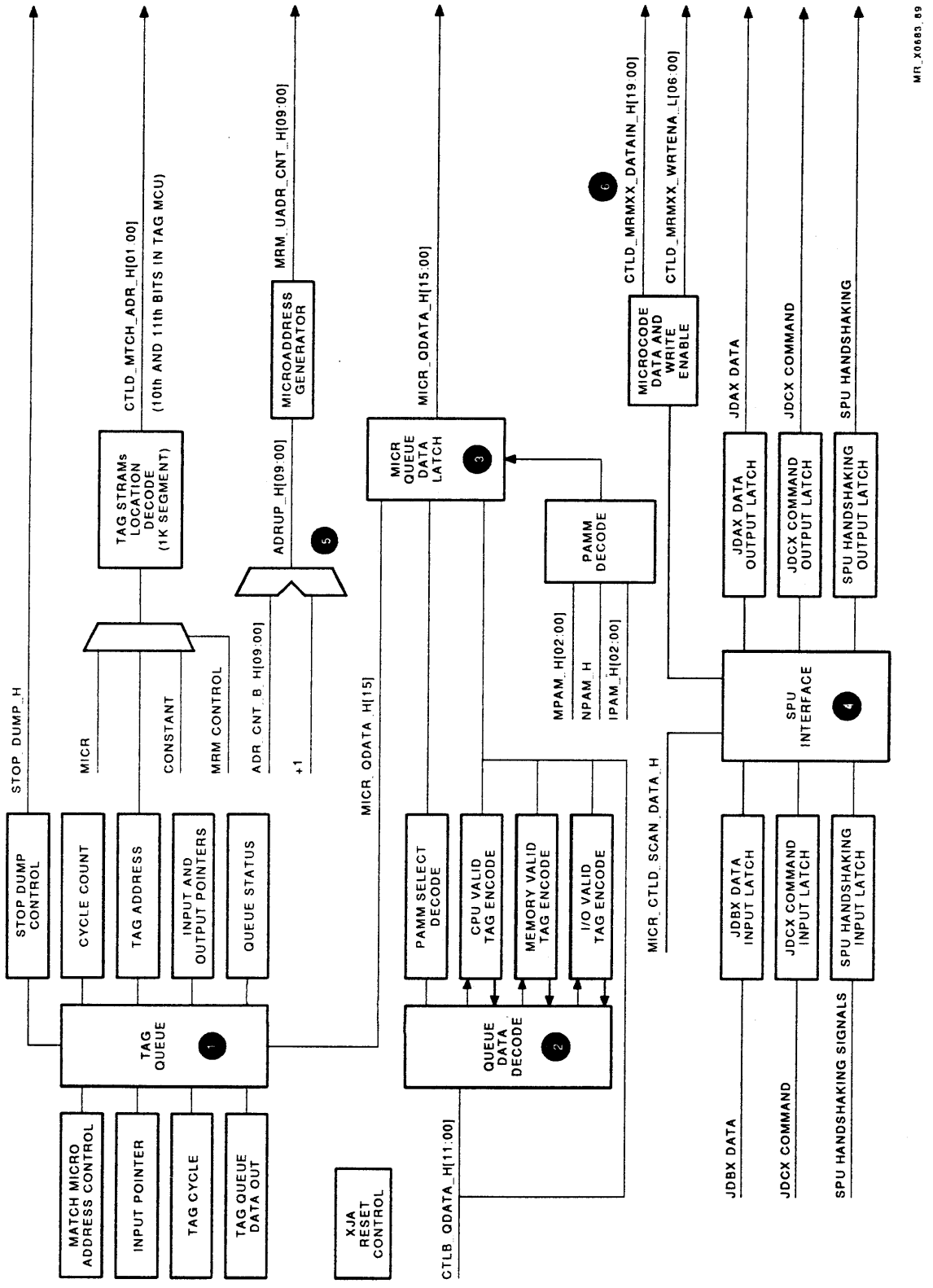


Figure 2–29 CTLD Block Diagram

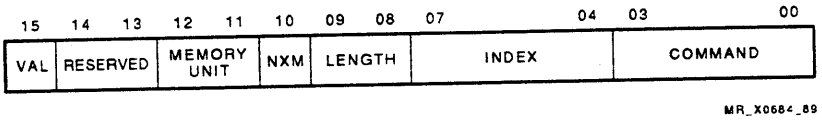


Figure 2–30 MICR Queue Data Fields

Table 2–13 MICR Queue Data Field Descriptions

Bit	Name	Description
03:00	Command	Holds the command for the port that has won arbitration.
07:04	Index	Identifies the port and its command buffer for the request.
09:08	Length	Specifies the data size for the data transfer to or from memory (as indicated by the command field).
10	Nonexistent memory (NXM)	Indicates that NPAMM has detected nonexistent memory for the address latched in the ADRX address hold latches.
12:11	Memory unit	Identifies the memory unit and segment as indicated by MPAMM.
14:13	–	Reserved.
15	Valid entry (VAL)	Indicates that the tag queue has a valid entry.

2.3 JBox Data Paths and Data Path Control (DSXX and DSCT MCAs)

The JBox uses a multipath data switch to connect CPUs, ACUs, and ICUs. The data switch consists of the DSXX MCAs. DS00, DS01, DS02, DS03, DS04, and DS05 support CPU0, CPU1, ACU0, and ICU0. DS06, DS07, DS08, DS09, DS10, and DS11 support CPU2, CPU3, ACU1, and ICU1. Data switch crossing is required when all DSXX MCAs are used to provide paths for all inputs to be switched to any output.

The data switch allows multiple CPU, I/O, and SPU transactions to be executed simultaneously, as long as the transactions do not involve the same port. The CPUs, I/O units, and SPU can exchange data with main memory using the data switch.

The DSCT MCA receives, buffers, and decodes data transaction commands and determines the source ports (data switch inputs) and destination ports (data switch outputs). DSCT monitors the availability of data paths and sends the status to the JBox resource checking logic. The DSCT MCA also receives clock control signals from the clock module and sends these control signals to the MCUs.

The JBox keeps the ports active and resolves cache conflicts by handling communication between the ports and main memory. SCU handles a number of requests that may require the same data switch paths. These paths are considered resources. The JBox arbitration of port requests uses SCU resources in the most efficient manner and reserves previously unavailable resources for subsequent processing.

The JBox has 64-bit-wide data interfaces to the ACUs, CPUs, and ICUs. Figure 2–31 shows the inputs and outputs of the data switch. Inputs to the data switch can be switched to any output.

There are independent 8-byte-wide data interfaces in each direction. Data is transmitted over these interfaces every 16 ns (500 Mbytes/s). The data interfaces between the CPU and SCU originate and terminate at the MBox in the CPU. Figures 2–32 and 2–33 show the MBox-to-JBox and JBox-to-MBox data formats, respectively.

The data interfaces between the ACUs and JBox originate and terminate at the MDPX MCAs and DSXX MCAs. Write data flows from the DSXX MCAs to the MDPX MCAs. Read data flows from the MDPX MCAs to the DSXX MCAs.

The data interfaces between the ICUs and JBox terminate and originate at the JDBX and JDAX MCAs and the DSXX MCAs. Write data flows from the JDAX MCAs to the DSXX MCAs. Read data flows from the JDBX MCAs to the DSXX MCAs. Figures 2–34 and 2–35 show the JDAX-to-DSXX and DSXX-to-JDBX data formats, respectively.

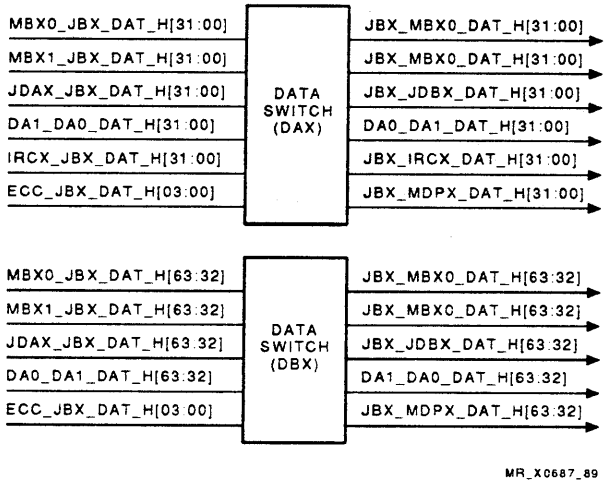


Figure 2–31 Data Switch

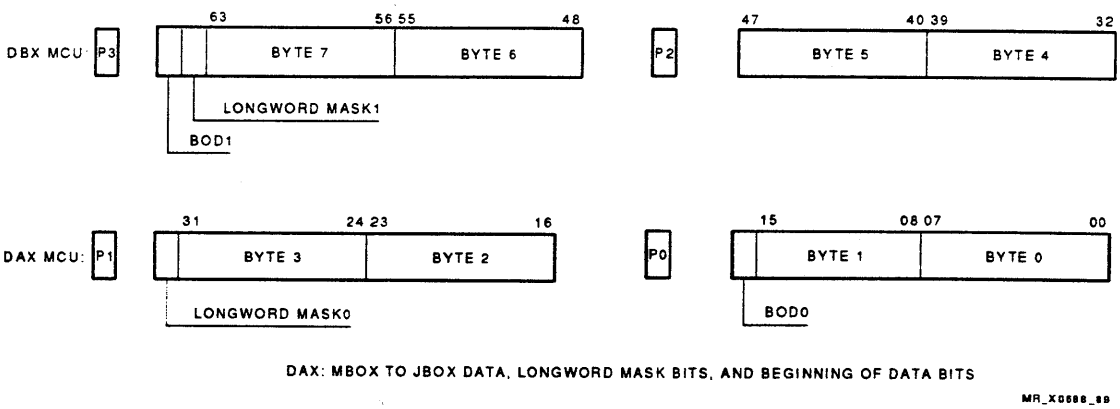


Figure 2–32 MBox-to-JBox Data Format

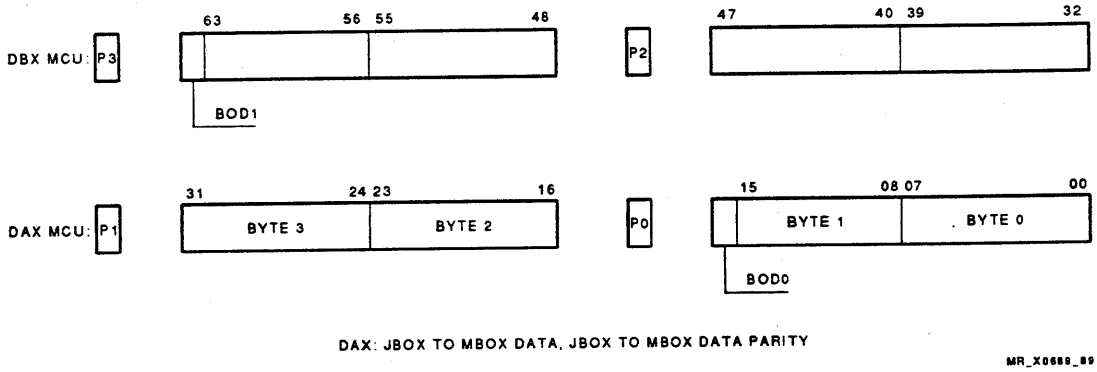


Figure 2–33 JBox-to-MBox Data Format

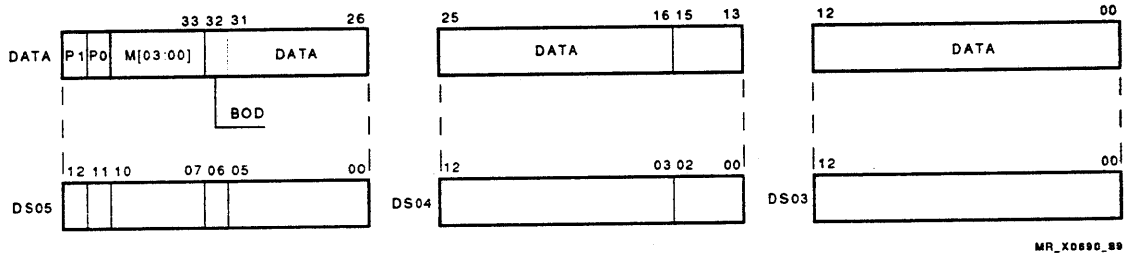


Figure 2–34 JDAX-to-DSXX Data Format

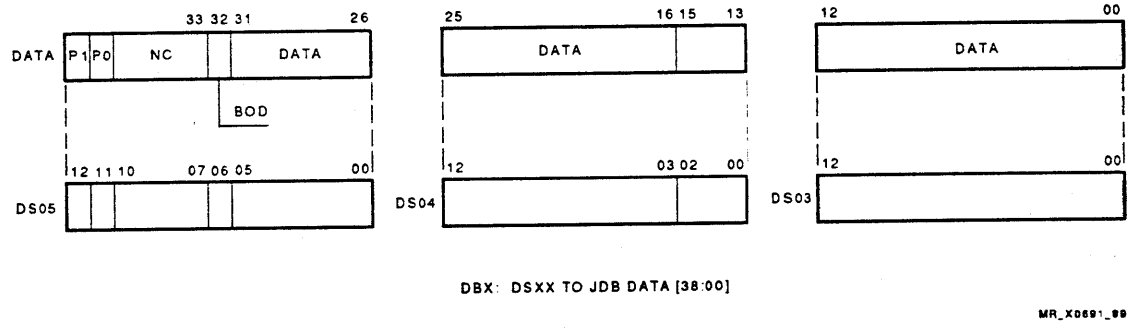


Figure 2–35 DSXX-to-JDBX Data Format

2.4 JBox Address Paths

The tag MCU contains the ADRX and MTCH MCAs, the PAMMs, and global tag STRAMs. The tag MCU receives and transmits addresses to and from the ports. The ADRX MCAs contain the address switch and send physical address bits to the following seven destinations:

- **Memory** — The address switch sends row and column address bits (scan determines row and column [32:26, 07, 06]). The row and column address bits address the following:

  Block and quadword within memory
  Unit, segment, and bank as defined by MPAMM

- **MTCH** — The address switch sends PA [32:06]. MTCH drives the addresses and data of the tag STRAMs. MTCH also compares the physical address with the contents of the tag STRAMs and determines if there is a match.
- **MBox** — Each ADRX MCA sends two 4-bit address slices to each MBox. This requires two cycles.
- **I/O controller** — The address switch sends two 4-bit address slices to each I/O controller. This requires two cycles.
- **MPAMM** — The address switch (two ADRX MCAs) sends a 10-bit address to MPAMM. ADR0 sends bits [09:02], which are PA [33:26]. ADR1 sends bits [01:00], which are PA [07, 06].
- **NPAMM** — Same as MPAMM.
- **IPAMM** — The address switch (three ADRX MCAs) sends a 10-bit address to IPAMM. ADR0 sends bits [09:07] which are PA [28:26]. ADR1 sends bits [02:00], which are PA [21:19], and ADR3 sends bits [06:03] which are PA [25:22].

2.4.1 ADRX MCAs

ADR0, ADR1, ADR2, and ADR3 each deal with one-quarter (eight bits) of the address bits. Each receives one-quarter (eight bits) of the address field from the four CPU ports and two I/O ports and transmits one-quarter (eight bits) of the address fields to the same ports. Figure 2–36 shows the outputs of ADRX.

The ADRX MCAs source row and column addresses to the main memory units. ADRX receives addresses from MBox and I/O in two cycles. The row and column addresses sent to MMU also require two cycles. Row address is sent first, then column. ADR3, for example, receives PA bits [25:22] on the second cycle of the transfer from the MBox or I/O. Using row and column addresses [08, 07], ADRX transmits the bits to MMU in two cycles.

During the first cycle of the transfer, ADRX sends PA bits [22] and [24] and uses row and column 7 and 8, respectively. During the second cycle of the transfer, ADRX sends PA bits [23] and [25] to MMU, and again uses row and column 7 and 8, respectively.

Table 2–14 lists the ADRX MCAs and the corresponding physical address bits generated during the two cycles.

ADR1 sends PA bits [07, 06] and [32:26] to ADR0. These bits are programmable row and column bits [11:09] that ADR0 sends.

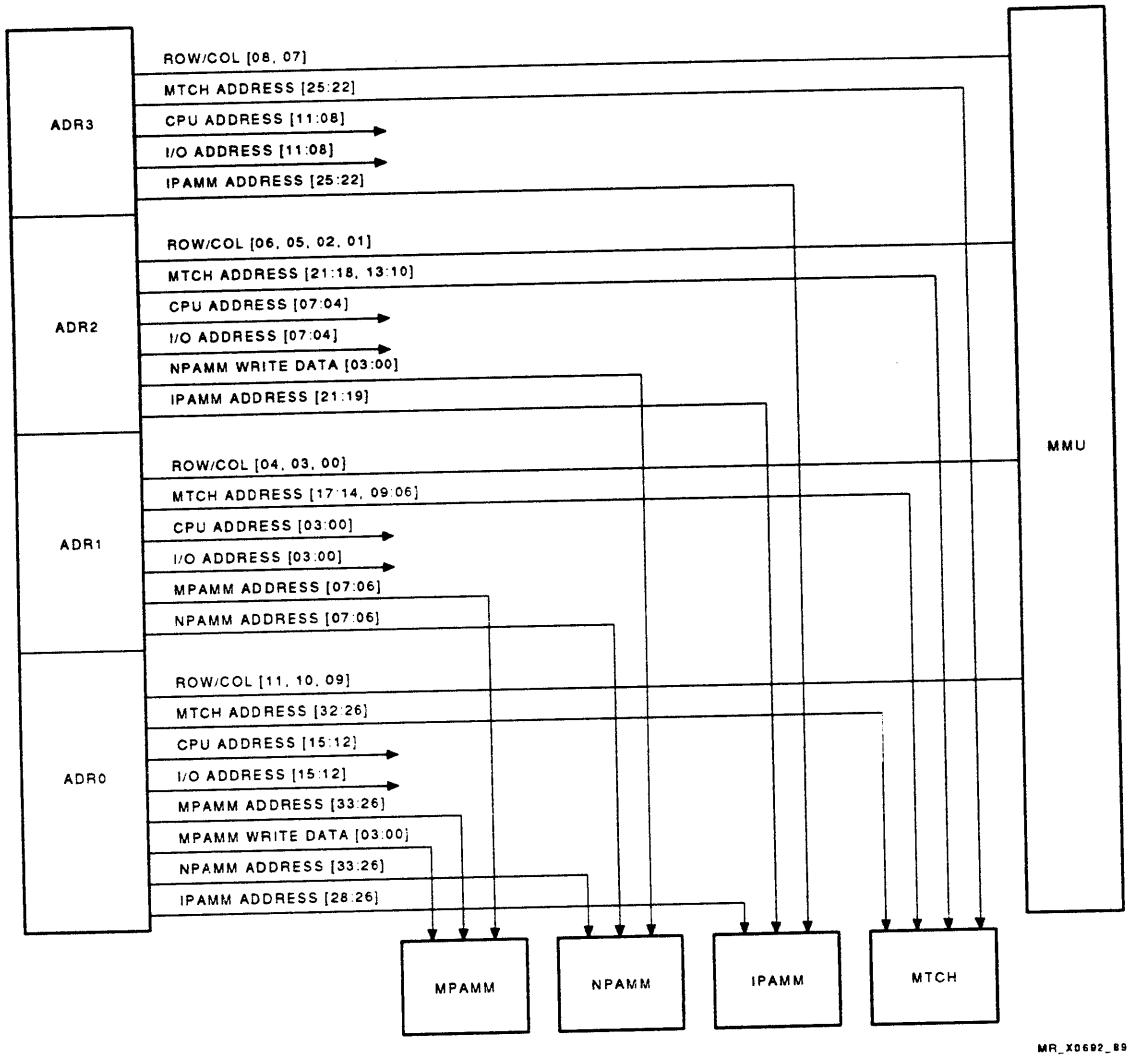


Figure 2–36 ADRX Address Outputs

Table 2–14 ADRX PA Bits

MCA	1st Cycle	2nd Cycle
ADR0	29:26	33:30
ADR1	06:09	17:14
ADR2	13:10	21:18
ADR3	05:02	25:22

2.4.2 Address Receive Latches

Each ADRX MCA contains three address receive latches for each CPU port and two address receive latches for each I/O port. The ADRX MCAs have a total of 16 address receive latches. Figure 2–37 shows the address receive latches in the ADRX MCA. Each latch receives four address bits and one parity bit. The ADRX MCA receives the 32-bit physical address in two cycles.

The 16 latch addresses are wired in parallel to address memory, PAMM, CPU, and MTCH MCA.

CTLA receives the load command from the port and sends CTLA_ADRX_HLD_RLAT_H[15:00] to latch the address corresponding to that command. Each ADRX MCA receives CTLA_ADRX_HLD_RLAT_H[15:00] and sends a 3-bit field to each of the four CPU address receive latches and a 2-bit field to each of the I/O address receive latches.

2.4.2.1 CPU Receive Latches

The address interface between the JBox and MBox is 2 bytes wide, and the complete address is transferred during two cycles. The address bits transmitted across this interface are PA [33:02].

Table 2–15 describes the mapping between the signal names and the physical address bits in the two address cycles.

Table 2–15 Mapping

Signal	1st Cycle PA Bit	2nd Cycle PA Bit
15	29	33
14	28	32
13	27	31
12	26	30
11	05	25
10	04	24
09	03	23
08	02	22
07	13	21
06	12	20
05	11	19
04	10	18
03	09	17
02	08	16
01	07	15
00	06	14

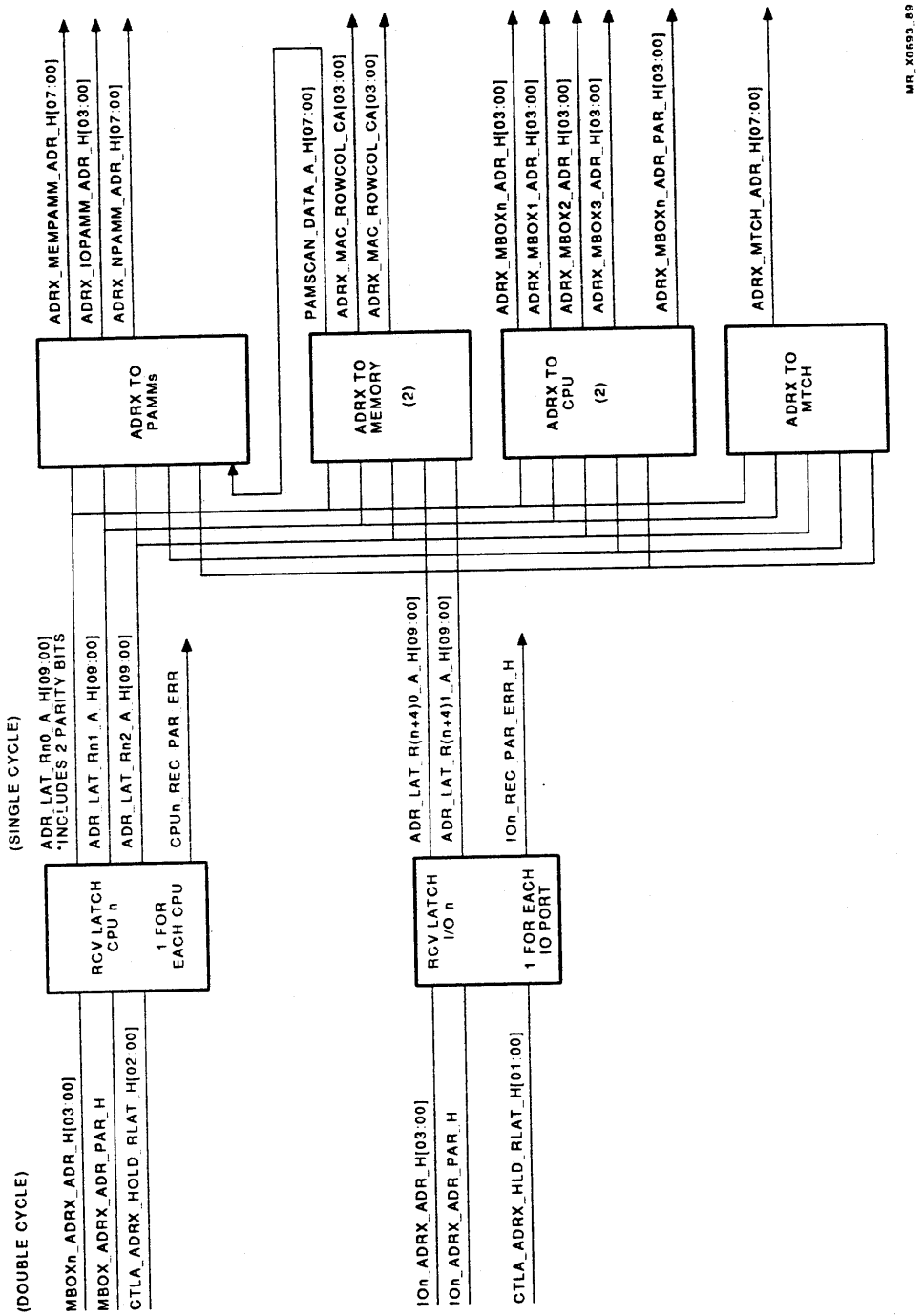


Figure 2–37 ADRX Receive Latches

The ADRX MCA contains CPU receive latches that accept four address bits and one parity bit from the MBox. The ADRX MCA has three CPU hold latches that correspond to CPU command buffers A, B, and C. Each hold latch is 8 bits wide and latches the address in two cycles. Each CPU receive latch outputs three 8-bit address slices plus parity.

The following steps summarize how the ADR3 MCA receives and latches PA bits from CPU0 during a single transaction involving CPU0:

1. ADR3 receives PA [05:02] and [25:22], MBOX0_ADRX_ADR_H, L[03:00], and associated parity bits (receivers are enabled by scan if CPU is present) from MBox 0. ADR3 latches the PA bits in the CPU0 receive latch in consecutive cycles.
2. Parity is checked at the receive latch and an error is reported as an input address parity error. An identical set is also provided in both I/O receive latches, for a total of six sets per ADRX MCA.
3. From the receive latches, the buffered address and parity bits are sent as ADR_LAT_R0n_A_H[09:00], where n = 0 – 2 and corresponds to the three CPU0 command buffers. ADRX receives bits [04:00], which are the four PA bits and one parity bit in the second cycle, and bits [09:05], which are the four PA bits and one parity bit in the first cycle. ADRX sends bits [09:00] to memory, PAMMs, MTCH MCA, and ADRX MCAs.
4. Bits [04:00] carry PA bits [25:22], plus byte parity, and bits [09:05] carry PA bits [05:02], plus byte parity. Only PA bits [25:22] are converted to row/column addresses on ADR0. MMC sends MMC_ADRX_INDEX_H to row and column selectors to select 1 of 16 addresses. That address (next memory address) is sent to the row and column multiplexer controlled by MMC_ADRX_COL_SEL_H to produce first row, then column, addresses, as MUX_RC0, 1, 2, and 3.
5. The next memory address is also sent to the row and column select multiplexer controlled by scan and the MMCX. These are not used by ADR3, but ADR0 uses the selectors to provide MUX_RC9, 10, and 11 (initialized by scan). The MUX_RCn bits are then sent to the row and column output multiplexer, controlled by SEL_HIGH_ORDER_RC_H (set by scan). Scan selects either physically mapped row/column bits or those initialized by scan. ADRX sends the row and column bits to memory.
6. On each ADRX MCA, the row and column output select multiplexer generates ADRX_MACn_ROWCOL_CA_H[03:00], the row address, and ADRX_MACn_ROWCOL_CB_H[03:00], the column address. In ADRX3, lines 0 and 3 are not connected. Lines 1 and 2 are connected, transmitting row and column bits [07] and [08], respectively, to MMU.

2.4.2.2 I/O Address Receive Latches

The I/O address interface is 2 bytes wide, and the complete address is transferred in two cycles. The ADRX MCAs in the JBox send and receive address bits to and from the JDBX and JDAX MCAs, respectively.

The ADRX MCAs have an I/O receive latch. Each latch receives four address bits and one parity bit from ICU. The ADRX MCAs have two hold latches in the I/O receive latch that correspond to the two ICU command buffers. Each hold latch is 8 bits wide and latches the address in two cycles.

2.4.3 JBox-to-Memory Address Interface

The address from the JBox to the memory system is divided into two parts:

- **Multiplexed row and column address** — JBox sends row and column addresses to the MACs in the MMUX. The address field can be up to 12 bits wide [11:00], depending on the dynamic RAM size.
- **Memory port select, segment select, and bank select fields** — JBox sends these fields to the MMCX MCAs.

The memory does not send return addresses to the JBox. The memory returns an index field that allows the JBox to associate the return data with 1 of 16 addresses in the ADRX receive latches.

2.4.3.1 DRAM Addressing

SCU initiates a memory read by setting the load command bit and sending the read command when the memory segment becomes available. ACU initiates a read command by selecting the row address from the address buffers in the tag MCU and then asserting row address strobe to the DRAMs. After asserting the row address strobe, ACU selects the column address from the address buffers and asserts the column address strobe to the DRAMs. Figure 2–38 shows how DRAM is addressed.

The JBox stores DRAM row and column addresses for all memory operations. The MMC provides control signals to the JBox for transmitting the row or column address at the correct time within the DRAM cycle. The following signals are used by the memory for this purpose:

- **Index** — Loaded into the segment command buffer.
- **Column address select** — Selects row or column address.
- **Index parity** — Indicates odd parity on the index and column address select lines.

ADRX MCAs provide the row and column address sent to MMU. Figure 2–39 shows MMC_ADRX_COL_SELECT_H selecting the row and column address bits. Row and column address lines can be either of two kinds of mapping: fixed or programmable. Two PA bits can be initialized to permit the two MMUs to have different DRAM sizes and to allow the memory to be reconfigured.

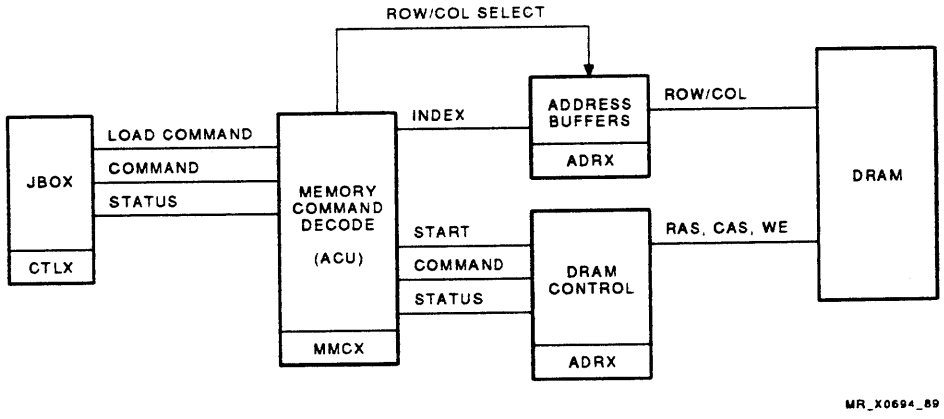


Figure 2–38 DRAM Addressing

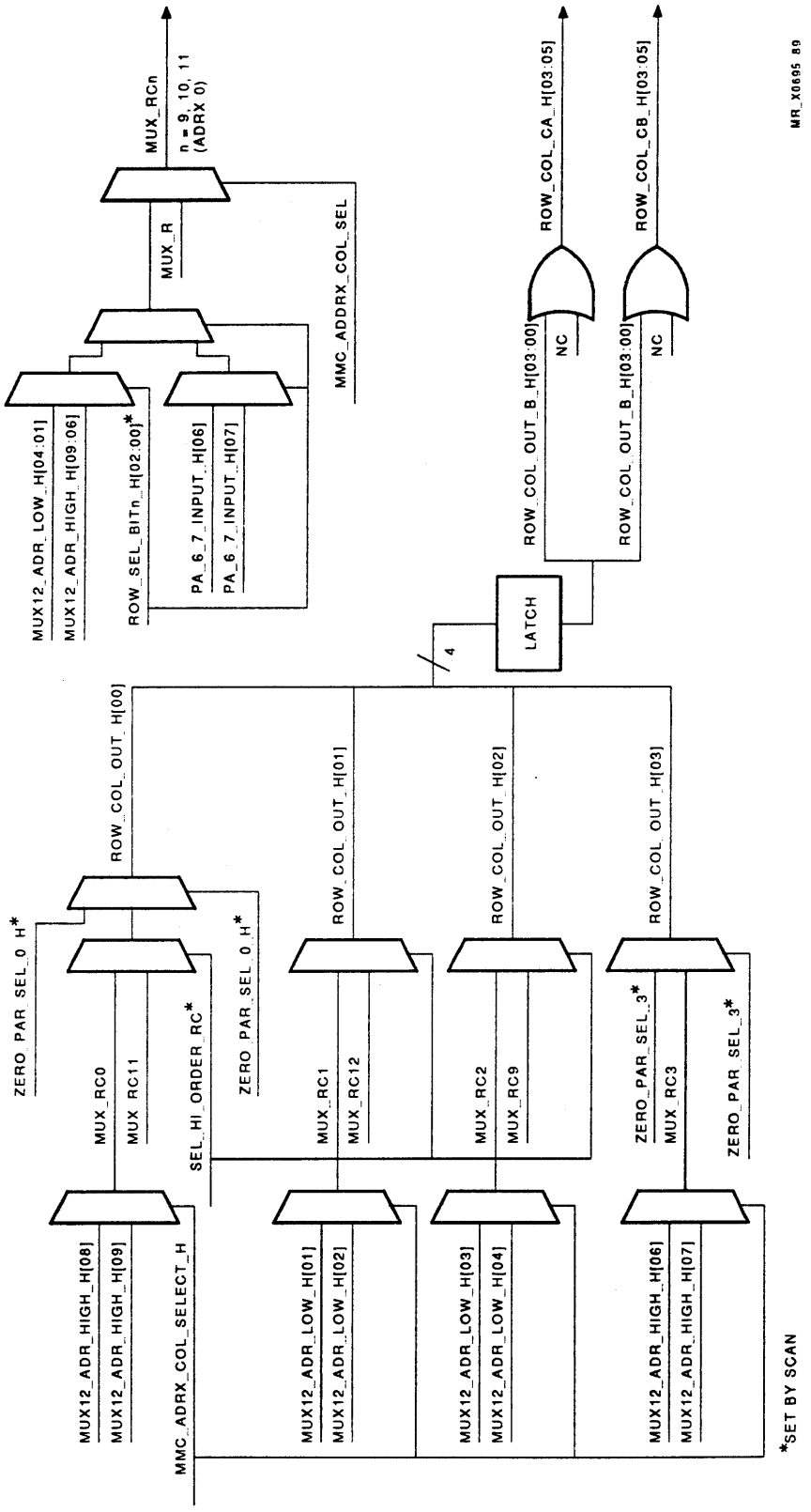


Figure 2–39 ADRX Row and Column Address Selection

Table 2–16 lists the ADRX MCAs and corresponding row and column address bits.

Table 2–17 lists the row and column bit number and the corresponding PA bit number for row and column bits.

Table 2–16 Row and Column Address Bits

MCA	Row/Column
ADR0	11:09
ADR1	04:03, 00
ADR2	06:05, 02:01
ADR3	08:07

Table 2–17 Mapping of Row/Column Lines to Physical Address Bits

Row/Column Bit Number	PA Bit for Row	PA Bit for Column
00	08	09
01	10	11
02	12	13
03	14	15
04	16	17
05	18	19
06	20	21
07	22	23
08	24	25
09	Any of [32:26, 07, 06]	Any of [32:26, 07, 06]
10	Any of [32:26, 07, 06]	Any of [32:26, 07, 06]
11	Any of [32:26, 07, 06]	Any of [32:26, 07, 06]

2.4.3.2 PAMM STRAMs Addressing

ADRX MCAs send ADRX_MPAMM_ADR_H[09:00], which addresses MPAMM. The JBox sends CCU_ADRX_INDEX_PAM_H[03:00] to ADRX and controls the addressing of the PAMMs. ADR0 sends [09:00], and ADR1 sends [01:00]. The ADRX-to-PAMM logic accepts 16 addresses, 3 from each of the 4 CPUs and 2 from each of the 2 ICU receive latches. The CCU index selects which address is to be sent to PAMM.

The ADRX MCAs have two dedicated scan latches for loading the PAMM STRAMs. One is for data, and the other is for the write enable. ADR0 sends MPAMM data and write enable. ADR2 sends NPAMM data and write enable. ADR1 sends IPAMM data and write enable.

CCU_ADRX_INDEX_PAM_H[03:00] controls the addressing of the PAMMs.

I/O mapping PAMM STRAMs decode bits PA [28:19] of the address field and determine if the I/O address points to an adapter on ICU0 or an adapter on ICU1. The PAMM STRAMs decode logic determines whether the address is a JBox or SPU register address.

2.5 JBox Interfaces

The JBox interfaces are as follows:

1. CPU (MBox)
2. ACU
3. ICU
4. SPU
5. MMU
6. ADRX MCAs

Figure 2–40 illustrates the JBox interfaces.

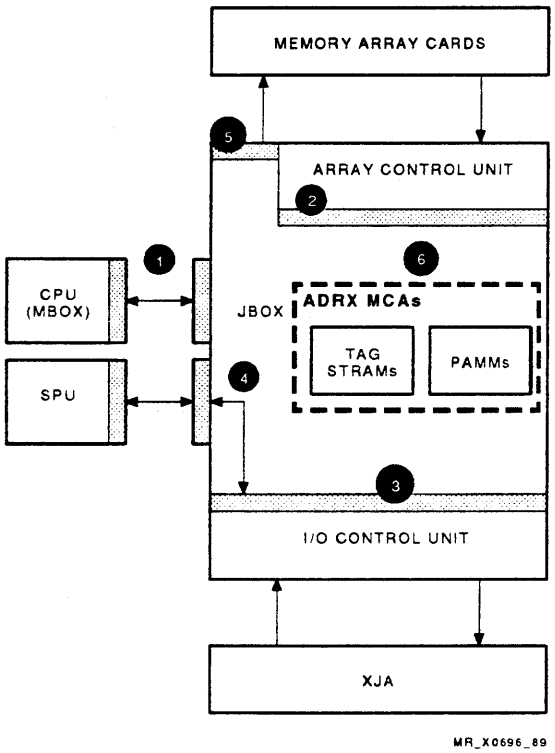


Figure 2–40 JBox Interfaces

2.6 CPU (MBox) Port Interface

From the perspective of a CPU, SCU looks like a memory controller. CPUs implement write back caches, and SCU must ensure cache data consistency. SCU accomplishes this through the use of duplicate cache tag stores for each of the four CPUs. The JBox portion of SCU implements the cache consistency algorithm.

Figure 2–41 shows the MBox-to-JBox command format. Table 2–18 lists the command fields and descriptions.

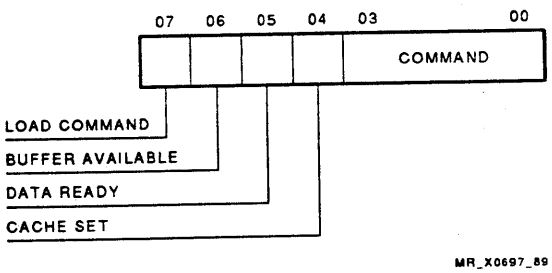


Figure 2–41 MBox-to-JBox Command Format

Table 2–18 MBox-to-JBox Command Field Descriptions

Field	Description
Load command	Notifies the JBox that a command is going to be sent.
Buffer available	Indicates ready to receive a command from the JBox.
Data ready	Indicates that data is in the write back buffer, ready to be transferred to SCU when SCU transmits send data.
Cache set	Indicates which cache set, 0 or 1, is involved.
Command	See Table 2–23.

CTLA latches the load command and sends it to the port state controller. Figure 2–42 shows CTLA receiving command bit fields from CTMV.

CTLB latches the command field and sends the command to the command arbitration logic. Figure 2–43 shows CTLB receiving command bit fields from WBBX. CTLB sends the command information to CTLC and CTLD. CTLC performs a resource check. CTLD forms queue data to send to MICR to determine the microaddress sent to the microcode and MICR command field controlling the CTLA, CTLB, CTLC, and CTLD MCAs for memory operations.

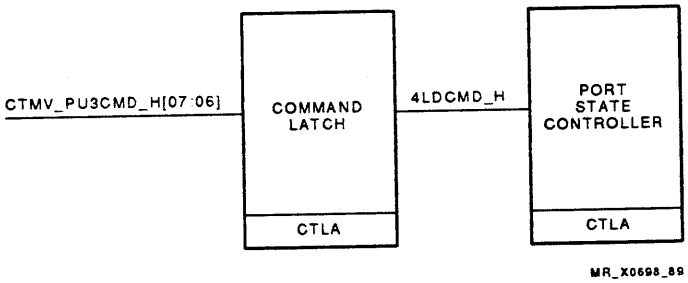


Figure 2–42 CTLA Receiving Command Bits from CTMV

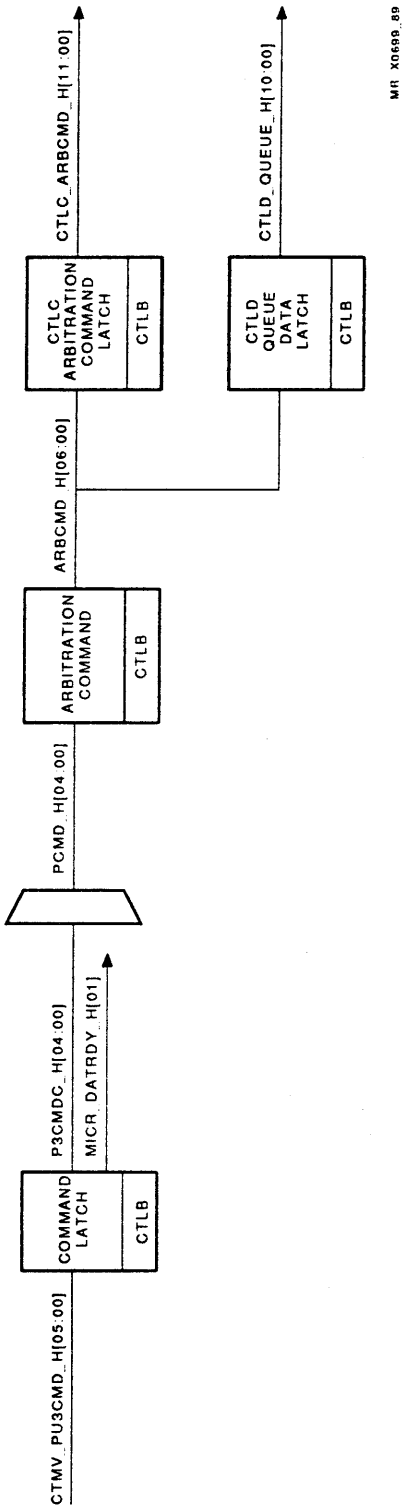


Figure 2–43 CTLB Receiving Command Bits to WBBX

Figure 2–44 shows the JBox-to-MBox command format. Table 2–19 lists the command fields and descriptions.

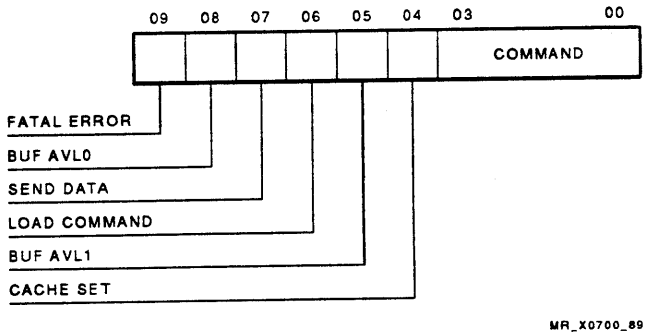


Figure 2–44 JBox-to-MBox Command Format

Table 2–19 JBox-to-MBox Command Field Descriptions

Field	Description
Fatal error	Indicates that JBox has detected a fatal error and has asserted the attention line to SPU.
Buffer available 0	Indicates the number of requests the JBox has retired. Works with buffer available 1 field. The output of the retire logic becomes buffer available. Simultaneously, a port can have three requests retire. This is a count of retired requests.
Send data	Unloads the data in the write back buffer.
Load command	Notifies the MBox that a command has been sent.
Buffer available 1	Indicates the number of requests that the JBox has retired. Works with the buffer available 0 field.
Cache set	Indicates which cache set, 0 or 1, is involved.
Command	See Table 2–21.

CTLA detects a fatal error, sets the fatal error status bit, and notifies CPU. Figure 2–45 shows CTLA generating command bit fields to WBBX. CTLA decodes the buffer available bits based on the retire logic output. CTLA latches the buffer available bits and sends these bits to CPU, indicating the buffers available.

CTLB contains the CPU port command generator. Figure 2–46 shows CTLB generating command bit fields to WBBX. PCMD receives the arbitration index, port command, control signals from CTLC based on the results of resource check, and the command field from the MICR field, which contains the unit, bank, segment, index, and cache set. CTLB generates the command field and the which cache field portion of the command.

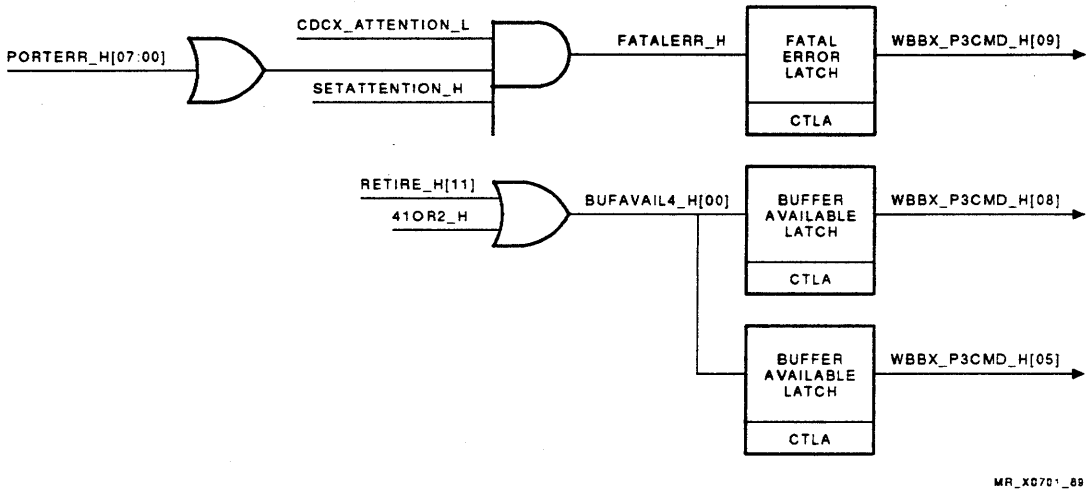


Figure 2–45 CTLA Generating Command Bits to WBBX

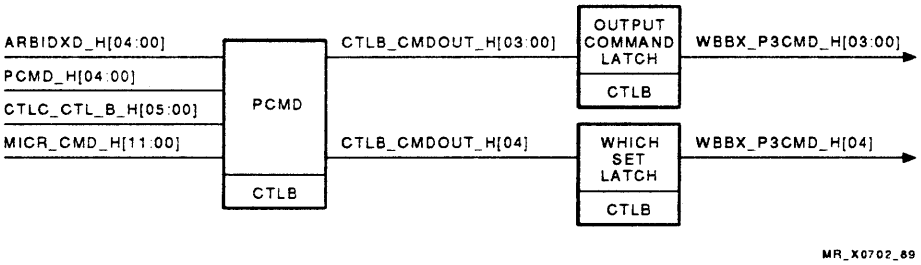


Figure 2–46 CTLB Generating Command Bits to WBBX

CTLC generates the send data field of the command interface. Figure 2–47 shows CTLC generating command bit fields to WBBX. TAKEN_H and bit [00] of the microcode enables the output send data bit to be decoded. The send data bit is latched and sent through the command interface.

CTLC also sends the output load command to CPU. The retire control logic looks at the primary, secondary, and immediate retire status, determines the command for the port, and sends the load command. This notifies CPU that a command field is sent.

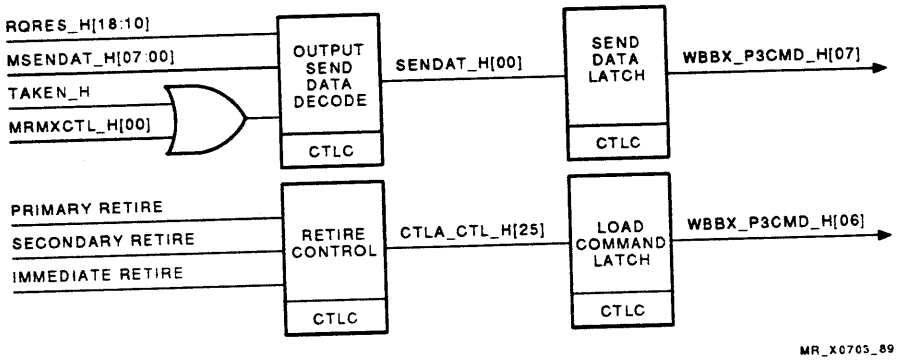


Figure 2–47 CTLC Generating Command Bits to WBBX

2.6.1 JBox Key Signals

Table 2–20 lists the key JBox-to-MBox signals and their descriptions.

Table 2–20 JBox-to-MBox Signals

Name	Description
JBX_MBXx_DAT_H[63:00]	Quadword of data. This data path transmits refill data and I/O register read data.
JBX_MBX_DPAR_H[03:00]	This is the data parity bit per word. The parity bits maintain odd parity and are asserted if an even number of data bits is asserted. The beginning of the data signals are included in two of the four parity bits. These parity bits are valid and checked on every cycle. The parity bits are as follows:

Parity Bit	Signals
JBOX_MBOX_DPAR_H[03]	D[63:48], BOD
JBOX_MBOX_DPAR_H[02]	D[47:32]
JBOX_MBOX_DPAR_H[01]	D[31:16]
JBOX_MBOX_DPAR_H[03]	D[15:00], BOD

JBX_MBXx_CMD_H[03:00]	This contains the JBox command that is sent to the MBox to be decoded.

Table 2–20 (Cont.) JBox-to-MBox Signals

Name	Description
JBX_MBXx_CPAR_H	This parity bit is valid, is checked on every cycle, and reflects odd parity over the command and control lines. The parity bit checks parity across the following signals: JBOX_MBOX_CMD_H[03:00] JBOX_MBOX_WCHSET_H[00] JBOX_MBOX_LD_CMD_H JBOX_MBOX_CMD_BUF_AVAIL_H JBOX_MBOX_SENDAT_H JBOX_MBOX_SPARE_H JBOX_MBOX_FATAL_ERROR_H
JBX_MBXx_WCHSET_H[00]	This bit indicates which set of the two sets in a given CPU cache is involved in the command. If equal to 0, set 0 and if equal to 1, set 1.
JBX_MBXx_LDCMD_H	The JBox sends a valid command and first half of the address to the MBox. The JBox is allowed to assert this signal if a suitable MBox buffer is available. The JBox sets a flag when it issues LDCMD to the MBox, and this prevents the JBox from sending another LDCMD, since the MBox has only one command and address buffer.
JBX_MBXx_CMD_AVAIL_H[01:00]	The MBox receives this signal, indicating the status of the three command and address buffers in the JBox that are assigned to this particular MBox.
JBX_MBXx_SENDAT_H	JBox forces the MBox to start sending data from its write back buffer or, in certain cases, directly from the cache.
JBX_MBXx_ADR_H[15:00]	These are double-cycled, enabling 32 address bits to be sent with each command. The first half of the address is sent in the same cycle as the command, and the second half of the address is sent in the following cycle. Mapping is shown in Table 2–15.
JBX_MBXx_APAR_H[03:00]	These are parity bits across the 16 address bits. The parity bits check odd parity across the following signals:

Parity Bit	Signal
JBOX_MBOX_APAR_H[03]	JBOX_MBOX_ADR_H[15:12]
JBOX_MBOX_APAR_H[02]	JBOX_MBOX_ADR_H[12:08]
JBOX_MBOX_APAR_H[01]	JBOX_MBOX_ADR_H[07:04]
JBOX_MBOX_APAR_H[03]	JBOX_MBOX_ADR_H[03:00]

JBX_MBX_BOD_A, B, C_H	This is asserted during the first of eight refill cycles. Eight bytes of data are transferred during each cycle, for a total of 64 bytes.
JBX_MBXx_FATAL_ERROR_H	This is asserted when the JBox detects a fatal error and sends the attention line to SPU.

2.6.2 JBox Commands

Table 2–21 lists the JBox-to-MBox commands and their descriptions.

NOTE
Written, read, and invalidate qualifiers to get data and return data commands refer to the ending cache status in the MBox and JBox tag stores.

Table 2–21 JBox-to-MBox Commands

Code	Name	Description
0	Get data written	Not used.
1	Get data read	SCU sends this command to get data from the MBox that is needed by another CPU or I/O. The JBox and MBox mark the status of the cache block as read.
2	Get data invalidate	SCU sends this command to get data from the MBox that is needed by another CPU or I/O. The JBox and MBox mark the cache block as invalid.
3	Return data read	SCU sends this command to return data as a result of a read refill command. SCU and the MBox mark the cache block as read.
4	Return data written	SCU sends this command to return data as a result of a write refill command. SCU and the MBox mark the cache block as written full.
5	OK to write	SCU sends this command in response to an aligned longword write update command or write refill and gives the MBox permission to write into the cache block (the MBox already has the cache block). SCU and the MBox mark the cache block as written partial or written full, respectively.
6	Invalidate read block	SCU sends this command to invalidate the cache block and change the status from read to invalid. For example, if the MBox has a cache block that has read status and another CPU needs to write to the same cache block, SCU sends this command to invalidate the cache block.
7	Return I/O register data	SCU sends this command and the register data in response to a read I/O register command.
8	Return read error status	SCU sends this command to send read error status to the MBox.
9	Lock acknowledge	SCU sends this command to acknowledge an MBox lock command in which the MBox already has the data but now needs to lock the cache block.
A	Memory read nonexistent memory	SCU sends this command to notify the MBox that the address the MBox sent with the command does not exist.
B	I/O read nonexistent memory	SCU sends this command to notify the MBox that the address the MBox sent with an I/O register command does not exist.
C	Lock denied	SCU sends this command in response to a lock command to indicate that the address sent by the MBox is already locked by another port.

Table 2–21 (Cont.) JBox-to-MBox Commands

Code	Name	Description
D	Invalidate written block	SCU sends the command to invalidate a cache block that has written full status.
E	Unused	–
F	Unused	–

2.6.3 MBox Key Signals

Table 2–22 lists the key MBox-to-JBox signals and their descriptions.

Table 2–22 MBox-to-JBox Signals

Name	Description
MBXx_JBX_DATA_H[63:00]	Quadword of data. This data path transmits write back data and I/O register write data.
MBXx_JBX_LWMSK_H[01:00]	This contains longword mask information for MBXx_JBX_DAT_H[63:00]. The longword mask bit was originally the valid bit in the cache tag store in the CPU. Written blocks can be partially valid. The MBox alerts the JBox as to which longwords should be written back to main memory. If the MBox is doing a write to an I/O register, the MBox must assert MBX_JBX_LWMSK_H[00]. [00] is the mask for [63:32] and [01] is the mask for [31:00].
MBXx_JBX_DPAR_H[03:00]	This is the data parity bit per word. The parity bits are odd parity and are asserted if an even number of data bits is asserted. The beginning of the data signals is also included in two of the four parity bits. These parity bits are valid and checked on every cycle. The parity bits check odd parity across the following signals:

Parity Bit	Signals
MBOX_JBOX_DPAR_H[03]	D[63:48], BOD, LWMSK [01]
MBOX_JBOX_DPAR_H[02]	D[47:32]
MBOX_JBOX_DPAR_H[01]	D[31:16], LWMSK [00]
MBOX_JBOX_DPAR_H[03]	D[15:00], BOD

MBXx_JBX_CMD_H[03:00]	This contains the MBox command that is sent to the MBox to be decoded. The commands are listed and described in Table 2–23.
MBXx_JBX_CPAR_H	This parity bit is valid and is checked on every cycle. It reflects odd parity over the the command and control lines. The parity bits check odd parity across the following signals: MBOX_JBOX_CMD_H[03:00] MBOX_JBOX_WCHSET_H[00] MBOX_JBOX_LD_CMD_H MBOX_JBOX_DATRDY_H

Table 2–22 (Cont.) MBox-to-JBox Signals

Name	Description
MBXx_JBX_WCHSET_ H[00]	This bit indicates which set of the two sets in a given CPU cache is involved with this command. If equal to 0, set 0 and if equal to 1, set 1.
MBXx_JBX_LDCMD_H	The MBox sends a valid command and the first half of the address to the JBox. The MBox is allowed to assert this signal if an MBox buffer is available. The JBox sets a flag when it issues LDCMD to the MBox, and this prevents the JBox from sending another LDCMD, since the MBox has only one command and address buffer.
MBXx_JBX_CMD_AVAIL_ H[01:00]	The MBox receives this information, indicating the status of the three command and address buffers in the JBox that are assigned to this particular MBox.
MBXx_JBX_ADR_H[15:00]	These are double-cycled, enabling 32 address bits to be sent with each command. The first half of the address is sent in the same cycle as the command, and the second half of the address is sent in the following cycle. Mapping is shown in Table 2–15.
MBXx_JBX_APAR_H[03:00]	These are parity bits across the 16 address bits. The parity bits check odd parity across the following signals:

Parity Bit	Signal
MBOX_JBOX_APAR_H[03]	MBOX_JBOX_ADR_H[15:12]
MBOX_JBOX_APAR_H[02]	MBOX_JBOX_ADR_H[12:08]
MBOX_JBOX_APAR_H[01]	MBOX_JBOX_ADR_H[07:04]
MBOX_JBOX_APAR_H[03]	MBOX_JBOX_ADR_H[03:00]

MBX_JBX_BOD_A, B_H	This signal deals with write back data. The MBox sends data to the JBox in 64-byte packets. This is asserted during the first of eight write back cycles.
MBXx_JBX_DATRDY_H	MBox asserts this when it has the data ready in response to a get data command from the JBox. SCU does not reserve the resources required for the get data write back until receipt of the data ready signal, which implies that the MBox is now ready to send the data.

2.6.4 MBox Commands

Table 2–23 lists the commands that the MBox can send to the JBox.

NOTE
Written, read, and invalidate qualifiers to the get data and return data commands refer to the ending cache status in the MBox and JBox tag stores.

Table 2–23 MBox-to-JBox Commands

Name	Description
Read refill	The MBox sends this command when a read miss has occurred. After SCU returns the data to the MBox, the MBox has a read-only block. If the MBox subsequently wants to write to this block, it must obtain permission by using the write refill or longword write update commands.
Write refill	There are two reasons for the MBox to issue this command: a block miss on a write or a block hit, write miss. In the first case, the MBox issues the write refill command, and SCU responds with refill data. The data is written after the fill is completed. The MBox now has write ownership of this block and can write to it at any time. In the second case, the MBox already has a read-only copy of the block but wants to write to it. The MBox issues the write refill command, but when SCU looks up the address of the cache block in the consistency tag STRAMs and discovers that the MBox status is read, SCU does not send refill data. SCU returns the OK to write response, which gives the MBox permission to proceed with the write.
Read refill linked with a write back	The MBox issues this command for the same reason it issues read refill, but in this case, the refill includes a write back. That is, both sets have written data but neither set matches the requested address. The JBox associates the write back command the MBox sends (which follows) with the read refill command. The memory segment controllers contain logic that coordinates the read refill and write back operations.
Write refill linked with a write back	This command is issued by the MBox when it block misses on a write because a write back is associated with the refill. SCU assumes that the next write back received from this MBox is the associated write back.
Write refill lock	This is the command that the MBox issues when it wants to do the read part of an interlock instruction. This is true regardless of the state of the cache block in the MBox. SCU does the tag lookup for this command, checking to see if the block is locked by another CPU. If it is, it returns lock denied, and the MBox tries again until the other CPU unlocks it.
Write refill unlock	If the MBox has lost write ownership of the cache block it issues a write refill unlock. Loss of write ownership is due to a get data or invalidate command from SCU.
Write refill linked lock	This command is the same as write refill lock but must also do a write back to free a block for the requested refill.
Write back	The MBox sends this command to notify SCU that written and valid data in the cache block must be written back to main memory. The JBox marks the cache block as invalid.
Longword write update	The MBox sends this command to notify SCU that a new cache block with only one longword with valid and written status has been created.

Table 2–23 (Cont.) MBox-to-JBox Commands

Name	Description
Read I/O register	The MBox sends this command to read the contents of an I/O register.
Write I/O register	The MBox sends this command to write data into an I/O register.
Longword write update linked	The MBox sends this command to link the longword write update with a write back command. The longword write update notifies SCU that a new cache block with only one longword with valid and written status has been created. The cache block originally had valid and written data that must be written back to main memory. The MBox sends this command to notify SCU that an aligned longword (in the cache block that has read status) has been modified and that this command is linked with a write back command.
Invalidate	The MBox sends this command to SCU during a cache sweep. When the MBox detects a cache block that is valid with read status, the MBox invalidates the block and notifies SCU.

2.7 ACU Port Interface

Each ACU is part of a separate memory subsystem and has its own JBox interface. Figure 2–48 shows the JBox-to-ACU interface.

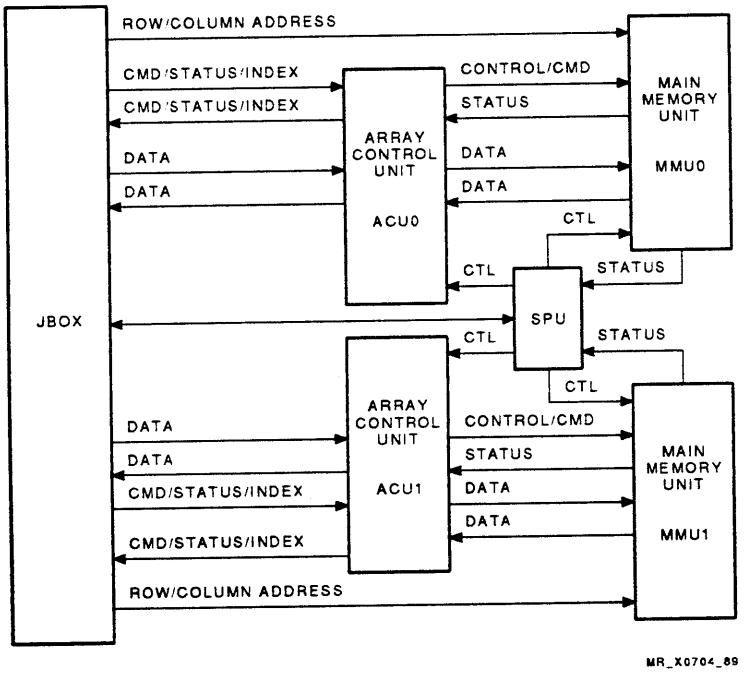


Figure 2–48 JBox-to-ACU Interface

A JBox-to-ACU interface permits the JBox to accept memory requests from CPUs or I/O and to send the requests to the memory segment controllers.

The ACU provides all communication between the JBox and main memory. The ACU performs the following functions:

- Accepts memory commands.
- Processes the commands to determine the availability of memory segments addressed by the commands.
- Sends commands to DRAM control.
- Determines the direction of data movement.

The ACU communicates with the JBox when any of the following conditions exist:

- A read request was made and the data is ready to send.
- An error was detected during the transfer of read data.
- An error was detected during the transfer of write data.
- A command buffer is available.

Figure 2–49 shows the DBX-to-CCU (memory-to-JBox) memory command format.

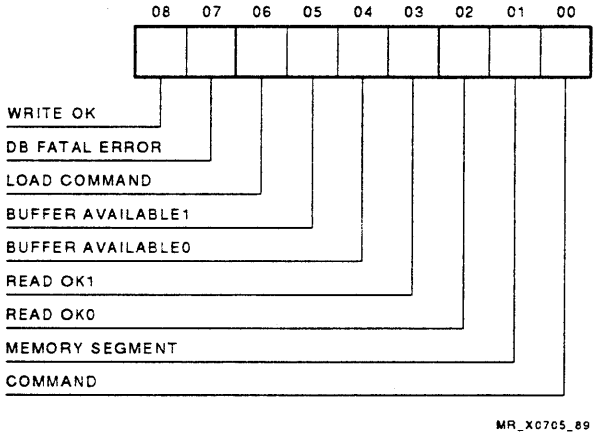


Figure 2–49 Memory-to-JBox Command Format

Table 2–24 lists the DBX-to-CCU command field descriptions.

Table 2–24 DBX-to-CCU Command Field Descriptions

Field	Description
Write OK	Not used.
DB fatal error	Indicates that DBX has detected a fatal error.
Load command	Notifies the JBox that a command is coming. JBox uses this to load command and segment information. This signal is asserted when the memory subsystem is sending a valid command.
Buffer available 1	Indicates that memory segment 1 is available.
Buffer available 0	Indicates that memory segment 0 is available.
Read OK 0	Implies that no ECC errors occurred during transfer of read data through the MDP for memory segment 0.
Read OK 1	Implies that no ECC errors occurred during transfer of read data through the MDP MCAs for memory segment 1.
Memory segment	Identifies the segment number.
Command	Uses a 1-bit field. If 0, means return read data, and if 1, means error report.

CTLA latches the load command and controls the command buffers for the memory command interface. Figure 2–50 shows the CTLA receiving the MMCX command bit fields.

CTLA latches the port ready field and sends it to the MICR port ready latch, which is then sent to the MICR. CTLA decodes the fatal error status sent by the memory.

CTLB latches and decodes the write OK. Figure 2–51 shows CTLB receiving the MMCX command bit fields. CTLB latches the command and segment fields of the command interface, sending them to command arbitration. The arbitration index selects the command.

CTLC latches OK to read status from MMC1 and MMC0. Figure 2–52 shows CTLC receiving the MMCX command bit fields. This status is sent to the M0, M1, M2, and M3 segment controllers.

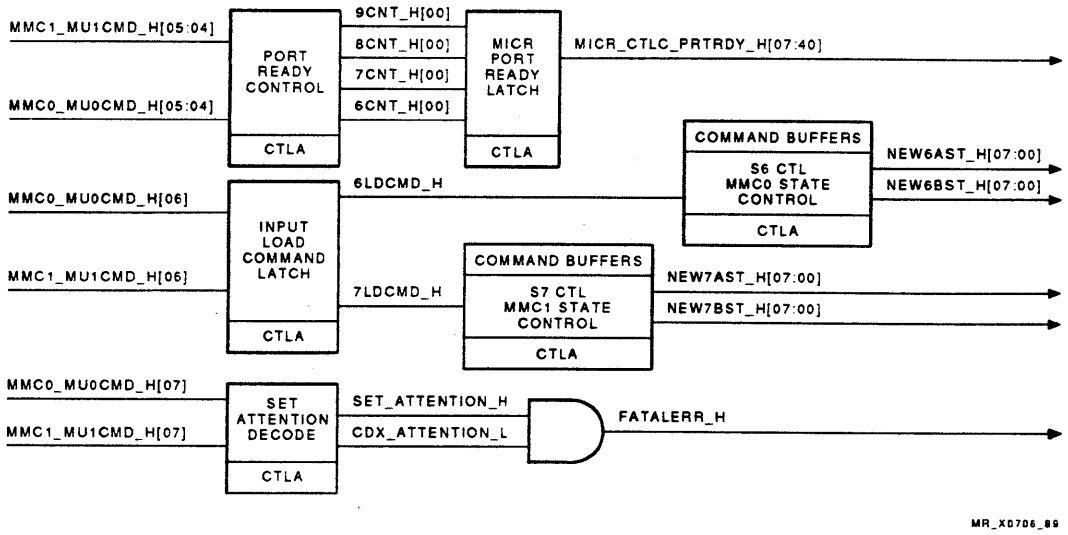


Figure 2–50 CTLA Receiving the MMCX Command Bits

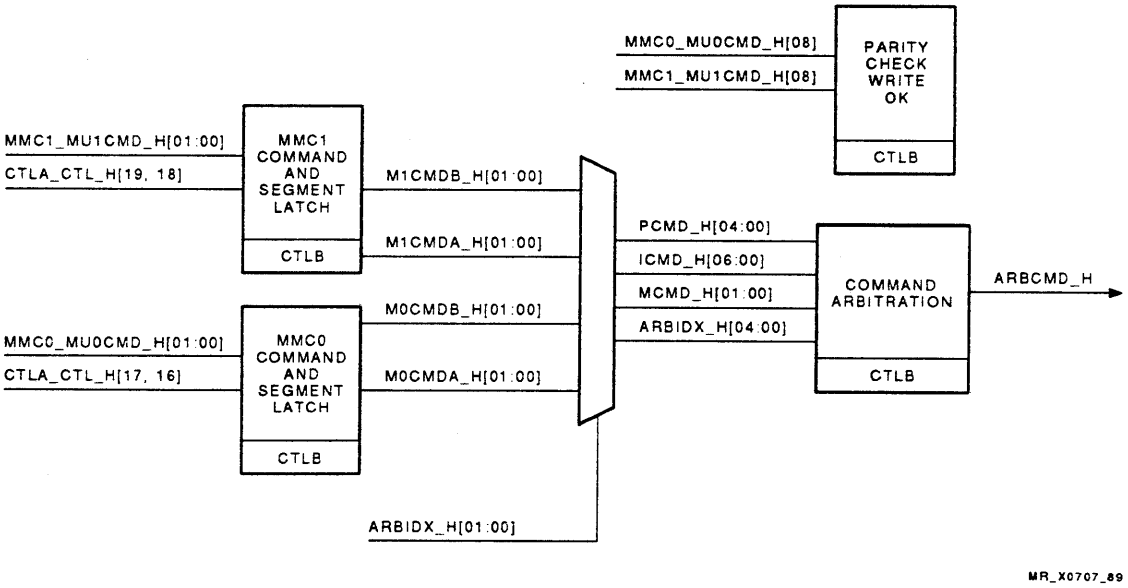


Figure 2–51 CTLB Receiving the MMCX Command Bits

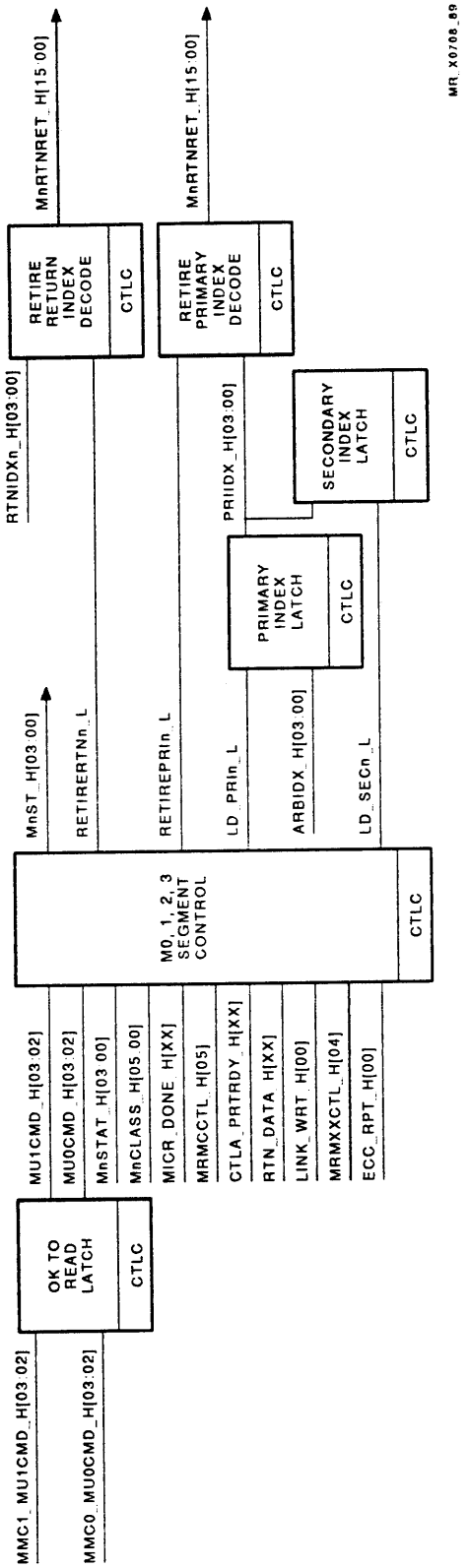


Figure 2–52 CTLC Receiving the MMCX Command Bits

Figure 2–53 shows the CCU-to-DBX (JBox-to-memory) memory command format. Table 2–25 lists the command fields and descriptions.

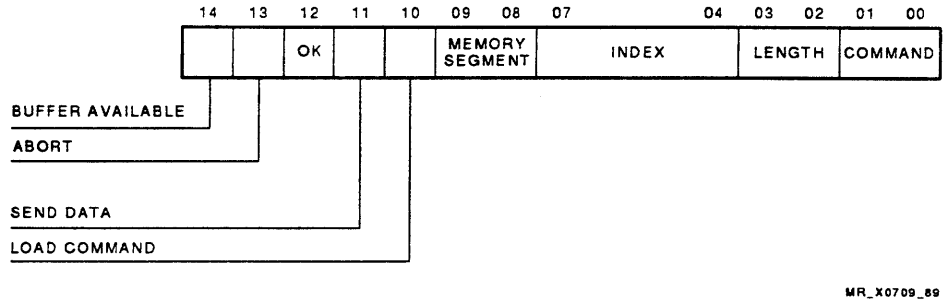


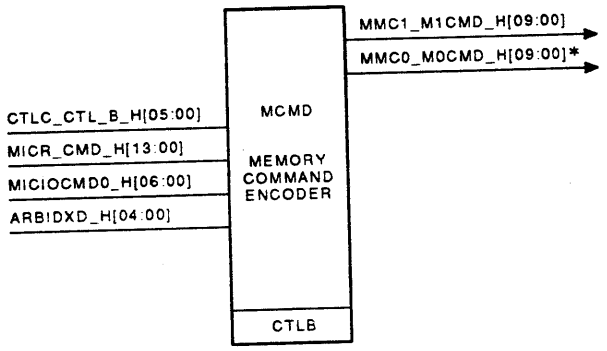
Figure 2–53 JBox-to-Memory Command Format

Table 2–25 CCU-to-DBX Command Field Descriptions

Field	Description
Buffer available	Indicates that a request has been retired and the buffer is available. The JBox can accept another command.
Abort	Indicates that the memory operation has been aborted. Cycle status bit specifies if a request should be canceled. When the memory cycle is started, the JBox determines where the latest copy of the data is located. If the latest copy of data is in another CPU cache (written cache block), the JBox sends abort status to the memory controller.
OK	Indicates that the memory operation should complete. Cycle status bit specifies if a request should continue. When the memory cycle is started, the JBox determines if the latest copy of the data is in main memory. If the latest copy of data is in main memory, the JBox sends OK status to the memory controller.
Send data	Notifies memory to send the data to the destination. Memory can transmit read data.
Load command	Indicates whether the command bits are valid. Notifies ACU that a command is coming. The JBox sets a flag to prevent the JBox from sending another load command. This flag is cleared when the memory system sends buffer available.
Memory segment	Specifies the memory segment involved with the memory request. Indicates segment and bank number.
Index	Identifies the request and the corresponding address associated with the request. The index is used to locate the address in the ADRX MCAs.
Length	Specifies the length of the data involved as the number of quadwords in the transfer. 0 = 4 quadwords, 1 = 8 quadwords, 2 = 1 quadword, and 3 = 2 quadwords.
Command	Specifies the ACU command. Command can be read, write, write read, or write pass.

CTLB generates the command sent to MMCX. Figure 2–54 shows CTLB generating the MMCX command bit fields. CTLB decodes the control lines sent by the CTLC, MICR, and arbitration index, before encoding the command.

CTLC decodes the retire logic and sends the output load command. Figure 2–55 shows CTLC generating the MMCX command bit fields. CTLC receives the microcode control that determines the status. The status of abort or OK is latched and sent to MMCX. CTLC decodes the requested resources and memory send data information to determine the send data field of the command interface to MMCX.



*NOTE

BIT	INPUTS USED
[01:00]	MICR
[03:02]	I/O CMD LENGTH FIELD OR MICR
[07:04]	MICR OR ARBIDX
[09:08]	MICR OR CTLC_CTLB_H[05:00]

Figure 2–54 CTLB Generating the MMCX Command Bits

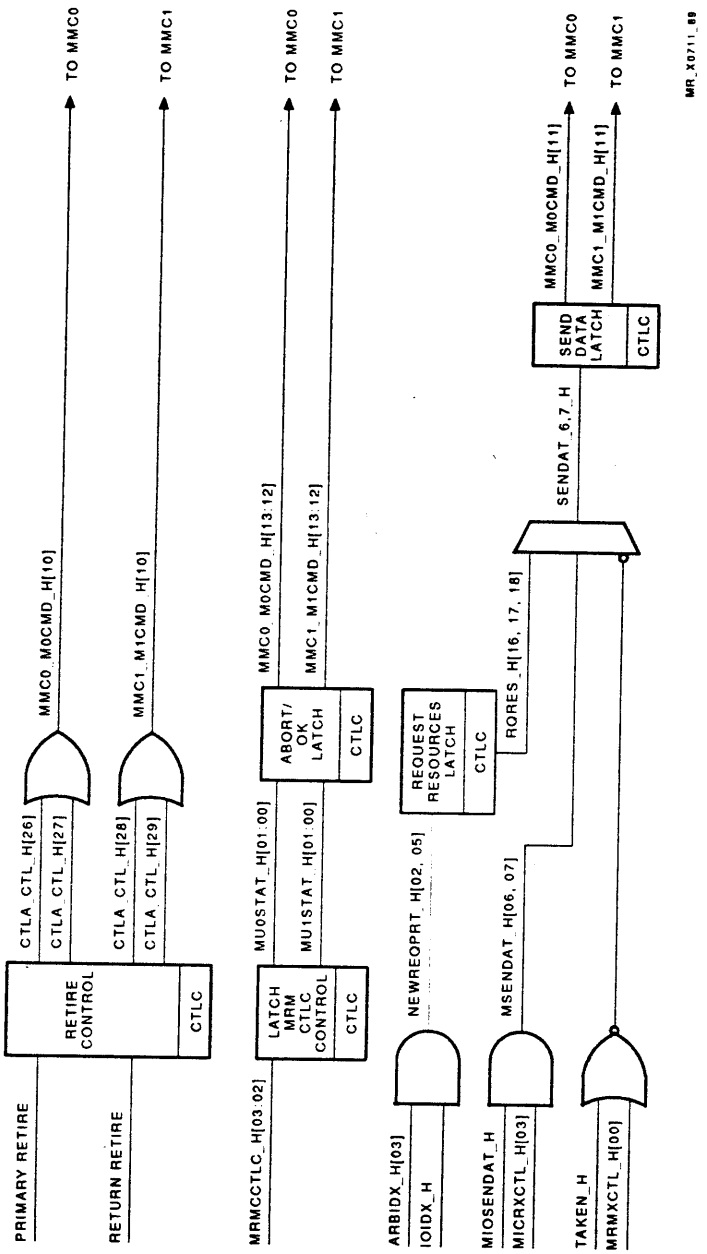


Figure 2–55 CTLC Generating the MMCX Command Bits

2.7.1 JBox Key Signals

Table 2–26 lists the key signals between the JBox and ACU.

Table 2–26 JBox-to-ACU Signals

Signal	Description
CTLX_MMCX_M0CMD_H[14:00]	Contains the command.
CTLA_MMCX_M0CMD_PAR_H	Contains the parity for the command.

2.7.2 JBox Commands

See Chapter 5.

2.7.3 ACU Key Signals

Table 2–27 lists the key signals between the ACU and JBox.

Table 2–27 ACU-to-JBox Signals

Signal	Description
MMCX_CTLX_MU0CMD_H[08:00]	Contains the command.
MMCX_CTLA_MU0CMD_PAR_H	Contains the parity for the command.

2.7.4 ACU Commands

See Chapter 5.

2.7.5 Read Refill — Example

To complete a read refill request, the JBox sends commands to and receives commands from the following:

CPU (MBox)
ACU

The following steps summarize a CPU read refill request:

1. The CPU (MBox) sends the load, cache set, and read refill commands over the MBox-to-JBox command interface.
2. The JBox starts the state machine for the next available command buffer and, after arbitration, generates an index that identifies the port (CPU) and command buffer (A, B, or C).
3. CTLA tells CTLB to latch and store the CPU command. CTLA tells ADRX to latch and hold the CPU physical address.
4. ADRX addresses MPAMM using the CPU physical address bits. The MPAMM output is sent to CTLA, CTLB, CTLC, and CTLD and is decoded into memory unit, segment, and bank.
5. CTLC checks for available resources for the JBox-to-ACU command and then for the JBox-to-MBox data transfer (for write).

6. The JBox sends the load, index, and memory read commands to ACU (MMCX).
7. CTLD forms the MICR queue data (index, memory unit, and command). The MICR MCA uses the MTCH status and queue data to form the microaddress sent to the microcode for the location corresponding to the CPU request (read refill).
8. ADRX sends PA bits to MTCH, which addresses the tag STRAMs. MTCH receives the address stored in the tag STRAMs and compares it to the CPU physical address. MTCH sends the results of the match to MICR. The status bits of the tag STRAMs are sent to MICR.
9. MICR command bits and microcode fields determine memory operation and status. These bits control the CTLA, CTLB, CTLC, and CTLD MCAs and memory segment controllers.
10. MMCX sends start, command, and status to DRAM control and the index to ADRX. DRAM control sends row address select, column address select, and write enables to the memory arrays. The index selects the address buffer that corresponds to the CPU command. The output of the address buffer addresses the memory array. MMCX sends write OK, load command, and return data read to the JBox.
11. The memory segment controller receives MICR_DONE_H and releases the memory segment for another request.
12. The JBox sends the load, which cache set, and return data read commands to CPU. The JBox sends the beginning of data bit followed by eight quadwords of data in eight cycles.

2.8 ICU Port Interface

Each ICU is part of a separate I/O subsystem and has its own JBox interface. Figure 2–56 shows the JBox-to-ICU interface.

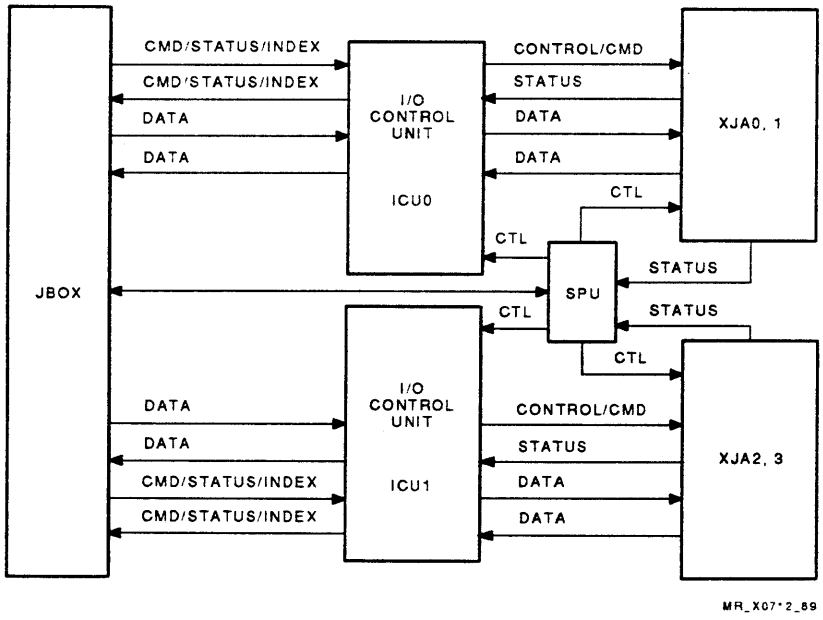


Figure 2–56 JBox-to-ICU Interface

The ICU port serves as an interface to the XJAs (over the JBox XMI data interface [JXDI] cable) and SPU, implementing the central system interrupt arbiter.

XJA and ICU provide an information path between the JBox and I/O devices. The SCU can have up to two ICUs, each capable of handling up to two XMI buses. To communicate with the XMI bus, a JXDI 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/s in both directions. Address, command, and data are time multiplexed onto these wires.

JDCX MCA controls the operation of JDAX and JDBX and coordinates the handshaking signals to and from JXDI and the JBox.

Figure 2–57 shows the ICU-to-JBox command format. Table 2–28 lists the command fields and descriptions.

CTLA checks fatal error, load command, and buffer available fields of the command interface. Figure 2–58 shows CTLA receiving the ICU command bit fields.

CTLB latches the ID, length, and command. Figure 2–59 shows CTLB receiving the ICU command bit fields. The ID is saved until the ID is sent with the return data. The length field is decoded, determining the data switch control for transferring the data through the data switch. The command is sent to command arbitration, where it determines the queue data sent to MICR and the command information sent to CTLC for the resource check.

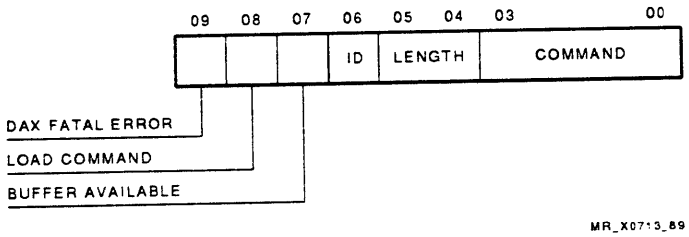


Figure 2–57 ICU-to-JBox Command Format

Table 2–28 DAX-to-CCU (ICU-to-JBox) Command Field Descriptions

Field	Description
DAX fatal error	DAX detected a fatal error.
Load command	DAX sent a command. ICU is sending a valid command.
Buffer available	The ICU command buffer has been unloaded and is available for another command.
ID	This field identifies the origin of the request.
Length	This field specifies the length of data as quadword or octaword.
Command	See Table 6–9.

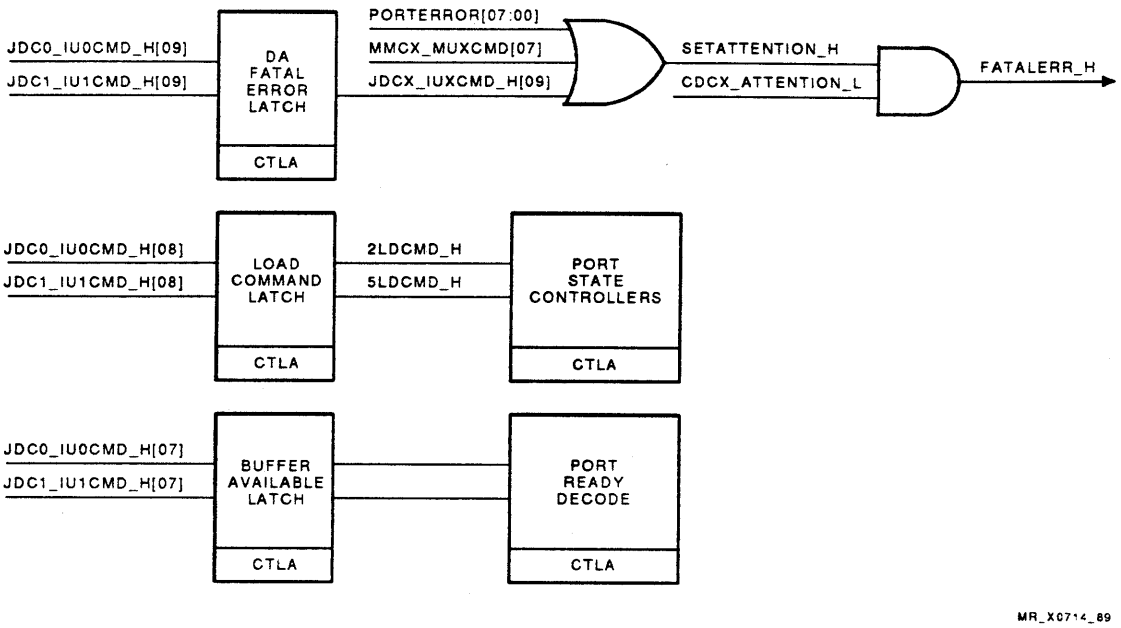


Figure 2–58 CTLA Receiving the ICU Command Bits

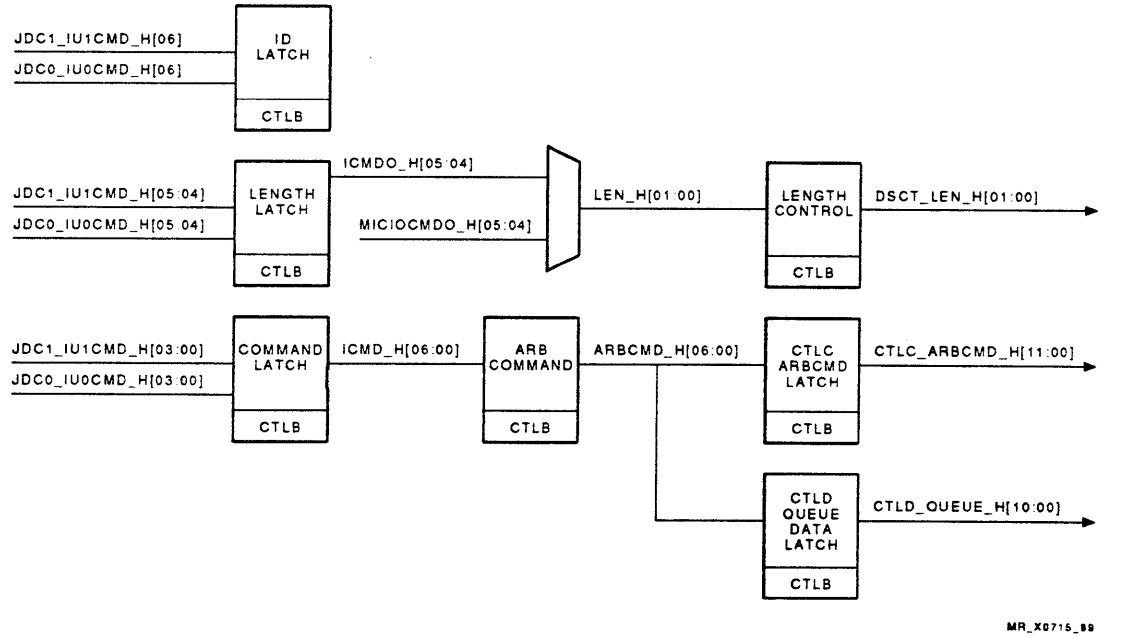


Figure 2–59 CTLB Receiving the ICU Command Bits

Figure 2–60 shows the JBox-to-ICU command format. Table 2–29 lists the JBox-to-ICU command fields and descriptions.

CTLA decodes the retire information sent from CTLC and the buffer available. Figure 2–61 shows the CTLA generating the ICU command bit fields.

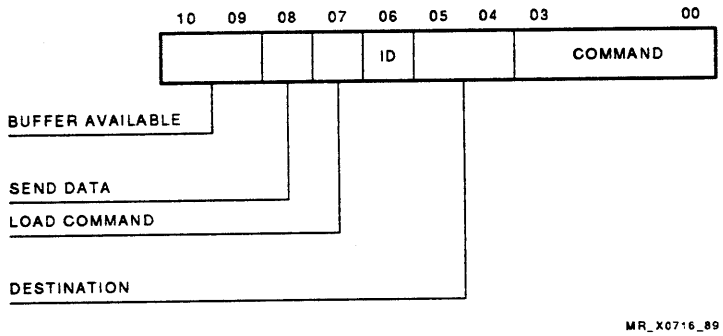


Figure 2–60 JBox-to-ICU Command Format

Table 2–29 CCU-to-DAX (JBox-to-ICU) Command Field Descriptions

Field	Description
Buffer available	Command has been retired and the command buffer is now available.
Send data	This field signals the ICU buffer to unload.
Load command	This field notifies ICU that a command is coming. The JBox asserts this signal when an ICU buffer is available and can accept a command that the JBox wants to send either to SPU or XJA. The JBox sets a flag when it sends a load command to prevent the JBox from sending another load command. This flag is cleared when ICU sends buffer available.
ID	ICU uses this field to track the original request and to identify the requester.
Destination	This field identifies the I/O device that is to receive the data.
Command	See Table 6–8.

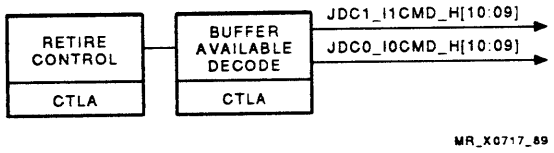


Figure 2–61 CTLA Generating the ICU Command Bits

CTLB generates the command field of the command interface. Figure 2–62 shows CTLB generating the ICU command bit fields. CTLB receives command arbitration, arbitration index, control from the CTLC, and MICR determines the ICU command.

CTLC sends the output load and send data commands. Figure 2–63 shows CTLC generating the ICU command bit fields.

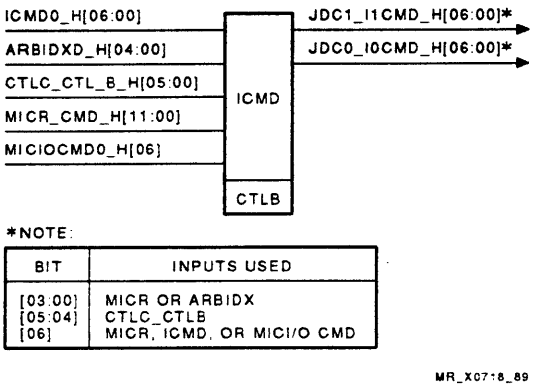


Figure 2–62 CTLB Generating the ICU Command Bits

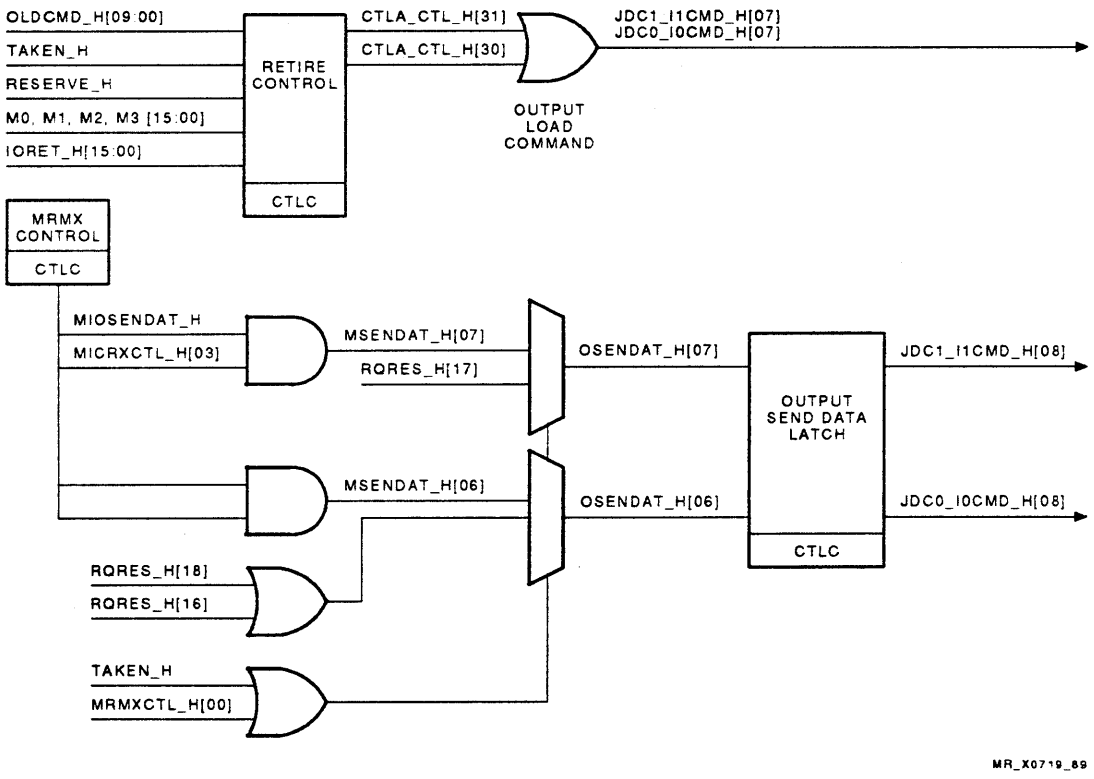


Figure 2–63 CTLC Generating the MMCX Command Bits

2.8.1 JBox Key Signals

Table 2–30 lists the key JBox-to-ICU signals.

Table 2–30 JBox-to-ICU Signals

Signal	Description
CCU_JDCX_CMD_ H[03:00]	Contains the command.
CCU_JDCX_IOSEL_ H[01:00]	Contains the I/O device selected.
CCU_JDCX_ID_H	Contains the ID that XJA uses to track the original request.
CCU_JDCX_BUF_ AVAIL_H[01:00]	As requests are retired, SCU sends the number of the buffer available.
CCU_JDCX_ SENDAT_H	JBox sends this to begin unloading the output buffer in which the data is stored.
CCU_JDCX_LDCMD	JBox wants to send a command to ICU.
CCU_JDCX_CTL_ PAR	A parity bit is sent with the control lines.

2.8.2 JBox Commands

See Chapter 6.

2.8.3 ICU Key Signals

Table 2–31 lists the key ICU-to-JBox signals.

Table 2–31 ICU-to-JBox Signals

Signal	Description
JDCX_CCU_CMD_ H[03:00]	Contains the command.
JDCX_CCU_DAX_ FATAL_L	Indicates that a fatal error has occurred in either the JDAX, JDBX, or JDCX MCAs.
JDCX_CCU_ LENGTH_H[01:00]	Indicates the length of the data.
JDCX_CCU_ID_H	Identifies one of two possible ID fields latched in the JDCX MCA. XJA sends an ID field with each packet that the JDCX MCA latches. The JDCX MCA sends a single bit (JDCX_CCU_ID_H) to CCU. 0 = one ID field, and 1 = the other ID field. When CCU sends a return command to the JDCX MCA, it sends the JDCX_CCU_ID_H to identify which ID field should be included in the packet. XJA uses the ID field to identify the original requester.
JDCX_CCU_BUF_ AVAIL_H	Indicates that the command buffer has been unloaded.
JDCX_CCU_CTL_ PAR_H	Contains the parity bit for the JDCX-to-CCU control lines.

2.8.4 ICU Commands

See Chapter 6.

2.9 SPU Port Interface

SPU's primary means of communication with the CPU is through the logical interface that connects SCU to CPU. These paths are unidirectional, byte-wide paths capable of transferring one byte every 128 ns. This provides about 3.5-Mbyte throughput for quadword transfers.

SPU is MicroVAX system-driven, and BI interface-based, providing service and maintenance support for the computer system. SPU serves as the operator console. SPU monitors the system and tests and diagnoses hardware faults.

ICU interfaces with the XJAs (over the JXDI cable) and SPU, implementing the central system interrupt arbiter.

SCU connects the service processor (console) to the rest of the system. Communication uses SCU registers. These registers have I/O space addresses that configure memory and I/O. They also contain status about interrupts and exceptions.

CTLD receives inputs from SPU, JDC0, and JDBX and sends outputs to JDAX, JDC0, and SPU. CTLD receives SPU data and sends the data to the JDAX ICU input buffers. CTLD receives SPU handshaking signals and sends this to JDCX for control decode.

Figure 2–64 shows the SPU-to-CCU handshaking format. Figure 2–65 shows the CCU-to-SPU handshaking format.

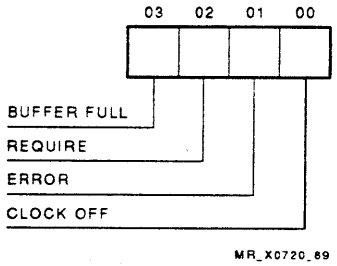


Figure 2–64 SPU-to-CCU Handshaking Format

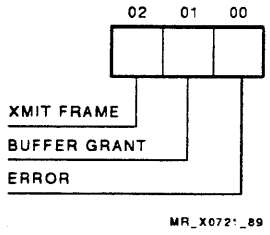


Figure 2–65 CCU-to-SPU Handshaking Format

Figure 2–66 shows CTLD receiving and generating the SPU interface. When ICU sends command, address, and data with parity to SPU, JDC0 sends handshaking signals to CTLD, JDBX sends data and address with parity to CTLD, and CTLD passes this onto SPU.

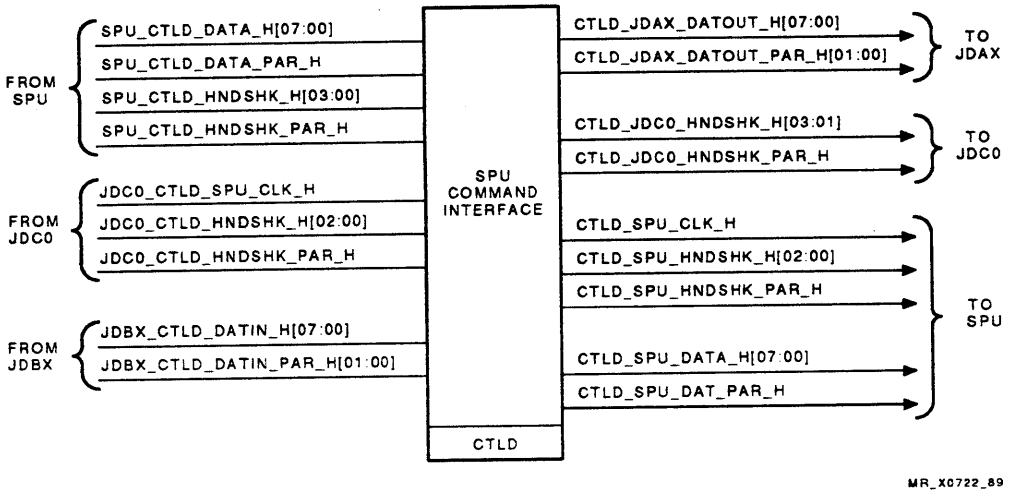


Figure 2–66 SPU Command Interface

2.9.1 JBox Key Signals

Table 2–32 lists the key CTLD-to-SPU signals.

Table 2–32 CTLD-to-SPU Signals

Signal	Description
CTLD_SPU_DATA_H, L[07:00]	Contains the data.
CTLD_SPU_DATA_PAR_H, L	Contains the parity for the data.
CTLD_SPU_HNDSHK_H, L[03:00]	Contains the handshaking signals.
CTLD_SPU_HNDSHK_PAR_H, L	Contains the parity for the handshaking signals.
CTLD_SPU_CLK_H	Provides the SPU output clock.

2.9.2 JBox Commands

See Chapter 6.

2.9.3 SPU Key Signals

Table 2–33 lists the key SPU-to-CTLD signals.

Table 2–33 SPU-to-CTLD Signals

Signal	Description
SPU_CTLD_DATA_H, L[07:00]	Contains the data.
SPU_CTLD_DATA_PAR_H, L	Contains the parity for the data.
SPU_CTLD_HNDSHK_H, L[03:00]	Contains the handshaking signals.
SPU_CTLD_HNDSHK_PAR_H, L	Contains the parity for the handshaking signals.
JDC0_CTLD_SPU_CLK_H	Provides the SPU input clock.
CTLD_JDAX_DATOUT_H[07:00]	Passes the data to the XJA buffer in JDAX.
CTLD_JDC0_HNDSHK_H[03:01]	Passes the handshaking signals to the JDC command buffer.
CTLD_JDC0_HNDSHK_PAR_H	Passes the parity for the handshaking signals to the JDC command buffer.

2.9.4 SPU Commands

See Chapter 6.

3
JBox Cache Consistency

This chapter defines cache consistency and inconsistency. It describes the CPU data cache, cache tags, SCU global tag STRAMs, and global tags. It describes how the system control unit (SCU) performs a global tag lookup and how SCU writes tags into the global tag STRAMs.

3.1 Overview

SCU ensures that if multiple copies of a cache block exist, the data in the cache block *is consistent or identical within multiple CPU cache sets 0 or 1*, and that a port (CPU or I/O) receives the *most recent copy* of the data.

SCU performs either normal operations or exceptions. A normal operation is performed when SCU receives a request for which memory has the most recent copy of the data. A copy of this data may reside in another CPU cache set 0 or 1, but the global tag status of the requester and other CPUs are consistent, in that copies of the data are identical.

An exception occurs when SCU receives a data request for which the data in memory is not the most recent, has been modified, and resides in either cache set 0 or 1 of another CPU. The tag status of the requester and the other CPUs are inconsistent. An exception operation must be performed in which the SCU:

- Obtains a copy of the most recent data from the other CPU cache
- Sends a copy of the data to the requester (if applicable)
- Writes the tag status into the global tag STRAMs
- Writes the data into main memory

3.2 CPU Cache Data STRAMs

The MBox contains the CPU data cache. The data cache capacity is 128 Kbytes. The data cache consists of eighty 4K × 4-bit STRAMs.

3.2.1 Cache Block

Each CPU has a two-way set associative, write back cache. The two sets are set 0 and set 1. Each set contains 64 Kbytes of data.

Figure 3–1 shows a cache block. Each cache block is 64 bytes long and has:

- One valid bit for each longword, for a total of 16 valid bits
- One written bit for the entire block

The valid and written bits are stored in a cache tag in the cache tag store.

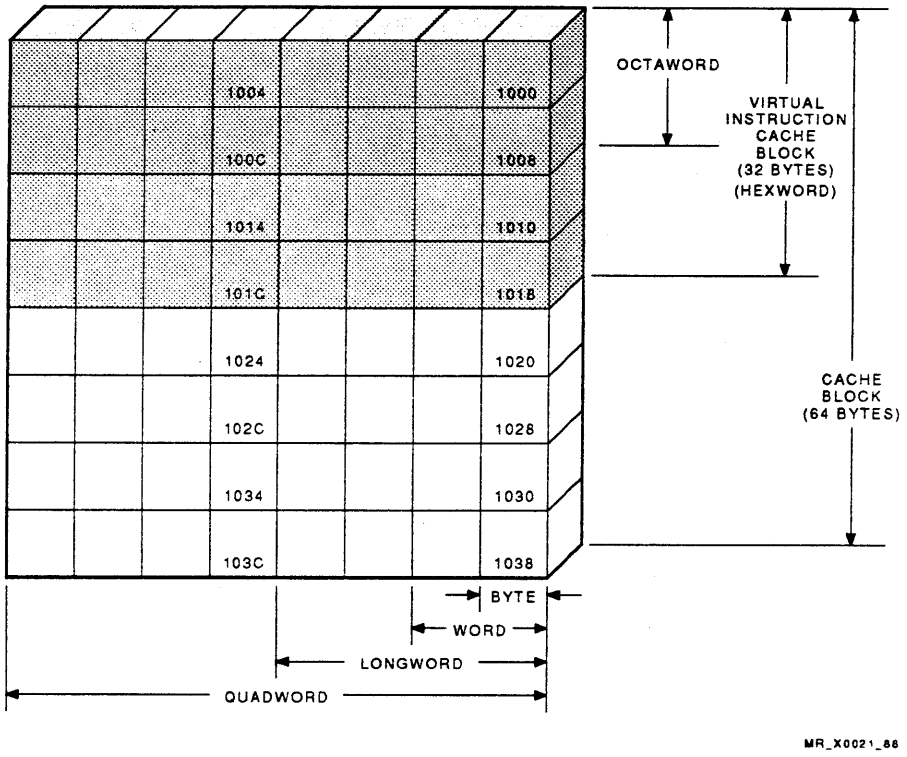


Figure 3–1 Data Sizes (Cache Block)

3.2.2 Cache Set 0 and 1

Each of the two data caches contains 64 Kbytes of data including byte parity. Each set is 8K lines deep and 8 bytes wide. Figure 3–2 shows the cache data found at each location in the data cache STRAMs. Each location contains 8 bytes (byte 0 through byte 7, or quadword), byte parity [07:00] and ECC [07:00].

Figure 3–3 shows the cache tag store, cache sets 0 and 1, and cache block.

For each CPU cache set, SCU has a 1K section in the global tag STRAMs that contains 1K cache block addresses and corresponding status bits. Figure 3–4 shows the locations in the SCU global tag STRAMs used for each cache set.

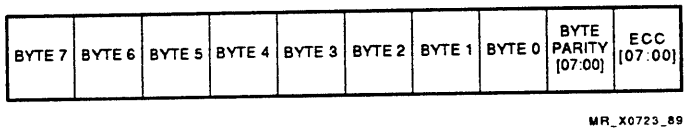


Figure 3–2 Cache Data (Quadword)

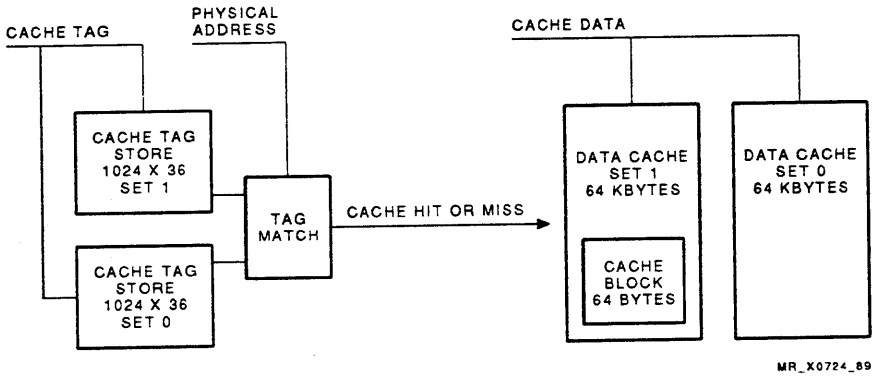


Figure 3–3 Cache Tag Store and Cache Set 0, 1

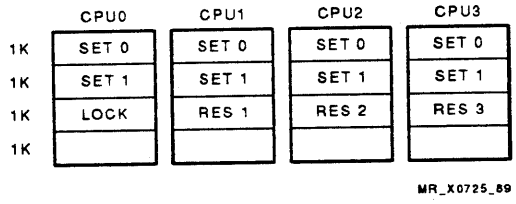


Figure 3–4 SCU Global Tag STRAMs Status Locations for Cache Set 0, 1

3.2.3 CPU Cache Lookup

A CPU cache lookup follows the successful translation of the virtual address into a physical address. The MBox addresses the cache tag STRAMs and determines whether the data is in either cache set 0 or 1 (hit) or a refill must take place to get the data from main memory (miss).

A cache miss results from a read miss or a write miss. A read miss occurs when an attempt to read cache results in a miss. The MBox sends a read refill command to the JBox, and the JBox responds with a return data read command and sends the data to the MBox. The MBox sets the valid bits in the CPU cache tag, and SCU writes read status from cache into the SCU global tag STRAMs.

A write miss occurs when an attempt to write to cache results in a miss. The MBox sends a write refill command, and the JBox responds with a return data written command and sends the data. The MBox sets the written bit in the cache tag, and SCU writes written full status into its global tag STRAMs.

3.2.4 CPU Cache Refill

When the IBox, EBox, or MBox (TB fixup unit) generates a read or write request resulting in a cache miss and the cache lookup fails to find the requested data in either set 0 or 1, the MBox sends either a read refill or a write refill command for a block of data (64 bytes) from SCU.

The read refill or write refill may be linked (read refill linked or write refill linked) with the write back command, if valid, and written data is found in the cache block needed for the refill. The read refill may also be part of a lock request (read refill linked lock or read refill lock), if the EBox (through the MBox) needs to notify SCU to lock a cache block and prevent other processors from writing to the block. To unlock the cache block, the MBox sends the write refill unlock command to SCU.

As the MBox receives data in the refill buffer, the data is wrapped; SCU sends the requested quadword first (return data read or return data written), followed by the seven remaining quadwords in the cache block. The MBox sends the data to the IBox or EBox, while unloading the refill buffer into the cache and updating the CPU cache tag store.

3.2.5 CPU Cache Write Back

Data is copied back to main memory from the MBox write back cache if another CPU requests the cache block and the block has valid, written data or if the CPU requests a cache sweep. The cache block may be needed when a CPU detects a cache miss during a read or write request and the cache lookup fails to find the requested data. This condition requires refilling the data cache and updating the cache tag store.

The MBox loads the valid, written data found in the cache block into the write back buffer and sends data ready and write back to SCU. SCU issues send data, get data read or get data invalidate, depending on the original refill request, unloads the write back buffer, and writes the data into main memory.

3.3 CPU Cache Tag STRAMs

The cache tag store is two-way set associative. Each of the tag stores for cache sets 0 and 1 contains 1024 tag entries. Figure 3–3 shows the cache tag stores. Each of the tag entries describes a 64-byte block in the data cache. Each tag entry is associated with a cache block and contains physical address bits, a written bit, and valid bits. Figure 3–5 shows the cache tag indicating the status for each address in the cache data STRAMs. Table 3–1 lists the cache tag bits and descriptions.

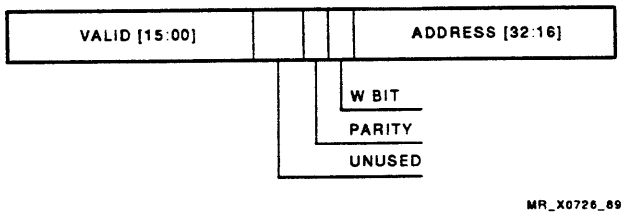


Figure 3–5 CPU Cache Tag Data

Table 3–1 Cache Tag Bit Descriptions

Bit	Description
Valid	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, and so on. All 16 bits are usually set when the block is refilled and checked during lookup to determine if the longword requested is valid in that cache (cache hit).
Parity	Parity bit for the tag. During refills it is generated and written, and during lookups it is read and checked. Odd parity is calculated on the entire contents of the 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.
Address	This 16-bit physical address field 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 that cache.

3.4 SCU Global Tag STRAMs

Each CPU cache set, 0 and 1, has a corresponding 1K section in the SCU global tag STRAMs. Figure 3–6 shows the global tag STRAMs containing the tag status for each CPU cache set. Each cache block within the cache set can have any status listed in Table 3–2. The global tag STRAMs consist of twenty-four 4K × 4-bit STRAMs, containing 16K locations. Two 1K sections of the 4K STRAMs are used for cache set 0 and 1. Of the remaining 2K sections, 1K is used for locks for interlock status and interlock reservations for CPUs and I/O devices (also shown in Figure 3–6), and the other 1K is unused.

SCU maintains the tag STRAMs. Whenever any CPU or I/O device makes a memory reference, SCU examines the global tag STRAMs and initiates the appropriate action to maintain cache consistency.

Figure 3–6 Global Tag STRAMs for CPUs Cache Set 0, 1

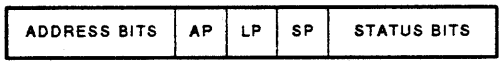
Table 3–2 Cache Block Status

Status	Description
Read (RD)	The cache block is marked as read as a result of a return data read or get data read. All the valid bits in the cache tag store are set.
Written full (WRTF)	The local cache block is marked as written full as a result of a return data written command. The CPU modifies the data in the cache block. If another CPU requests this block, SCU gets the data from the CPU and sends it directly to the port, without first writing the data into main memory. After sending the data, SCU writes it into memory.
Written partial (WRTP)	The written and valid bits are set for the 16 longwords in the cache block as a result of longword write updates. This block is marked as written partial and requires a merge of memory array data and the valid, modified longwords before the block can be sent to the requester.
Invalid (INV)	Valid bits in the local cache are marked zero. This block is marked invalid as a result of a write back initiated by the CPU, a get data invalidate by the JBox, or a simple invalidate request.

3.4.1 Global Tag Contents

Each tag STRAM location contains address, parity, and status bits. Figure 3–7 shows the global tag contents at one location in the global tag STRAMs.

Global tag STRAMs contain copies of each of the CPU cache tag stores. The global tag STRAMs indicate read, written partial, written full, invalid, and lock status of cache blocks.



MR_X0728_89

Figure 3–7 Global Tag Contents

3.4.2 Global Tag Address Bits

Figure 3–8 shows the address bits in the global tag STRAMs. The address match logic writes the address bits during a tag status write cycle and reads the address bits during a tag lookup read cycle.

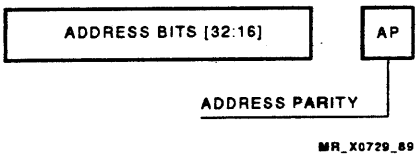


Figure 3–8 Global Tag Data

3.4.3 Global Tag Status Bits

Each tag location contains four status bits that define the state of that cache block in the CPU cache. Figure 3–9 shows the global tag status bits and parity. Table 3–3 lists the global tag status bits and their descriptions.

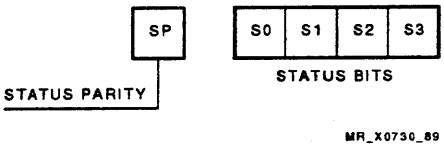


Figure 3–9 Global Tag Status Bits

Table 3–3 Global Tag Status Bit Descriptions

Status Bit	Description
01:00	Indicates the status of the cache block for the address: 0 = Invalid 1 = Read 2 = Written partial 3 = Written full
02:00	Used for interlocks. See Figure 3–21 and Table 3–12.

3.4.4 Global Tag Parity Bits

As shown in Figure 3–10, each tag location has three parity bits. Table 3–4 describes each parity bit.

Figure 3–10 Global Tag Parity Bits

Table 3–4 Global Tag Parity Bit Descriptions

Parity Bit	Generated and Checked By	Description
Address	MTCH MCA	A 1-bit parity bit. This parity is calculated across the PA [32:16], which becomes tag address [32:16]. See Figure 3–12.
Location	MTCH MCA	A 1-bit parity bit. This parity is calculated across the STRAM address bits used to look up the global tag location. See Figure 3–12.
Status	MICR MCA	A 1-bit parity bit. This parity is calculated across status bits [03:00].

3.5 SCU Global Tag Lookup

A global tag lookup follows port arbitration. SCU addresses the global tag STRAMs to determine whether the data is in one or more CPU caches. SCU also determines the status of the CPU caches (invalid, written full, written partial, read, or locked). An address match occurs when the port's physical address matches the address bits stored in the global tag STRAMs. SCU determines the microcode address to handle the request using the result of the tag lookup.

When a port requests data from SCU or when a CPU writes to a cache block for the first time, the port sends a physical address to SCU. The SCU passes the physical address through the address crossbar and writes the address and corresponding cache status into the global tag STRAMs.

Memory reads may be initiated immediately, without waiting for the tag lookup results.

During the global tag lookup, SCU reads status for eight cache sets (sets 0 and 1 for each CPU) in two read cycles. Table 3–5 lists the read cycles for the tag lookup and the write cycles for the tag status. For lock requests, the tag lookup requires three read cycles (one for set 0, one for set 1, and one for lock).

SCU performs global tag lookups for both data and nondata requests.

Table 3–5 lists the tag lookup cycles for CPU and I/O requests. For example, a CPU refill has three cycles: 0, 1, and 2. In cycle 0, the MBox requests refill data and SCU examines set 0. In cycle 1, SCU examines the other global tag location for cache set 1. In cycle 2, SCU writes the global tag status into the requester's set 0 or 1 tag STRAMs.

Table 3–5 Global Tag Lookup Cycles

	Cycles			
Command	**0**	**1**	**2**	**3**
Sweep (write back and invalidate)	Read global tag STRAMs for set 0	Read global tag STRAMs for set 1	Write tag status	–
DMA read, DMA write	Read global tag STRAMs for set 0	Read global tag STRAMs for set 1	Write tag status	–
Refill (link), longword update	Read global tag STRAMs for set 0	Read global tag STRAMs for set 1	Write tag status	–
DMA read lock, DMA write unlock	Read global tag STRAMs for set 0	Read global tag STRAMs for set 1	Read lock status	Write lock status
CPU read lock, CPU write unlock	Read global tag STRAMs for set 0	Read global tag STRAMs for set 1	Read lock status	Write lock status

3.5.1 Addressing the Global Tag STRAMs

The following steps summarize the global tag lookup operation:

1. ADRX receives the physical addresses from each CPU and I/O port and sends PA [32:16] to the MTCH MCA.
2. MTCH addresses the global tag STRAMs and receives the global tag data. Figure 3–11 shows the inputs and outputs of the ADRX MCAs, MTCH MCA, and global tag STRAMs.

   MTCH compares PA [32:16] and global tag [32:16] for each CPU (four global tag entries are read in each of two tag lookup cycles). If an address match occurs, MTCH sends address match [03:00] and the address match parity to MICR. Figure 3–12 shows the MTCH MCA, containing the address match, parity generator, and checker logic.
3. MTCH checks parity across PA [32:06].
4. MTCH generates and checks address parity for the global tags [32:16].
5. MTCH generates and checks location parity for the global tag locations.

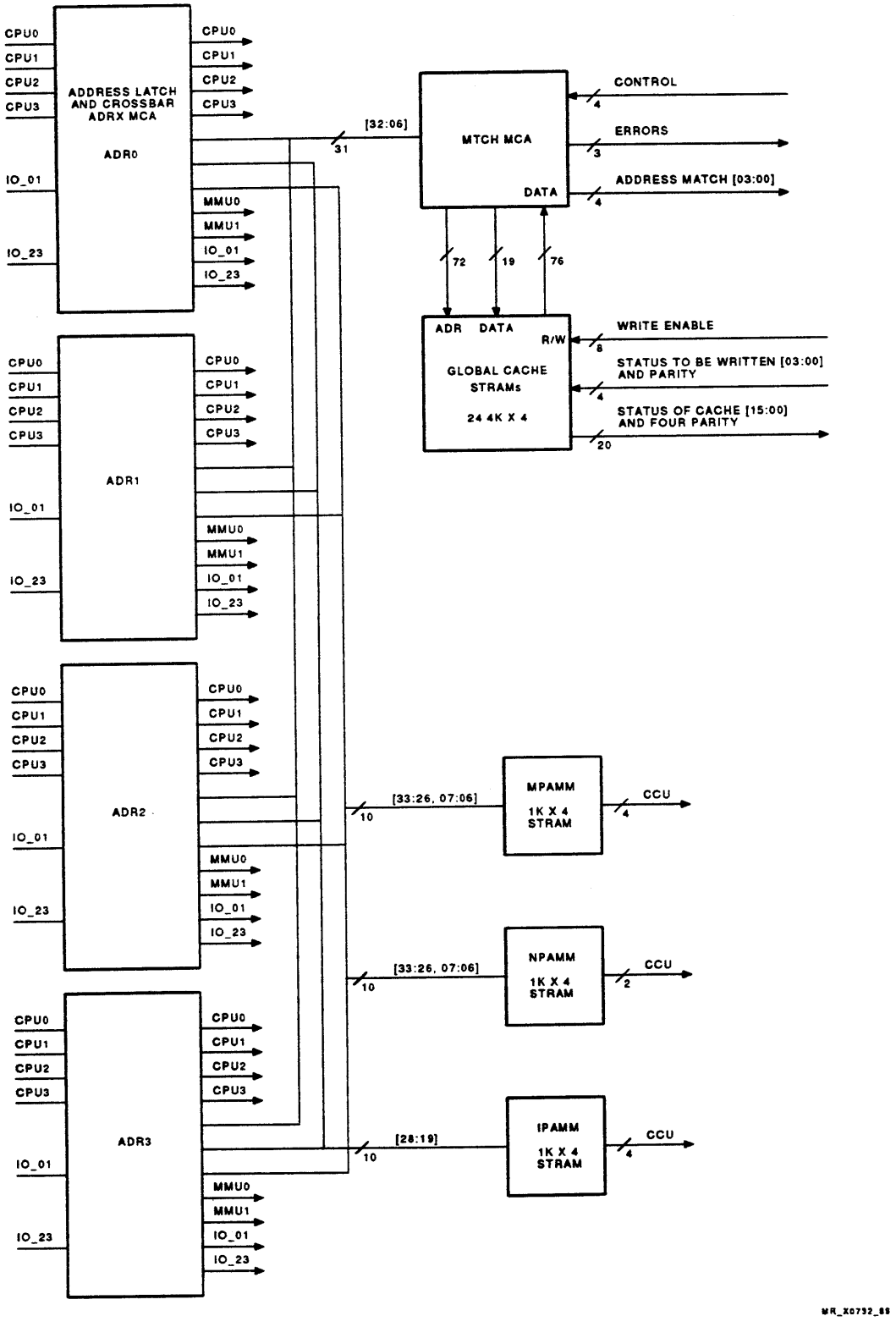


Figure 3–11 ADRX MCA Outputs

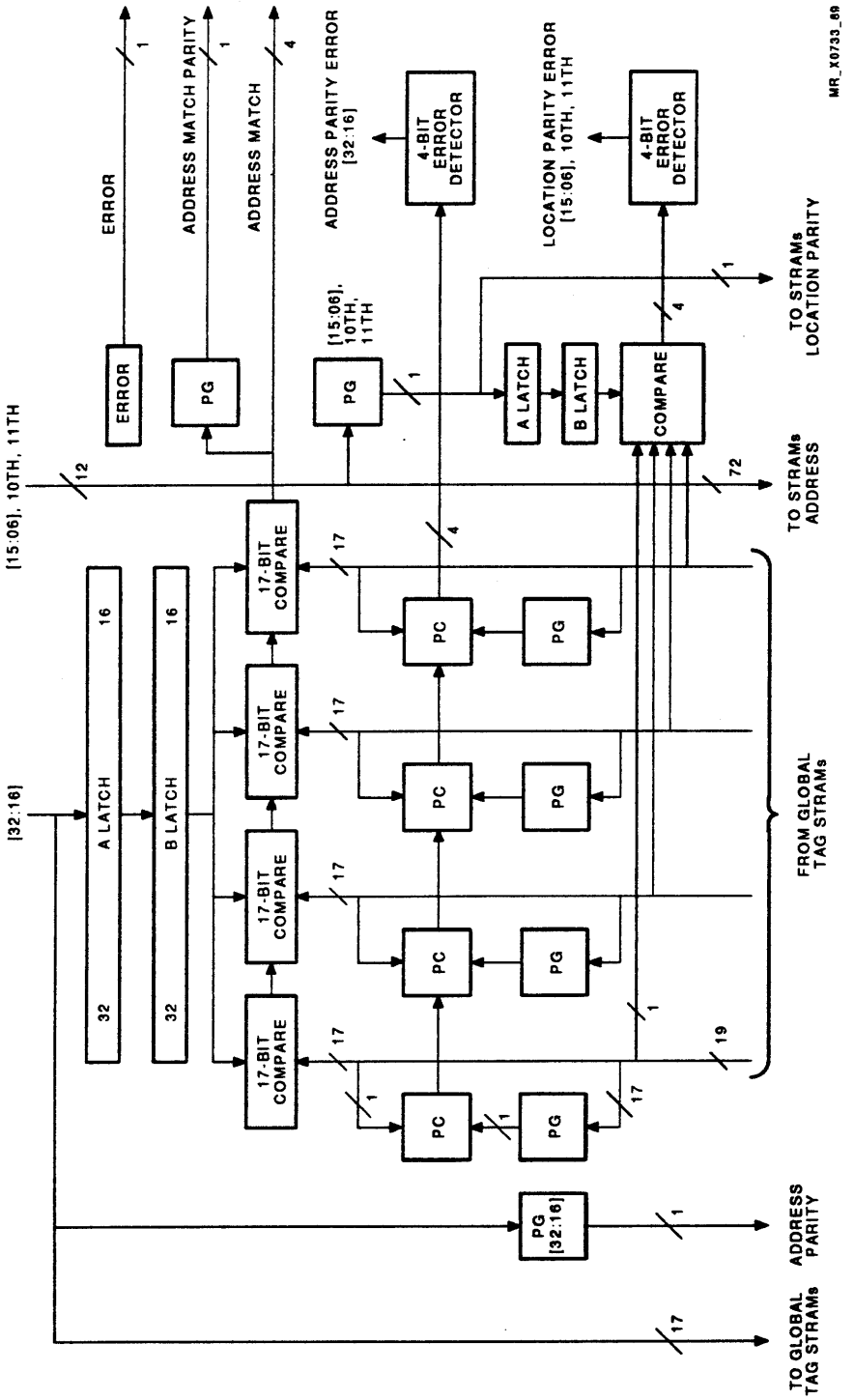


Figure 3–12 MTCH MCA

3.6 Reading Global Tags

The MTCH MCA addresses the global tag STRAMs, as shown in Figure 3–11. MTCH receives 19 data bits from each of 4 CPU global tag STRAMs, for a total of 76 data bits, as follows:

Seventeen address bits per CPU per cache set
One address parity bit per CPU per cache set
One location parity bit per CPU per cache set

Figure 3–13 shows the 16 status bits and 4 status parity bits. MICR receives 5 data bits from each CPU's global tag STRAMs, for a total of 20 data bits:

Four status bits per CPU per cache set
One status parity bit per CPU per cache set

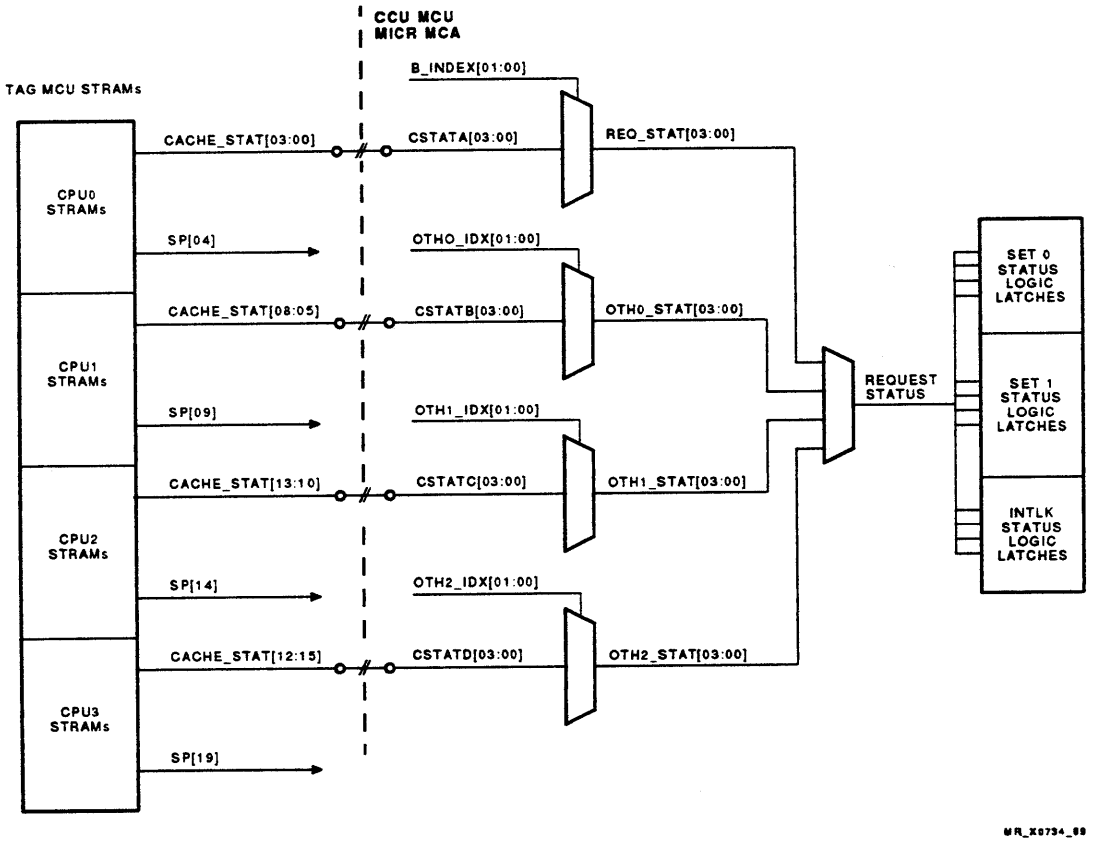


Figure 3–13 Sixteen Status Bits and Four Parity Bits

MICR selects and latches the four sets of status bits (one for each CPU). MICR can select any CPU as requester and refers to the three remaining CPUs as other 0, 1, and 2:

REQ_STAT[03:00]
OTH0_STAT[03:00]
OTH1_STAT[03:00]
OTH2_STAT[03:00]

Each CPU port can have up to three commands, and each I/O port can have up to two commands, in their respective command buffers. As shown in Figure 3–13, B_INDEX[01:00] and OTHx_IDX[01:00] select status bits CSTATA, B, C, and D[03:00], which determine REQ_STAT[03:00] and OTHx_STAT[03:00]. Table 3–6 lists the CSTATA code and corresponding port and command buffer.

Table 3–6 Tag Status

Code	Port	Command Buffer
0	CPU0	A
1	CPU0	B
2	CPU0	C
3	CPU1	A
4	CPU1	B
5	CPU1	C
6	IO0	A
7	IO0	B
8	CPU2	A
9	CPU2	B
A	CPU2	C
B	CPU3	A
C	CPU3	B
D	CPU3	C
E	IO1	A
F	IO1	B

MICR latches and decodes the four sets of status bits to determine status for the following:

- **STAT0** — Represents cache set 0 for all CPUs.
- **STAT1** — Represents cache set 1 for all CPUs.
- **Lock and lock reserve** — An address can have lock status or lock reserve status. If the address has lock status, SCU sends lock deny for other lock requests until the address is unlocked. When reserve lock status has been written into the global tag STRAMs for a port, SCU sends the port lock deny, and the port must send another lock request for the address. When SCU receives subsequent lock requests, MICR checks the lock reserve status and grants the lock if the port has the highest priority reserve status.

As shown in Figure 3–14, MIC match receives STAT0 and STAT1 from the set status latches. With this information, plus address match [03:00] from MTCH, MIC match determines which status bits are from the global tag STRAM location in which the match occurred. MICR match decodes the set status output, generates valid status, and sends STAT0 and STAT1 to MIC tag.

MIC tag receives the valid status signal and generates fork status bits for the fork address, which determines the SCU microaddress. MIC tag also checks for illegal status conditions. MIC lock decodes the set status output and generates lock status and lock error.

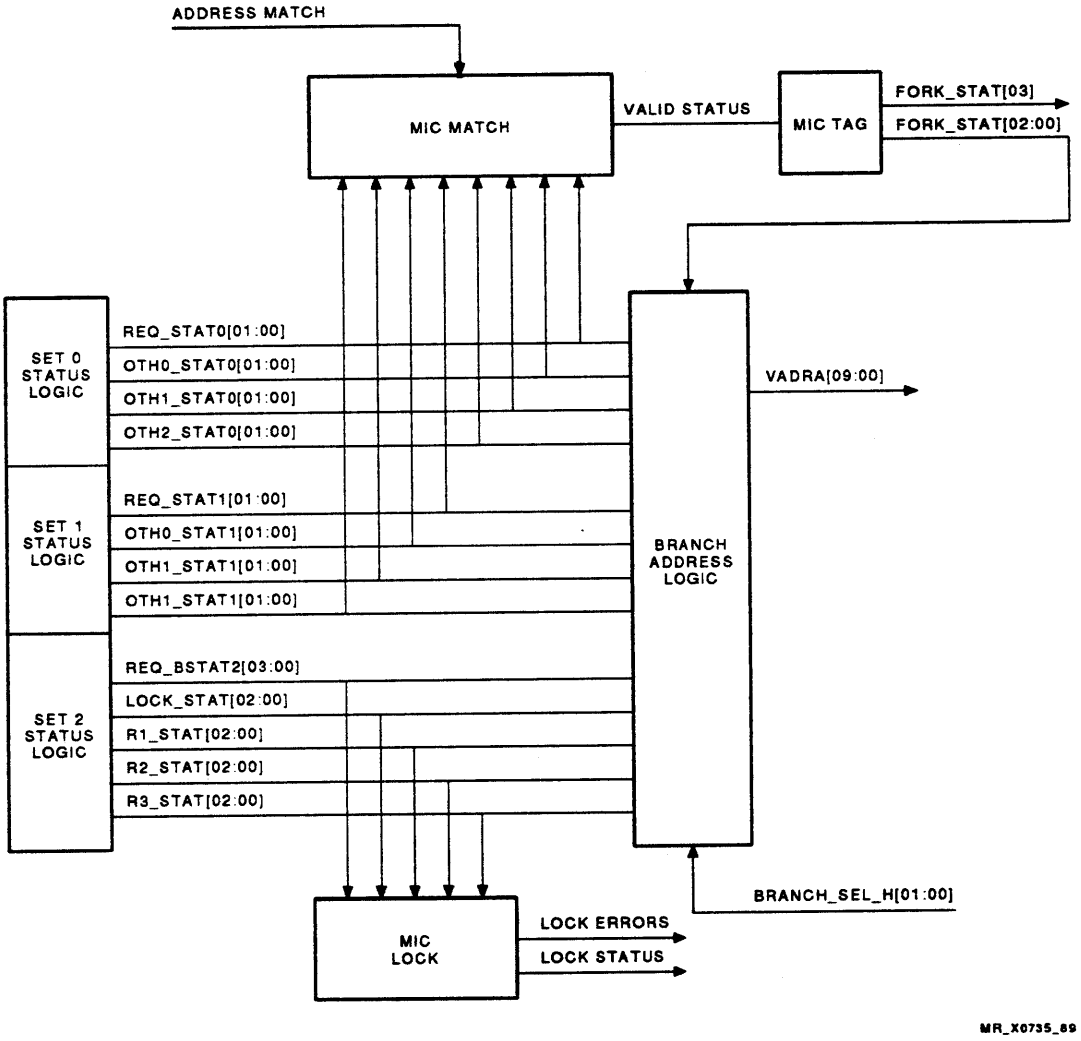


Figure 3–14 Set 0, Set 1, and Lock Status Latches

3.6.1 SCU Microcode

Figure 3–15 shows the fix command and tag status fields of the microword. SCU uses these fields to send commands to memory, JBox, and tag logic during fixup operations, in which a memory request results in an inconsistent cache status. Examples of some of the fix commands that SCU uses during fixup operations are listed in Table 3–7. See Chapter 4 for more details.

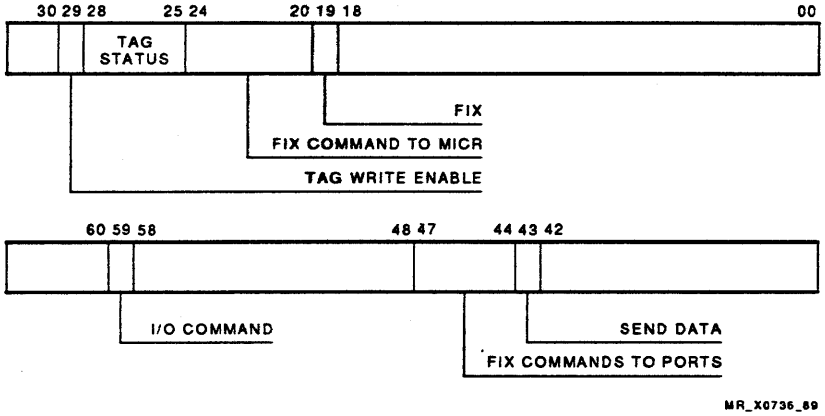


Figure 3–15 Microword Fix Command and Tag Status Fields

Table 3–7 Microcode Fix Command — Examples

Request	Requester	Other	Fix Command
Write refill	Invalid	Written full	Write pass invalidate. The JBox assembles get data invalidate/get data read and sends the command to the other CPU. The CPU responds with the data, and memory performs a write pass operation. This operation sends the data to the JBox, which then sends the data to the requester. SCU writes the tag status into the tag STRAMs.
Read refill	Invalid	Written full	Write pass read. The JBox assembles get data invalidate and sends the command to the other CPU. The CPU responds with the data, and memory performs a write pass operation. This operation sends the data to the JBox, which then sends the data to the requester. SCU writes the tag status into the tag STRAMs.
Write refill	Invalid	Written partial	Write read invalidate. The JBox assembles get data invalidate and sends the command to the other CPU. The CPU responds with the data, and memory performs a write read operation. This operation merges the data received from the other CPU with data in the memory arrays and sends the data to the JBox, which then sends the data to the requester. SCU writes the tag status into the tag STRAMs.

3.6.2 Writing Tag Status

Figure 3–16 shows the write enable, address, and status inputs to the global tag STRAMs. As MTCH addresses the global tag STRAMs, MICR uses the tag status, tag write enable, and address [01:00] fields of the microcode to write the cache status and lock status and to select the set number.

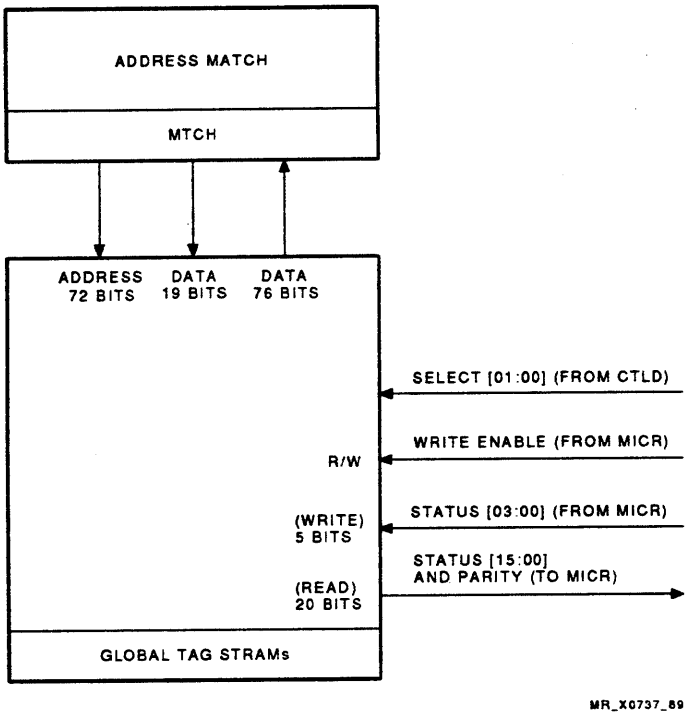


Figure 3–16 Writing Global Tag Status

The following steps summarize a read refill request in which the requester (cache status is invalid) sends a read refill command to SCU and the data is in another CPU cache block (cache status is written partial). SCU must get the data from the other cache and send the data to the requester. To do this, SCU first reads the global tag STRAMs and, before retiring the request, writes the tag status in the global tag STRAMs.

In this example:

1. SCU receives the read refill, address, and cache set to be refilled from the MBox.
2. MTCH compares the requester's physical address with the contents of the global STRAMs to determine if there is a match.
3. MTCH sends the match information to MICR.
4. MICR reads the status bits of the global tag STRAMs. MICR uses the command and status bits to form a microaddress to send to the microcode. The microword loads the fixup queue and determines which commands the JBox sends to memory and MBox. Microcode flows and fixup flows cannot occur simultaneously. Microcode notifies CTLC when to retire the request and continue arbitration.
5. MICR writes the tag status into the tag STRAMs for the requester as read and for the other CPUs as invalid.
6. JBox sends get data invalidate to the other CPU to get the data for the requester. The CPU invalidates the cache block by clearing the valid and written bits when it loads the buffer for a write back.
7. CPU sends data ready to MICR.
8. MICR starts the memory write.
9. JBox sends return data written to the requester. The CPU sets the written bit in the local cache tag store for the block.

3.7 Errors

The following is a list of global tag lookup errors:

- **JBox tag error** — If more than one CPU has written data for the same address or if one CPU has read, while another CPU has written, data for the same address.
- **Other set valid** — If the other set from the requester has valid data and match.
- **Request status error** — If the data is written in the requested set, and no address matches. (A linked command should have been sent.)
- **MBox error** — If a CPU requests read refill and already has read status or if the CPU requests write refill and already has written full status for that address.
- **Lock error** — If a CPU requests lock refill and already has that address locked or if a CPU requests unlock refill and has unlocked that address already.
- **Other status error** — If another CPU has both sets valid and match.

3.8 Maintaining Consistent Global Tag Status

All requests for data are made to SCU, which must retrieve the latest copy of the data and return it in response to read refill or write refill. SCU maintains cache status for every address in each CPU cache. For example, if the MBox sends write refill, SCU examines the cache status of all CPUs to check whether the data is in main memory or another CPU cache. If the data is in main memory, SCU reads the data from the memory arrays, sends the cache block to the MBox (return data written), and marks the cache block as written full.

If the data is in the other CPU's cache, SCU does the following:

1. Sends get data invalidate to the other CPU to get the data and invalidate the cache block.
2. Marks the cache block in the other CPU invalid.
3. Sends return data written to the requester.
4. Marks the requester cache block as written full.

Table 3–8 lists the requests and cache conditions in which no cache conflicts exist.

Table 3–8 Consistency

Request	Requester	Other
Write refill	Invalid	Invalid
Read refill	Invalid	Invalid
Read refill	Invalid	Read

3.9 Handling Inconsistent Global Tag Status

Two factors control SCU's handling of cache inconsistency. One factor is write back caches, and the other is having multiple copies of data available. SCU maintains global tag STRAMs, checks status on memory accesses, and ensures that *processors are not sharing written status for the same cache block simultaneously.*

Inconsistency exists when a request, a requester's cache status, and another cache status result in the possibility of different copies of data existing for the same cache block. Table 3–9 lists requests and cache conditions for the requester and other CPUs that result in inconsistency.

Table 3–9 Fixup Operations

Before		After			
Request	**Requester**	**Other**	**Requester**	**Other**	**Fixup Operations**
Read refill	Invalid	Written full	Read	Read	Fix command: Write pass invalidate. Memory: Write pass. JBox: Get data invalidate to other, return data read to requester. MICR: Writes tag status for requester as read and other as read.
Read refill	Invalid	Written partial	Read	Invalid	Fix command: Write read invalidate. Memory: Write read. JBox: Get data invalidate to other, return data read to requester. MICR: Writes tag status for requester as read and other as invalid.
Write refill	Invalid	Read	Written full	Invalid	Fix command: Read. Memory: Memory read. JBox: Invalidate to other, merge data in memory, return data written to requester. MICR: Writes tag status for requester as written full and other as invalid.
Write refill	Invalid	Written full	Written full	Invalid	Fix command: Write pass invalidate. Memory: Write pass. JBox: Get data invalidate to other, return data written to requester. MICR: Writes tag status for requester as written full and other as invalid.
Write refill	Invalid	Written partial	Written full	Invalid	Fix command: Invalidate other, OK to write to requester. Memory read aborted, no return data.

The following is an example of how SCU handles inconsistent global tag status:

1. In Figure 3–17, as a result of a read refill request, CPU0 has a cache block containing location 1000.
2. CPU1 has the same cache block as a result of a read refill request.
3. CPU0 and CPU1 share a cache block with status as read in the SCU tag STRAMs.
4. CPU2 detects a write miss and requests the same cache block. CPU2 needs a copy of the cache block to write data into.
5. In Figure 3–18, SCU updates the cache status of CPU0 and CPU1 to invalid and of CPU2 to written full.

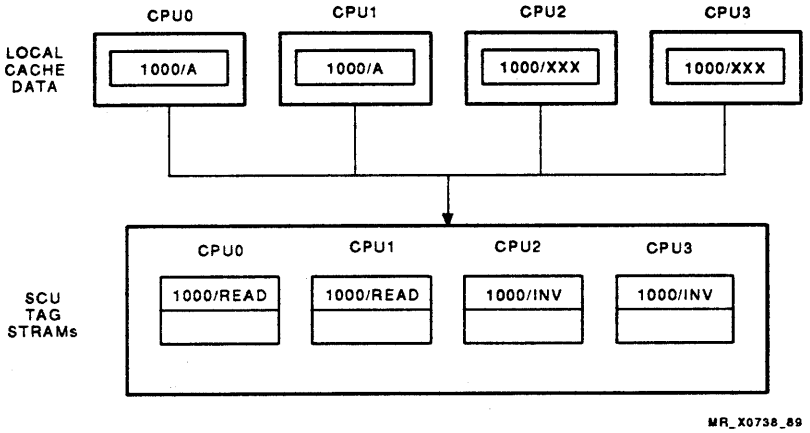


Figure 3–17 CPU Local Cache STRAMs After Read Refill

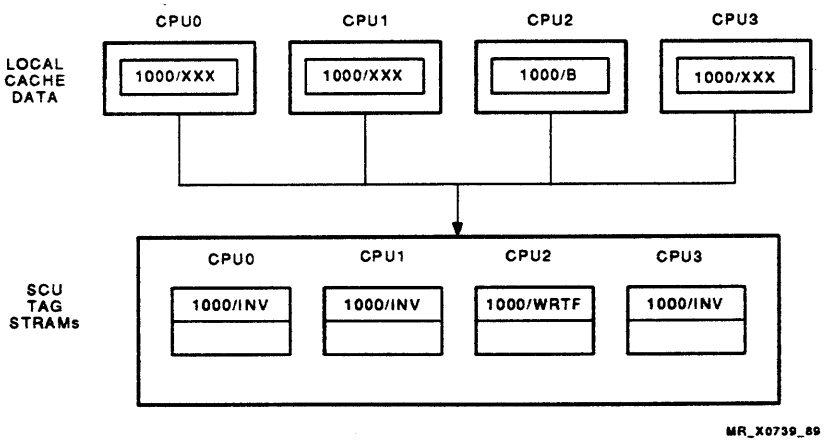


Figure 3–18 CPU Local Cache STRAMs After SCU Updates Cache

Figure 3–19 shows how SCU handles inconsistent tag status for a CPU write refill request. The following sequence summarizes the events:

❶ CPU0 sends write refill, address, and which cache set, 0 or 1, is to be refilled.

❷ ADRX selects the CPU0 physical address.

❸ ADRX sends the CPU0 physical address to MTCH.

❹ MTCH addresses the global tag STRAMs.

❺ MTCH receives the address bits from the tag location.

❻ MTCH compares the physical address from CPU0 with the tag contents, and if there is a match, MTCH sends address match [03:00] to MICR.

❼ MICR receives the address match and tag status.

❽ MTCH addresses the global tag STRAMs and provides the address bits. MICR writes the tag status bits into the tag location.

❾ MICR generates a microaddress for the microcode.

❿ The microword contains tag status, commands for the CTLX (such as get data invalidate), and a command for memory (write pass). The microcode loads a command into the fixup queue to handle the inconsistency.

⓫ MICR generates write enable and tag status bits.

⓬ The MICR sends the get data invalidate command to CPU1. CPU1 sends data ready and the data to SCU. After MICR sends send data and write to memory, CTLC sends the return data written command to CPU0 with the data.

Table 3–10 lists the status of the requester and other CPUs before and after a CPU write refill request is sent to SCU.

Table 3–10 Status of Tag STRAMs for CPU Write Refill Requests

Before		After	
Requester	**Other**	**Requester**	**Other**
Invalid	Invalid	Written full	Invalid
Read	Invalid	Written full	Invalid
Written full	Invalid	Error	
Read	Read	Written full	Invalid
Invalid	Read	Written full	Invalid
Invalid	Written partial	Written full	Invalid
Invalid	Written full	Written full	Invalid

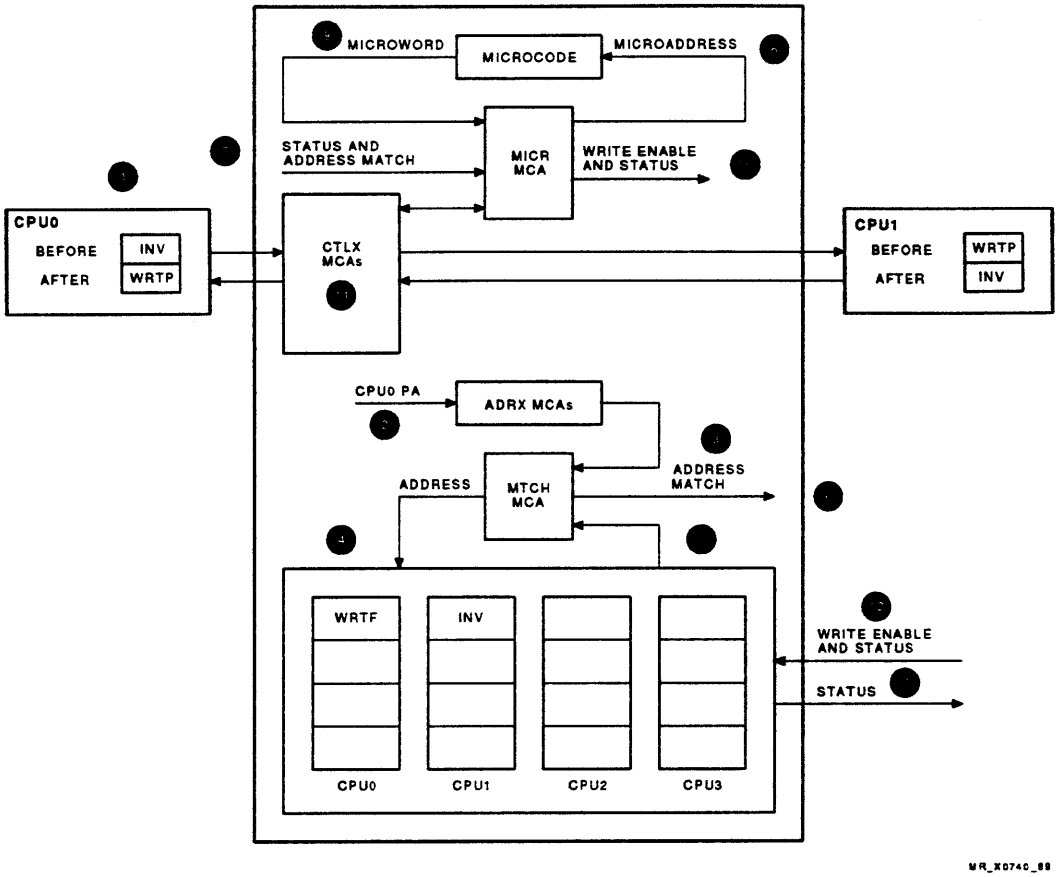


Figure 3–19 CPU Write Refill Request

The following list describes how the SCU updates the status of the tag STRAMs for CPU write refill requests. See Table 3–10.

- **Requester: Read/Other: Invalid** — MMCX receives memory abort. MICR asserts tag write enable. The status bits are written into the cache set block indicated by the index pointer. The tag MCU updates the tag status of the requester, changing it from read to written full.
- **Requester: Invalid/Other: Read** — MMCX receives OK to read the block from main memory. The fixup logic executes an invalidate command to the other CPUs that have a read status. The fixup logic handles the fixup sequence. MICR loads the fix queue. The tag MCU writes invalid status for each CPU having a read status for that block.

  Earlier in the fork cycle, the JBox sends an invalidate command to each MBox to change the status for the block from read to invalidate. The microcode sends an index value and notifies the tag MCU that an address must accompany the invalidate command. The tag MCU uses the index value to select the address in one of the address receive latches. JBox sends a return data written to the requester. The tag MCU updates the requester's tag status from invalid to written full.

When the last invalidate is sent to the other CPUs and when the tag status for both the requester and other CPUs has been updated, CTLC receives CTLC done and determines that MICR has completed the required tag write. CTLC retires the request and arbitration continues. CTLD, after receiving fixup, proceeds to the next tag lookup.

- **Requester: Invalid/Other: Written Partial** — Memory abort is sent to MMCX to stop the memory read from being completed. MICR loads the fixup queue. The fixup logic handles the fixup sequence, and executes a write read invalidate command. CTLC stops arbitration.

  The fixup logic also executes a write read data ready command. The fixup queue starts the memory write. MMCX performs a read and merges the written partial data with the memory read data. JBox sends the data to the CPU or ICU, which then sends the data to XJA.

  CTLC stops arbitration. The tag MCU replaces invalid status with the written partial status for the other CPU in the fork cycle. If the other CPU has posted a write back, it is aborted by MICR when the write back arbitrates after the get data command is completed. The JBox sends an invalidate command to the other CPU to change the status from written full to invalidate. The microcode sends an index value and notifies the tag MCU that an address must accompany the invalidate command. The tag MCU uses the index value to select the appropriate address in one of the address receive latches. JBox sends a return data written to the requester. The tag MCU changes the requester's tag status from invalid to written full.

  When the last invalidate is sent to the other CPUs and when the tag status for both the requester and other CPUs has been updated, CTLC receives CTLC done and determines that MICR has completed the required tag write. CTLC retires the request and arbitration continues. CTLD, after receiving fix hold/yes and fix increment/yes, proceeds to the next tag lookup.

- **Requester: Invalid/Other: Written Full** — Memory abort is sent to MMCX to stop the memory read from being completed. SCU selects the index of the other CPU that has the WRTF block needed by the requester. MICR loads the fixup queue. The fixup logic handles the fixup sequence and executes a write pass invalidate command.

  The fixup logic also executes a write read data ready command. The branch select field contains data ready. The microcode flow branches when MBox sends data ready during the get data flow. The data switch passes written full data to the requester.

  CTLC stops arbitration. MICR replaces invalid status with the written partial status for the other CPU. The JBox sends an invalidate command to each MBox to change the read status for the block to invalidate. As a nonmemory command, the tag MCU is notified that an address must accompany the invalidate command. When the invalidate is sent to the MBox, CTLC receives CTLC done and determines that MICR has completed the required tag write. CTLC retires the request and arbitration continues. CTLD, after receiving fix hold/yes and fix increment/yes, proceeds to the next tag lookup.

3.9.1 Written Full and Written Partial Examples

The following steps summarize fixup operations in which the requester has a cache block with invalid status and the other cache block has written partial cache status:

1. CTLA arbitrates the command.
2. If the memory is ready, CTLC starts the memory read and sends the command to CTLD.
3. CTLD starts the tag lookup and sends the command to MICR.
4. MICR generates a fork address for the microcode (command and status). If an error is detected (that is, a JBox tag error), the fork address (118) causes the microcode to send a fatal error command to the MICR MCA.
5. The microcode aborts the memory read. At the same time, the microcode writes the tag status (request = written full) and loads the fixup queue in MICR with the write read and invalidate fix commands.
6. As soon as the other CPU command buffer is available, the fixup queue interrupts the tag lookup microcode fork pipeline for one cycle.
7. The microcode sends a get data invalidate fix command to the other CPU by interrupting CTLC arbitration and startup for one cycle. The microcode writes the tag (other = invalid) and loads the fixup queue with the new command, write read data ready.
8. The fixup queue now waits for a data ready command from the other CPU. When data ready arrives in MICR, the fixup queue again interrupts the pipeline for one cycle.
9. The microcode loads a new fix command into the fixup queue, write read send data. This command tells the fixup queue to start reserving the data path resources needed.
10. As soon as the paths and memory command buffer are available, the fixup queue again interrupts the pipeline.
11. The microcode sends send data to the CPU, which unloads its write back buffer, stops CTLC arbitration, and sends the write read command to memory. The microcode sends a CTLC done bit to CTLC. The fixup operation is completed.
12. Some time later, memory sends a return data read command to the CTLA MCA.

3.10 Interlocks

The memory system services multiple CPUs and I/O processors. If these processors share data structures stored in main memory, the access must be controlled to guarantee the correct sequences of operation. One mechanism for reading from and writing to shared data structures is the use of memory interlocks. The interlocked read operation sets the interlock status, and the interlock write releases it.

When a processor does an interlocked read to memory, no other port can get interlocked access to the same location until the first processor does a write unlock to release it. Using interlocks, the processors can share data structures in a controlled manner without race conditions.

Physically, interlocks are implemented in SCU with interlock status bits associated with the locked memory locations.

3.10.1 Interlock Instructions

Interlock instructions control access to shared writeable data.

VAX architecture supports seven interlock instructions. The instructions are listed in Table 3–11.

Table 3–11 Interlock Instructions

Instructions	Description
BBSSI, BBCCI	These instructions read a single byte, test and modify a bit within that byte, and then write the byte in an interlocked sequence.
ADAWI	This instruction performs a read and then a write operation to a single, aligned sequence.
INSQHI, INSQTI, REMQHI, REMQTI	These instructions manipulate interlock queue headers, allowing queues to be maintained in a multiprocessor system.

3.10.2 SCU Supporting Interlocks

SCU is the only unit that sees all memory traffic and monitors lock traffic. SCU supports interlock reads and writes from CPUs and I/O devices. SCU uses the cache block, which is the unit of memory data allocation, as the unit for interlocks. SCU does not monitor memory accesses of finer granularity than a cache block. SCU can lock up to 1000 cache blocks and deny lock requests from other ports until the blocks are unlocked by the port that owns the lock.

The SCU hardware allows memory to be interlocked by a write refill transaction and unlocked by a write transaction and allows synchronization of processes in quad-CPU and dual-ICU configurations that access shared areas of memory. When SCU receives an interlock read transaction, SCU writes the lock status in the tag STRAMs and performs the usual read function. However, SCU does not accept any further interlock read transaction for addresses in the locked cache block, until the interlock has been removed by a write unlock transaction. When SCU receives a write interlock transaction, it clears the lock status and performs the usual write transaction. SCU tracks the interlocked addresses by storing them in the upper 2K area of the tag STRAMs.

During the time the cache block is interlocked, SCU accepts typical read transactions and all write transactions. However, when SCU receives a read interlock request for the locked block, it denies the request and writes a reserve status into the tag STRAMs. SCU can have up to three reservations for addresses in a locked cache block.

3.10.3 Types of Interlocks

SCU receives two types of interlock requests, read and write. A write interlock request is preceded by a nonlock refill.

3.10.3.1 Interlock Reads

CPU read interlocks either require a refill or do not. When SCU receives a CPU read interlock requiring a refill, SCU performs a tag lookup in three cycles and writes the tag status in two cycles. Table 3–5 shows the tag lookup and the tag status cycles for a CPU read lock and write unlock. In cycle 0, when the MBox requests refill data, SCU examines cache set 0. In cycle 1, SCU examines the cache set 1 tag. In cycle 2, SCU examines the lock status.

If the cache block is not already locked, SCU accepts the read lock and, in cycle 4, writes the lock status into the tag STRAMs.

If the address in the lock request matches the contents of the lock portion of the global tag STRAMs and the valid bit is set (hit), SCU denies the read lock. In cycle 3, SCU writes reserve status in the tag STRAMs. In cycle 4, SCU writes the cache block global tag status. SCU continues to deny a read lock request until the port sends a write unlock for the cache block.

An unlock that does not require a refill has four cycles, three read cycles, and one cycle in which SCU writes the lock status.

3.10.3.2 Interlock Writes

The CPU write interlock requires five cycles, and the I/O write interlock requires four cycles. Table 3–5 shows the tag lookup and the tag writing status cycles for a CPU read lock and write unlock. If SCU receives a write unlock, it clears the lock status in the tag STRAMs. An interlock error occurs if no lock was granted previously.

3.11 Interlock Storage

Figure 3–20 shows the location of the lock status in the tag STRAMs.

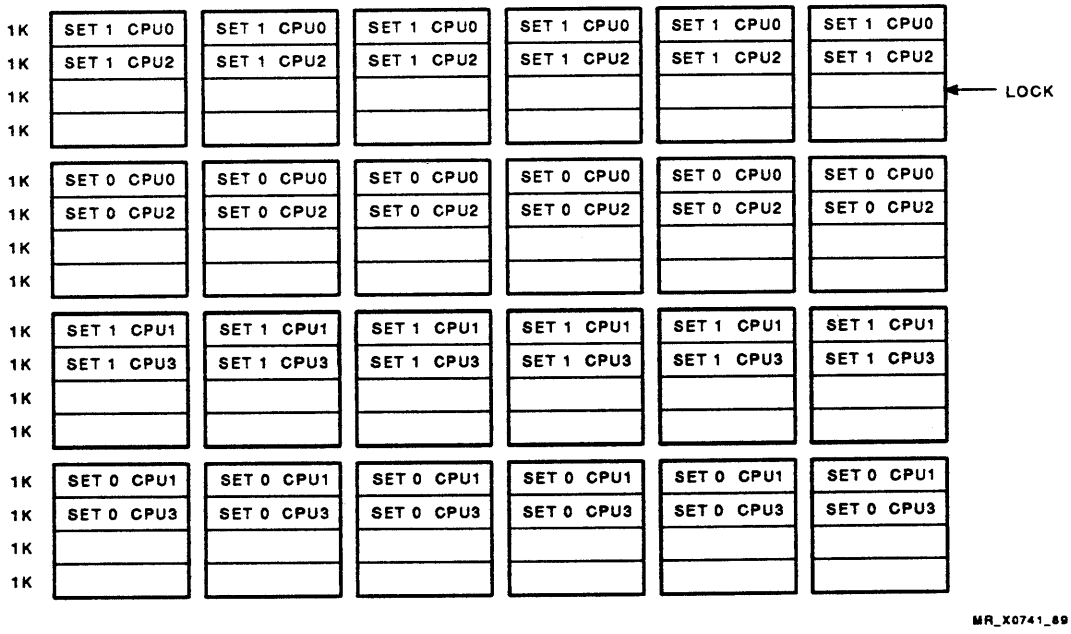


Figure 3–20 Lock Status Storage

3.11.1 Lock Status Bits

Figure 3–21 shows the lock status bits. The lock status bits are described in Table 3–12.

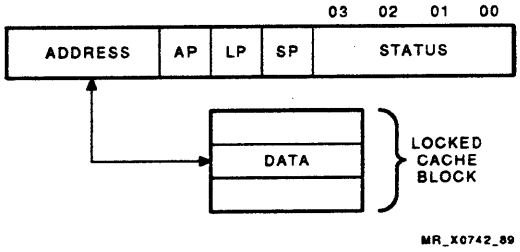


Figure 3–21 Lock Status Bits

Table 3–12 Lock Status Bit Description

Status Bits	Description
01:00	Indicate the cache status, as follows: 0 = Invalid 1 = Read 2 = Written partial 3 = Written full
02:00	Indicate which port is locked, as follows: 0 = Not reserved or locked 1 = ICU0 2 = Reserved 3 = ICU1 4 = CPU0 5 = CPU1 6 = CPU2 7 = CPU3
03	Reserved.

3.12 Global Tag Lookup for Lock Request

After a port wins arbitration, SCU addresses the global tag STRAMs, determining first where the most up-to-date copy of the data is and then whether any ports have a lock on that data. SCU determines the status of the requester and other CPU cache blocks (invalid, written full, written partial, or read) at that address.

When a port requests a read lock, the port sends a physical address to SCU. The SCU passes the physical address through the address crossbar and writes the address bits and lock status bits to the global tag STRAMs.

For lock requests, the typical global tag lookup requires two cycles to read the cache status for the eight cache sets (sets 0 and 1 for each CPU) and a third read cycle for the lock portion.

Figure 3–22 shows CPU0 sending a read refill lock request for a cache block that is in CPU1 and is written partial.

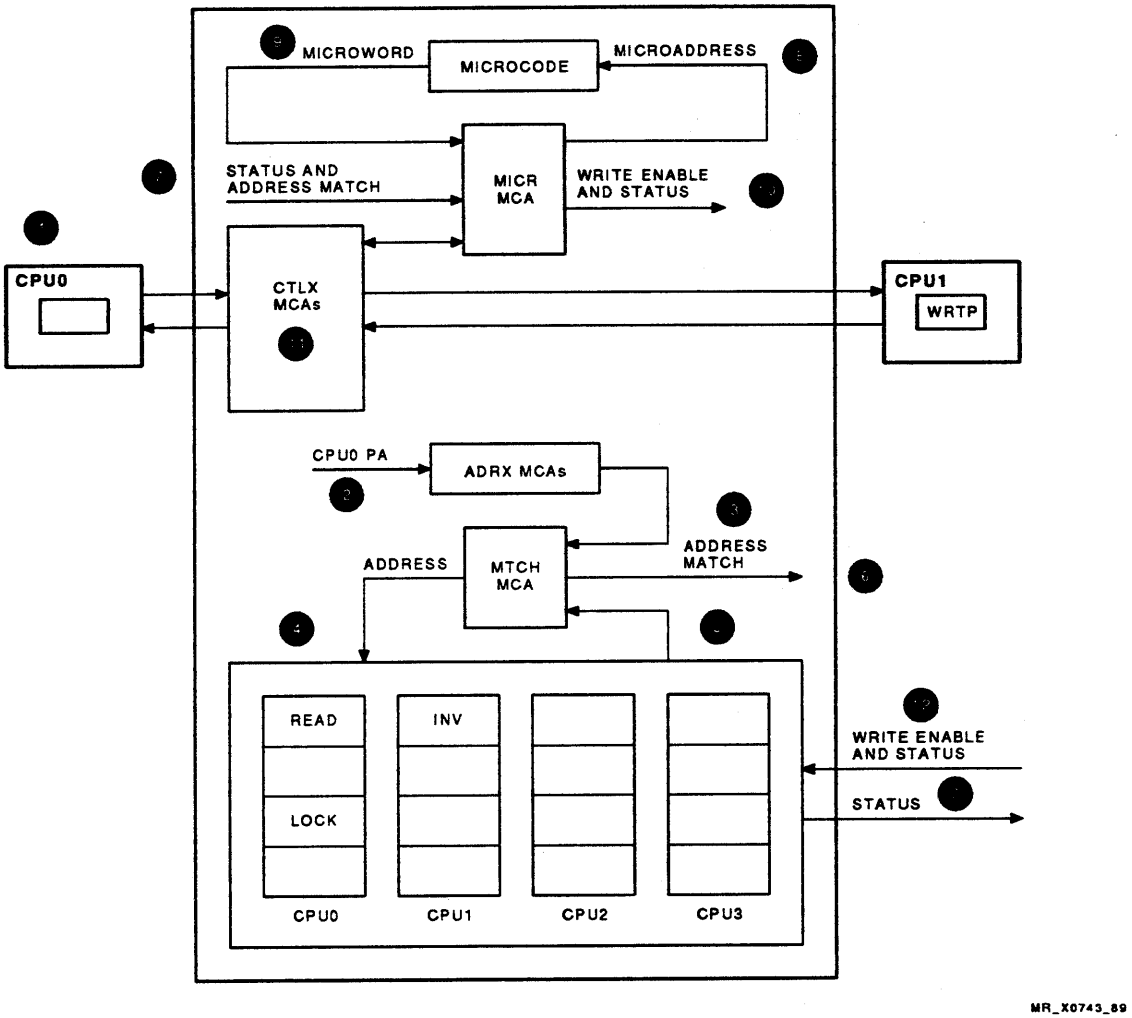


Figure 3–22 CPU Lock Request for a Block Written Partial

❶ CPU0 sends read refill, address, and which cache set, 0 or 1, is being refilled.

❷ ADRX selects CPU0 physical address.

❸ ADRX sends the CPU0 physical address to MTCH.

❹ MTCH addresses the global tag STRAMs.

❺ MTCH receives the address bits from the tag location.

❻ MTCH compares the CPU0 physical address with the tag contents. MTCH sends address match [03:00] to MICR.

❼ MICR receives the address match and tag status.

❽ MICR generates a microaddress to be sent to the microcode. SCU generates a microaddress using the lock information shown in Figure 3–23.

❾ The microword contains the tag status and memory command (write read). The microcode loads the fixup queue, which handles any inconsistency.

⑩ MICR generates write enable and tag status bits.

⑪ CTLX sends get data invalidate to CPU1. CPU1 sends the data. CTLX sends return data read to CPU0 with the data.

⑫ MTCH addresses the global tag STRAMs and provides the address bits. MICR writes the tag status bits for CPU0 as read, CPU1 as invalid, and the block in the CPU0 cache as locked.

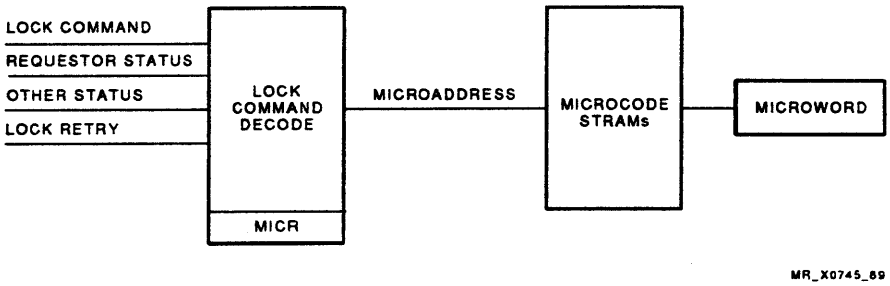


Figure 3–23 Generating a Microaddress for a Lock Request

3.12.1 Reading Lock Status

The MTCH MCA addresses the global tag STRAMs. MTCH receives 17 address bits, 1 address parity bit, and 1 location parity bit from each of 4 STRAMs. MICR receives 4 lock status bits and 1 status parity bit from each of 4 sets of global tag STRAMs. The 4 sets include lock status and three levels of lock reservation status.

3.12.2 Writing Lock Status

MTCH addresses the global tag STRAMs. MICR uses the tag status and tag write enable fields of the microcode to write the cache status and lock status into the global tag location.

3.13 Lock Request Timeouts

The CPU (EBox) requesting the lock either receives the data from the MBox or times out. If another CPU or I/O port fails to unlock the block that is now being requested, the CPU experiences a keep-alive failure when attempting to read lock that cache block. SPU polls the EBox. If the EBox is executing instructions, it responds to SPU at the end of the next instruction. If SPU receives no response, it declares that a keep-alive failure has occurred. A CPU times out 1000 cycles after the EBox makes a memory request and does not receive the data.

SCU does not track timeouts. SCU sends lock deny to the MBox if the cache block is locked or reserved. Receiving lock deny, the MBox restarts the command as a new request. XJA also restarts the command and sends a new request. SCU continues to update the global tag STRAMs with reserve lock status and grants the lock to the port with the highest reserve status at the time the lock becomes available.

If a DMA lock is not cleared within the time expected, SPU sends an SPU write unlock command to SCU. The SCU clears the lock status and accepts the lock request.

3.14 Lock Errors

SCU detects the following types of lock errors:

- **Lock error** — SCU detects a write unlock to an unlocked address. SCU forces the microcode to branch to a location to handle the lock error or send the lock request to an address already locked by that CPU.
- **Lock nonexistent memory** — SCU detects a port's attempt to lock an address in an area in memory that does not exist.

3.15 CPU Interlock Requests

A lock request can be a part of a read refill or write refill. The EBox issues a read lock command to the MBox, followed by a write unlock command, when executing an interlock instruction. The MBox translates the read lock command into a write refill request. The CPU (EBox) can lock only one cache block at a time. The MBox sends one of the following interlock requests to SCU:

CPU write refill lock
CPU write refill unlock
CPU write refill linked lock

Table 3–13 lists the status of the requester and other CPUs before and after a CPU write refill lock request.

Table 3–14 lists the status of the requester and other CPUs before and after a CPU write refill unlock request.

Table 3–13 Status of Tag STRAMs for CPU Write Refill Lock Request

Before		After	
Requester	**Other**	**Requester**	**Other**
Invalid	Invalid	Lock, written full	Invalid
Read	Invalid	Lock, written full	Invalid
Written partial	Invalid	Lock, written full	Invalid
Written full	Invalid	Lock, written full	Invalid
Read	Read	Lock, written full	Invalid
Invalid	Read	Lock, written full	Invalid
Invalid	Written partial	Lock, written full	Invalid
Invalid	Written full	Lock, written full	Invalid

Table 3–14 Status of Tag STRAMs for CPU Write Refill Unlock Request

Before		After	
Requester	**Other**	**Requester**	**Other**
Lock invalid	Invalid	Unlock, written full	Invalid
Lock read	Invalid	Unlock, written full	Invalid
Lock written partial	Invalid	Unlock, written full	Invalid
Lock written full	Invalid	Unlock, written full	Invalid
Lock read	Read	Unlock, written full	Invalid
Lock invalid	Read	Unlock, written full	Invalid
Lock invalid	Written partial	Unlock, written full	Invalid
Lock invalid	Written full	Unlock, written full	Invalid

Table 3–15 lists the status of the requester and other CPUs before and after a CPU write refill linked lock request.

Table 3–15 Status of Tag STRAMs for CPU Write Refill Linked Lock Request

Before		After	
Requester	**Other**	**Requester**	**Other**
Invalid	Invalid	Lock, written full	Invalid
Read	Invalid	Error	–
Written partial	Invalid	Error	–
Read	Read	Error	–
Invalid	Read	Lock, written full	Written full
Invalid	Written partial	Lock, written full	Invalid
Invalid	Written full	Lock, written full	Invalid

3.15.1 Lock Request — Cache Block Is Not Locked

The following sequence summarizes the CPU lock request when the block is not locked by any other processor or I/O device:

1. CPU0 sends the following:
   a. Read refill lock (MBXx_JBX_CMD_H[03:00])
   b. Address (MBXx_JBX_ADR_H[15:00])
   c. Cache set (MBXx_JBX_WCHSET_H[00]) to be refilled
   d. Load command (MBXx_JBX_LDCMD_H)
2. ADRX selects the CPU0 physical address and sends the address to MTCH, which compares the physical address with the contents of the cache consistency STRAMs containing the lock status and cache status. If the block in which the address resides is not locked and no cache conflicts exist, MICR writes the lock status bits into the lock status STRAMs and writes the appropriate cache status into the requester's tag STRAM location.
3. CTLX receives send data and the command mask and sends the following commands and information to CPU0:
   a. Return data read (JBX_MBXx_CMD_H[03:00])
   b. Cache set (JBX_MBXx_WCHSET_H[00]) to be refilled
   c. Load command (JBX_MBXx_LDCMD_H)
   d. Beginning of data (JBXX_MBXx_BOD_A, B, C, D)
   e. Refill data (JBX_MBXx_DAT_H[63:00])

3.15.2 Lock Request — Cache Block Is Locked

If the block is locked by another processor or I/O device, SCU sends lock denied to the port, and MICR writes reserve lock status bits into the reserve lock status STRAMs. MICR can write up to three lock reservations. Figure 3–24 shows how SCU writes the lock status and reservation status. The following sequence summarizes the process:

❶ SCU accepts the CPU0 lock request and writes a 3-bit code specifying that CPU0 has locked the cache block.

❷ When CPU3 sends a lock request for the same cache block, SCU examines the status of the block and finds that the lock is busy. SCU sends lock denied to the port and uses the first write cycle to write a reserve value specifying CPU3 in the global tag STRAMs.

❸ When CPU2 sends a lock request for the same cache block, SCU examines the status of the block, finds that the lock is busy, and uses the first write cycle to write a reserve value for CPU2 and CPU3 in the global tag STRAMs. The CPU3 reserve status shifts into the next reserve STRAM, and when CPU0 releases the lock, SCU accepts the CPU3 request before the CPU2 request.

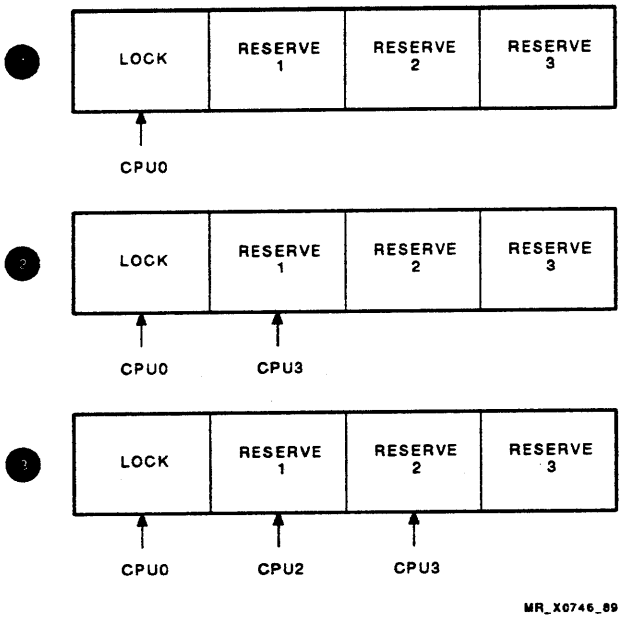


Figure 3–24 Lock Reservations

3.15.3 Lock Request — Cache Block Partially Written or Written Full

The following sequence summarizes a CPU lock request in which the block is written partial or written full in another CPU:

1. CPU0 sends read the refill lock, address, which cache set is to be refilled, and load command.
2. ADRX selects the CPU0 address and sends it to MTCH, which compares it with the contents of the cache consistency STRAMs. MTCH sends the address match (MTCH_MICR_ADR_MATCH_H[03:00]) to MICR. The cache tag STRAMs send status bits to MICR. The status of CPU0 is invalid, and the status of CPU1 is written partial.
3. CTLX sends get data invalidate to CPU1, which loads the write back buffer with the data (if the write back buffer is full, SCU receives data directly from the cache over the bypass path).
4. CTLX sends JBX_MBXx_SEND_DAT_H. CPU1 unloads the write back buffer and sends the data to DSXX.
5. The microcode sends a write read command to memory for a partially written block and sends a write pass for a written full block. For a write read, memory control merges the written partial block coming from CPU1 with valid memory data. For a write pass, SCU sends the data directly to the port first and then writes the data into main memory. CTLX sends the data to CPU0. MICR writes the lock status as locked for CPU0, cache status for CPU0 as read, and the cache status for CPU1 as invalid.

3.16 CPU Lock Acknowledge

When the MBox issues a lock request, the JBox can send:

- Return data if status is invalid or written partial
- OK to write if status is read
- Lock acknowledge if status is written full
- Lock deny if lock is not available

Lock acknowledge originates in the microword corresponding to the lock request. The JBox sends lock acknowledge when the MBox sends write refill lock or write refill linked lock, the lock is available, and the MBox has the block as written full. For a written partial block, merged data is returned to the MBox.

SCU updates the global tag with lock status and continues to update the tag STRAMs with reserve lock status for lock requests for the same address.

3.17 CPU Unlock Requests

CPU sends write refill unlock or write refill linked unlock to SCU. MICR clears the lock status. SCU releases the lock and accepts the lock request from the port with the highest level of lock reservation.

3.18 XJA Interlock Requests

XMI transactions that select SJA as the responder node are forwarded to ICU as DMA-type transactions. The DMA transactions can be reads, writes, read locks, or write unlocks.

3.18.1 Interlock Commands

The XJA sends one of the following interlock commands:

DMA read lock
DMA write unlock

Table 3–16 lists the lock and cache status before and after a DMA read lock request.

Table 3–17 lists the lock and cache status before and after a DMA write unlock request.

Table 3–16 DMA Read Lock Request — Cache Status

Before Read Lock Request	After Read Lock Request
Written full	Lock, read
Written partial	Lock, invalid
Read	Lock, read
Invalid	Lock, invalid

Table 3–17 DMA Write Unlock — Cache Status

Before Write Unlock Request	After Write Unlock Request
Lock, read	Unlock, invalid
Lock, invalid	Unlock, invalid
Lock, written full	Unlock, invalid
Lock, written partial	Unlock, invalid

3.18.2 Lock Request — Cache Block Is Not Locked

XJA sends the DMA read lock to SCU to lock a cache block. Figure 3–25 shows the DMA lock request and the following steps summarize the request:

1. XJA sends XJA_COMDAVAIL_H to SCU and sends DMA read lock and the address (XJA_DAT_H[15:00]).
2. ADRX selects the XJA physical address and sends the address to MTCH, which compares the physical address with the contents of the global tag STRAMs containing the lock status and the cache status.
3. If the block in which the address resides is not locked and no cache conflict exists, MICR writes the lock status bits into the lock status portion of the global tag STRAMs.
4. ICU sends ICU_COMDAVAIL_H, the return read lock data command, and the address (ICU_DAT_H[15:00]) to XJA.
5. XJA sends XJA_CLKx_H[02:00] and unloads the output buffer. After XJA receives the data, it sends XJA_BUFEMPTD_H to SCU.

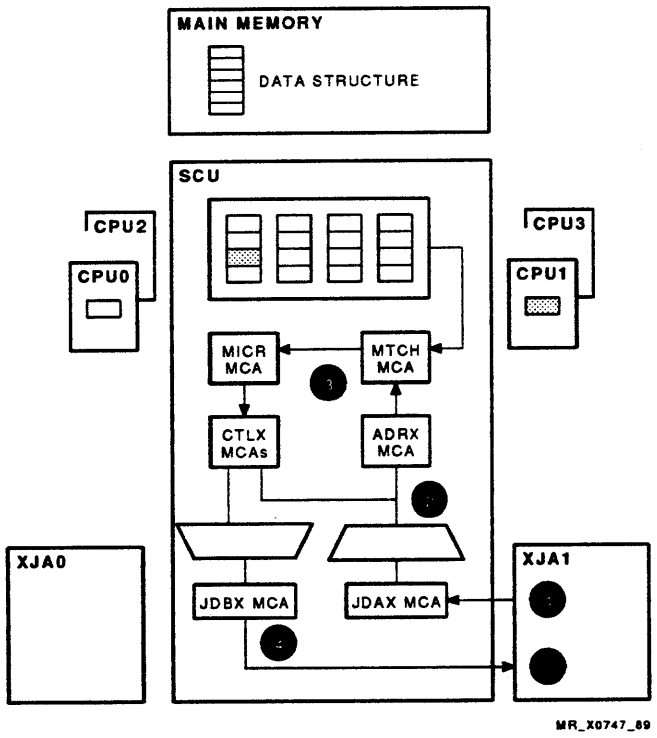


Figure 3–25 XJA Lock Request

3.18.3 Lock Request — Cache Block Is Locked

If the block is locked by another processor or I/O device, MICR writes reserve lock status bits into the reserve lock status STRAMs. MICR can write up to three lock reservations. Figure 3–24 shows how SCU writes the lock status and reservation status. Figure 3–26 shows the lock denied response.

❶ XJA sends DMA read lock to SCU.

❷ ADRX sends the XJA physical address to MTCH.

❸ MTCH compares the contents of the global tag STRAMs to the XJA physical address and sends the results to MICR. MICR receives the cache and lock status from the tag STRAMs. MICR uses the cache status, lock status, and command to generate a fork address for the microcode (lock busy).

❹ CTLX decodes the microword fields and sends lock deny to XJA.

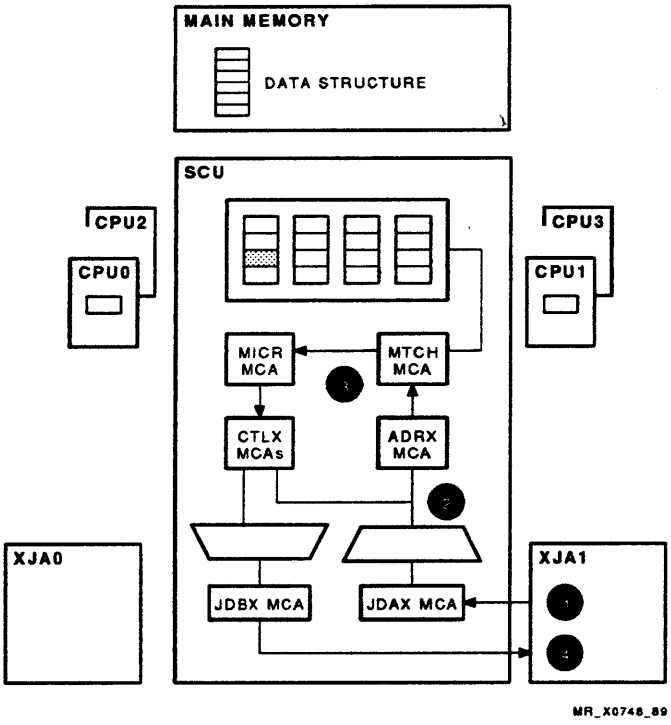


Figure 3–26 XJA Deny Lock

3.18.4 Lock Request — Cache Block Partially Written or Written Full

The following sequence summarizes an XJA lock request in which the block is written partial or written full in the cache of another CPU:

1. XJA sends XJA_CMDAVAIL_H and DMA read lock (XJA_DAT_H[15:00]).
2. ADRX selects the XJA address and sends the address to MTCH, which compares the XJA address with the contents of the global tag STRAMs. MTCH sends an address match (MTCH_MICR_ADR_MATCH_H[03:00]) to MICR. The cache tag STRAMs send status bits to MICR. MICR finds that the cache block requested by XJA is partially written or written full and is locked by the CPU.
3. CTLX sends a get data invalidate command to the CPU that owns the cache data.
4. The microcode sends a write read command to memory for a partially written block, and sends write pass for a fully written block.

   For a write read, memory merges the written partial block from the CPU with valid memory data. For a write pass, SCU sends the data directly to the XJA, and then writes the data into main memory. ICU sends the data to XJA. MICR writes the lock status as locked and the cache status for the CPU as invalid.

3.19 XJA Unlock Requests

XJA sends DMA write unlock and address (XJA_DATA_H[15:00]) to SCU. MICR clears the lock status. SCU releases the lock and accepts the lock request from the port with the highest lock reservation.

3.19.1 Cycles

A DMA read request requires four cycles: three read cycles and one lock status cycle. A DMA write unlock also requires four cycles. Table 3–5 lists the DMA read request cycles.

3.20 SPU Interlock Requests

SPU sends an interlock request when it needs to manipulate a queue for communication between the operating system and SPU. The SPU can also release a memory lock that was obtained for an I/O device that has since failed.

3.20.1 Interlock Commands

The SPU sends the following interlock commands:

SPU read lock
SPU write unlock

Table 3–18 lists the lock and cache status before and after an SPU read lock request.

Table 3–19 lists the lock and cache status before and after an SPU write unlock request.

Table 3–18 SPU Read Lock Request — Cache Status

Before Read Lock Request	After Read Lock Request
Written full	Lock, read
Written partial	Lock, invalid
Read	Lock, read
Invalid	Lock, invalid

Table 3–19 SPU Write Unlock — Cache Status

Before Write Unlock Request	After Write Unlock Request
Lock, read	Unlock, invalid
Lock, invalid	Unlock, invalid
Lock, written full	Unlock, invalid
Lock, written partial	Unlock, invalid

3.20.2 Lock Request — Cache Block Is Not Locked

SPU sends an SPU read lock command to SCU to lock a cache block. Figure 3–27 shows the SPU lock request. The following steps summarize the request:

❶ SPU sends SPU_JBOX_BUF_RQST_H, command (SPU read lock), and address bytes on SPU_CTLD_DATA_H[07:00] to SCU.

❷ The JBox, receiving the SPU command and address, sends the following:

a. The SPU physical address sent to ADRX, which selects the SPU physical address and sends the address to MTCH. The MTCH compares the physical address with the contents of the global tag STRAMs containing the lock status and the cache status.

b. The SPU command is sent to CTLX.

❸ If the block in which the address resides is not locked and no cache conflict exists, MICR writes the lock status bits into the lock status STRAMs and writes the appropriate cache status into the global tag STRAMs of the CPUs.

❹ All CCU MCU-to-SPU commands go to ICU first. The data and command use the same wires. SCU sends JBOX_SPU_CLK, the return read lock data command, data, and an address on JBOX_SPU_DATA_H[07:00].

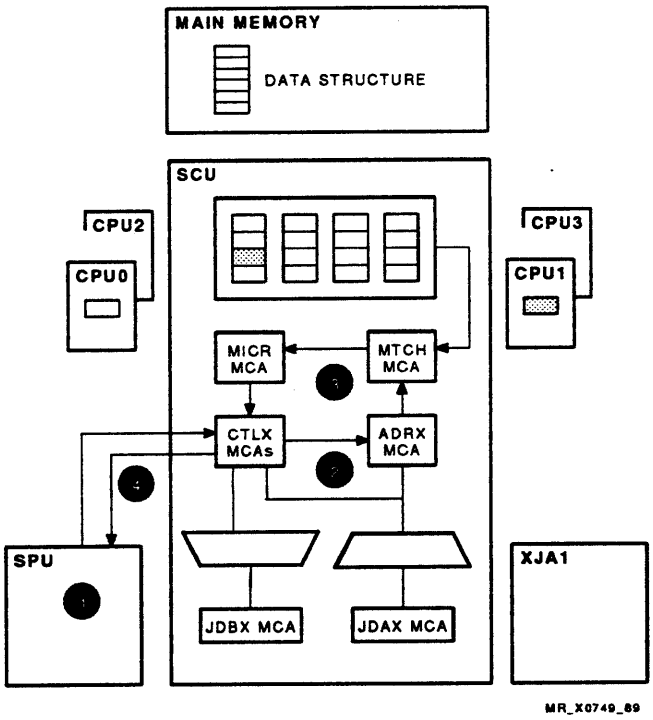


Figure 3–27 SPU Lock Request

3.20.3 Lock Request — Cache Block Is Locked

If the block is locked by another processor or I/O device, MICR writes reserve lock status bits into the reserve lock status STRAMs. MICR can write up to three lock reservations. Figure 3–24 shows how SCU writes the lock status and reservation status. Figure 3–28 shows the lock denied response.

❶ SPU sends the SPU read lock command to SCU (CTLD).

❷ ADRX sends the SPU physical address to MTCH.

❸ MTCH compares the contents of the global tag STRAMs to the SPU physical address and sends the results to MICR. The MICR receives the cache and lock status from the tag STRAMs. MICR uses the cache status, lock status, and command to generate a fork address for the microcode (lock busy).

❹ CTLX decodes the microword fields and sends a lock deny command to SPU.

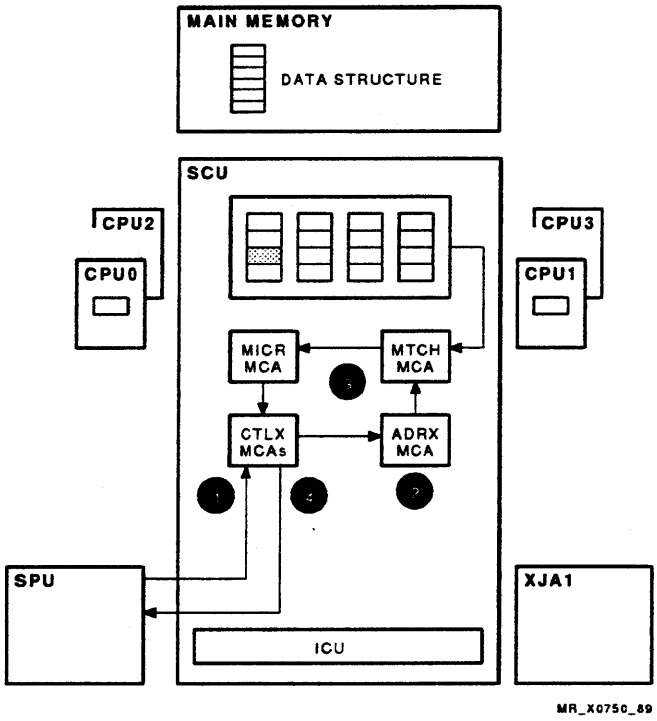


Figure 3–28 Deny SPU Lock

3.20.4 Lock Request — Cache Block Partially Written or Written Full

The following sequence summarizes an SPU lock request in which the block is written partial or written full in a CPU cache:

1. SPU sends the command and address on SPU_CTLD_DATA_H[07:00].
2. ADRX selects the SPU address and sends the address to MTCH, which compares the SPU address with the contents of the global tag STRAMs. MTCH sends an address match (MTCH_MICR_ADR_MATCH_H[03:00]) to MICR. The cache tag STRAMs send status bits to MICR. MICR finds that the cache block that SPU needs is partially written or fully written and locked by the CPU.
3. CTLX sends get data invalidate to the CPU that owns the cache data.
4. The microcode sends a write read command to memory for a partially written block and sends a write pass command for a written full block.

   For a write read, memory merges the written partial block coming from CPU with valid memory data. For a write pass, SCU sends the data directly to ICU and then writes the data into main memory. (The JBox sends the data to SPU.) MICR writes the lock status as locked and the cache status for the CPU as invalid.

3.21 SPU Unlock Requests

SPU sends a SPU write unlock command and address (XJA_DATA_H[15:00]) to SCU. MICR clears the lock status. SCU releases the lock and accepts the lock request from the port with the highest lock reservation, if any are pending.

3.21.1 Cycles

An SPU read and unlock requires four cycles: three read cycles and one lock status cycle. Table 3–5 lists the DMA lock cycles.

4
Micromachine Control

This chapter describes the micromachine control. It describes the physical features of the control store STRAMs array and covers control store space allocation, addressing, and data format.

This chapter describes how the control store is loaded at initialization and how it is usually accessed. A detailed description of the microword field definitions is included. The structure and organization of the microcode listing are described, including microcoding examples of symbolic, hex, and bit encoding. This chapter also explains how to analyze information using bit field description tables.

4.1 Overview

The micromachine consists of the control store STRAMs array and the MICR MCA.

4.2 JBox Control Store

This section describes how the JBox loads and addresses the control store STRAMs. It identifies the fields of the microword. It describes how the JBox distributes the microword bits and checks parity.

4.2.1 Control Store STRAMs

The control store consists of a fifteen 1K × 4-bit STRAMs array that stores one thousand 60-bit microwords. The control store also includes a three 1K × 4-bit STRAMs array, which stores microword addresses of the last 1K microwords addressed by the micromachine. Figure 4–1 shows the control store STRAMs array.

4.2.1.1 Space Allocation

Figure 4–2 shows the control store space allocation.

Each CPU and ICU port request, such as a CPU read refill or a DMA read, has eight entries into the control store and uses the base addresses listed in Table 4–1.

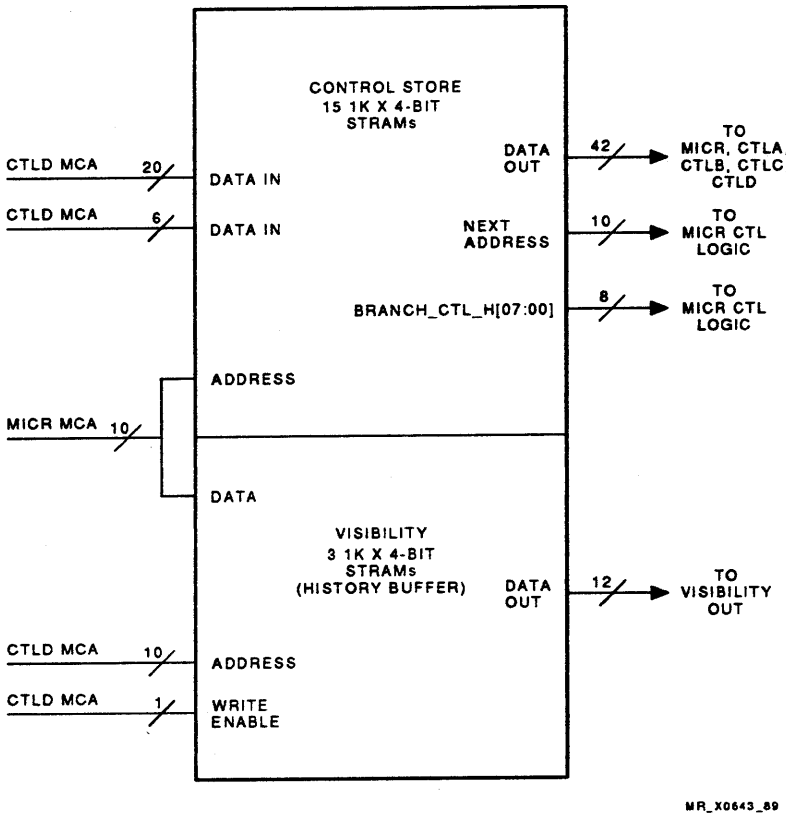


Figure 4–1 Control Store STRAMs Array

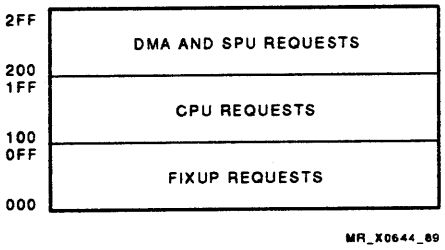


Figure 4–2 Control Store Space Allocation

Table 4–1 Base Addresses

Address	Request
000	Idle
001–0FF	Addressable by fixup queue only
100	CPU read refill
110	CPU write refill
120	CPU read refill linked
130	CPU write refill linked
140	CPU write refill lock
150	CPU write refill unlock
160	CPU write back
170	CPU longword write update
180	CPU read I/O register
190	CPU write back link
200	DMA read
210	DMA read lock
230	SPU write unlock
240	DMA write
250	DMA write unlock
260–270	Illegal DMA
280	SPU read I/O register
290	SPU write I/O register
2A0–2F0	Illegal DMA

For example, in the case of a CPU read refill, the microcode has nine fork addresses that use 100 as a base address. Figure 4–3 shows the CPU read refill fork addresses.

- Four addresses, 100, 105, 106, and 107, correspond to the status combinations listed in Table 4–2.
- Six addresses correspond to the error conditions listed in Table 4–3.

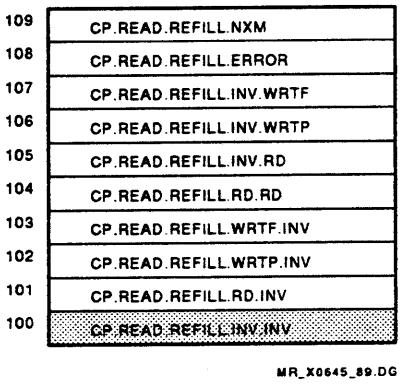


Figure 4–3 Base Address — 100

Table 4–2 Cache Status Combinations for Microaddresses

Address	Requester	Other CPUs
107	Invalid	Written full
106	Invalid	Written partial
105	Invalid	Read
100	Invalid	Invalid

Table 4–3 Error Entries

Address	Error	Description
109	NXM	NPAMM detects nonexistent memory for the address that was sent with the request.
108	ERROR	An MBox error has occurred. There are two types of MBox errors: more than one CPU has the same cache block with written status or the lock bit is already set for that address and port.
104	RD.RD	The CPU sends a read refill request for a cache block that already has read status.
103	WRTF.INV	The CPU sends a read refill request for a cache block that already has written full status.
102	WRTP.INV	The CPU sends a read refill request for a cache block that already has written partial status.
101	RD.INV	The CPU sends a read refill request for a cache block that already has read status.

4.2.2 Control Store Data

The CTLA, CTLB, CTLC, CTLD, and MICR MCAs latch the control store data as shown in Figure 4–4.

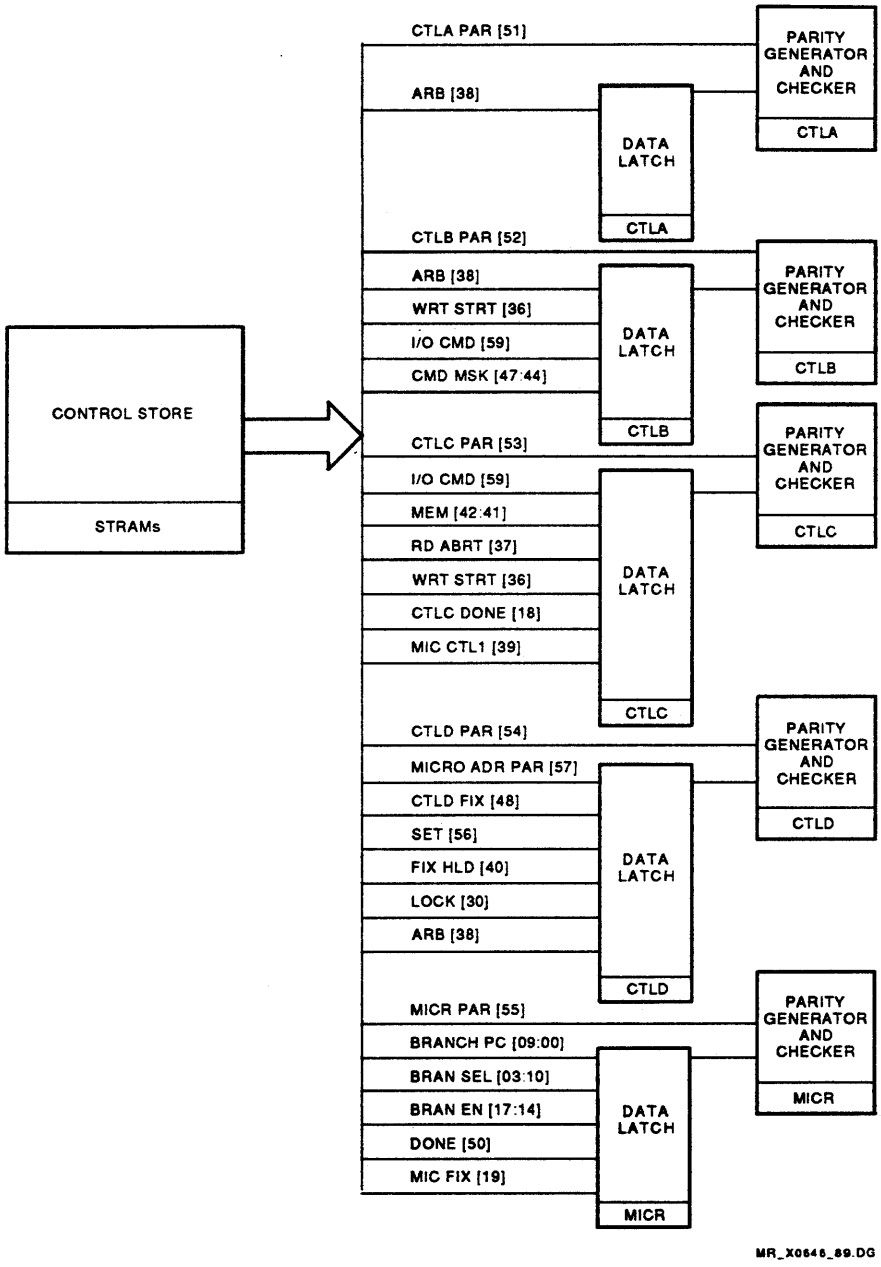


Figure 4–4 Control Store Data Latch and Parity Checking

4.2.3 Control Store Parity Checking

The CTLA, CTLB, CTLC, CTLD, and MICR MCAs perform the parity checking for the control store data shown in Figure 4–4. Each microword contains the following six 1-bit parity fields that the MCAs use when latching the control store data:

CTLA parity
CTLB parity
CTLC parity
CTLD parity
MICR parity
Microaddress parity

4.2.4 Control Store Addressing

MICR_FORK_SEL_H[01:00] selects one of the following sources of microaddresses. Figure 4–5 shows the microaddress selection.

- **MICR branch address [09:00]** — From the results of the branch select and enable microword fields, MICR samples the logic signals for available resources or status and uses the result to modify the least significant bits of the microaddress.
- **MICR fork fix command [09:00]** — The microcode loads one of three fixup queues with a fix command. The fixup queue uses the fix command bits to form the low-order bits of the microaddress. The MICR fork fix command [09:00] addresses the fixup microcode in the lower section of the control store.
- **MICR fork address [09:00]** — MICR receives the tag queue data, match results, and the tag STRAMs cache and lock status bits. MICR then forms a fork address into the microcode.

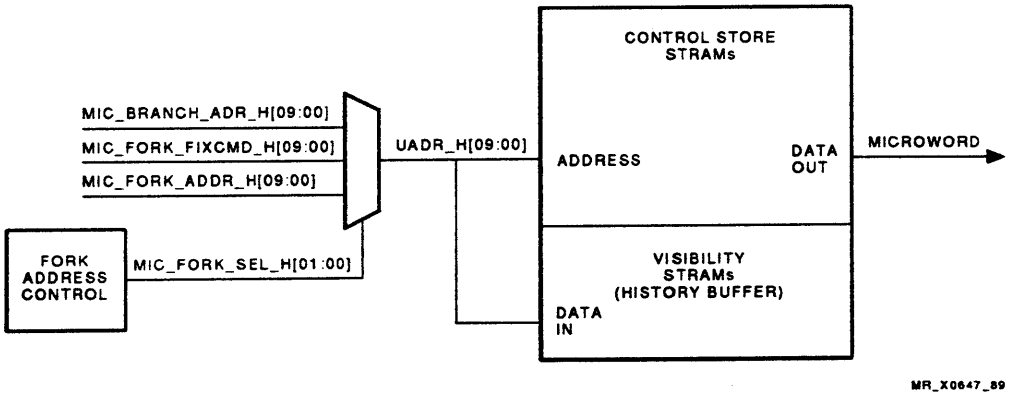


Figure 4–5 Microaddress Sources

4.2.4.1 Branch Address

The branch enable, select, and PC fields of the microword determine the next microaddress. If the enable bits are set, the MICR branch logic tests the selected signal and branches to the microaddress specified in the branch PC field. The microcode uses the branch field logic to handle a fourth request that requires a fixup microcode (three fixups are in progress).

4.2.4.2 MICR Address

For each port request, the JBox places an entry into the tag queue in the CTLD MCA. The MICR MCA addresses the control store for each entry in the tag queue and increments the input pointer for the next entry. If the MICR sends a fork address for a microword indicating a fixup is required, MICR loads one of three fixup queues with a fix command. The microcode flow returns to idle or branches to another microword.

The microcode and fixup queues alternate sending addresses to the control store. In this way, more than one request can be in progress. When the microcode flow returns to idle, MICR takes the next tag entry and addresses the control store, while the fixup queue handles the previous request.

MICR uses the tag STRAMs status bits, MTCH results, and CTLD tag queue data to form the microaddress. During a tag lookup, MICR receives status bits from the tag STRAMs, match from MTCH MCA, and CTLD_MICR_QDATA_H[15:00] from the tag lookup queue.

MICR uses the tag status to form the low-order bits of the microaddress and uses the queue data to form the high-order bits. Figure 4–6 shows the queue data.

Table 4–4 lists the tag queue data fields and their descriptions.

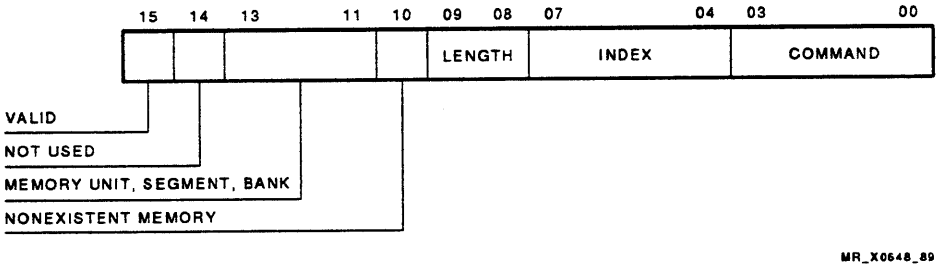


Figure 4–6 Tag Queue Data

Table 4–4 Tag Queue Data Bit Descriptions

Bit	Name	Description
03:00	Command	Holds the command for the port that has won arbitration.
07:04	Index	Identifies the port and its command buffer for the request.
09:08	Length	Specifies the data size for the data transfer to or from memory (as indicated by the command field).
10	Nonexistent memory	Indicates that NPAMM has detected nonexistent memory for the address latched in the ADRX address hold latches.
13:11	Memory unit	Identifies the memory unit and segment as indicated by MPAMM.
14	–	Not used.
15	Valid	Indicates that the tag queue has a valid entry.

CTLD sends CTLD_MICR_QDATA_H[15] one cycle after the tag lookup begins. MICR addresses the control store after every tag lookup. MICR reads QDATA[03:00] (command) and QDATA[04:07] (index) to determine how many cycles the tag lookup takes, as well as the type of request. Due to the STRAM cycle time, the match signal and status bits do not arrive until two cycles after the tag lookup.

When the tag lookup is complete, MICR generates a fork address for a microword that specifies load the fixup queue, return to idle or branch.

4.2.4.3 Fixup Queue Address

The fixup queue loads the fix command of the previous microword to form the low-order bits of the microaddress that handles the fixup. Requests that require fixup include the following cases:

- A CPU has a cache block, written full or partial, that is needed by another CPU. The JBox must get the data, send it to the requester, and update the tag STRAMs.
- One of two CPUs, sharing the same cache block with read status, wants to write to the cache block. The JBox must change the status of both CPUs' cache blocks from read to written full and invalid.
- NPAMM has detected nonexistent memory or a nonexistent device for either a CPU or DMA request, respectively.

The fixup queue interrupts the usual flow of microaddressing. (The microcode returns to idle after each microword is addressed so that microaddresses for different requests in progress can be sent to the control store.) The fixup queue forms a microaddress using the bits listed in Table 4–5. The fixup microwords never initiate a branch and force the microcode to idle.

Table 4–5 Fixup Microaddress

Bit	Name	Description
04:00	Next address	The fixup queue uses the fix command field of the previous microword to form the 5-bit next address field of the microcode. The fixup queue accesses 001 through 0FF of the control store for fixup handling.
05	Last CPU	This bit indicates that the last CPU, in a series of at least two CPUs, has a cache block that must be invalidated. If more than one CPU cache must be invalidated, this bit remains 0, until the last CPU. 0 = Not the last CPU, another CPU has a cache block that must be invalidated. 1 = This is the last CPU.
06	Request CPU	This bit indicates that the CPU involved in the fixup is the requesting CPU. 0 = Not the requesting CPU. 1 = Requesting CPU.
07	DMA request	This bit indicates that the request is a DMA request. 0 = Not a DMA request. 1 = A DMA request.

4.2.5 Control Store Loading

Figure 4–7 shows how the control store is loaded at system initialization.

❶ CTLD receives SCAN_DATA_IN from SPU and sends the control store data to the data input of the control store STRAMs array. CTLD also latches the write enable as scan data and sends it to the control store.

❷ BRANCH_ADR[09:00] from the MICR control logic determines the the microaddress for the STRAMs. The addresses to the control store are also sent to the data inputs of the visibility STRAMs.

❸ Using a counter, CTLD addresses the three 1K × 4-bit visibility STRAMs and sends write enable. As the MICR MCA generates the microaddress, the visibility STRAMs store the microaddress. The visibility STRAMs store the last 1K microaddresses.

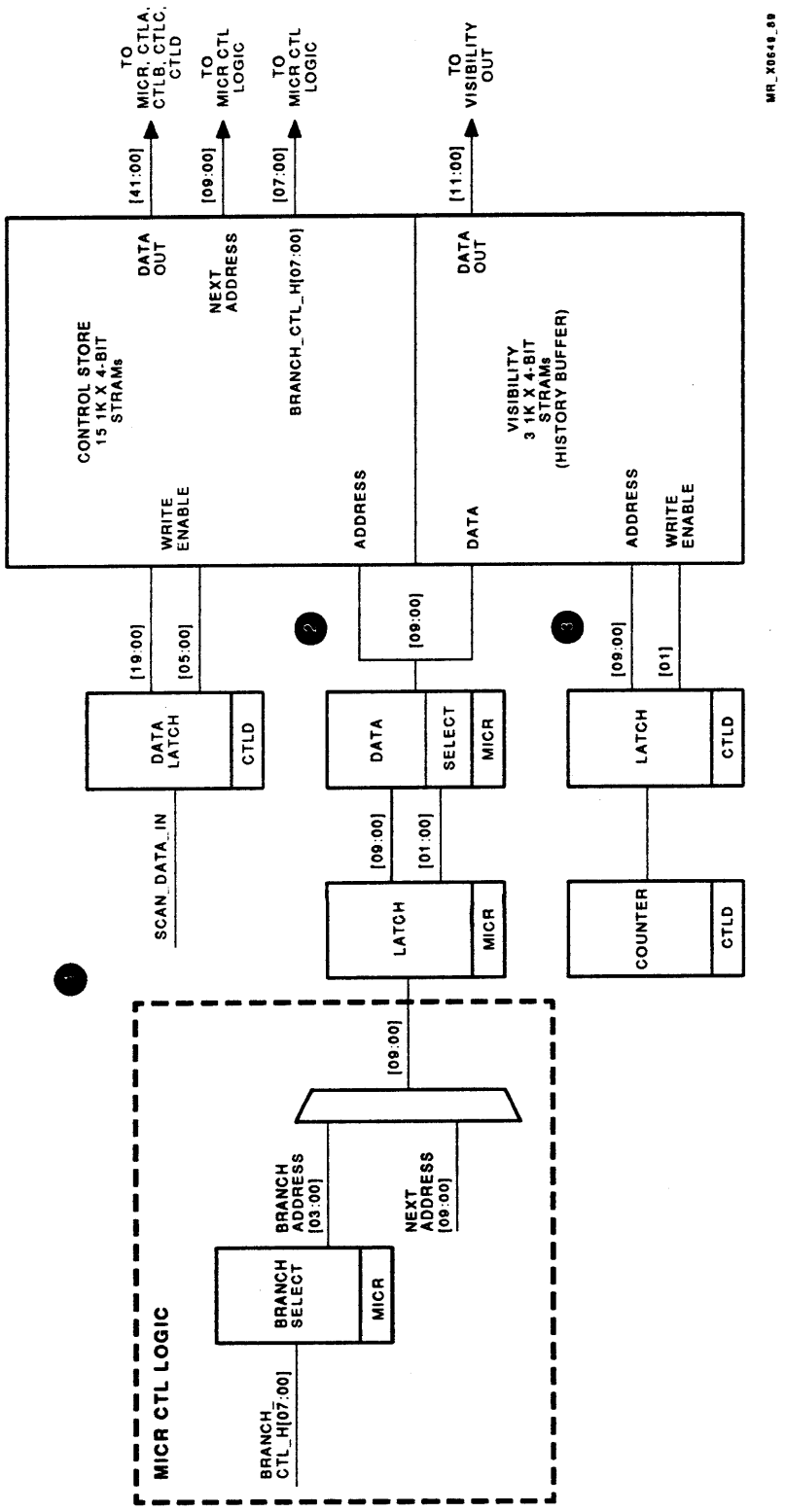


Figure 4–7 Loading the Control Store

4.3 MICR MCA

The MICR MCA contains the microcontrol logic. Figure 4–8 shows the MICR MCA block diagram. The MICR MCA contains the following:

❶ **Tag STRAMs cache status latch and decode** — The MICR latches MIC_REQ_STATn_H[01:00] and MIC_OTHn_STATn_H[01:00] from the tag STRAMs. MICR uses the status bits to determine the status of the requester and to determine the status of each of the other CPUs for the address corresponding to the request.

❷ **Tag STRAMs lock status latch and decode** — The MICR MCA latches and decodes the command to determine whether the lock bit should be set or cleared, or whether an error has occurred. MICR MCA uses the status bits to determine the lock status.

❸ **MTCH MCA tag match results latch and decode** — The MICR MCA latches and decodes MIC_MATCHn_H[03:00]. MTCH MCA compares the port's physical address with the address stored in the tag STRAMs to determine whether a match has occurred. MTCH sends the match results to MICR.

❹ **Error latch and decode** — The MICR MCA detects cache status, lock status, and parity errors. MICR checks parity across:

DSCT_AVAIL_H[31:00]
MIC_REQ_STATn_H[01:00]
MIC_OTHn_STATn_H[01:00]
MIC_MATCHn_H[03:00]
CTLC_RSRV_H[32:00]
CTLD_QDATA_H[15:00]

❺ **Tag cycle and fork address control** — The MICR MCA tag cycle and fork address control generate MIC_LOOKUP_H and MIC_CYC_INC_H[03:00] to control the tag lookup and tag status write cycles and MIC_FORK_SEL_H[01:00] to control the microaddress selection.

❻ **Available resources latch and decode** — The MICR MCA latches and decodes available resources.

- **PRTRDY_H[09:00]** — The MICR MCA decodes the port ready status and determines if the I/O, requester, memory, or other port command buffer is available.
- **DSCT_AVAIL_H[32:00]** — The MICR MCA decodes the data switch control lines and determines whether the data paths, source and destination, are available.
- **CTLC_USB_H[02:00]** — The MICR MCA decodes the unit, segment, and bank control lines to determine if the memory segment is available.

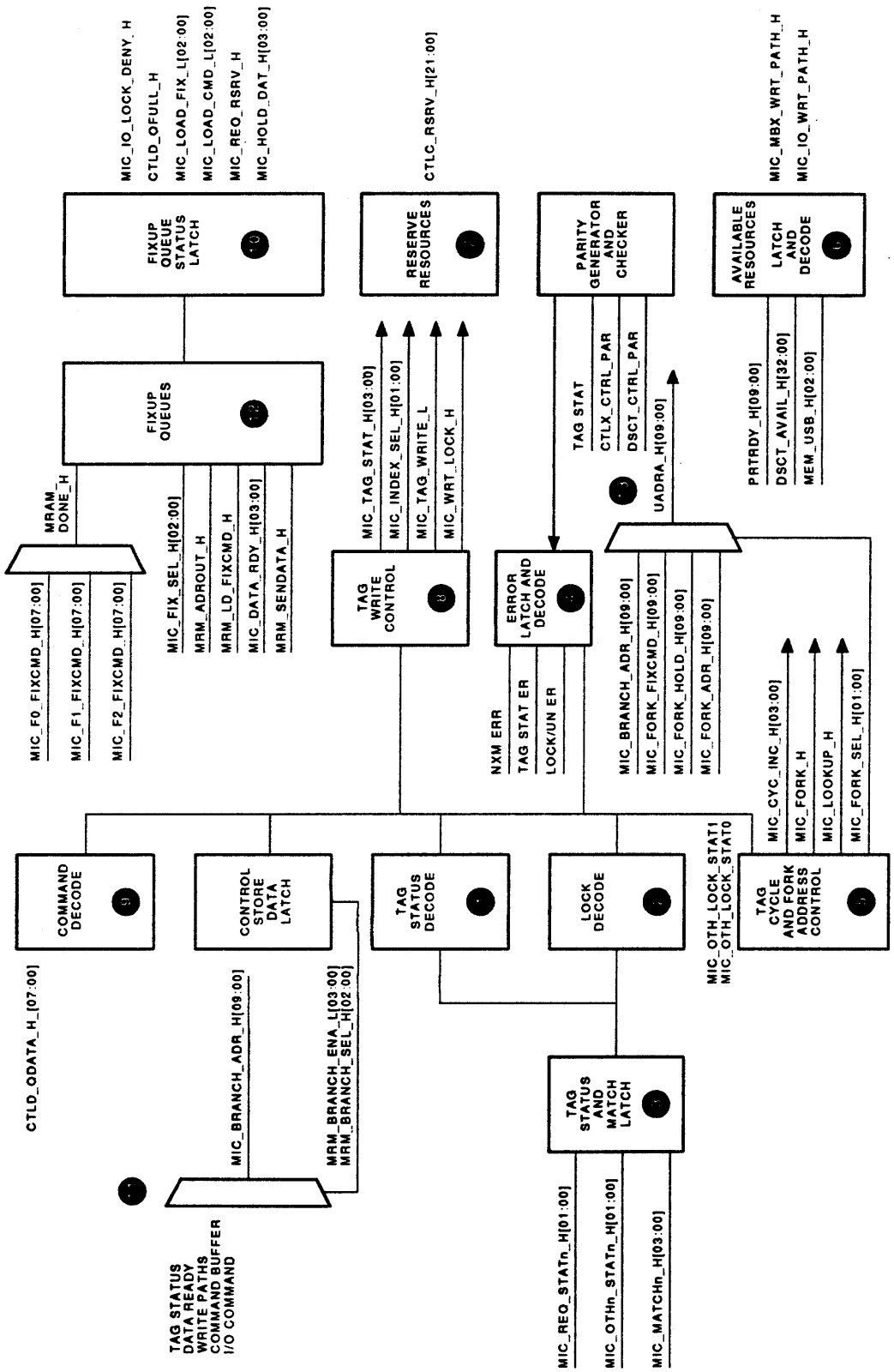


Figure 4–8 MICR MCA Block Diagram

❼ **Reserve latch and decode** — The MICR MCA determines if the data path is in use and sends CTLC_RSRV_H[21:00] to CTLC reserve resources logic.

❽ **Tag status latch and write enable** — The MICR MCA decodes the command to determine its type. If the command is a lock command, the MICR MCA reads the lock status and determines if there is a lock or unlock error, or if the lock is busy. MICR sends this information to the tag status latch and write enable logic. MICR generates MIC_TAG_STAT_H[03:00] (tag status), MIC_TAG_WRITE_L (tag STRAMs write enable), MIC_INDEX_SEL_H[01:00] (CPU 0, 1, 2, or 3), and MIC_WRT_LOCK_H (write lock status).

❾ **Command decode** — The MICR MCA latches CTLD_QDATA_H[07:00], MIC_VAL_CMD_H[03:00], and MIC_VAL_INDEX_H[03:00], decodes the command, determines if the command is valid, and generates MIC_FORK_ADR_H[09:04].

❿ **Fixup queue status latch** — The MICR MCA has three fixup queues. MICR latches and decodes MRM_LD_FIXCMD_H (load the fixup queue), MRM_FIXUP_CMD_H[03:00] (fixup command), and MIC_FIX_SEL_H[02:00] (fixup queue 0, 1, or 2).

The MICR MCA also latches MRAM_DONE_H (clear fixup queue), MRM_ADROUT_H (reserve the address out resources), MIC_B_DATA_RDY_H[03:00] (port has data in its output buffer), and MRM_SENDATA_H (port has received send data).

⓫ **Branch address, select, and enable latch and decode** — The MICR MCA latches and decodes (determines the microword address for) the branch address, select, and enable fields of the microword and checks parity across these fields.

⓬ **Three fixup queues** — The MICR MCA has three fixup queues that handle cache inconsistencies, nonexistent memory errors, and lock busy and lock deny requests. The fixup queue latches the fix command field from the microword and generates a microaddress to address the fixup microcode in the control store.

⓭ **Microaddress generator** — The MICR MCA selects one of the following microaddresses to the control store STRAMs array:

- MIC_BRANCH_ADR_H[09:00]
- MIC_FORK_FIXCMD_H[09:00]
- MIC_FORK_ADR_H[09:00]

4.4 Microword Definition

Most of the logic within SCU is controlled directly or indirectly by bits within the microword. All functions executed by SCU are controlled by generating specific microwords.

4.4.1 Microword Format

The JBox microword is 60 bits wide. Figure 4–9 shows the format of the microword.

Branch PC [09:00] — This 10-bit field is sent to the MICR MCA and is used to determine the next address in the microcode flow. It specifies the base address of the next microword to be accessed. This address may be modified by the microcontrol logic as a result of the next two fields. Table 4–8 lists the branch select and enable values and their descriptions.

Branch select [13:10] — This 4-bit field is sent to the MICR MCA and selects a bit to sample. Table 4–6 lists the values and their descriptions. Table 4–8 lists the branch select and enable values and their descriptions.

Table 4–6 Branch Select [13:10]

Value	Name	Description
0	Status	Reserved.
1	Branch select	Identifies and selects a logical bit to determine its status.

Branch enable [17:14] — This 4-bit field is sent to MICR to enable the current microword to sample the state of specific logic signals and modify the low-order bits of the next microaddress. The next microaddress is based on the state of the selected signals. Table 4–7 lists the values and their descriptions. Table 4–8 lists the branch select and enable values and their descriptions.

Table 4–7 Branch Enable [17:14]

Value	Name	Description
00	None	The branch select is not enabled.
0F	All	Reserved.

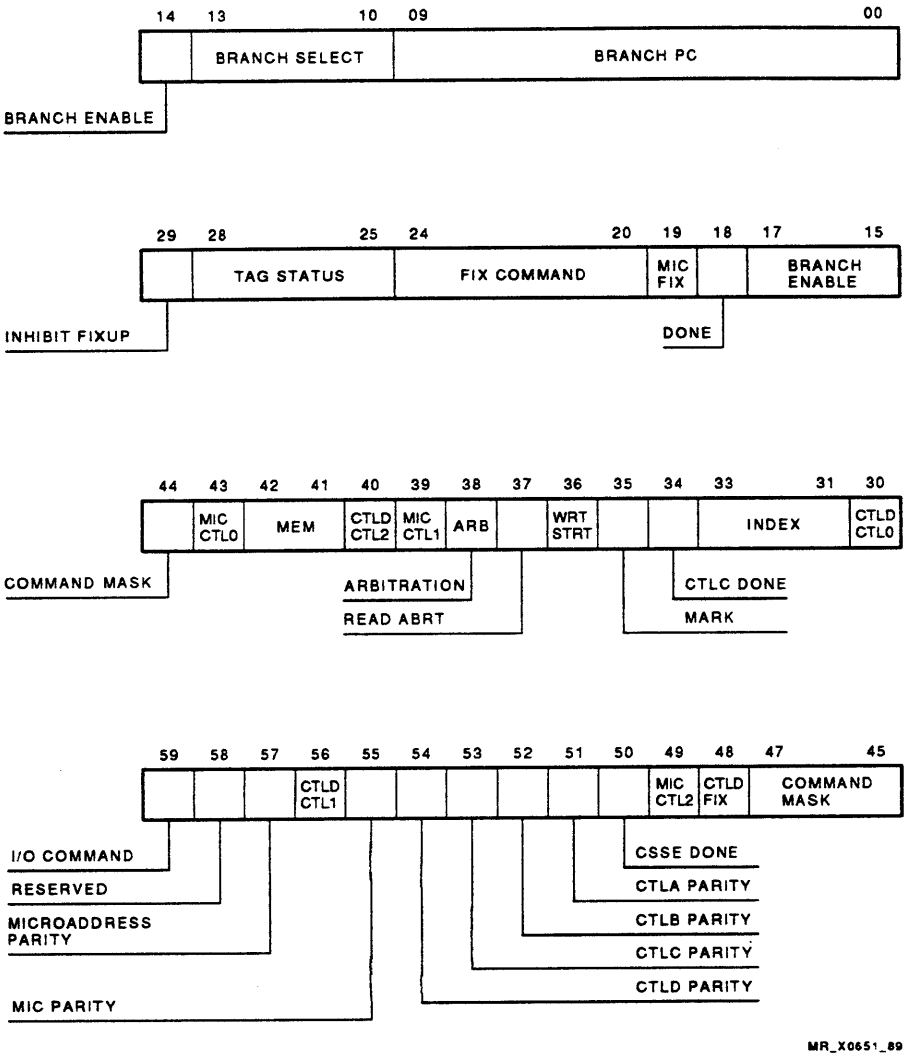


Figure 4–9 Microword Format

Branch select and branch enable [17:10] — This 8-bit field is sent to MICR and is used for forks and branching. Table 4–8 lists the branch select and enable values and their descriptions.

Table 4–8 Branch Select and Enable [17:10]

Value	Name	Description
10	Read done	Determines if memory has return data available.
11	Request write path and memory	Determines if the requester's write path and memory segment are available.
12	Address out or data ready	Determines if the address path is available or if the port has sent data ready.
13	Other command buffer	Determines if the other port's command buffer is available to receive a command.
14	Memory write path	Determines if the memory write path is available.
15	Memory command buffer	Determines if the memory command buffer is available to receive a command.
16	Request command buffer	Determines if the requester's command buffer is available to receive a command.
17	I/O command buffer	Determines if the I/O port command buffer is available to receive a command.
20	DMA command	Determines if the request is an I/O command.
38	DMA status	Determines the DMA status.
41	Request write path	Determines if the requester's write path is available.
70	Status	Determines the status of the request.
82	Write pass path	Determines if the write pass path is available.
84	Other write path	Determines if the other write path is available.
80	Memory unit number	Determines the memory unit number.
86	Request read path	Determines if the requester's read path is available.
E1	Index	Determines if the index information is available.
F8	Request status	Determines if the request status is available.
F9	Other 0 status	Determines if the other 0 status is available.
FA	Other 1 status	Determines if the other 1 status is available.
FB	Other 2 status	Determines if the other 2 status is available.
FC	Request lock status	Determines if the requester lock status is available.
FD	Other 0 lock status	Determines if the other 0 lock status is available.
FE	Other 1 lock status	Determines if the other 1 lock status is available.
FF	Other 2 lock status	Determines if the other 2 lock status is available.

Done [18] — This 1-bit field is sent to the MICR MCA to indicate that this is the last cycle of the flow. Done allows MICR to start reading the status for the next request. The microcode is going to the idle state. Table 4–9 lists the values for this field and the corresponding descriptions. The next cycle is idle. Tag control increments the input pointer to the tag queue. This bit also controls the fixup queue.

Table 4–9 Done [18]

Value	Name	Description
0	No	Not the last cycle of the flow. The request cannot be retired. The fixup queue sends additional microaddresses to handle the fixup.
1	Yes	Last cycle of the flow. The MICR can obtain another entry from the tag queue and load the fixup queue with a fix command (if applicable).

MIC fix [19] — This 1-bit field is sent to the MICR MCA to indicate that the fixup queue is active. When MIC fix [19] is first set in the flow, it indicates that the fixup queue is to be loaded. When bit 19 is again set in the flow, it indicates that the fixup queue is to be cleared. Table 4–10 lists the values for this field and the corresponding descriptions.

Table 4–10 MIC Fix [19]

Value	Name	Description
0	No	Do not load or clear the fixup queue. Fixup queue is not active.
1	Yes	Load or clear the fixup queue (depending on where the microword is located in the fixup flow). Fixup queue is active.

Fix command [24:20] — This 5-bit field is sent to the MICR MCA and holds the fix command that is loaded into one of three fixup queues. The fix command field is used to encode the low-order address bits of the *next* fixup microword to perform the fix command in a *previous* microword. Table 4–11 lists the fix command values and their descriptions.

Table 4–11 Fix Command [24:20]

Code	Command	Description
0	Clear	Clears the fixup queue.
1, 2	Reserved	Reserved for no tag fixes.
3	Write retire	Loads the memory command buffer with a write command, sends write start to the memory segment controller and CPU send data, and retires the request.
4, 5, 6, 7	Reserved	Reserved for the send data command.
9, A, B	Reserved	Reserved for the get data command.
0A	Write invalidate	Stops arbitration, writes invalid status, places the invalidate command into the CPU's command output buffer, and sends the address latched in ADRX for the request to the CPU.
0C	Invalidate	Stops arbitration, loads the CPU command output buffer with invalidate, sends the address latched in ADRX for the request, and writes invalid status.
0D	OK to write	Stops arbitration, loads OK to write into the CPU command output buffer, sends address, and writes written partial status.
0E	Lock deny	Denies lock request.
0F	Lock acknowledge	Stops arbitration, loads port command output buffer with lock acknowledge, sends address, and retires the request.
10	Write data ready	Samples the CPU port's data ready line and loads the fixup queue with the write send data command for the next fixup microaddress.
11	Write read data ready	Samples the CPU's port data ready and loads the fixup queue with the write read send data command for the next fixup microaddress.
12	Write pass data ready	Samples the CPU's port data ready line and loads the fixup queue with the write pass data ready command for the next fixup microaddress.
13	DMA write data ready	Samples the I/O port's data ready line and loads the fixup queue with the DMA write send data command for the next fixup microaddress.
14	Write send data	Stops arbitration, loads the memory command output buffer with the write command, sends the CPU send data, sends write start to the memory segment controller, and retires the request.

Table 4–11 (Cont.) Fix Command [24:20]

Code	Command	Description
15	Write read send data	Stops arbitration, loads the memory command output buffer with the write read command, sends the CPU send data, sends write start to the memory segment controller, and retires the request.
16	Write pass send data	Stops arbitration, loads the memory command output buffer with the write pass command, sends the CPU send data, sends write start to the memory segment controller, and retires the request.
17	DMA write send data	Stops arbitration, loads the memory command output buffer with the write command, sends the I/O port send data, sends write start to the memory segment controller, and retires the request.
18	Write invalidate	Stops arbitration, writes invalid status, loads get data invalidate into the CPU command output buffer, sends the address, and loads the fixup queue with the write data ready command for the next fixup microword address.
19	Write read invalidate	Stops arbitration, writes invalid status, loads get data invalidate into the CPU command buffer, and loads the fixup queue with the write read data ready command for the next fixup microword address.
1A	Write pass invalidate	Stops arbitration, writes invalid status, loads get data invalidate into the CPU command output buffer, and loads the fixup queue with the write pass data ready command for the next fixup microword address.
1B	Write pass read	Stops arbitration, writes read status, loads get data read into CPU command output buffer, and loads the fixup queue with the write pass data ready command for the next fixup microword address.
1C	Invalidate write	Stops arbitration, loads the fixup queue with the invalidate write command for the next fixup microword address, writes invalid status, and loads the port command output buffer with invalidate.
1D	Read written	Stops arbitration, loads the fixup queue with the OK to write command for the next fixup microword address, and loads the command output buffer with invalidate.
1E	Memory read nonexistent memory	Loads the port command output buffer with the memory read NXM command, sends address, and retires the request.
1F	I/O read nonexistent memory	Loads the port command output buffer with the I/O read NXM command, sends address, and retires the request.

Tag status [28:25] — In this 4-bit field, which is sent to the MICR MCA, [28] is the write enable, [27:25] is lock status, and [26:25] is cache status. If [28] is asserted, MICR updates the tag STRAMs as soon as the lookup in progress has finished. The MICR MCA writes the tag status bits into the cache set block as indicated by the index field [33:31]. Table 4–12 lists the tag status values and their descriptions. Table 4–13 lists the tag status [28] and index [33] values and their descriptions.

Table 4–12 Tag Status [28:25]

Value	Name	Description
0	NOP	No operation.
8	Invalid or unlock	Indicates invalid when writing cache status and indicates unlock when writing lock status.
9	Read	Indicates read when writing cache status.
A	WRTP or IO0	Indicates written partial when writing cache status and indicates IO0 when writing lock status.
B	WRTF or IO1	Indicates written full when writing cache status and indicates IO1 when writing lock status.
C	CPU0	Indicates CPU0 when writing lock status.
D	CPU1	Indicates CPU1 when writing lock status.
E	CPU2	Indicates CPU2 when writing lock status.
F	CPU3	Indicates CPU3 when writing lock status.

Table 4–13 Writing Tag Status

Tag [28]	Index [33]	Description
0	0	No write.
0	1	Index [32:31] defines the CPU. Use the MIC_LOCK_ STATUS to write the lock status.
1	0	Tag [27:25] is written. Index [32:31] is not used.
1	1	Tag [27:25] is written using index [32:31]. 0 = Write RAM0 1 = Write RAM1 2 = Write RAM2 3 = Write RAM3

Inhibit fixup queue [29] — This 1-bit field inhibits fixups but does not stop lookups. This is used in two-cycle operations. Table 4–14 lists the values for this field and the corresponding descriptions.

Table 4–14 Inhibit Fixup Queue [29]

Value	Name	Description
0	No	Do not inhibit fixups.
1	Yes	Inhibit fixup in the next cycle.

CTLD CTL0 [30] — This 1-bit field works with CTLD CTL1 [56] and CTLD CTL2 [40]. Table 4–15 lists the values for this field and the corresponding descriptions.

Table 4–15 CTLD Control Bits [40, 56, 30]

CTLD CTL2 [40]	CTLD CTL1 [56]	CTLD CTL0 [30]	Description
0	0	0	NOP.
0	0	1	INC SET. Indicates an interrupted read, which can resume from where it left off when the microcode is finished. Writes the inverse of the set number that MICR sends to CTLD.
0	1	0	INC LOCK. Indicates an interrupted read, which can resume from where it left off when the microcode is finished. Writes lock status in the section of the tag STRAMs that is reserved for lock bits.
0	1	1	INC. Indicates an interrupt read, which can resume from where it left off when the microcode is finished.
1	0	0	Unused.
1	0	1	HOLD SET. Indicates an interrupt read, which must start over when the microcode is finished. Writes the inverse of the set number that MICR sends to CTLD.
1	1	0	HOLD LOCK. Indicates an interrupt read, which must start over when the microcode is finished. Writes lock status in the section of the tag STRAMs that is reserved for lock bits.
1	1	1	HOLD. Indicates an interrupt read, which must start over when the microcode is finished.

Index [33:31] — This 3-bit field indicates in which CPU tag location, requester or other, to write the tag status field [28:25]. If [33] = 0, then [32:31] indicates the following: 0 = all CPUs with valid status, 1 = requester only, 2 = fixup queue (refresh), 3 = fixup queue increment (refresh minus the selected or taken CPU). If [33] = 1, [32:31] indicates the following: write 0 = RAM0, 1 = RAM1, 2 = RAM2, and 3 = RAM3. Table 4–16 lists the values for this field and the corresponding descriptions. Table 4–13 lists tag status [28] and index [33] values and their descriptions.

Table 4–16 Index [33:31]

Value	Name	Description
0	Requester	Uses the index field to identify the command buffer and port.
1	Fix	Uses the index field to identify the command buffer and port.
2	–	Reserved.
3	–	Reserved.
4	Lock	Sets or clears the bit using the index field to identify the command buffer and port.
5	Reserve 1	Sets or clears the bit using the index field to identify the command buffer and port.
6	Reserve 2	Sets or clears the bit using the index field to identify the command buffer and port.
7	Reserve 3	Sets or clears the bit using the index field to identify the command buffer and port.

CTLC done [34] — This 1-bit field is sent to the memory segment controllers in the CTLC MCA. Table 4–17 lists the values of this field and the corresponding descriptions. Each of the four memory segment controllers uses [34] as a qualifier for write start [36] and read abort [37]. The microword sends either [36] or [37], along with [34], to indicate the status of the memory operation.

The memory segment controller uses CTLC to do the following:

- Retire the request.
- Determine if the tag status has been written.
- Allow arbitration to continue.
- Determine if fixup is done (if applicable).

Table 4–17 CTLC Done [34]

Value	Name	Description
0	No	Request is still in progress. The port command buffer and ADRX address latch still hold the command and address.
1	Yes	Done. Retires the request.

Mark [35] — Reserved.

Write start [36] — This 1-bit field is sent to the CTLC MCA. CTLC receives [36] and CTLC done [34] to indicate the beginning of a memory write operation. The memory segment controller sends this information to MMCX for DRAM to send RAS, CAS, and write enable when ADRX sends row and address bits to the memory array. Table 4–18 lists the values for this field and the corresponding descriptions. Bit [36] is set for longword updates and DMA write requests and forces the JBox to send a get data command.

Table 4–18 Write Start [36]

Value	Name	Description
0	NOP	No operation.
1	Yes	Starts memory write operation.

Read abort [37] — This 1-bit field is sent to the CTLC MCA when the JBox must stop a read that has already started. CTLC receives [37] and CTLC done [34] to indicate that the JBox has detected a cache inconsistency or parity error. If [37] = 1, the JBox does not perform a get data. The requester already has the cache block and wants to change the status from read to written (write refill). The JBox responds with OK to write with no data transfer. Table 4–19 lists the values for this field and the corresponding descriptions.

Table 4–19 Read Abort [37]

Value	Name	Description
0	NOP	No operation.
1	Yes	Aborts memory read operation.

Arbitration [38] — This 1-bit field controls port arbitration. The microcode and startup logic share resources (address and data paths). If [38] is set, the microcode controls the address and data paths (source and destination). When the microcode completes the request, it releases the address and data paths by resetting [38] in the microword. Table 4–20 lists the values for this field and the corresponding descriptions.

Table 4–20 Arbitration [38]

Value	Name	Description
0	OK	Continues arbitration.
1	Stop	Stops arbitration.

MIC CTL1 [39] — This 1-bit field works with MIC CTL0 [43] and MIC CTL2 [49]. Table 4–21 lists the values for this field and the corresponding descriptions.

Table 4–21 MIC Control Bits [49, 43, 39]

MIC CTL2 [49]	MIC CTL1[39]	MIC CTL0 [43]	Description
0	0	0	NOP
0	0	1	Requesters command buffer
0	1	0	CPU send data
0	1	1	Clear data ready
1	0	0	Address out
1	0	1	Error
1	1	0	I/O send data
1	1	1	Tag status [03]

CTLD CTL2 [40] — This 1-bit field works with CTLD CTL 0 [30] and CTLD CTL 1 [56]. Table 4–15 lists the values for this field and the corresponding descriptions.

Memory [42:41] — This 2-bit field is sent to the memory segment controllers in CTLC. Table 4–22 lists the values for this field and the corresponding descriptions.

Table 4–22 Memory [42:41]

Value	Name	Description
0	NOP	No operation.
1	OK	Indicates the request status. The microword sends OK when a memory operation is in progress that allows the memory operation to continue until completed.
2	Abort	Indicates the request status. The microword sends abort when a memory operation is in progress to stop the memory operation.
3	–	Reserved.

MIC CTL0 [43] — This 1-bit field works with MIC CTL1 [39] and MIC CTL2 [49]. Table 4–21 lists the values for this field and the corresponding descriptions.

Command mask [47:44] — This 4-bit field is sent to CTLB to form the command field of the port's command interface. The JBox samples the port buffer's available lines before loading the command [47:44] into the port's output buffer. Table 4–23 lists the command mask values and their descriptions.

Table 4–23 Command Mask [47:44]

Value	Name	Description
0	Get data write	Loads the CPU command output buffer with the get data write command.
1	Get data read	Loads the CPU command output buffer with the get data read command.
2	Get data invalidate	Loads the CPU command output buffer with the get data invalidate command.
5	Read to written	Loads the CPU command output buffer with the OK to write command.
6	Invalidate	Loads the CPU command output buffer with the invalidate command.
7	SPU read nonexistent memory	Loads the I/O command output buffer with the SPU NXM command.
9	Lock acknowledge	Loads the port command output buffer with the lock acknowledge command.
A	Memory read nonexistent memory	Loads the CPU command output buffer with the memory read NXM command.
B	I/O read nonexistent memory	Loads the I/O command output buffer with the I/O read NXM command.
C	Lock deny	Loads the port command output buffer with the lock deny command.
D	Write read	Loads the memory port output command buffer with the write read command.
E	Write	Loads the memory port output command buffer with the write command.
F	Write pass	Loads the memory port output command buffer with the write pass command.

CTLD fix [48] — This 1-bit field notifies the CTLD not to send queue data to MICR. Bit [48] is set when the fixup queue is loaded and is set again when the fixup queue is cleared. Table 4–24 lists the values for this field and the corresponding descriptions.

Table 4–24 CTLD Fix [48]

Value	Name	Description
0	No	Do not send tag queue data to MICR.
1	Yes	Send tag queue data to MICR.

MIC CTL2 [49] — This 1-bit field works with MIC CTL0 [43] and MIC CTL1 [39]. Table 4–21 lists the values for this field and the corresponding descriptions.

CSSE done [50] — This 1-bit field is another copy of the done bit and is sent to the CSSE connector. Table 4–9 lists the values for this field and the corresponding descriptions.

CTLA parity [51] — This 1-bit field contains the odd parity sent to the CTLA MCA for the microbits listed in Table 4–25. CTLA uses [51] to check the parity across the control store data.

Table 4–25 CTLA Parity [51]

Bit	Field
38	ARB

CTLB parity [52] — This 1-bit field contains the odd parity sent to the CTLB MCA for the microbits listed in Table 4–25. CTLB uses [52] to check the parity across the control store data. Table 4–26 lists the values for this field and the corresponding descriptions.

Table 4–26 CTLB Parity [52]

Bit	Field
38	ARB
36	WRT STRT
59	I/O command
47:44	Command mask

CTLC parity [53] — This 1-bit field contains the odd parity sent to the CTLC MCA for the microbits listed in Table 4–27. CTLC uses [53] to check the parity across the control store data.

Table 4–27 CTLC Parity [53]

Bit	Field
59	I/O command
42:41	MEM
37	Read abort
36	Write start
18	CTLC done
39	CTL1

CTLD parity [54] — This 1-bit field contains the odd parity sent to the CTLD MCA for the microbits listed in Table 4–28. CTLD uses [54] to check the parity across the control store data it latches.

Table 4–28 CTLD Parity [54]

Bit	Field
57	ADR PAR
48	CTLD fix
56	Set
40	Fix hold
30	Lock
38	ARB

MIC parity [55] — This 1-bit field contains the odd parity sent to the MICR MCA for the microbits listed in Table 4–29. MICR uses [55] to check the parity across the control store data.

Table 4–29 MIC Parity [55]

Bit	Field
13:10	Branch select
17:14	Branch enable
9:0	Branch PC
50	Done
19	MIC fix

CTLD CTL1 [56] — This 1-bit field works with CTLD CTL0 [30] and CTLD CTL2 [40]. Table 4–15 lists the values for this field and the corresponding descriptions. If (MICR_SET = 1 and MRM_SET = 0) or (MICR_SET = 0 and MRM_SET = 1), then set 0 = 1. If lock, set 1 = 1.

Microaddress parity [57] — This 1-bit field is the parity across the microaddress of this microword.

Reserved [58].

I/O command [59] — This 1-bit field is sent to CTLB and CTLC and indicates if the microword is for an I/O request. Table 4–30 lists the values for this field and the corresponding descriptions.

Table 4–30 I/O Command [59]

Value	Name	Description
0	No	Not an I/O command.
1	Yes	I/O command.

4.4.2 Microcoding Examples

This section examines microcoding examples and shows how to analyze the information contained in the JBox microcode listing. The bit description tables in this chapter support the examples. Each example includes the following:

- Request definition
- Flowchart
- Symbolic encoding
- Hex encoding
- Bit encoding

4.4.2.1 CPU Read Refill — No Fixup Required

In this example, CPU0 sends a read refill to JBox for the cache block starting at address 1000. The global tag STRAMs' status indicates that this block is invalid in all four CPU caches. The following steps summarize the read refill request operation:

1. CPU0 sends a load command, read refill command, address 1000, cache set 0 in the first cycle, and in the second cycle, the remainder of address 1000.
2. MICR latches and decodes the tag STRAM status bits, MTCH MCA match results, and CTLD tag queue data. MICR then generates microaddress 100.
3. The microword specifies a memory read from the memory unit, segment, and bank identified by MPAMM. It also specifies which address in the address latches in ADRX is to generate the row and column address bits to the memory array cards.
4. The microword specifies to write the tag status as read. MICR writes the tag status for the cache block status for address 1000 as CPU0 = read; CPU1, 2, and 3 = remain invalid.

Figure 4–10 shows the microcode flow for the read refill request. Example 4–1 shows the symbolic encoding for the read refill request.

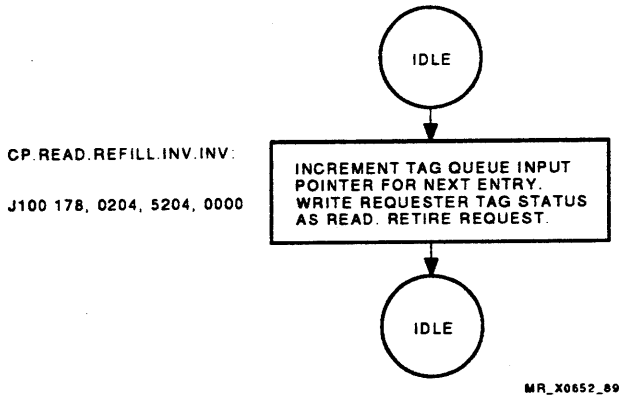


Figure 4–10 Read Refill Flow

```
HEX ENCODING:  J100  178,0204,5204,0000

 SYMBOLIC ENCODING:

 CP.READ.REFILL.INV.INV:   INC,
                           TAG STAT/RD,
                           CTLC DONE/YES,
                           GO TO [IDLE]
```

Example 4–1 Read Refill Symbolic Encoding

The symbolic encoding can be interpreted as follows:

- **MEM/OK** — Continue the memory read operation.
- **INC** — The tag is being written. Increment the tag queue input pointer for the next entry. If not a tag write, the CTLD tag queue input pointer is incremented automatically.
- **TAG STAT/RD** — Write read tag status for CPU0.
- **CTLC DONE/YES** — Retire the request.
- **GO TO [IDLE]** — Return to the idle state.

The bit encoding is shown in Table 4–31.

Table 4–31 Microaddress 100 CPU Read Refill Microword Bits

Bit	Name	Value	Description
09:00	Branch PC	00 0000 0000	–
13:10	Branch select	0000	–
17:14	Branch enable	0000	NOP.
18	Done	1	Indicates that MICR can obtain another request from the tag queue.
19	MIC fix	0	Does not load the fixup queue.
24:20	Fix command	0 0000	NOP.
28:25	Tag status	1001	Writes read status using the requester's index.
29	Inhibit fixup	0	Does not inhibit fixups.
30	CTLD CTL0	1	Allows a read in progress to resume from where the microcode left off.
33:31	Index	000	–
34	CTLC done	1	Retires request.
35	Mark	0	–
36	Write start	0	NOP.
37	Read abort	0	NOP.
38	ARB	0	Does not stop arbitration.
39	MIC CTL1	0	NOP.
40	CTLD CTL2	0	Allows a read in progress to resume from where the microcode left off.
42:41	MEM	01	Status OK.
43	MIC CTL0	0	NOP.
47:44	Command mask	0000	NOP.
48	CTLD fix	0	Sends tag queue data to MICR.
49	MIC CTL2	0	NOP.
50	CSSE done	0	–
51	CTLA parity	1	–
52	CTLB parity	1	–
53	CTLC parity	1	–
54	CTLD parity	1	–
55	MIC parity	0	–
56	CTLD CTL1	1	Allows a read in progress to resume from where the microcode left off.
57	ADR parity	0	–
58	–	–	Reserved.
59	I/O command	0	Signifies that this is not an I/O request.

4.4.2.2 CPU Read Refill — With Fixup Required

In this example, the cache block with the starting address 1000 has written full status in the CPU0 global tag STRAM and invalid status for CPU1, CPU2, and CPU3. CPU1 sends a read refill to the JBox for the cache block. The CPU1 tag STRAM status for location 1000 is changed from invalid to read. The CPU0 tag STRAM status for location 1000 is changed from written full to read. The following steps summarize the read refill request operation when a fixup is required:

1. CPU1 sends a load command, read refill command, address 1000, and cache set 0 in the first cycle, and in the second cycle, the remainder of address 1000.
2. MICR latches and decodes the tag STRAM status bits, MTCH MCA match results, and CTLD tag queue data. MICR then generates microaddress 107.
3. The microword at 107 specifies that a fixup is required, aborting the memory read operation that has already started (memory read is already in progress to optimize performance). The cache tag status informs MICR that more recent data is in the CPU0 cache and loads the fixup queue with the write pass read command. The fixup queue uses the fix command to form the low-order bits of the next microword address, 03B. The microword also specifies that the CPU1 tag status is to be written as read.
4. The microword at 03B specifies that arbitration is to be stopped, the CPU0 status is to be written as read, and a get data read command is to be sent to CPU0 with the address latched in ADRX. When CPU0 needs to write to the cache block (now with read status), it sends a write refill command. The JBox sends CPU0 an OK to write command and changes the cache status from read to written. The JBox also loads the fixup queue with a write pass data ready command to form the next microword address, 032.
5. The microword at 032 samples the data ready line from CPU0 and sends a write pass send data fix command to the fixup queue. CPU0 loads the write back buffer and sends the JBox data ready.
6. The microword at 036 specifies that the memory port is to receive a write pass memory command. The JBox unloads CPU0's cache block at address 1000, passes it to the CPU1 data switch lines, and then writes the data into memory. The microcode needs the data paths (source and destination) and stops port arbitration (until the data transfer is complete). The tag queue input pointer is incremented to point to the next entry in the tag queue. The fixup queue is cleared and the CTLC retire logic retires the request.

   A memory write is performed using the memory unit, segment, and bank identified by MPAMM. The address in ADRX generates the row and column address bits for the memory array cards.

Figure 4–11 shows the microcode flow for the read refill request. Example 4–2 shows the symbolic encoding for the read refill request.

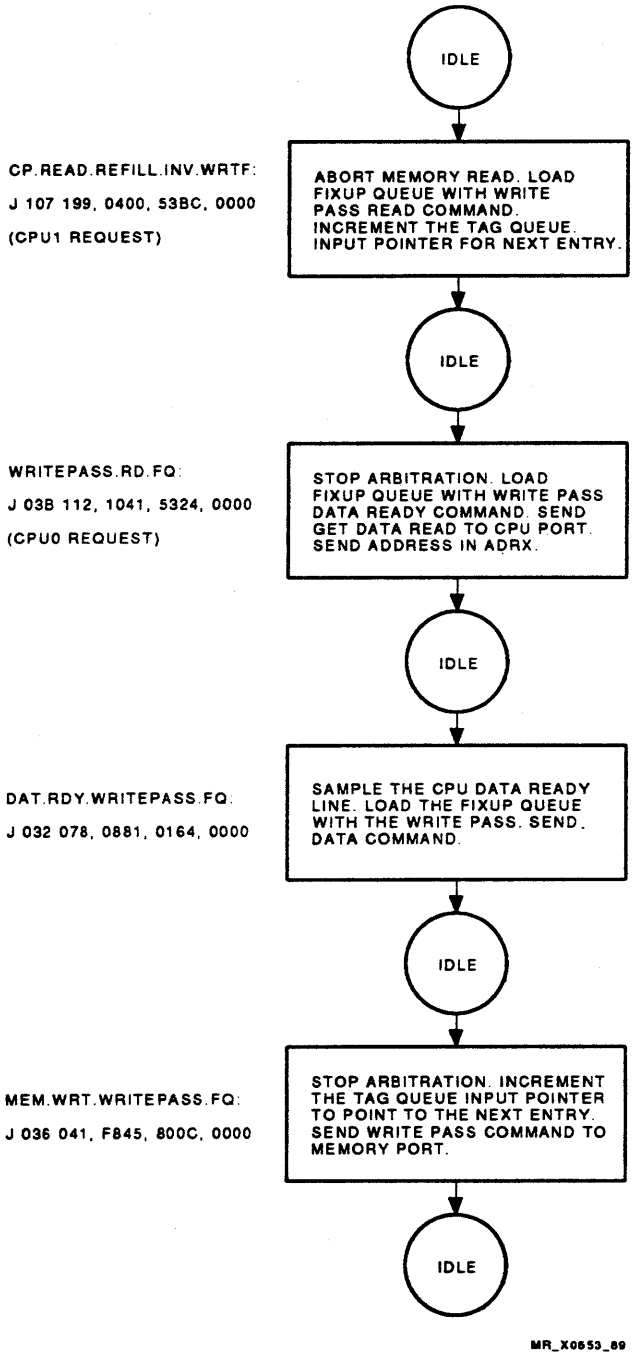


Figure 4–11 Read Refill — With Fixup Flow

```
HEX ENCODING:  J 107    199,0400,53BC,0000

  SYMBOLIC ENCODING:

  CP.READ.REFILL.INV.WRTF:  FIXUP,
                            INC,
                            TAG STAT/RD,
                            MEM/ABORT,
                            INDEX/VAL,
                            FIXCMD/WRITEPASS RD,
                            GO TO [IDLE]

HEX ENCODING:  J 03B    112,1041,5324,0000

  SYMBOLIC ENCODING:

  WRITEPASS.RD.FQ:  INC,
                    INDEX/FIX,
                    ARB/STOP,
                    TAG STAT/RD,
                    CMDMSK/GET DATA RD
                    FIXCMD/WRITEPASS DATRDY,
                    ADROUT,
                    GO TO [IDLE]

HEX ENCODING:  J 032    078,0881,0164,0000

  SYMBOLIC ENCODING:

  DAT.RDY.WRITEPASS.FQ:  DATRDY,
                         FIXCMD/WRITEPASS SENDAT,
                         INDEX/FIX,
                         GO TO [IDLE]

HEX ENCODING:  J 036    041,F845,800C,0000

  SYMBOLIC ENCODING:

  MEM.WRT.WRITEPASS.FQ:  ARB/STOP,
                         CMDMSK/WRITEPASS,
                         INDEX/FIX INC,
                         CLEAR FIX,
                         CTLC DONE/YES,
                         CP SENDATA,
                         GO TO [IDLE]
```

Example 4–2 Read Refill — With Fixup Symbolic Encoding

The symbolic encoding for microaddress 107 can be interpreted as follows:

- **FIXUP** — CPU1 needs the cache block (written full) in the CPU0 cache set. The microcode indicates that the fixup queue needs to resolve the conflict.
- **INC** — The tag is being written. Increment the tag queue input pointer for the next entry. If not a tag write, the CTLD tag queue input pointer is incremented automatically. The JBox can have four fixup requests in progress.
- **TAG STAT/RD** — Write read tag status for CPU1.
- **MEM/ABORT** — Stop the memory read already started. The latest copy of the data that CPU1 needs is not in memory but in CPU0 cache set 0.
- **INDEX/VAL** — This is used to load the fixup queue with the CPU number to be reserved, CPU0 in this example.
- **FIXCMD/WRITEPASS RD** — The microcode loads a write pass read fix command into the fixup queue. The fixup queue uses this command field and generates the low-order bits of the next microaddress (03B).
- **GO TO [IDLE]** — Return to idle.

The bit encoding for microaddress 107 is shown in Table 4–32.

Table 4–32 Microaddress 107 CPU Read Refill (With Fixup) Microword Bits

Bit	Name	Value	Description
09:00	Branch PC	00 0000 0000	–
13:10	Branch select	0000	–
17:14	Branch enable	0000	Does not enable branch select logic.
18	Done	1	Indicates that MICR can read the next entry in the tag queue.
19	MIC fix	1	Loads the fixup queue.
24:20	Fix command	1 1011	Indicates that write pass read is the fix command loaded into the fixup queue and forms the low-order bits of the next microaddress.
28:25	Tag status	1001	Writes read status (CPU1).
29	Inhibit fixup	0	Does not inhibit fixup queue.
30	CTLD CTLn	1	Allows an interrupted read to resume from where it left off when the microcode is finished.
33:31	Index	000	Uses the requester's index value (CPU1).
34	CTLC done	0	Indicates that a request is still in progress.
35	Mark	0	NOP.
36	Write start	0	NOP.
37	Read abort	0	NOP.
38	ARB	0	Does not stop arbitration.
39	MIC CTL1	0	NOP.
40	CTLD CTLn	0	Allows an interrupted read to resume from where it left off.
42:41	MEM	10	Indicates abort status.
43	MIC CTL0	0	NOP.
47:44	Command mask	0000	NOP.
48	CTLD fix	1	Loads fixup queue with the fix command.
49	MIC CTL2	0	NOP.
50	CSSE done	0	NOP.
51	CTLA parity	1	–
52	CTLB parity	1	–
53	CTLC parity	0	–
54	CTLD parity	0	–
55	MIC parity	1	–
56	CTLD CTLn	1	Allows an interrupted read to resume from where it left off.
57	ADR parity	0	–

Table 4–32 (Cont.) Microaddress 107 CPU Read Refill (With Fixup) Microword Bits

Bit	Name	Value	Description
58	–	–	Reserved.
59	I/O command	0	Signifies that this is not an I/O command.

The symbolic encoding for microaddress 03B can be interpreted as follows:

- **INC** — The tag is being written. Increment the tag queue input pointer for the next entry. If not a tag write, the CTLD tag queue input pointer is incremented automatically.
- **INDEX/FIX** — This reloads the reserved CPU number into the fixup queue. It also generates the write enable for the tag write (CPU0).
- **ARB/STOP** — Data and address paths are now controlled by the microcode. Port arbitration stops.
- **TAG STAT/RD** — Write the read tag status for CPU0.
- **CMDMSK/GET DATA READ** — Get the data from CPU0 and leave the cache block in the cache data STRAMs with read status. If CPU0 needs to write to this cache block, it sends a write refill. JBox knows that CPU0 already has the cache block and can send an OK to write command and that the MICR can change the cache status from read to written full.
- **FIXCMD/WRITEPASS DATRDY** — The fixup queue uses the fix command to form the low-order bits for the next microword (032).
- **ADROUT** — The JBox uses the address in the ADRX hold latches when it sends a get data read command to CPU0. CPU0 decodes the get data read command and uses the address to address the cache block in the cache data STRAMs in the MBox.
- **GO TO [IDLE]** — Return to the idle state.

The bit encoding for microaddress 03B is shown in Table 4–33.

Table 4–33 Microaddress 03B CPU Read Refill (With Fixup) Microword Bits

Bit	Name	Value	Description
09:00	Branch PC	00 0000 0000	–
13:10	Branch select	0000	–
17:14	Branch enable	0000	Does not enable branch select logic.
18	Done	1	Indicates that MICR can get an entry from the tag queue.
19	MIC fix	0	NOP.
24:20	Fix command	1 0010	Loads fixup queue with write pass data ready command.
28:25	Tag status	1001	Writes tag status as read.
29	Inhibit fixup	0	Does not inhibit fixups.

Table 4–33 (Cont.) Microaddress 03B CPU Read Refill (With Fixup) Microword Bits

Bit	Name	Value	Description
30	CTLD CTLn	1	Allows an interrupted read to resume from where it left off when the microcode is finished.
33:31	Index	001	Writes the status for the other (CPU1).
34	CTLC done	0	Indicates that a request is still in progress.
35	Mark	0	NOP.
36	Write start	0	NOP.
37	Read abort	0	NOP.
38	ARB	1	Stops arbitration.
39	MIC CTL1	0	Sends the address in the ADRX hold latch with the port command (get data read).
40	CTLD CTLn	0	Allows an interrupted read to resume from where it left off when the microcode is finished.
42:41	MEM	00	Indicates OK status.
43	MIC CTL0	0	Sends the address in the ADRX hold latch with the port command.
47:44	Command mask	0001	Loads CPU0 command output buffer with the get data read command.
48	CTLD fix	0	NOP.
49	MIC CTL2	1	Sends the address in the ADRX hold latch with the port command.
50	CSSE done	0	NOP.
51	CTLA parity	0	–
52	CTLB parity	1	–
53	CTLC parity	0	–
54	CTLD parity	0	–
55	MIC parity	0	–
56	CTLD CTLn	1	Allows an interrupted read to resume from where it left off when the microcode is finished.
57	ADR parity	0	–
58	–	–	Reserved.
59	I/O command	0	Signifies that this is not an I/O command.

The symbolic encoding for microaddress 032 is as follows:

- **DATRDY (CLEAR)** — This tells the fixup queue not to wait for data ready any longer.
- **FIXCMD/WRITEPASS SENDAT** — The microcode loads the fixup queue with write pass send data fix command. The fixup uses the fix command and generates the next microaddress (036).
- **INDEX/FIX** — Generates the write enable for the tag write.
- **GO TO [IDLE]** — Return to idle.

The bit encoding for microaddress 032 is shown in Table 4–34.

Table 4–34 Microaddress 032 CPU Read Refill (With Fixup) Microword Bits

Bit	Name	Value	Description
09:00	Branch PC	00 0000 0000	–
13:10	Branch select	0000	–
17:14	Branch enable	0000	Does not enable branch select logic.
18	Done	1	Indicates that MICR can get an entry from the tag queue.
19	MIC fix	0	NOP.
24:20	Fix command	1 0110	Loads fixup queue with write pass send data command.
28:25	Tag status	0000	NOP.
29	Inhibit fixup	0	Does not inhibit fixups.
30	CTLD CTLn	0	NOP.
33:31	Index	010	Invalidates the next CPU if applicable.
34	CTLC done	0	Indicates that a request is still in progress.
35	Mark	0	NOP.
36	Write start	0	NOP.
37	Read abort	0	NOP.
38	ARB	0	Does not stop arbitration.
39	MIC CTL1	1	Clears the data ready line.
40	CTLD CTLn	0	NOP.
42:41	MEM	00	Indicates OK status.
43	MIC CTL0	1	Clears the data ready line.
47:44	Command mask	0000	NOP.
48	CTLD fix	0	NOP.
49	MIC CTL2	0	Clears the data ready line.
50	CSSE done	0	NOP.
51	CTLA parity	1	–
52	CTLB parity	1	–
53	CTLC parity	1	–

Table 4–34 (Cont.) Microaddress 032 CPU Read Refill (With Fixup) Microword Bits

Bit	Name	Value	Description
54	CTLD parity	1	–
55	MIC parity	0	–
56	CTLD CTLn	0	NOP.
57	ADR parity	0	–
58	–	–	Reserved.
59	I/O command	0	Signifies that this is not an I/O command.

The symbolic encoding for microaddress 036 is as follows:

- **ARB/STOP** — Stop arbitration.
- **CMDMSK/WRITEPASS** — Send a write pass command to the memory port.
- **INDEX/FIX INC** — Generates the write enable for the tag write and increments the tag queue input pointer.
- **CLEAR FIX** — Clear the fixup queue.
- **CTLC DONE/YES** — Retire the CPU1 read refill request.
- **CP SENDATA** — Send a send data command to CPU0.

The bit encoding for 036 is shown in Table 4–35.

Table 4–35 Microaddress 036 CPU Read Refill (With Fixup) Microword Bits

Bit	Name	Value	Description
09:00	Branch PC	00 0000 0000	–
13:10	Branch select	0000	–
17:14	Branch enable	0000	Does not enable branch select logic.
18	Done	1	Indicates that MICR can get an entry from the tag queue.
19	MIC fix	1	Clears the fixup queue.
24:20	Fix command	0 0000	Clear.
28:25	Tag status	0000	NOP.
29	Inhibit fixup	0	Does not inhibit fixups.
30	CTLD CTLn	0	NOP.
33:31	Index	011	–
34	CTLC done	1	Retires a request.
35	Mark	0	NOP.
36	Write start	0	NOP.
37	Read abort	0	NOP.
38	ARB	1	Stops arbitration.

Table 4–35 (Cont.) Microaddress 036 CPU Read Refill (With Fixup) Microword Bits

Bit	Name	Value	Description
39	MIC CTL1	0	Reserves the requester's output command buffer.
40	CTLD CTLn	0	NOP.
42:41	MEM	00	Indicates OK status.
43	MIC CTL0	1	Reserves the requester's output command buffer.
47:44	Command mask	1111	Loads the memory port command buffer with the write pass command.
48	CTLD fix	1	Clears the fixup queue.
49	MIC CTL2	0	Reserves the requester's output command buffer.
50	CSSE done	0	NOP.
51	CTLA parity	0	–
52	CTLB parity	0	–
53	CTLC parity	0	–
54	CTLD parity	1	–
55	MIC parity	0	–
56	CTLD CTLn	0	NOP.
57	ADR parity	0	–
58	–	–	Reserved.
59	I/O command	0	Signifies that this is not an I/O command.

4.4.2.3 CPU Write Refill — Without Fixup

In this example, the cache block with the starting address 1000 has invalid status in the CPU0, CPU1, CPU2, and CPU3 tag status STRAMs. CPU0 sends a write refill to the JBox to request the cache block. The following steps summarize the write refill request operation:

1. CPU0 sends a load command, write refill command, address 1000, cache set 0 in the first cycle, and in the second cycle, sends the remainder of address 1000.
2. MICR latches and decodes the tag STRAMs status bits, MTCH MCA match results, and CTLD tag queue data to generate microaddress 110.
3. The microword specifies a memory write from the memory unit, segment, and bank identified by MPAMM, and specifies that the address in ADRX is to generate the row and column address bits to be sent to the memory array cards.
4. The microword specifies that the tag status is to be written as written full for the cache block at address 1000 for CPU0.

The symbolic encoding can be interpreted as follows:

- **INC** — The tag is being written. Increment the tag queue input pointer for the next entry. If not a tag write, the CTLD tag queue input pointer is incremented automatically.
- **TAG STAT/WRTF** — Write written full tag status for the requester.
- **MEM/OK** — Continue the memory read operation.
- **WRT** — Reserved.
- **CTLC DONE/YES** — Retire the request.
- **GO TO [IDLE]** — Return to the idle state.

4.4.2.4 DMA Read — Without Fixup

In this example, the cache block with the starting address 1000 has invalid status in the CPU0, CPU1, CPU2, and CPU3 tag status STRAMs. ICU0 has received an XJA0 DMA read command, for address 1000, for 32 bytes (hexword) of data. Example 4–3 shows the symbolic encoding for the read refill request. The following steps summarize a DMA read request operation:

1. XJA0 sends a DMA$READ_REQUEST, address 1000, XMI ID, and length (octaword). ICU sends an acknowledgment to XJA0.
2. ICU sends the JBox the load command, DMA read command, and address 1000. ICU sends a buffer available command to XJA0.
3. MICR latches and decodes the tag STRAMs status bits, MTCH MCA match results, and CTLD tag queue data to generate microaddress 200.
4. The microword specifies a DMA read from the memory unit, segment, and bank identified by MPAMM and specifies that the address in ADRX (in the I/O address latch) is to generate the row and column address bits to be sent to the memory array cards.
5. ICU sends DMA$READ_DATA_RETURN, return address 1000, length (hexword), and four quadwords, 0, 1, 2, and 3, to XJA0. XJA0 sends an acknowledgment to ICU. When XJA0 unloads its command/address/data receive buffer, it sends a buffer available command to ICU.

```
HEX ENCODING:  J 200    07C,0204,0004,0000

  SYMBOLIC ENCODING:

  DMA.READ.INV.INV:  SEG,
                     MEM/OK,
                     CTLC DONE/YES,
                     GO TO [IDLE]
```

Example 4–3 DMA Read Symbolic Encoding

The symbolic encoding can be interpreted as follows:

- **SEG** — Reserved.
- **MEM/OK** — The memory read operation can continue.
- **CTLC DONE/YES** — Retire the request.
- **GO TO [IDLE]** — Return to the idle state.

4.4.2.5 DMA Read — With Fixup

In this example, the cache block with the starting address 1000 has written full status in the CPU1 tag status STRAM and invalid status in CPU0, CPU2, and CPU3. ICU0 sends a DMA read to the JBox. Figure 4–12 shows the microcode flow for the DMA read request. Example 4–4 shows the symbolic encoding for the read refill request. The following steps summarize the DMA read request operation when a fixup is required:

1. ICU0 sends a load command, DMA read command, and address 1000.
2. MICR latches and decodes the tag STRAMs status bits, MTCH MCA match results, and CTLD tag queue data to generate microaddress 207.
3. The microword at 207 specifies an abort memory read (memory read is already in progress to optimize performance; the cache tag status informs MICR that more recent data is in the CPU0 cache) and loads the fixup queue with write pass read command. The fixup queue uses the fix command field to generate the next microaddress, 0BB.
4. The microword at 0BB writes the CPU1 status as read. If CPU1 has to write to the cache block, it sends write refill. The tag STRAMs indicate that CPU1 has the cache block with read status. The JBox sends OK to write to the CPU and changes the tag status from read to written full. The microcode loads the fixup queue with a write pass send data command to generate the next microword address, 0B2.
5. The microword at 0B2 samples the data ready line from CPU1 and sends write pass send data fix command to the fixup queue. The fixup queue uses the fix command to form the next microword address, 0B6. CPU1 loads the write back buffer and sends the JBox data ready.
6. The microword at 0B6 specifies that the memory port receives a write pass memory command. The JBox unloads CPU1's cache block at address 1000, passes it to the ICU0 data switch lines, and then writes the data into memory. The microcode needs the data paths (source and destination) and stops port arbitration until the data transfer is completed. The tag queue input pointer is incremented to point to the next entry in the tag queue. The fixup queue is cleared and the CTLC retire logic retires the request.

   A memory write is performed using the the memory unit, segment, and bank identified by MPAMM, and the address in ADRX generates the row and column address bits sent to the memory array cards.

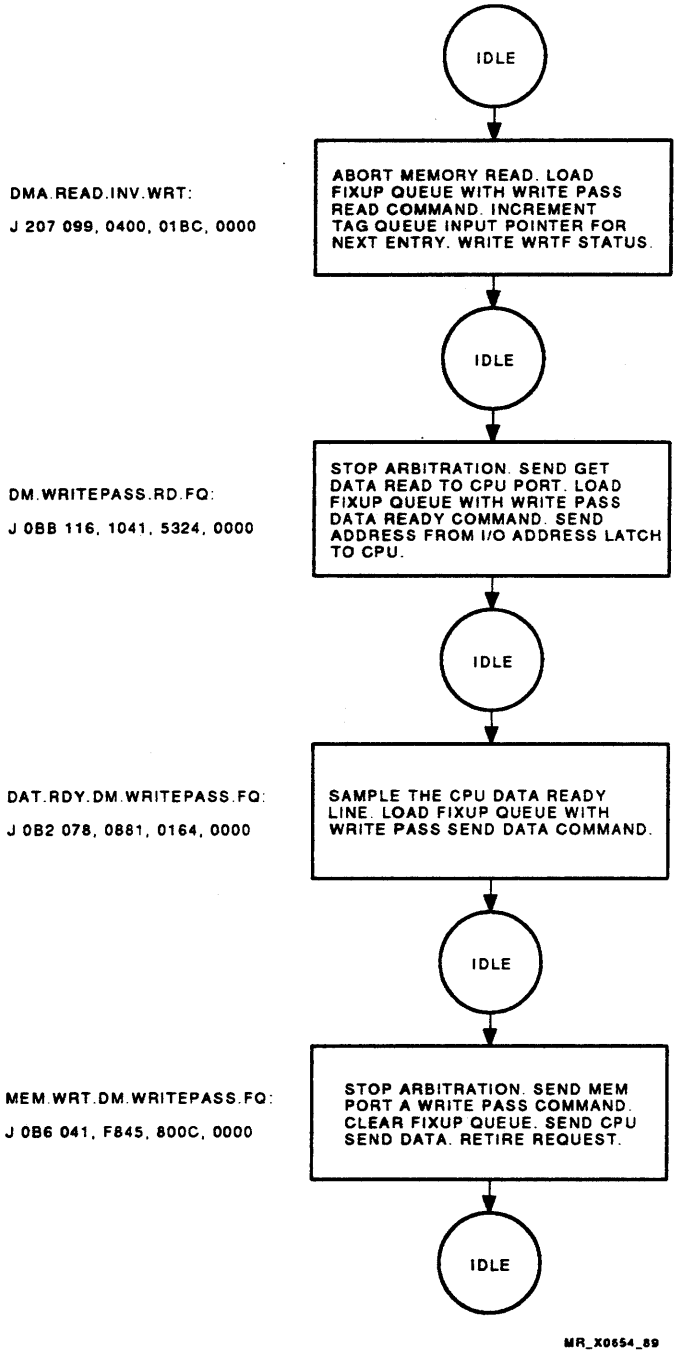


Figure 4–12 DMA Refill — With Fixup Flow

```
HEX ENCODING:  J 207    099,0400,01BC,0000

  SYMBOLIC ENCODING:

  DMA.READ.INV.WRTF:  MEM/ABORT,
                      INDEX/VAL,
                      FIXCMD/WRITEPASS RD,
                      FIXUP,
                      GO TO [IDLE]

HEX ENCODING:  J 0BB,116,1041,5324,0000

  SYMBOLIC ENCODING:

  DM.WRITEPASS.RD.FQ:  SEG,
                       INC,
                       INDEX/FIX,
                       ARB/STOP,
                       TAG STAT/RD,
                       CMDMSK/GET DATA RD,
                       FIXCMD/WRITEPASS DATRDY,
                       ADROUT,
                       GO TO [IDLE]

HEX ENCODING:  J 0B2,078,0881,0164,0000

  SYMBOLIC ENCODING:

  DAT.RDY.DM.WRITEPASS.FQ:  DATRDY,
                            FIXCMD/WRITEPASS SENDAT,
                            INDEX/FIX,
                            GO TO [IDLE]

HEX ENCODING: J 0B6,    041,F845,800C,0000

  SYMBOLIC ENCODING:

  MEM.WRT.DM.WRITEPASS.FQ:  ARB/STOP,
                            CMDMSK/WRITEPASS,
                            INDEX/FIX INC,
                            CLEAR FIX,
                            CTLC DONE/YES,
                            CP SENDATA,
                            GO TO [IDLE]
```

Example 4–4 DMA Read — With Fixup Symbolic Encoding

5
Array Control Unit and Main Memory Unit

The SCU can have two array control units (ACUs), ACU0 and ACU1. ACU0, located on the DA0 and DB0 MCUs, consists of the MMC0, MCD0, MDP0, and MDP1 MCAs and supports main memory unit 0 (MMU0). ACU1, located on the DA1 and DB1 MCUs, consists of the MMC1, MCD1, MDP2, and MDP3 MCAs and supports main memory unit 1 (MMU1).

5.1 Overview

Each ACU contains built-in self-test (BIST) logic to support the service processor unit (SPU) during self-tests. The ACUs send and receive commands and data to and from the following:

- **SPU** — Using a cable that connects SPU to the SCU planar module, the two ACUs receive SPU control signals for testing the memory modules.
- **Two main memory units** — Using cables that connect the SCU planar module with MMU, the two ACUs send and receive commands and data to and from main memory units 0 and 1.
- **JBox** — Using the logical interface on the SCU planar module, ACU sends and receives memory commands, control, and status information to and from the CCU MCU in the JBox. The JBox receives CPU and I/O memory requests and sends the requests to ACU. The JBox contains a port controller that controls two command buffers for commands received from ACU and the four memory segment controllers to which ACU sends the status of each memory segment in each MMU. In the JBox, the tag MCU holds the row and column addresses for the DRAMs. The ACU selects the appropriate address latch in the tag MCU.

The memory communicates with the JBox when any of the following events occur:

- A read request is made and the data is ready to be sent.
- An error is detected during the transfer of read data.
- An error is detected during the transfer of write data.
- A command buffer is available.

5.2 Memory Subsystem

The memory subsystem consists of the following logic:

- **Array control unit** — ACU0 supports MMU0. ACU1 supports MMU1.
- **Main memory unit** — MMU0 contains four memory modules: M0, M1, M2, and M3. MMU1 contains four memory modules: M0, M1, M2, and M3.
- **Service processor unit** — SPU controls testing the memory modules.

Figure 5–1 shows the two memory subsystems, MMU0 and MMU1.

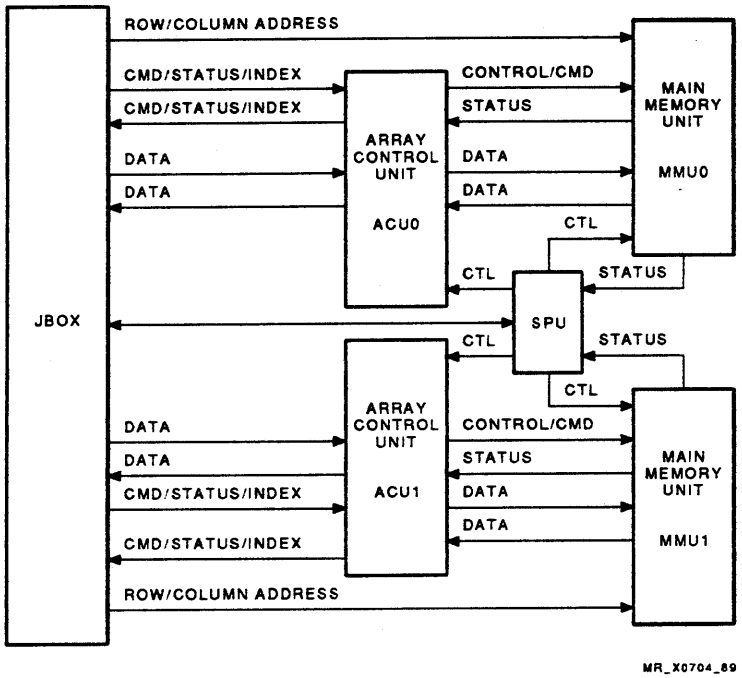


Figure 5–1 Memory Subsystems

5.2.1 Array Control Unit

ACU resides on the SCU planar module and consists of the following MCAs and MCUs (Figure 5–2 shows the SCU planar module and its MCAs, MCUs, STRAMs, and cable connections):

- **MMCX (main memory control) MCA** — This MCA provides the command, data, and address control and the status interface to the JBox. MMCX sends data path control signals and DRAM control commands to the MCDX MCA. The MMCX MCA provides error detection on all MMCX MCA control lines and supports the BIST operation. MMC0, located on DB0, supports ACU0. MMC1, located on DB1, supports ACU1.
- **MCDX (memory control DRAMs) MCA** — This MCA provides DRAM control signals, RAS, CAS, and WE, for the data path and memory modules. MCDX provides DRAM control timing (except in step mode), sends commands to MMCX during BIST operations, and provides error detection on all MCDX MCA control lines. While in step mode, this MCA translates memory commands into DCA (DRAM control and address) step mode commands.

- **MDPX (memory data path) MCA** — This MCA provides a 4-byte path in each direction, check bit generation for write data, and a byte merge path. MDPX detects and corrects single-bit errors on read data, but it can only detect, not correct, double-bit errors. This MCA contains a linear feedback shift register (LFSR) that generates data patterns during BIST operations. LFSR is also called an address and data pattern generator.

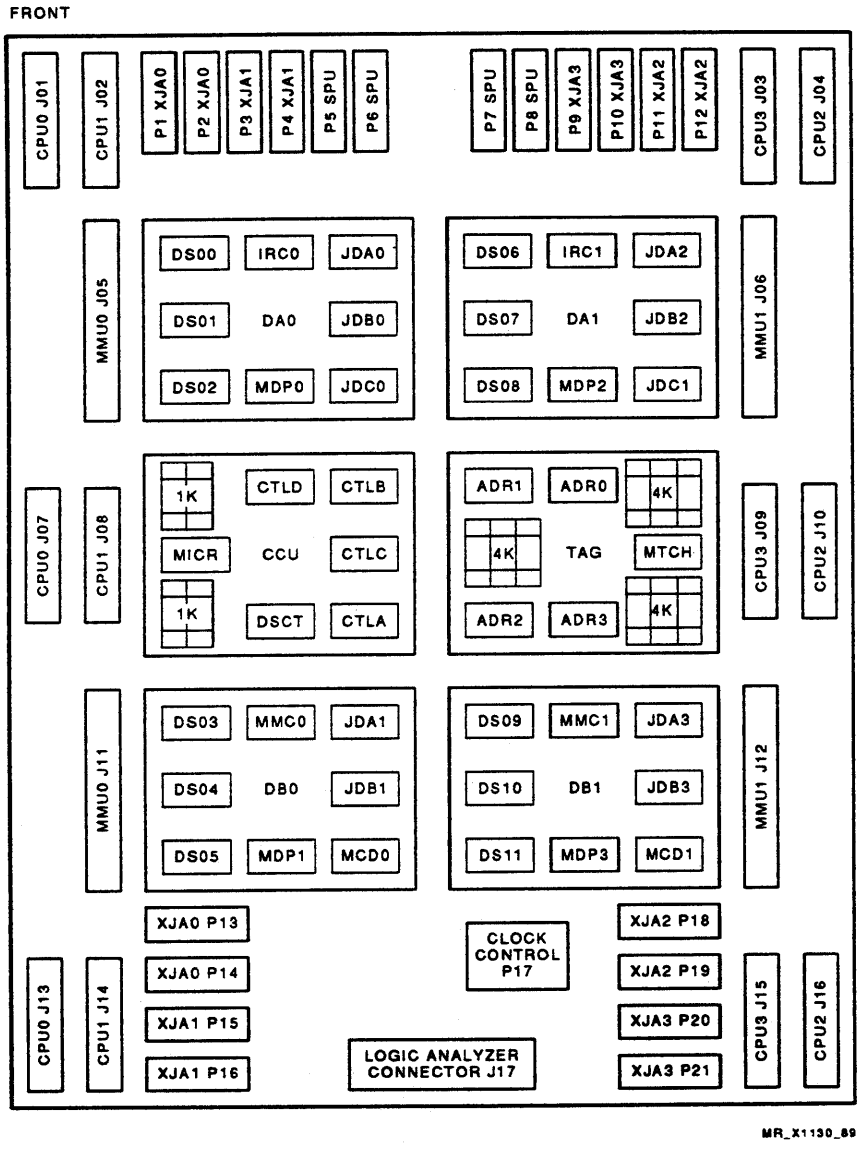


Figure 5–2 SCU Planar Module

5.2.2 Main Memory Unit

The main memory subsystem is a nonbussed, block-oriented, high-bandwidth system using application-specific integrated circuits (ASIC) and board technologies. Cables connect MMU to the SCU planar module. The ACUs provide the data path between the JBox and MMUs. The tag MCU on the SCU planar module provides the address path between SCU and the MMUs.

Parameters for a fully configured (MMU0 and MMU1) memory include:

- Four-way interleaving.
- Maximum capacity of 512 Mbytes using 1-Mbit DRAMs. (Each MMU provides 256 Mbytes using 1-Mbit DRAMs.)
- Read and write bandwidth of 500 Mbytes/s.
- 280-ns read latency (from receipt of command to transmission of data to the JBox).
- Memory expansion support.

Figure 5–3 shows the ACU and MMU data interface, and Figure 5–4 shows the ACU and MMU address, command, control, and status interfaces.

Table 5–1 lists the power requirements for the memory subsystem.

Table 5–1 Power Requirements for the Memory Subsystem

Description	Voltage (Vdc)	Maximum Active Current	Maximum Standby Current	Minimum Standby Current
BBU	+5.0	26.23	5.98	1.223
VCC	+5.0	5.00	NA	NA
VEE	–5.2	5.00	NA	NA

MMU has four extended hex memory modules that reside in a card cage. Each memory module consists of a main array card (MAC) and two daughter array cards (DACs). The MAC is 15.688 in × 11.9 in. The DAC is 7.4 in × 5.6 in.

The two DACs plug onto each MAC using two 260-pin, four row-in-line connectors, and are positioned below the MAC DRAMs.

Each MMU has two segments. Each segment contains two banks, 0 and 1. Control and address lines operate independently across segments. The write path and read data path are common to both segments. The paths are separated into a read and write path of 20-bits per memory module (four memory modules = 80 bits). Each segment receives 11 control signals from the MCDX MCA and 12 row and column [00:11] address lines from the ADRX MCAs.

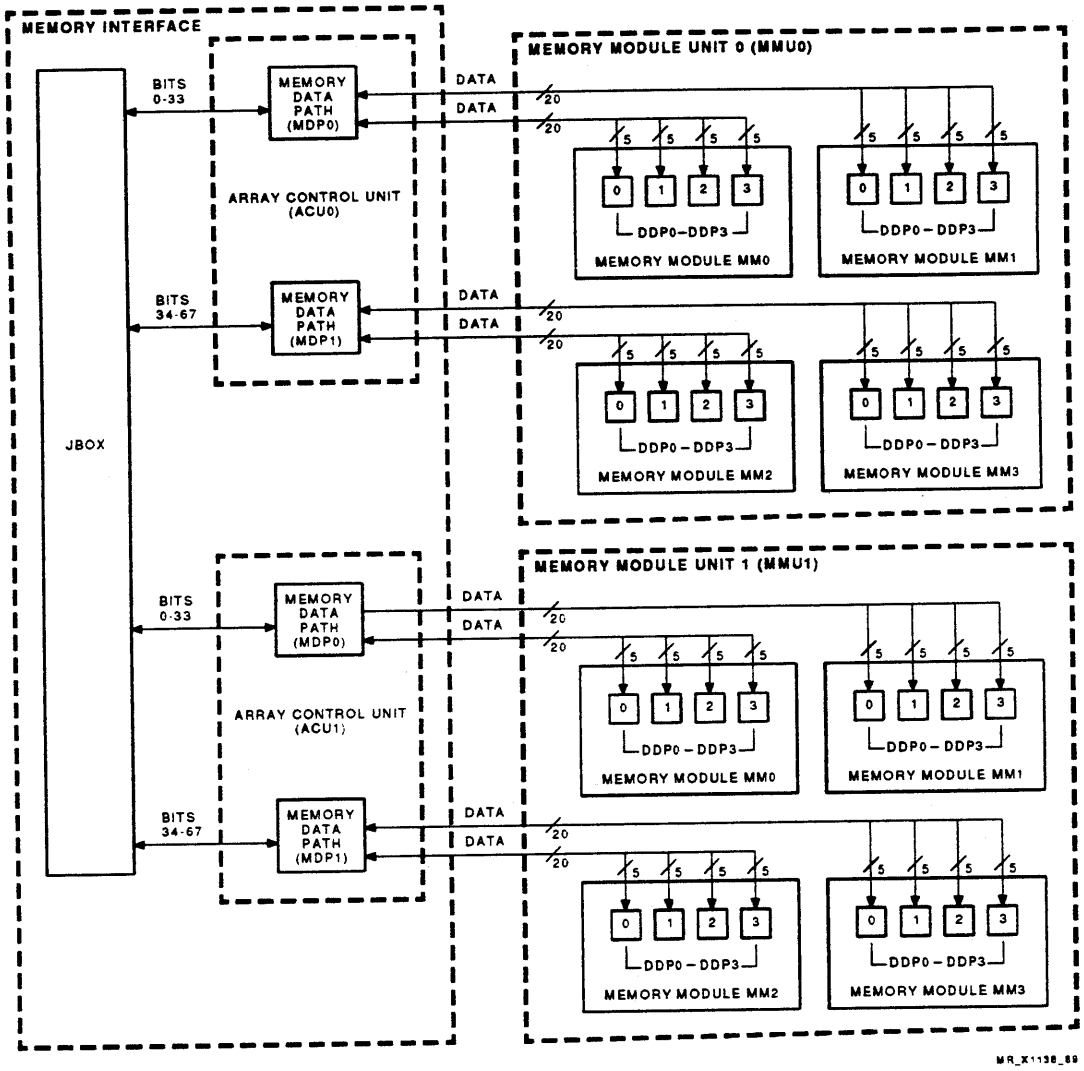


Figure 5–3 ACU-to-MMU Data Interface

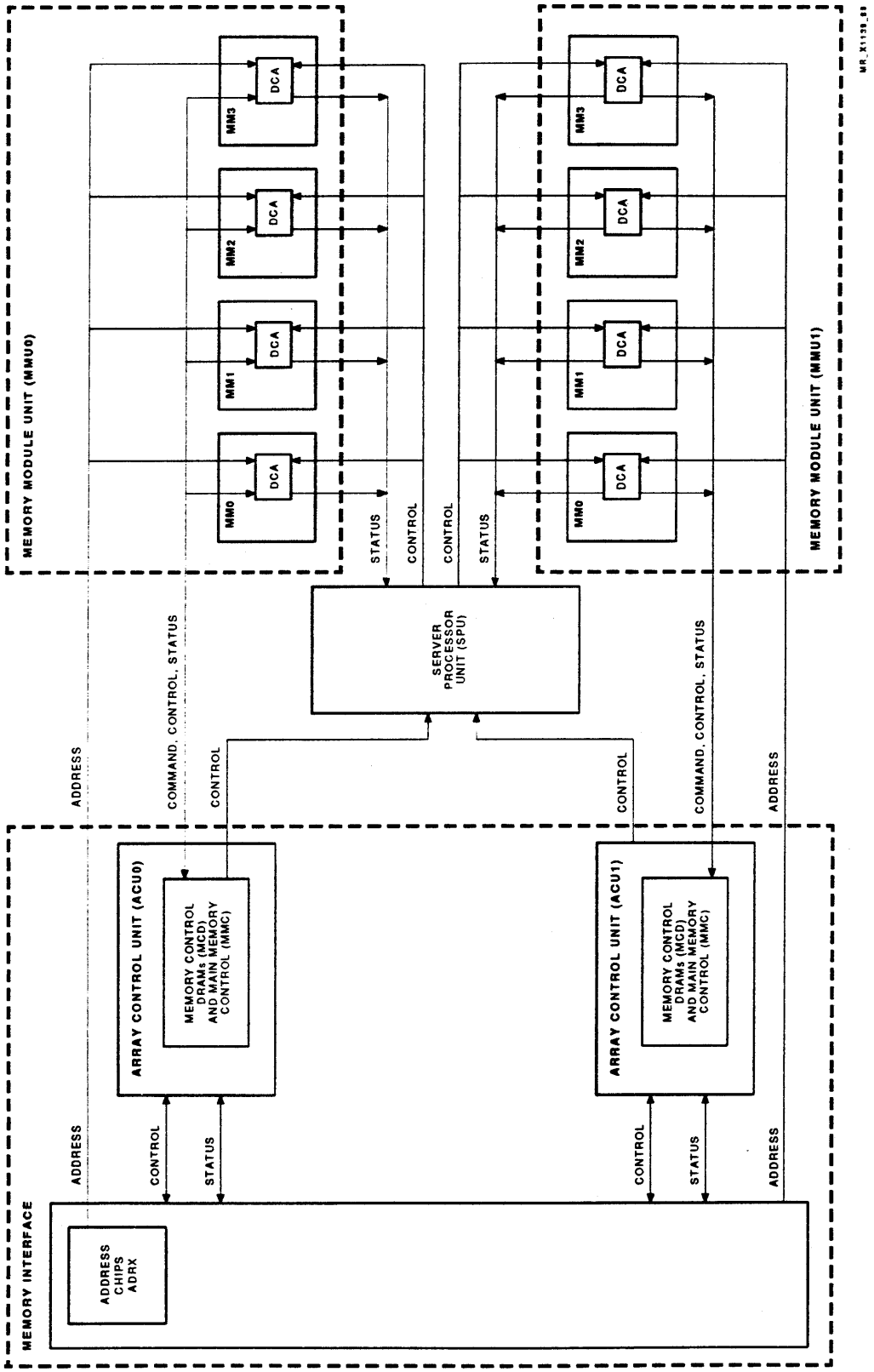


Figure 5–4 ACU-to-MMU Command, Status, and Control Interface

5.2.2.1 Memory Module

The memory module is a nonstandard, hex size module with a 480-pin, right-angle connector at the edge of the card. Each module stores 640 million data bits. Figure 5–5 shows the physical layout of the memory module. The memory module has the following hardware:

Six hundred forty double-sided, surface-mounted DRAMs
Four DRAM data path (DDP) gate arrays
One DRAM control and address (DCA) gate array

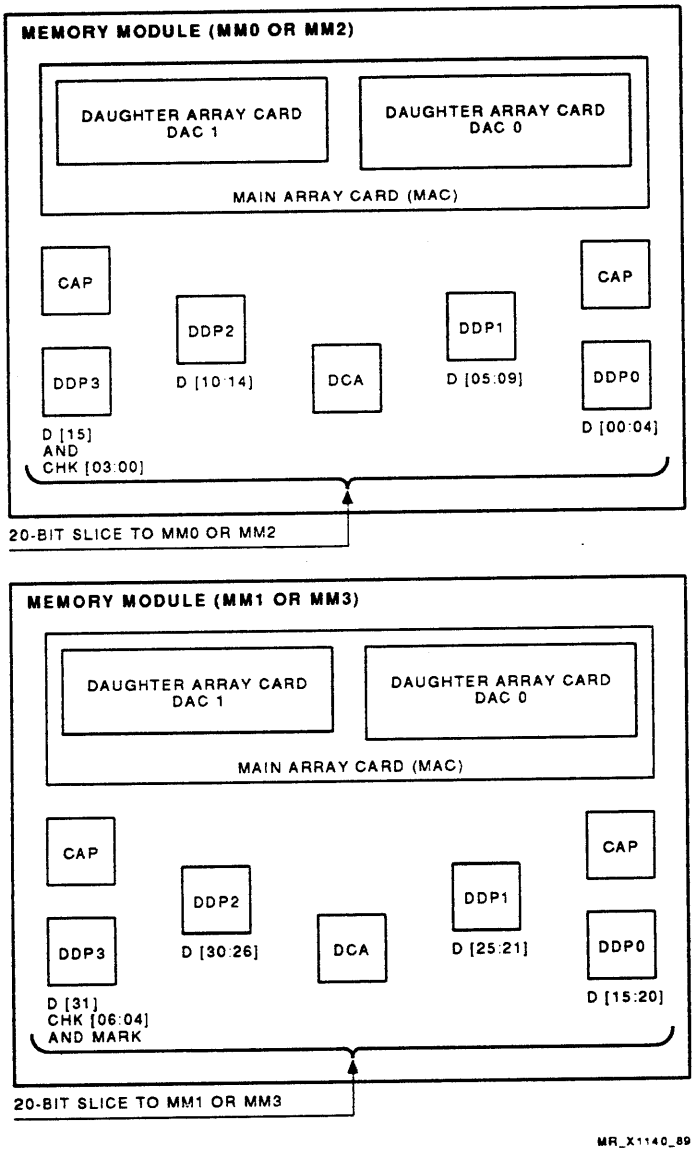


Figure 5–5 Memory Module — Physical Characteristics

The parameters of a memory module are as follows:

Two-way interleaving (across segments)
Two segments
Two banks to a segment
Maximum 64-Mbyte memory size (1-Mbit DRAM)

Figure 5–6 shows a conceptual level functional block diagram of the memory module.

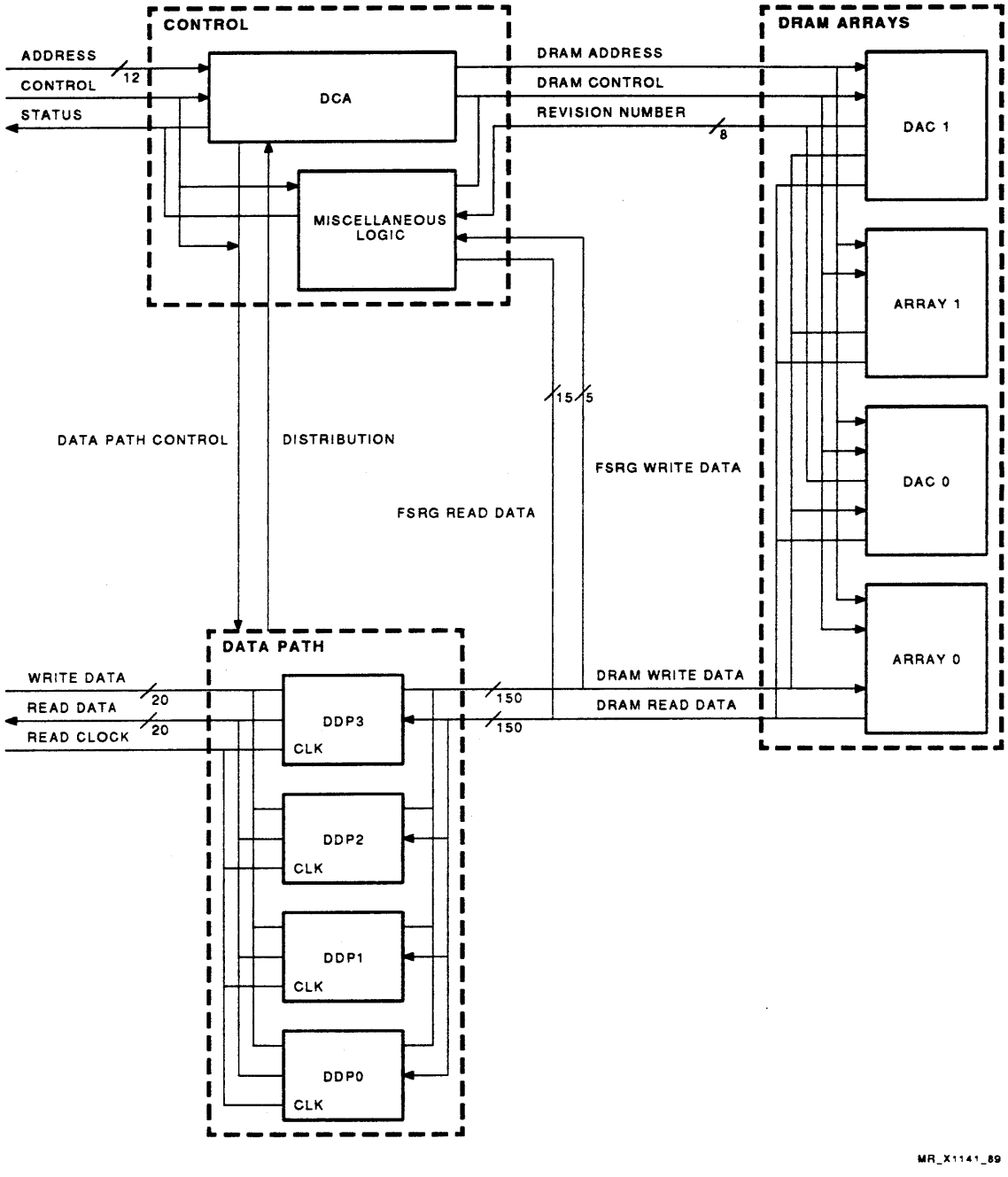


Figure 5–6 Memory Module — Conceptual Level Functional Block Diagram

5.2.2.2 Dynamic RAMs

The MMU contains 4 × 640 (2560) DRAMs. On each of the 4 main array cards, there are 320 DRAMs, and 160 DRAMs are on each of the 8 DACs. Figures 5–7 and 5–8 show the DRAM arrays functional block diagrams for DAC1 array 1 and DAC0 array 0, respectively, on the memory module.

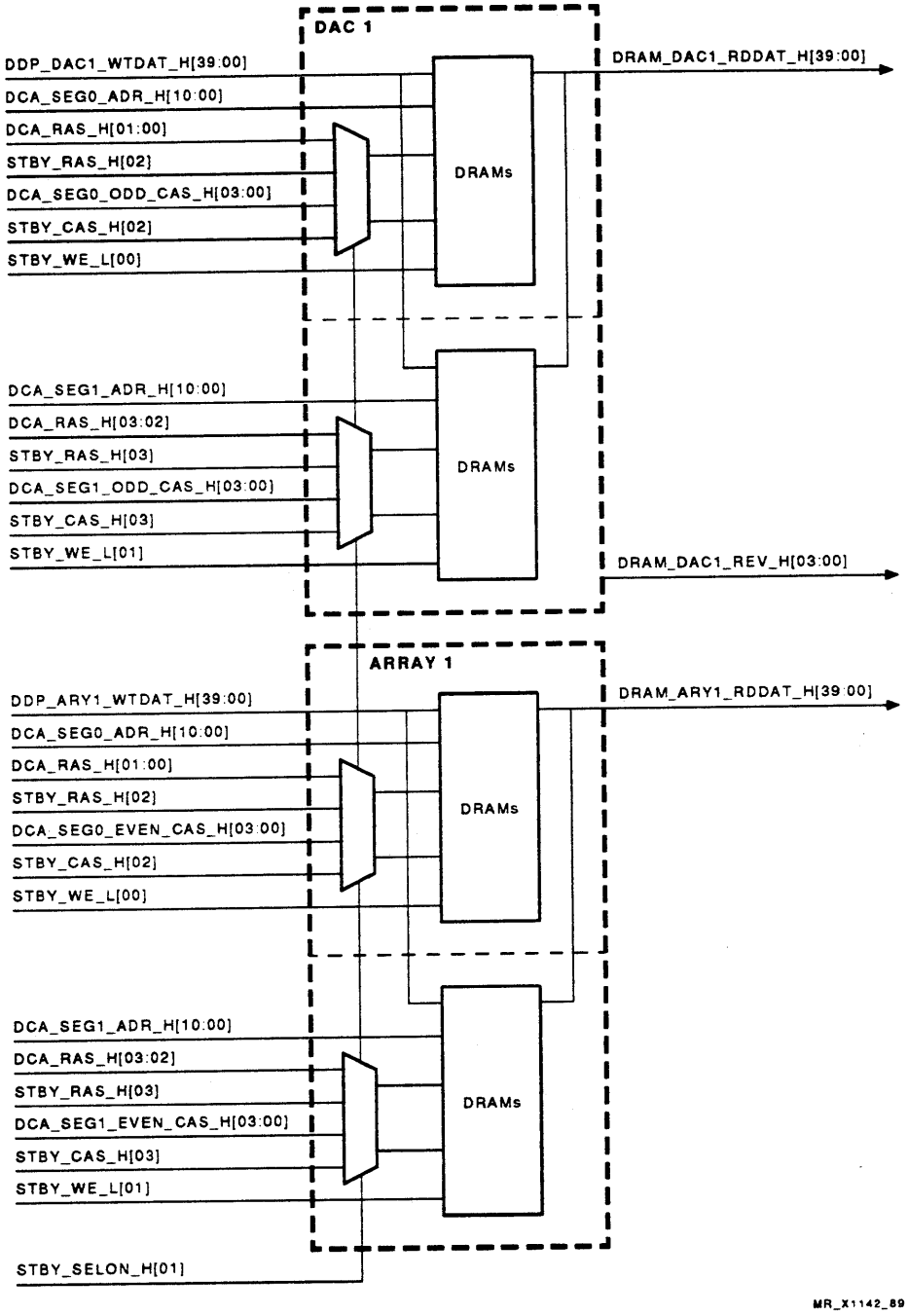


Figure 5–7 DRAM Arrays — DAC1 Array 1

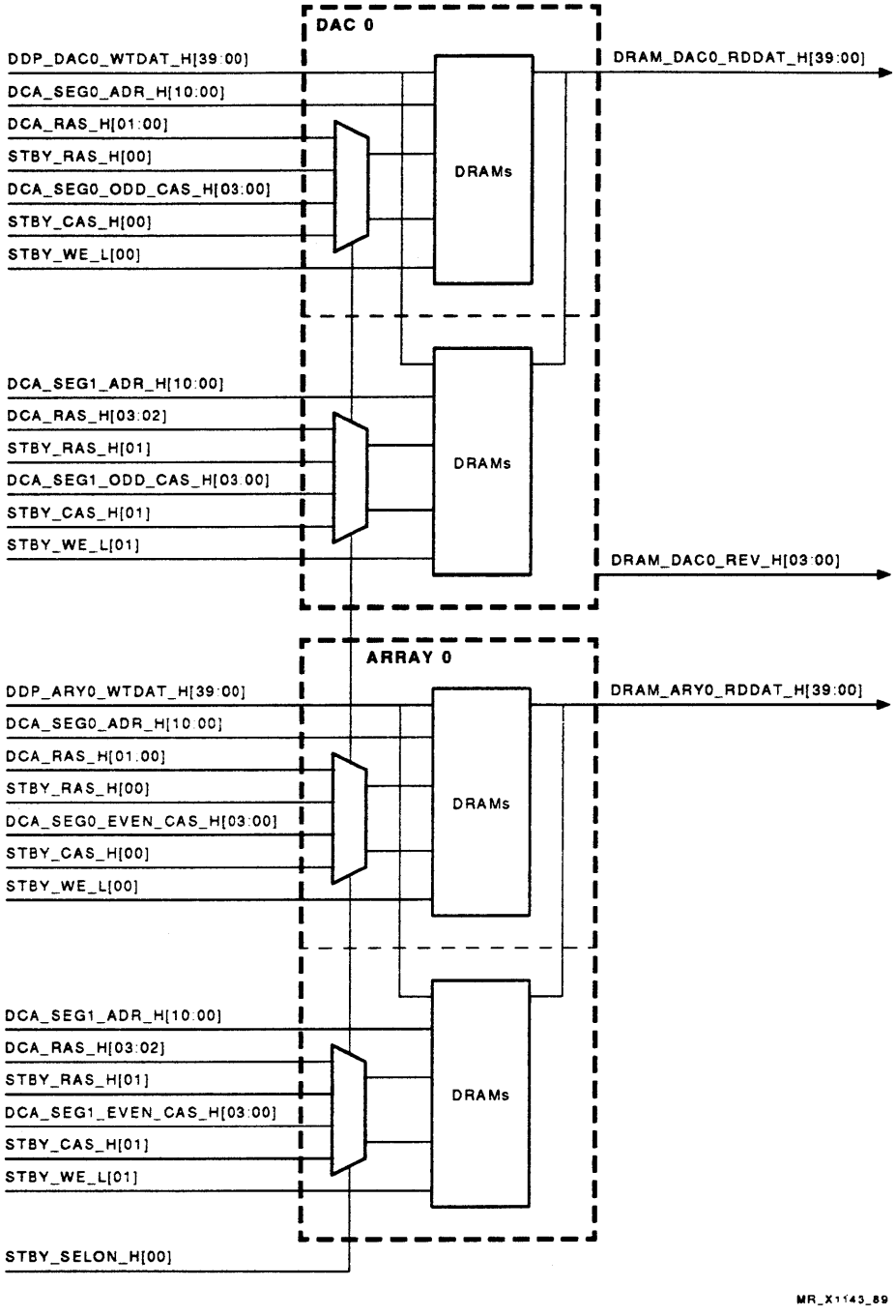


Figure 5–8 DRAM Arrays — DAC0 Array 0

5.2.2.3 Clocks

Three types of clocks exist in the memory subsystem: gated write clocks, a free-running read clock, and an on-board memory module clock.

The memory module receives gated write clocks, called write buffer strobes, from the MMCX control lines [16:13]. MMCX asserts write buffer strobes when write data is loaded into the data input latch (write buffer 0) of the memory module.

The read clock is a free-running, differential clock. The 8 memory modules receive 16 read clocks from SCU. The CDCX MCA generates read clocks (also called STRAM clocks) in groups 0, 1, 2, and 3, and each memory module receives two read clocks. (Two DDPs receive one read clock.) The leading edge of the read clock opens the data output latch (read buffer 0) in the memory module, while the trailing edge clocks the read select and read latch enable signals in the memory module. Figure 5–30 shows the data output latch, read select logic, and read latch enable logic.

The STRAM clocks are programmable and originate in the following MCUs (DAX and DBX MCUs do not generate the STRAM clock groups):

- **Tag MCU** — This MCU provides four STRAM clocks from group 1.
- **CCU MCU** — This MCU provides six STRAM clocks from group 0 and six STRAM clocks from group 2.

Each memory module contains its own clock. A 10-MHz oscillator provides the clock reference for the following logic:

- **DCA step mode clock logic** — DCA uses the clock during step mode. DCA does not use the on-board clock during any other mode.
- **MMCX MCA and DCA refresh clock logic** — The memory module uses clock circuitry to provide a refresh signal to MMCX and DCA. MMCX uses the refresh signal during normal operation but not during step mode. The DCA uses the refresh signal when in step mode and not during normal operation.
- **Standby clock logic** — The standby logic on the memory module uses the clock as a reference. The standby circuit provides CAS signals before RAS signals to refresh the DRAMs at 12.5-ns intervals. This circuit is engaged during standby mode.

5.2.2.4 Main Array Card

MMU0 contains four MACs: MM0, MM1, MM2, and MM3. MMU1 contains four MACs: MM4, MM5, MM6, and MM7. MAC is an extended hex module that contains surface-mounted DRAMs on both sides. MAC contains five gate arrays (four DDPs and one DCA), an EEPROM (for serial number, revision level, and manufacturing support), some MSI logic (for standby operation), electrolytic capacitors, and resistor terminations. Each MAC does the following:

- Provides 32 Mbytes of DRAM storage.
- Buffers write data.
- Buffers read data.
- Ensures DRAM data integrity during power loss.
- Performs read and write cycles independently of ACU (during single-step operations).
- Provides connections and logic support for two DACs.

5.2.2.5 Daughter Array Card

The DAC contains surface-mounted DRAMs on both sides. The DAC provides 16 Mbytes of DRAM storage. Each MAC has two DACs, DAC0 and DAC1. Figure 5–9 shows the inputs to the daughter array card. Figure 5–10 shows the DRAM data path bits across the DAC0 and DAC1 for each quadword in a cache block.

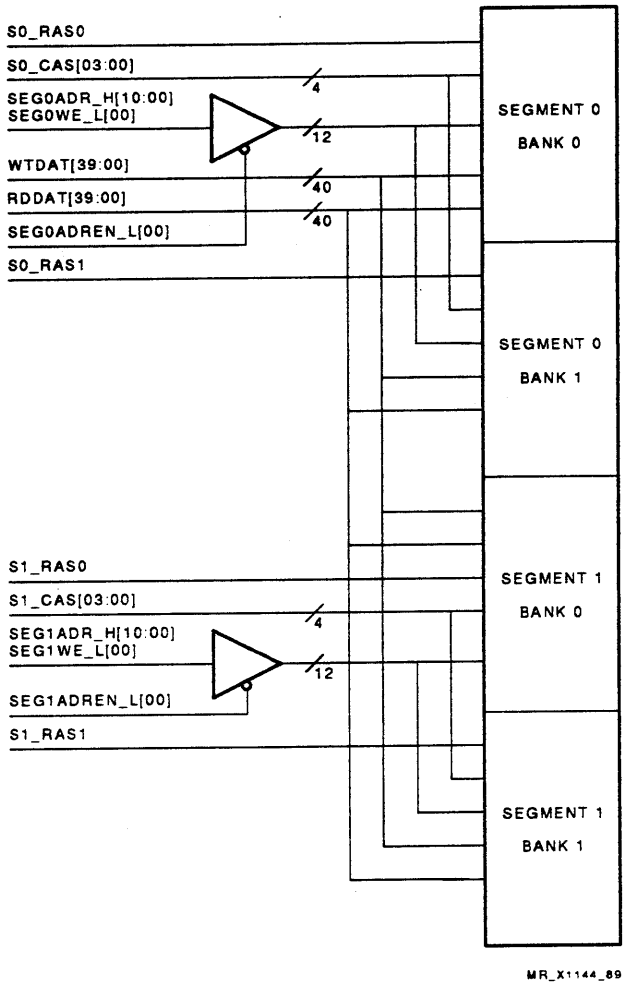


Figure 5–9 Daughter Array Card Inputs

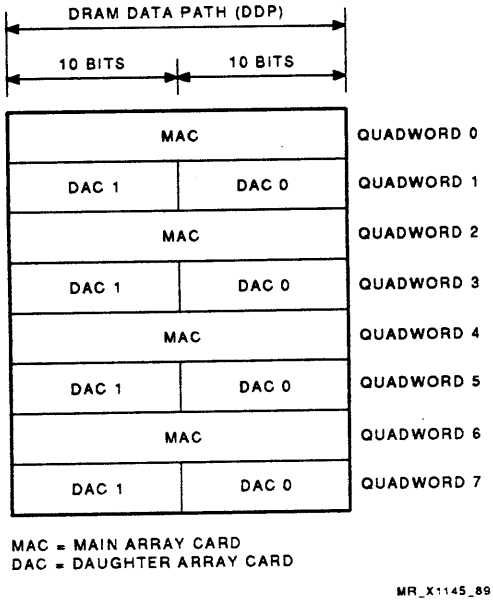


Figure 5–10 DRAM Data Bits for DAC0 and DAC1

5.2.2.6 DRAM Data Path Gate Array

Each MAC has four DDPs: DDP0, DDP1, DDP2, and DDP3. Figure 5–11 shows the four DDPs on the memory module. Figures 5–12 through 5–15 show each of the DDP gate arrays.

The DDP performs the following activities:

- Translates ECL to TTL and TTL to ECL.
- Provides a read data path (Figure 5–16).
- Buffers read data.
- Provides a write data path (Figure 5–17).
- Buffers write data.
- Provides a DRAM bypass path (used during BIST mode and the transfer of cache blocks having written full status).
- Provides single-step control timing and address pattern generations for self-tests.

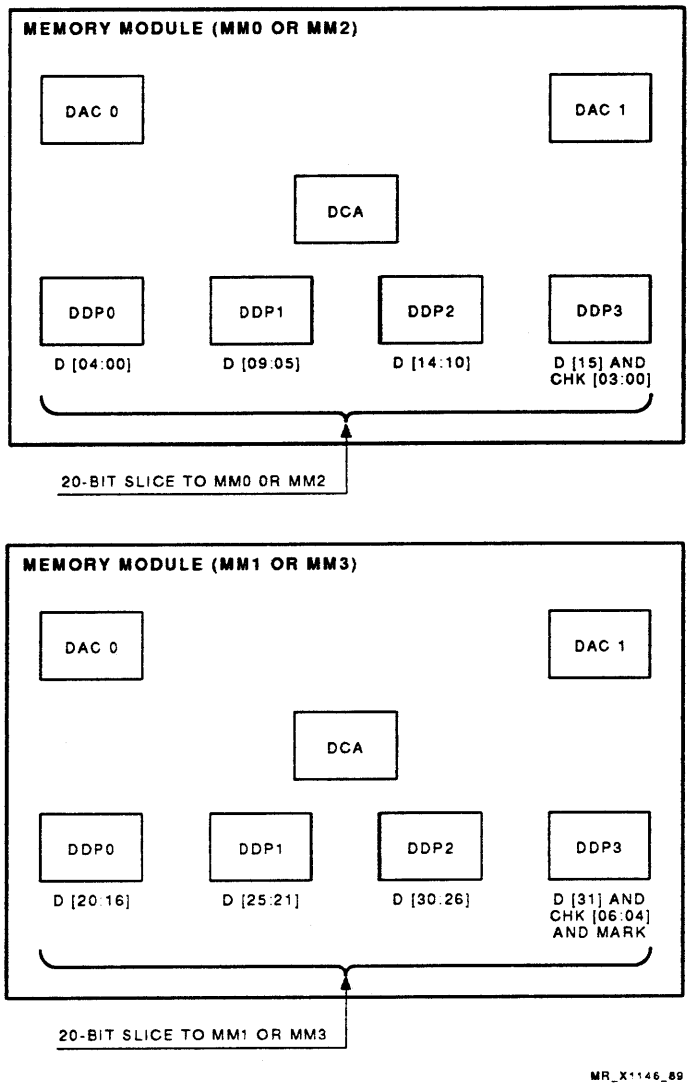


Figure 5–11 DDPs on the Memory Module

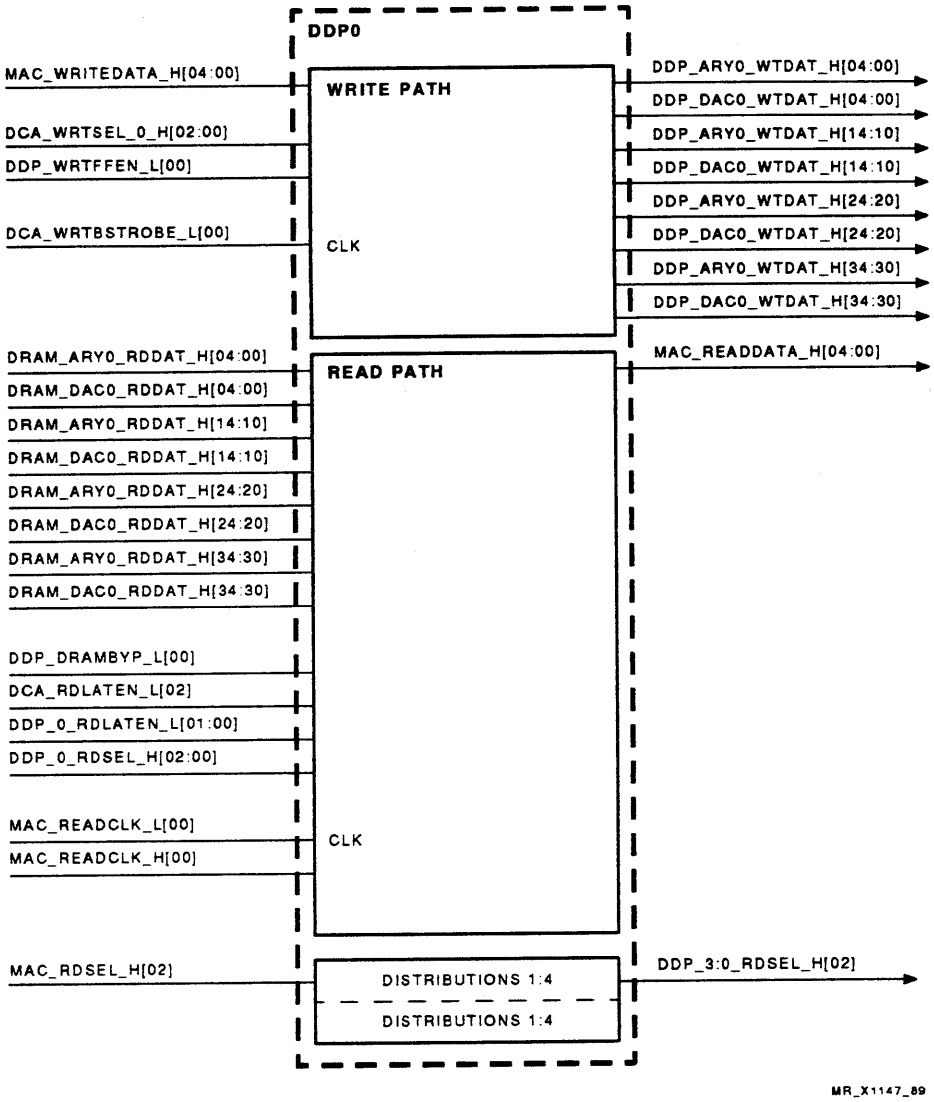


Figure 5–12 DDP0 Functional Block Diagram

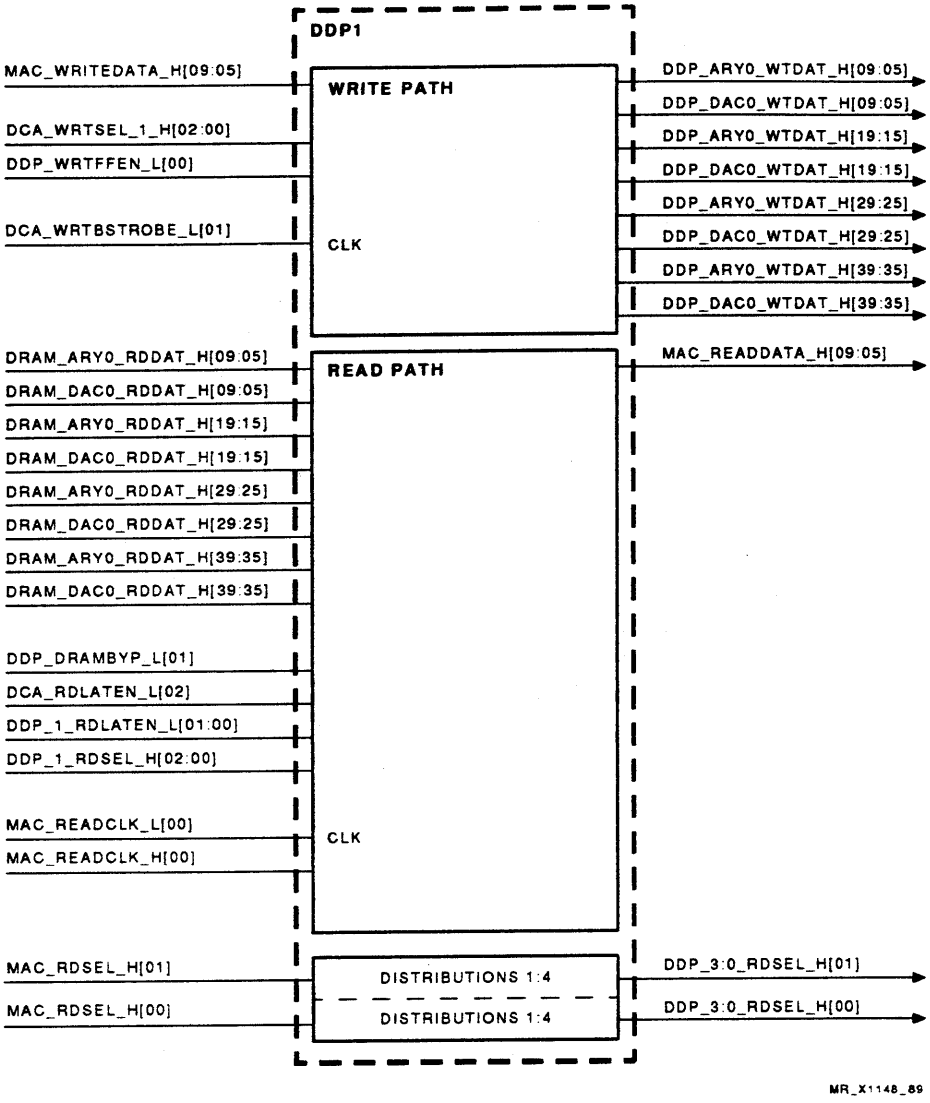


Figure 5–13 DDP1 Functional Block Diagram

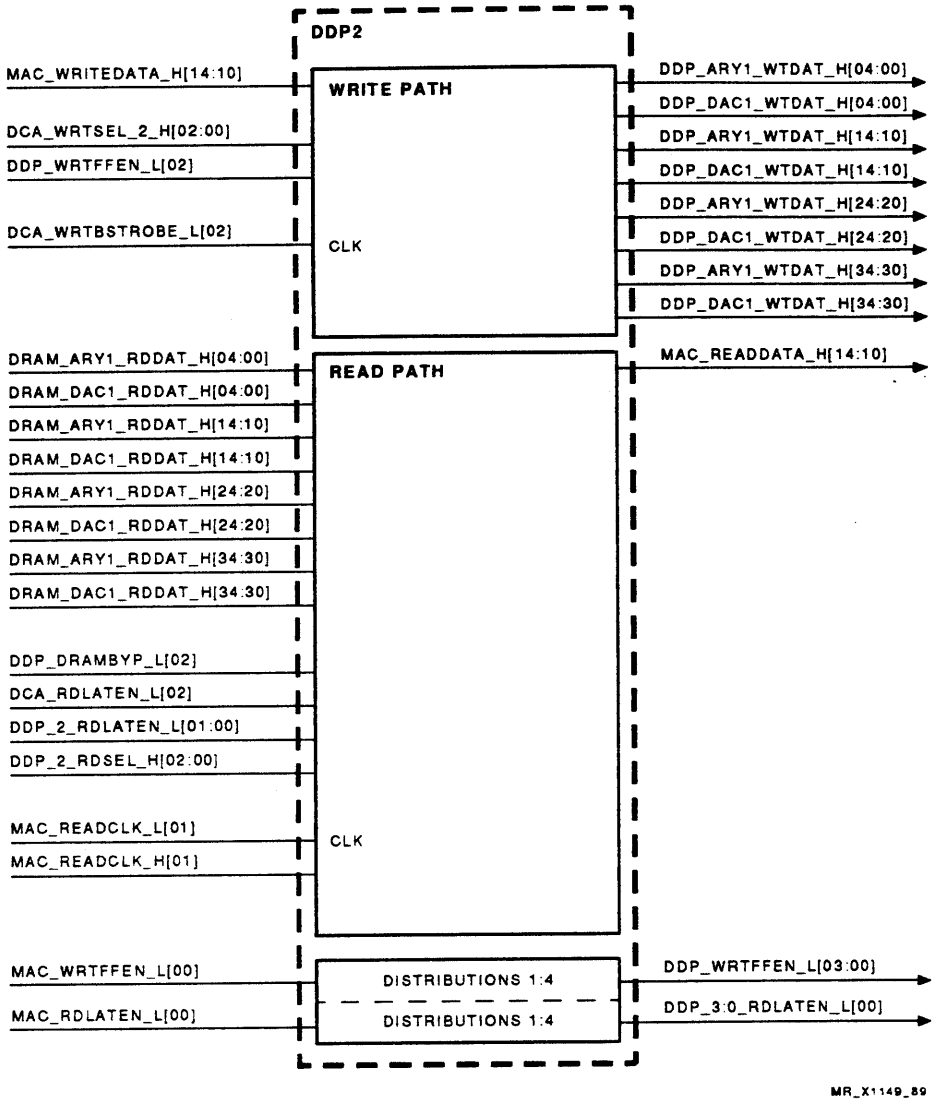


Figure 5–14 DDP2 Functional Block Diagram

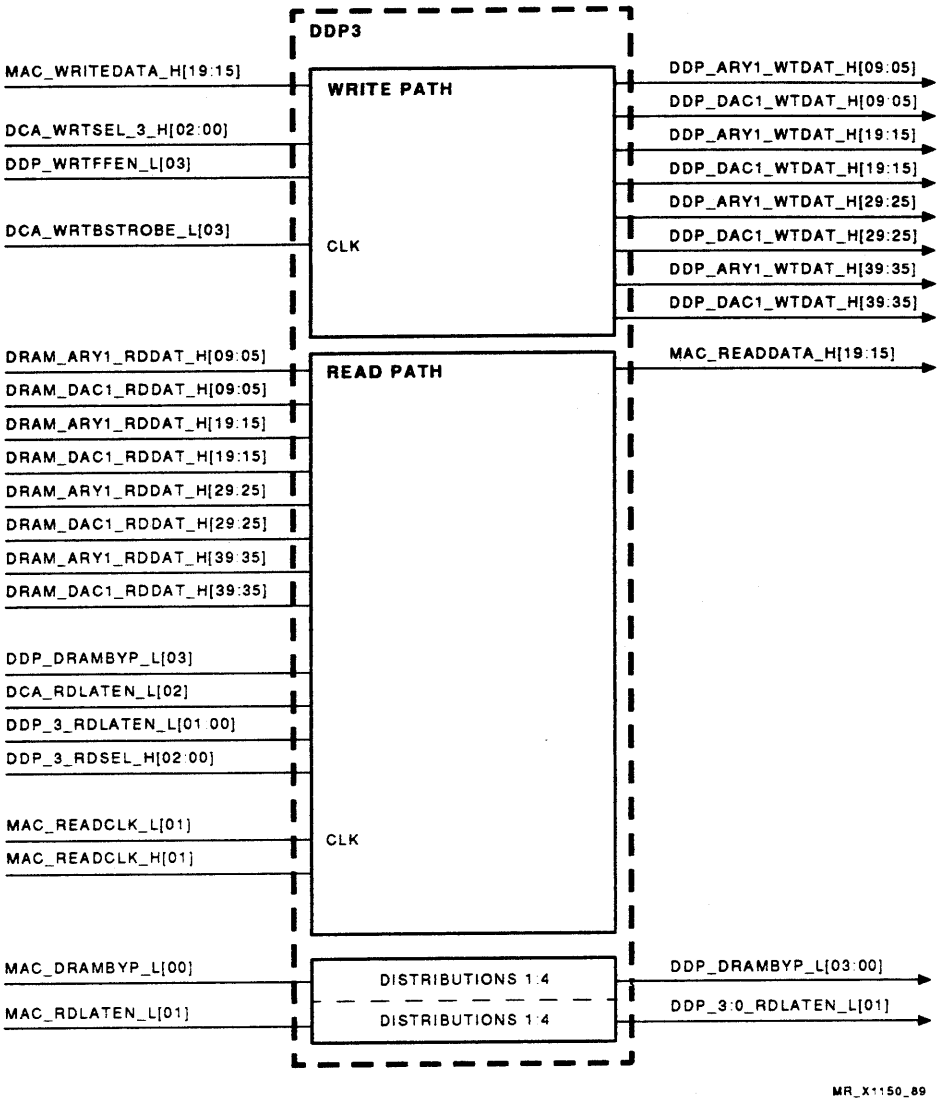


Figure 5–15 DDP3 Functional Block Diagram

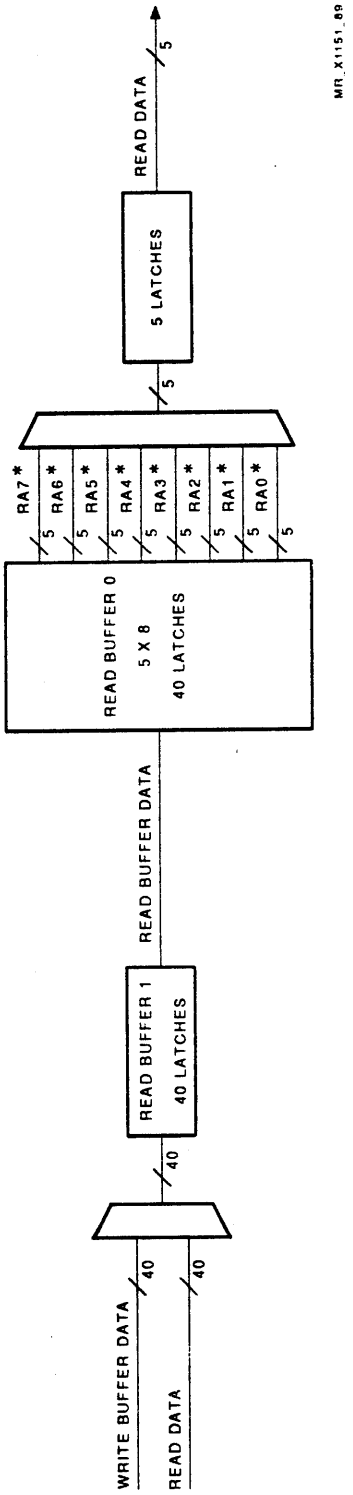


Figure 5–16 DDP Read Data Path

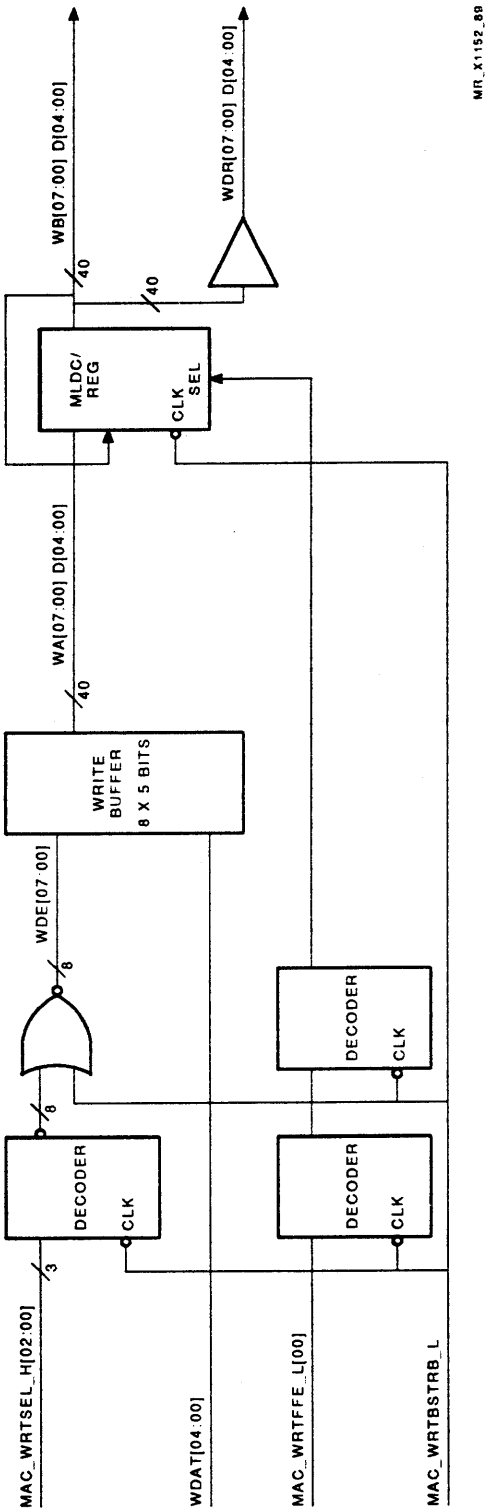


Figure 5–17 DDP Write Data Path

5.2.2.7 DRAM Control and Address Gate Array

One DCA is located on the MAC. The DCA contains the step mode controller. Figures 5–18 and 5–19 show the DCA gate array.

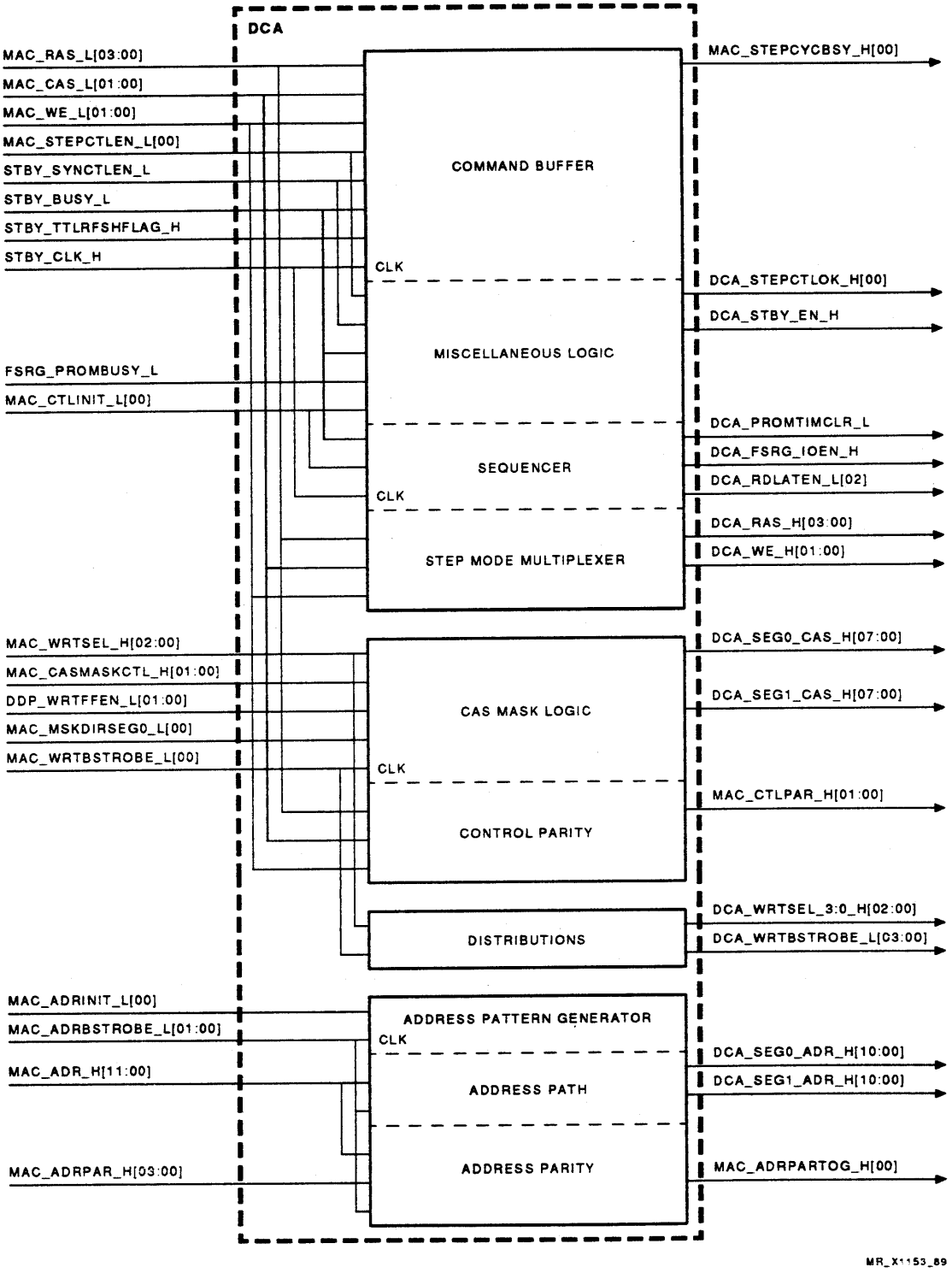


Figure 5–18 DCA Functional Block Diagram

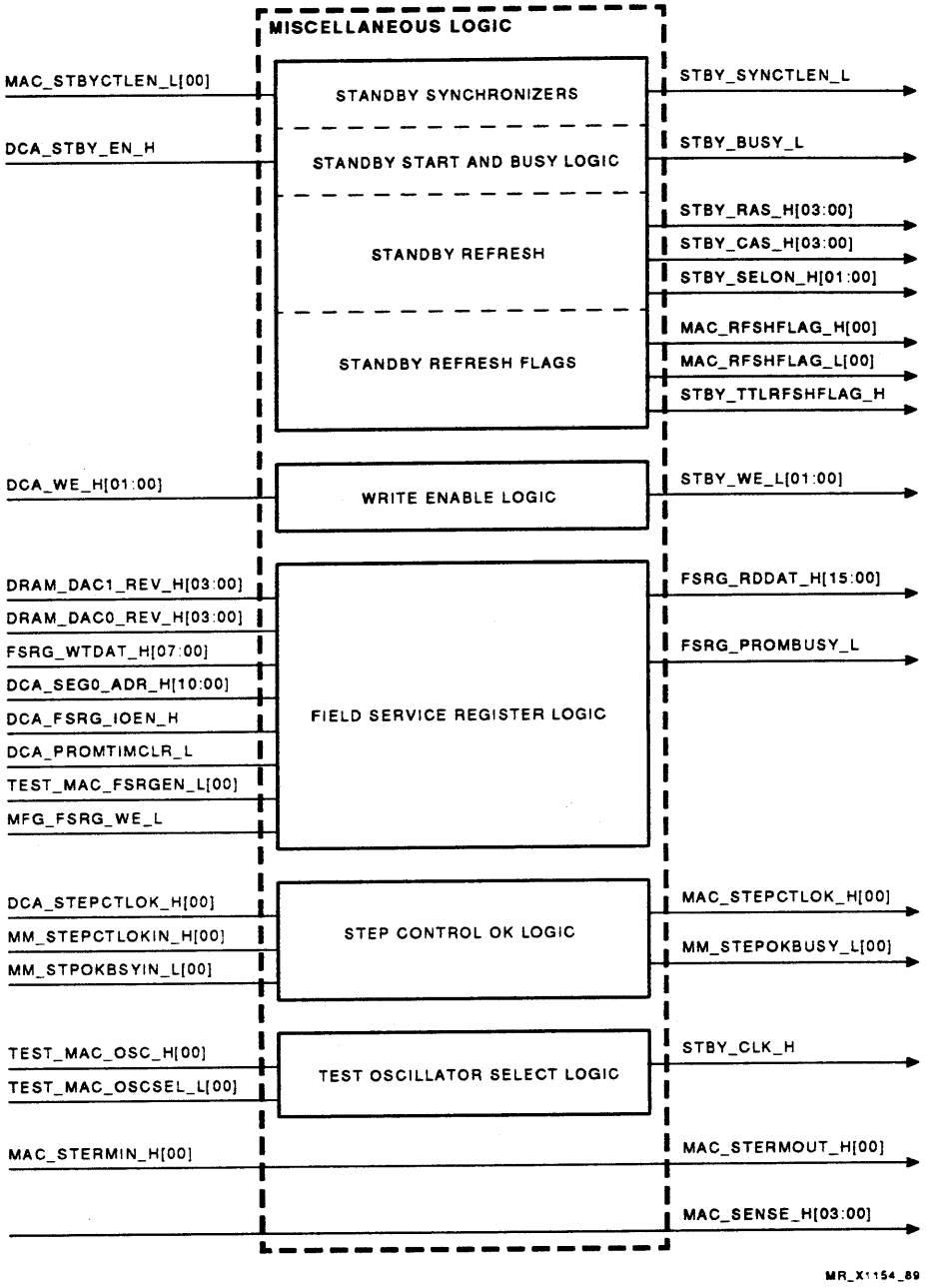


Figure 5–19 DCA Miscellaneous Control Logic

The DCA does the following:

Translates ECL to TTL and TTL to ECL.
Contains the CAS mask registers.
Executes step mode commands (read, write, EEPROM, revision read).
Buffers control signals to the DDPs.
Executes handshaking sequences when switching between timing modes.
Supports standby and Customer Services operations.

5.2.2.8 Interleaving

The SCU can interface to two completely independent memory subsystems. One subsystem consists of ACU0 and MMU0, the other of ACU1 and MMU1. If the DRAMs are the same size in both subsystems, the two subsystems can be interleaved and can have alternating addresses.

The two segments within each memory subsystem can also be interleaved. Figure 5–20 shows how the segments within the memory subsystem can be interleaved.

Each MMU has its own port on SCU. Each port has two segments. The two segments are interleaved on block boundaries. SCU can cycle four segments in parallel and can permit four simultaneous memory references. ACU can accept a memory request from the JBox on behalf of any of the CPU or I/O devices and can pass the request to the designated segment in memory.

The interleaving of segments is based on matching the memory access block size to the size of the cache blocks used in the system CPUs. Each memory module has two segments, 0 and 1. Each segment has two banks, 0 and 1. The memory addresses for the memory banks are interleaved on block boundaries, as listed in Table 5–2.

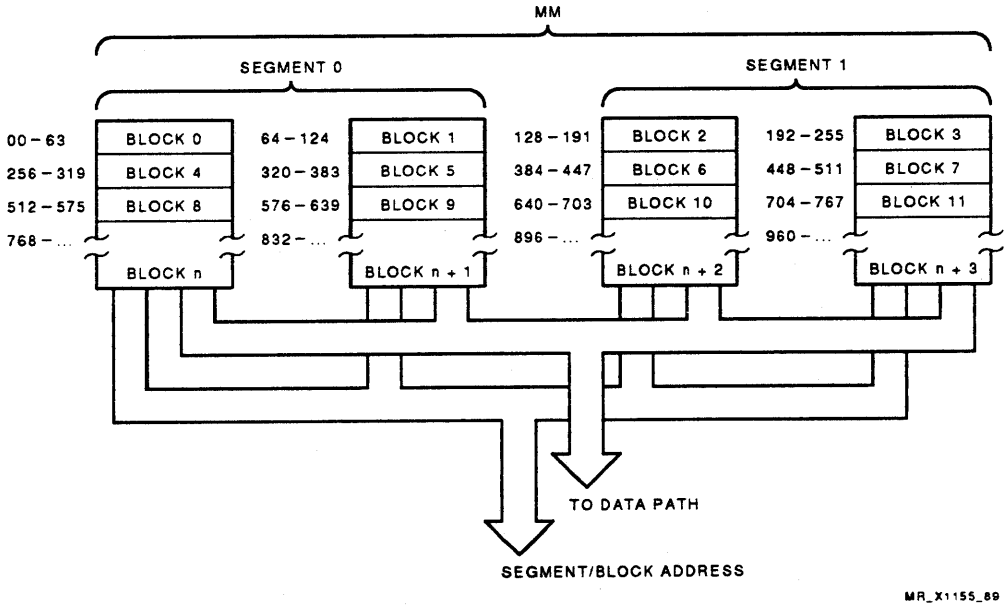


Figure 5–20 Interleaving Segments within the Memory Subsystem

Table 5–2 Block Boundaries

Block	Bytes	Segment	Bank
0	0–63	0	0
1	64–127	0	1
2	128–191	1	0
3	192–255	1	1

Table 5–3 lists the types of interleaving. Four-way is the maximum degree of interleaving, which can be used with MMU0 and MMU1. If the system has only one MMU, two-way interleaving is the maximum interleaving that is possible.

All banks in the same MMU must be interleaved in the same way. If one bank in an MMU is two-way interleaved, then the other three banks in that MMU must also be two-way interleaved. If one bank in an MMU is noninterleaved, the other used banks in that MMU must be noninterleaved.

Table 5–3 Degrees of Interleaving

Degree	Description
Noninterleaved	A bank is defined as noninterleaved if consecutive block addresses in that bank are 64 bytes apart.
Two-way interleaved	A bank is defined as two-way interleaved if consecutive block addresses in that bank are 2 × 64 bytes apart.
Four-way interleaved	A bank is defined as four-way interleaved if consecutive block addresses in that bank are 4 × 64 bytes apart.

5.2.2.9 ADRX Row and Column Address Bits for Interleaving

The ADRX MCAs latch and hold physical address bits. Under MMCX control, the ADRX MCAs address the MPAMM (for unit, segment, and bank) and send row and column address bits to the MMU. Tables 5–4 through 5–6 show how PA [32:26, 07, 06] are used to address unit, segment, and bank for non-, two-, and four-way interleaving. These tables use the following conventions:

B = Bit selecting the segment's bank
Cxx = Column bit xx
Rxx = Row bit xx
S = Bit selecting the unit's segment
U = Unit

Table 5–4 Noninterleaving

PA Bit	64 Mbytes	256 Mbytes	1024 Mbytes
06	R9	R9	R9
07	C9	C9	C9
26	Bank	R10	R10
27	Segment	C10	C10
28	Unit	Bank	R11
29	–	Segment	C11
30	–	Unit	Bank
31	–	–	Segment
32	–	–	Unit

Table 5–5 Two-Way Interleaving

	One Unit			Two Units		
PA Bit	64 Mbytes	256 Mbytes	1024 Mbytes	64 Mbytes	256 Mbytes	1024 Mbytes
06	Segment	Segment	Segment	Unit	Unit	Unit
07	C9	C9	C9	C9	C9	C9
26	R9	R9	R9	R9	R9	R9
27	Bank	C10	C10	Bank	C10	C10
28	Unit	R10	R10	Segment	R10	R10
29	–	Bank	C11	–	Bank	C11
30	–	Unit	R11	–	Segment	R11
31	–	–	Bank	–	–	Bank
32	–	–	Unit	–	–	Segment

Table 5–6 Four-Way Interleaving

PA Bit	64 Mbytes	256 Mbytes	1024 Mbytes
06	Unit	Unit	Unit
07	Segment	Segment	Segment
26	R9	R9	R9
27	C9	C9	C9
28	Bank	R10	R10
29	–	C10	C10
30	–	Bank	R11
31	–	–	C11
32	–	–	Bank

Table 5–7 shows the mapping for row and column bits for a noninterleave.

Table 5–7 Noninterleave Mapping

Row/Column Bit	PA Bit for Row	PA Bit for Column
09	06	07
10	26	27
11	28	29

Table 5–8 shows the mapping for the row and column bits for a four-way interleave.

Table 5–9 shows the mapping for the row and column bits for a two-way interleave.

Table 5–10 lists the addresses for two MMUs that are four-way interleaved.

Table 5–8 Four-Way Interleave Mapping

Row/Column Bit	PA Bit for Row	PA Bit for Column
09	26	27
10	28	29
11	30	31

Table 5–9 Two-Way Interleave Mapping

Row/Column Bit	PA Bit for Row	PA Bit for Column
09	26	07
10	28	27
11	30	29

Table 5–10 Two MMUs, Four-Way Interleaved

Memory	Address
MMU0 SEG0 BANK0	0, 4 ... 3FFFFC
MMU1 SEGO BANK0	1, 5 ... 3FFFFD
MMU0 SEG1 BANK0	2, 6 ... 3FFFFE
MMU1 SEG1 BANK0	3, 7 ... 3FFFFF
MMU0 SEGO BANK1	400000, 400004 ... 7FFFFC
MMU1 SEGO BANK1	400001, 400005 ... 7FFFFD
MMU0 SEG1 BANK1	400002, 400006 ... 7FFFFE
MMU1 SEG1 BANK1	400003, 400007 ... 7FFFFF

Table 5–11 lists the addresses for one MMU that is two-way interleaved.

Table 5–11 One MMU, Two-Way Interleaved

Memory	Addresses
MMU0 SEGO BANK0	0, 2 ... 1FFFFE
MMU0 SEG1 BANK0	1, 3 ... 1FFFFF
MMU0 SEG0 BANK1	200000, 200002 ... 3FFFFE
MMU0 SEG1 BANK1	200001, 200003 ... 3FFFFF

If the system has two MMUs, and one MMU has a broken bank, the remaining seven good banks can be configured in several alternative ways. These alternative ways range from minimum interleaving with maximum usable memory to maximum interleaving with considerably reduced usable memory.

To maintain maximum memory bandwidth for a memory subsystem in which a bank is only partly broken and one-fourth of its address space is bad, the following sequence of events occurs:

1. Retain four-way interleaving.
2. Map out the bad one-fourth bank.
3. Map out the corresponding one-fourth in three other banks.

Mapping out parts of banks is done by the SPU, not the SCU hardware.

Table 5–12 lists the addresses for two MMUs in which one bank is broken. MMU0 uses four banks that are two-way interleaved and MMU1 uses three banks that are noninterleaved. No interleaving exists between MMU0 and MMU1.

Table 5–12 Two MMUs, One Bank Broken — No Interleaving Between Banks

Memory	Interleave	Address
MMU0 SEGO BANK0	Two-way	0, 2 ... 1FFFFE
MMU0 SEG1 BANK0	Two-way	1, 3 ... 1FFFFF
MMU0 SEG0 BANK1	Two-way	200000, 200002 ... 3FFFFE
MMU0 SEG1 BANK1	Two-way	200001, 200003 ... 3FFFFF
MMU1 SEGO BANK0	One-way	400000, 400001 ... 4FFFFF
MMU1 SEG0 BANK1	One-way	500000, 500001 ... 5FFFFF
MMU1 SEG1 BANK0	One-way	600000, 600001 ... 6FFFFF
MMU1 SEG1 BANK1	Broken	

Table 5–13 lists the addresses for two MMUs in which one bank is broken. MMU0 uses three banks that are two-way interleaved and MMU1 uses three banks that are two-way interleaved. One good bank is not used so that two-way interleaving can be maintained across the remaining six banks.

Table 5–14 lists the addresses for two MMUs in which one bank is broken. MMU0 uses two banks that are four-way interleaved and MMU1 uses two banks that are four-way interleaved. Three good banks are not used so that four-way interleaving can be maintained across the remaining four banks.

Table 5–13 Two MMUs, One Bank Broken — Two-Way Interleaving Between Banks

Memory	Address
MMU0 SEGO BANK0	0, 2 ... 1FFFFE
MMU1 SEG0 BANK0	1, 3 ... 1FFFFF
MMU0 SEG0 BANK1	200000, 200002 ... 3FFFFE
MMU1 SEG0 BANK1	200001, 200003 ... 3FFFFF
MMU0 SEG1 BANK0	400000, 400002 ... 5FFFFE
MMU1 SEG1 BANK0	400001, 400003 ... 5FFFFF
MMU0 SEG1 BANK1	Not used
MMU1 SEG1 BANK1	Broken

Table 5–14 Two MMUs, One Bank Broken — Four-Way Interleaving Between Banks

Memory	Address
MMU0 SEGO BANK0	0, 4 ... 3FFFFC
MMU1 SEG0 BANK0	1, 5 ... 3FFFFD
MMU0 SEG1 BANK0	2, 6 ... 3FFFFC
MMU1 SEG1 BANK0	3, 7 ... 3FFFFF
MMU0 SEG0 BANK1	Not used
MMU1 SEG0 BANK1	Not used
MMU0 SEG1 BANK1	Not used
MMU1 SEG1 BANK1	Broken

Table 5–15 lists the addresses for two MMUs that are two-way interleaved. MMU0 uses 4-Mbit chips and MMU1 uses 1-Mbit chips. No interleaving exists between MMU0 and MMU1.

Table 5–16 lists the addresses for one MMU with one broken bank. The MMU uses three banks that are noninterleaved.

Table 5–17 lists the addresses for one MMU with one broken bank. The MMU uses two banks that are two-way interleaved. One good bank is not used.

Table 5–15 Two MMUs, 4-Mbit MMU0 and 1-Mbit MMU1

Memory	Address
MMU0 SEG0 BANK0	0, 2 ... 7FFFFE
MMU0 SEG1 BANK0	1, 3 ... 7FFFFF
MMU0 SEG0 BANK1	800000, 800002 ... FFFFFE
MMU0 SEG1 BANK1	800001, 800003 ... FFFFFF
MMU1 SEG0 BANK0	1000000, 1000002 ... 11FFFFE
MMU1 SEG1 BANK0	1000001, 1000003 ... 11FFFFF
MMU1 SEG0 BANK1	1200000, 1200002 ... 13FFFFE
MMU1 SEG1 BANK1	1200001, 1200003 ... 13FFFFF

Table 5–16 One MMU, Three Banks Used — Noninterleaved

Memory	Address
MMU0 SEG0 BANK0	0, 1 ... FFFFFF
MMU0 SEG0 BANK1	100000, 100001 ... 1FFFFF
MMU0 SEG1 BANK0	200000, 200001 ... 2FFFFF
MMU0 SEG1 BANK1	Broken

Table 5–17 One MMU, Two Banks Used — Two-Way Interleaved

Memory	Address
MMU0 SEG0 BANK0	0, 2 ... 1FFFFE
MMU0 SEG1 BANK0	1, 3 ... 1FFFFF
MMU0 SEG0 BANK1	Not used
MMU0 SEG1 BANK1	Broken

5.2.2.10 Data Organization

The memory module is divided into two segments, 0 and 1. Each segment contains two banks, 0 and 1. Each of the four banks contains 160 DRAMs. Figure 5–21 shows the partitioning of the DRAM data on the MAC and DAC. Bank 0 of segment 0 consists of 80 MAC DRAMs, 40 DAC0 DRAMs, and 40 DAC1 DRAMs.

The memory subsystem simultaneously accesses the same bank in all four MACs. When bank 0 of segment 0 is accessed in MM0, bank 0 of segment 0 in MM1, MM2, and MM3 are also accessed. One-fourth of the DRAMs in each memory module are accessed simultaneously.

Each memory module processes 20-bit slices of the eight quadwords (640 bits) in a 64-byte block transfer. (Four modules simultaneously write 640 DRAMs.) For memory writes, the MDPX MCA sends 20-bit slices to the memory module DDP gate arrays, which buffer the write data. DDPs on each memory module store eight 20-bit transfers and send all 160 bits to the DRAMs. For memory reads, the DRAMs send 160 bits to the DDPs, which send eight 20-bit slices to the MDPX MCAs. Figure 5–22 shows how these slices are sent.

5.2.2.11 Memory Module Bit Configuration

The memory subsystem stores slices of a quadword in four memory modules. Figure 5–23 shows bit positions for a 20-bit slice of a quadword within a memory module. Bit [00] corresponds to the following:

Data bit [00] of the lower longword (even longword)
Data bit [16] of the upper longword (odd longword)

Bit [19] corresponds to the following:

Check bit [03] of the lower longword
Mark bit on the upper longword

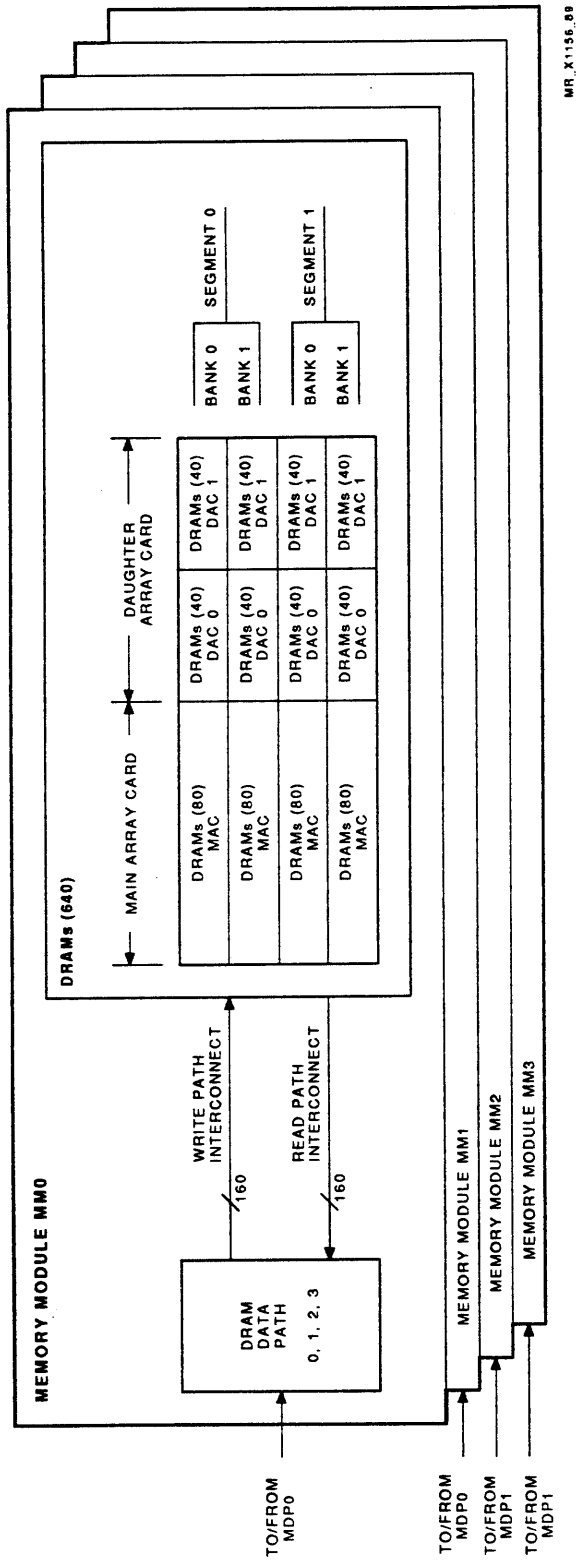


Figure 5–21 Data Partitioning

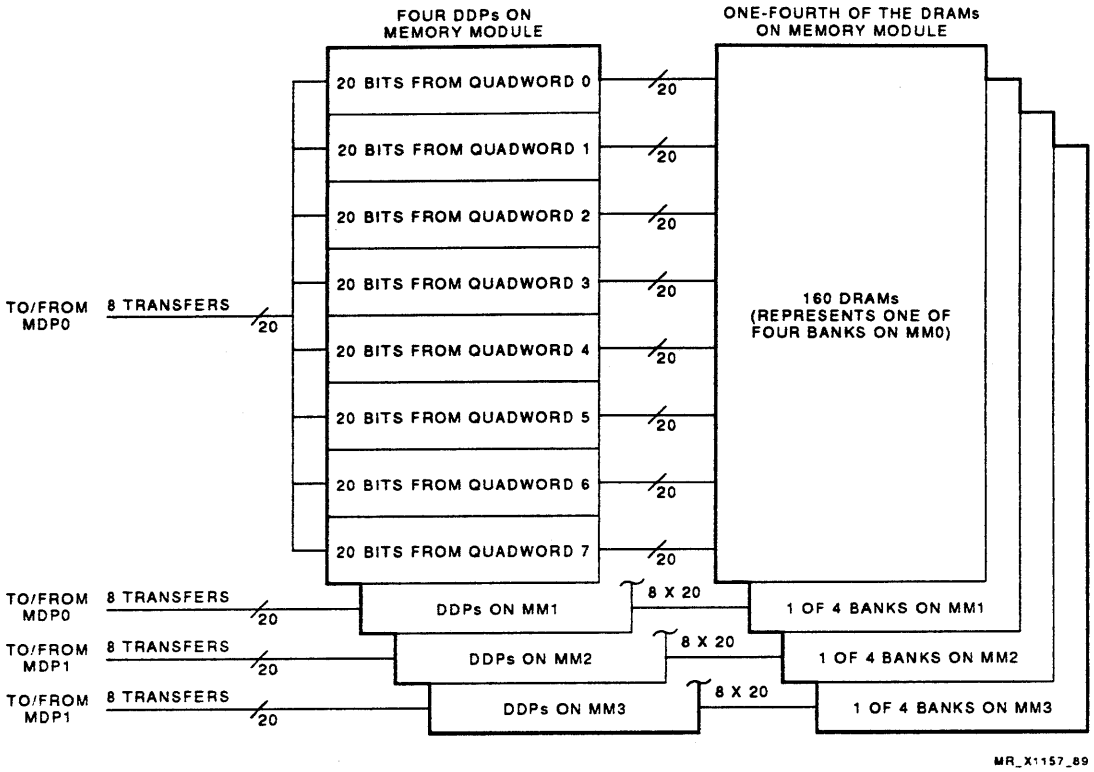


Figure 5–22 DDPs Sending Eight 20-Bit Slices

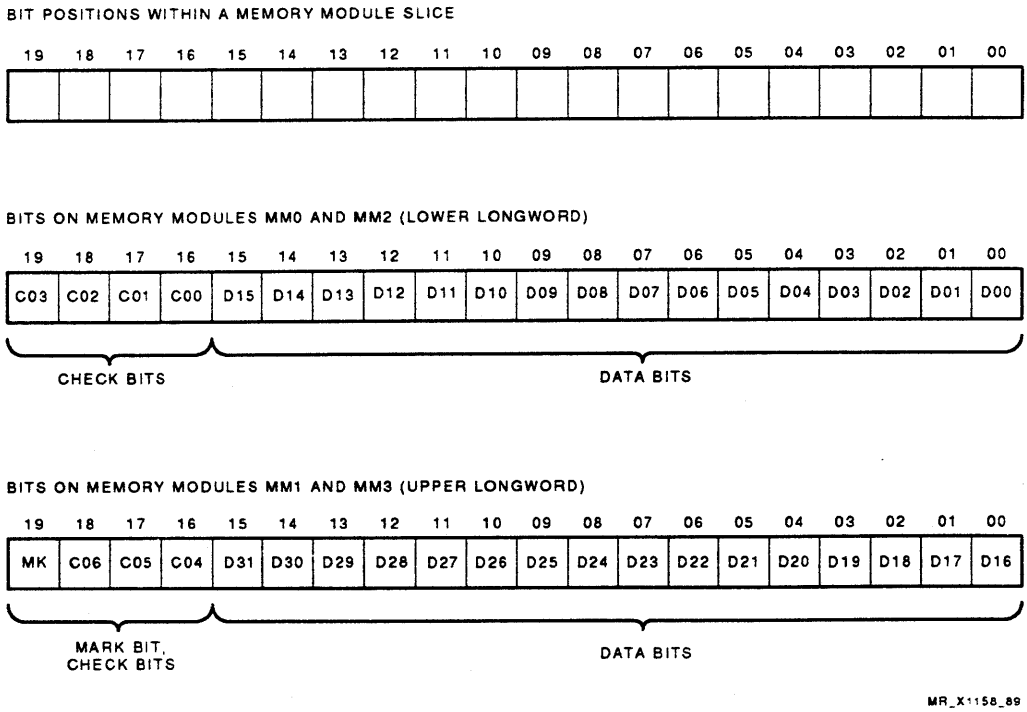


Figure 5–23 Quadword Bit Configuration

Figure 5–24 shows how the quadword data bits are distributed in the MACs and DACs.

5.2.2.12 Storing Quadwords

Figure 5–24 shows the even and odd longwords. MM0 and MM1 store the odd longword. MM2 and MM3 store the even longword. MM0 and MM2 store the same bits within their respective longwords. MM1 and MM3 store the same bits within their respective longwords.

For quadword 0, MM0 receives a 20-bit slice and MM1 receives a 20-bit slice of the odd longword's 40 bits. MM2 receives a 20-bit slice and MM3 receives a 20-bit slice of the even longword's 40 bits.

For quadword 1, MM0 and MM1 receive the even longword (40 bits). In MM0 and in MM1, DAC0 receives 10 bits and DAC1 receives 10 bits. MM2 and MM3 receive the odd longword (40 bits). In MM2 and in MM3, DAC0 receives 10 bits and DAC1 receives 10 bits.

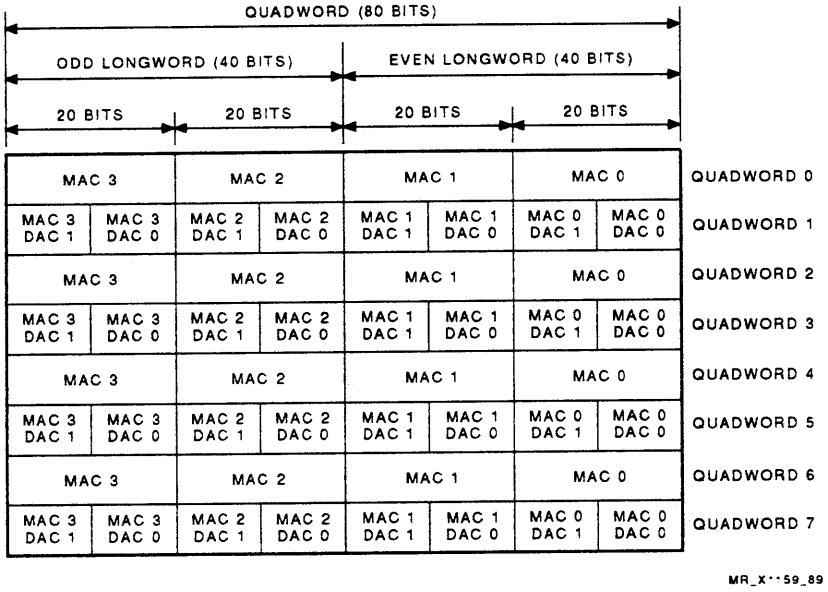


Figure 5–24 Distribution of Quadword Data Bits

Figure 5–25 shows [19:00] of a 20-bit slice. MAC DRAMs store [19:00] of quadwords 0, 2, 4, and 6. DAC0 stores [09:00] of quadwords 1, 3, 5, and 7. DAC1 stores [19:10] of quadwords 1, 3, 5, and 7.

DDP0, DDP1, DDP2, and DDP3 each send five bits of data to the DRAMs. For example, DDP0 sends [04:00] of quadwords 0, 2, 4, and 6 to the MAC DRAMs. DDP0 sends [04:00] of quadwords 1, 3, 5, and 7 to DAC0 DRAMs.

DDP3 sends [19:15] of quadwords 0, 2, 4, and 6 to the MAC DRAMs. DDP3 sends [19:15] of quadwords 1, 3, 5, and 7 to the DAC1 DRAMs.

The DDP gate arrays contain independent read and write buffers that allow a read data transfer to occur in one segment while a write data transfer occurs in another segment. Figure 5–30 shows the independent read and write buffers, the DRAM bypass path, and the write select decoder.

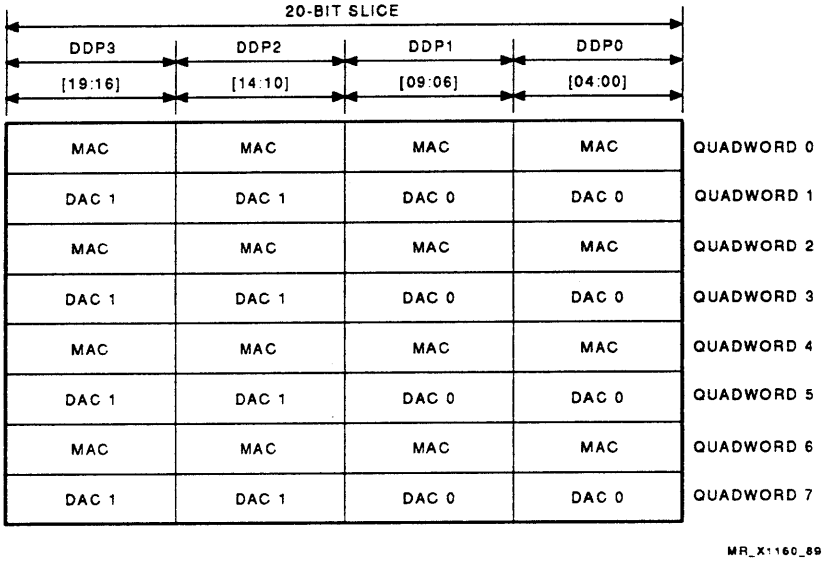


Figure 5–25 DRAMs Storing Bits [19:00]

5.2.3 Service Processor Unit

The SPU is based on a BI MicroVAX system installed in a single BI card cage. The processor consists of a service processor module (SPM), 2 Mbytes of ECC memory, KFBTA (AIO — disk controller), DEBNT (AIE — NI/TK50 controller), and a scan control module (SCM).

SPM contains the SPU-to-JBox adapter (SJA) MCA. Two cables, one for the logical (test) interface and the other for the scan interface, plug onto the front edge of the SPM module and the SCM module and are connected to the SCU planar module. Figure 5–2 shows the SPU connectors on the SCU planar module. SPU asserts the following signals:

- **Request step** — The memory modules enter the step mode. This supports single-stepping of the system clocks.
- **Request standby** — The memory modules enter standby mode. This mode supports the scan operation and powerfail.

5.2.3.1 Initializing MMUs

SPU initializes each MMU by sending an SPU_MAC_CTLINIT_L signal to the memory modules. The following summarizes the initializing sequence:

1. Power is applied to the machine. The memory modules are placed in standby mode.
2. Scan-based testing of ACU is performed.
3. ACU is initialized into step mode.
4. SPU releases MMU from standby by way of a handshake sequence. MMU and ACU are both in step mode.
5. ACU and MMU exit step mode using a handshake sequence.
6. MMU and ACU enter their usual operating mode.

5.2.3.2 Modes of Operation

The memory subsystem is in one of the following *timing* modes:

Normal
Step
Standby
Address pattern generation

These timing modes support the following levels of system operation:

Normal — Normal timing
Burst clocking — Step mode timing
Scan — Standby mode timing
Power loss — Standby mode timing
Built-in self-test — Address pattern generation timing

SPU can switch between modes. To preserve data integrity, mode switching must be done in a specific order. For more detail, see Section 5.8.

5.3 Array Control Unit — Functional Description

This section provides a functional description of the ACU MCUs (DAX and DBX) and and the ACU MCAs (MMCX, MCDX, and MDPX). The ACU controls the main memory unit, DRAMs, and DRAM data paths on the memory modules. ACU also provides the SCU memory data path from the memory modules to the JBox data switch. For addressing the DRAMs, the JBox holds the row and column addresses in the ADRX MCAs. ACU uses an index value that the JBox sends with the command to select the appropriate address in the ADRX MCA. Figure 5–26 shows the ACU block diagram.

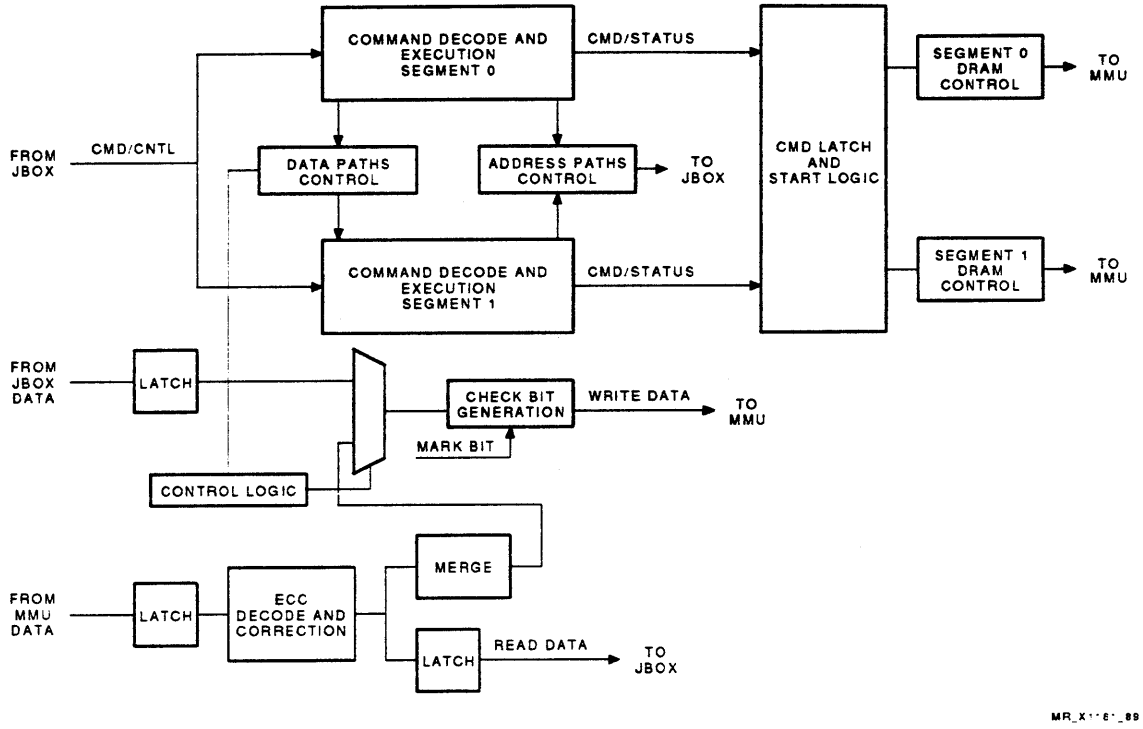


Figure 5–26 ACU Block Diagram

5.4 MMCX MCA

The MMCX MCA provides the main memory control for the memory operations described in Section 5.9. MMC0, located on the DB0 MCU, controls the receipt and transmission of memory commands to and from the CCU MCU, MMU0 memory modules, and SPU. MMC1, located on the DB1 MCU, controls the receipt and transmission of memory commands to and from the CCU MCU, MMU1 memory modules, and SPU. Figure 5–27 shows the the major control areas in the MMCX MCA.

The MMCX MCA contains the following logic:

- **Input buffer** — The input buffer receives the CCU command, tag starting address, data switch mask bits, MMUX and MDPX status, and MCU fatal errors. The input buffer decodes the status and commands, sending internal commands to the command latch, command buffer control, read buffer control, write buffer control, and output buffer.
- **Command buffer control** — The command buffer control logic contains the segment 0 and 1 controllers, segment 0 and 1 address strobe enables, and segment 0 and 1 command start controls. The command buffer control also contains the refresh control, column address select, and step mode controller.
- **Command latch** — The command latch contains the segment 0 and 1 command buffers. The command latch sends the index to CCU, the read command to the read buffer control, the write command to the write buffer control, the EEPROM read command to the read buffer control, and the EEPROM write command to the write buffer control.
- **Write buffer control** — The write buffer control logic contains the segment 0 and 1 write controllers, the CAS mask control, and the write select control.
- **Read buffer control** — The read buffer control logic contains the read data latch controller (read buffer 1), data output latch controller (read buffer 0), error controller, quadword counter, read select control, and read-modify-write select control.
- **Output buffer** — The output buffer sends commands to the CCU MCU; control information to the MDPX, MCDX, and ADRX MCAs; and set attention to the CDCX MCA.

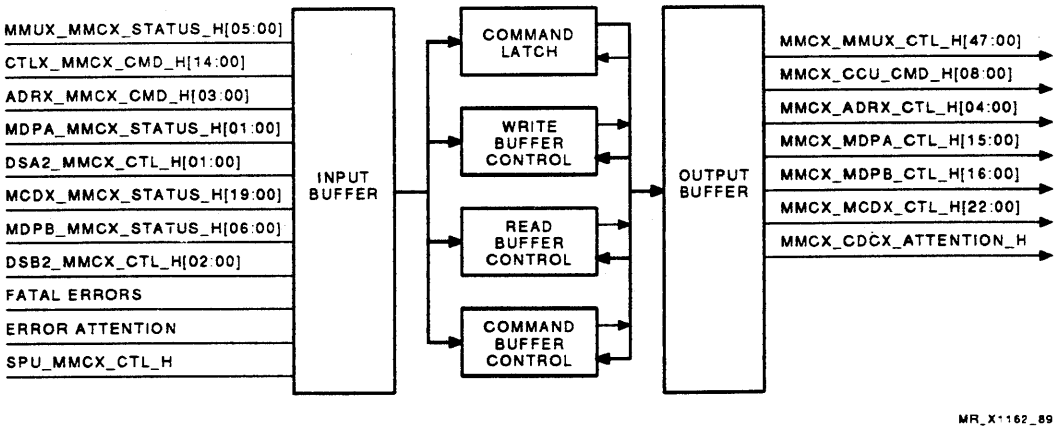


Figure 5–27 MMCX Control Areas

5.4.1 Command Buffer Control

The command buffer control inputs and outputs are shown in Figure 5–28. The command buffer control in the MMCX MCA receives the following internal MMCX signals:

- Write done, read bus busy, read data ready, single-step mode, and BIST from the MMCX input buffer
- Commands from the MMCX command latch
- Data received in the data input latch, write data ready, and read-modify-write status from the MMCX write buffer control
- Read latch busy and single-step status from read buffer control

The command buffer control in the MMCX MCA sends the following:

- Address parity error to the input buffer control
- Address select, command latch select, and single-step mode to the command latch
- Command buffer status to the write buffer control
- Single-step mode, OK received status, and command buffer status to the read buffer control

Through the MMCX output buffer, the command buffer control in the MMCX MCA sends the following:

- Address strobe and step control enable to the MMUX
- Column address select to the ADRX MCA
- Normal buffer available to the CCU MCU
- Refresh required, segment abort, single-step mode, start, and BIST buffer available to the MCDX MCA

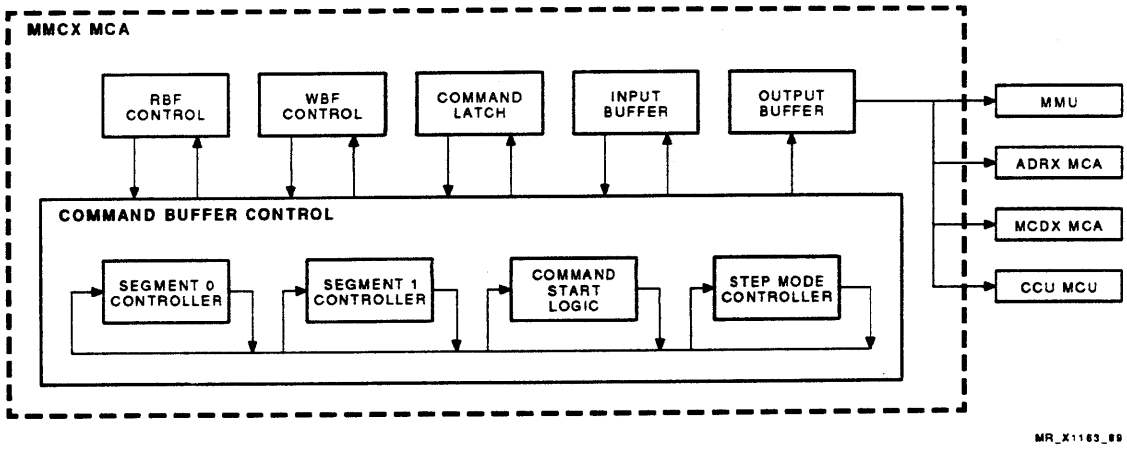


Figure 5–28 Command Buffer Control

5.4.1.1 Command Buffer Controller

Each segment has a command buffer controller in the MMCX MCA. The command buffer controller controls the segment's command latch. Figure 5–29 shows the states for the command buffer controller.

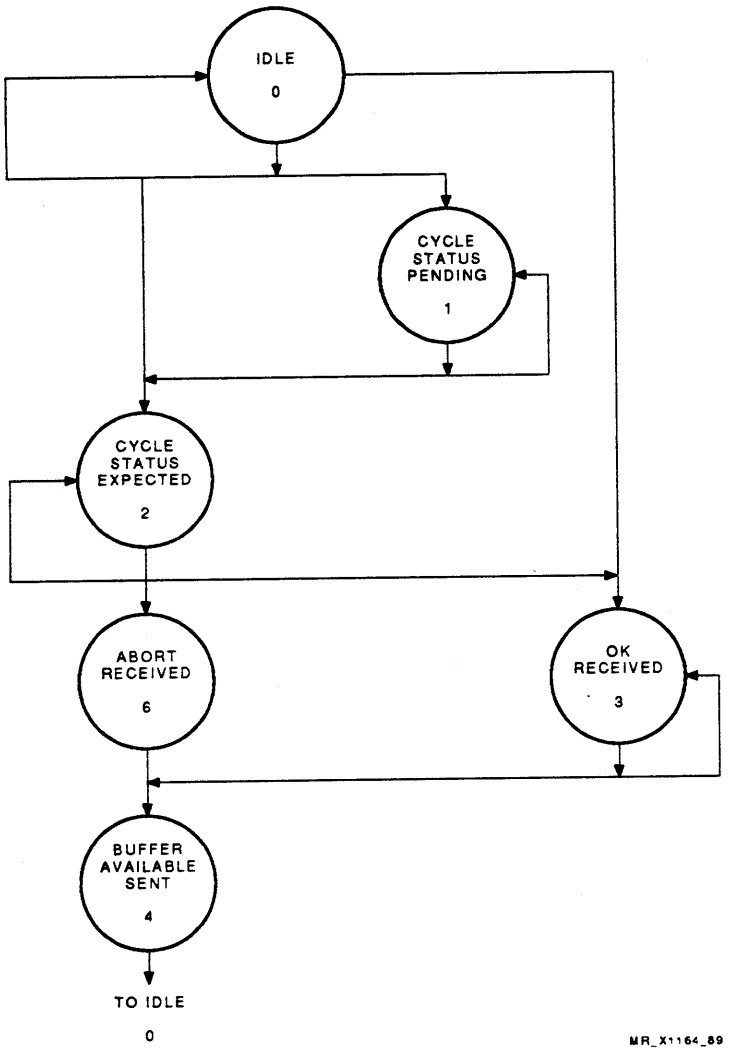


Figure 5–29 Command Buffer Controller

For reads and writes, the controller enters the following states:

1. **Idle state.**
2. **Cycle status pending or cycle status expected states** — The MMCX monitors read and match the status with the appropriate read command. Each segment can have an outstanding return data read command. CCU sends cycle status to MMC.
3. **OK received state or abort state** — CCU can send OK or abort cycle status to the MMCX.
4. **Buffer available sent state** — The MMCX sends buffer available for each segment.

For more detail on the memory operations, see Section 5.9.

5.4.2 Segment Controller

The segment controller starts the segment in the memory module during memory operations. The segment receives row and column addresses for each operation from the ADRX MCAs. The segment controller controls the loading of the row and column addresses into the DRAM control and address gate arrays in the memory module.

The states of the controller can be divided into the following three areas:

Starting the segment controller
Loading the row address
Loading the column address

The MMCX MCA sends the following signals:

- **Index value, row, and column select to the ADRX MCA** — The ADRX MCA uses the index value to select an address receive latch in which a physical address is stored. The ADRX MCA uses the physical address to generate row and column addresses to MMU.
- **Address strobes to the MMUX** — The DCA in the memory module uses the address strobes to latch the row and column address bits coming from the ADRX MCA.
- **Start to the MCDX MCA** — MCDX receives the command and start from the MMCX, decodes the command, and generates RAS (for bank and segment), CAS (for segment), and WE (for segment) to address the DRAMs.

Figure 5–30 shows the segment address lines addressing the DRAMs in each segment of the memory module.

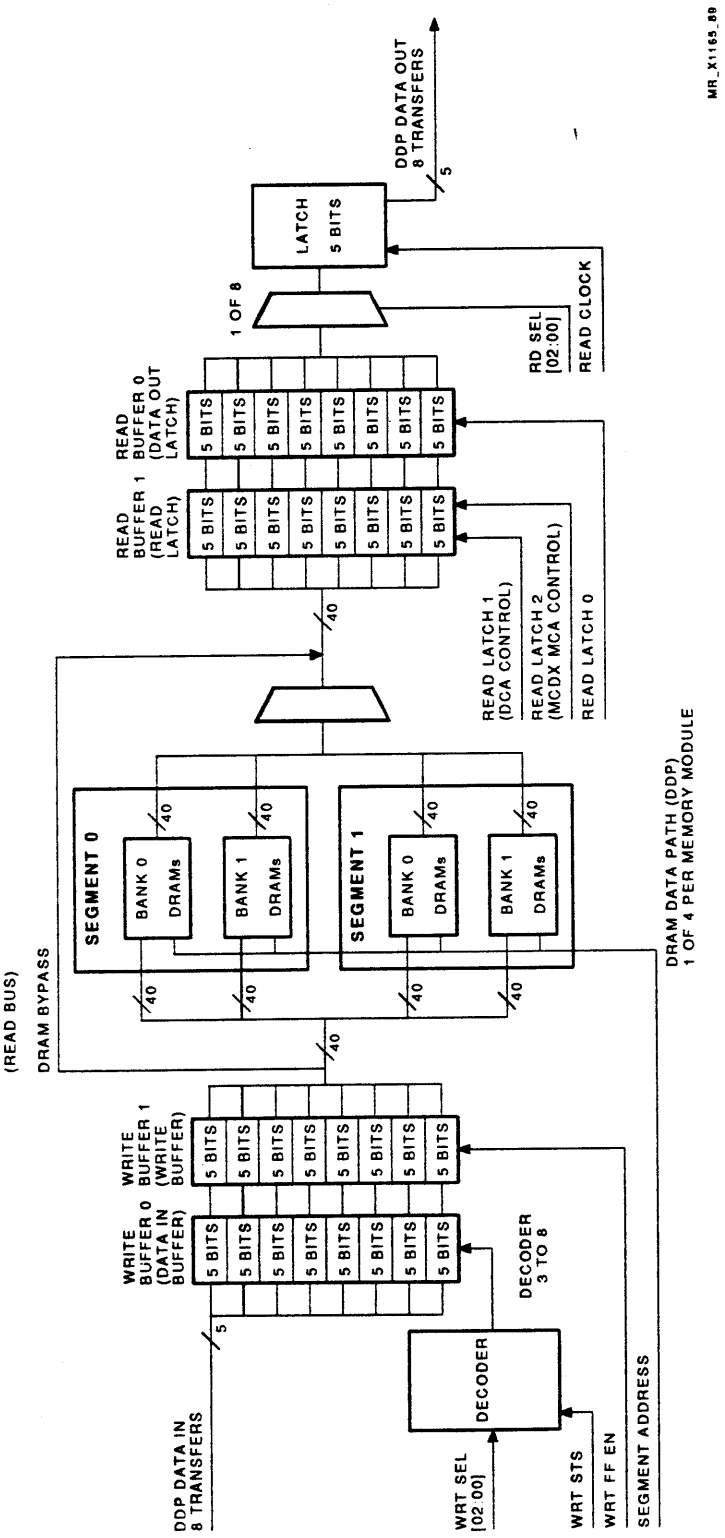


Figure 5–30 Read and Write Data Paths in the Memory Module

5.4.2.1 Starting the Segment Controller

If the segment is ready and the MMCX command latch has received a command, the segment controller starts the operation. Figure 5–31 shows the starting segment controller states.

For reads, the segment controller moves from the idle state to the load row address state.

For writes, the segment controller moves from the idle state to the waiting for resources state. The source and destination data paths to and from the data switch must be available.

MCDX generates the DRAM control signals (RAS, CAS, WE) and determines when the data is valid. MCDX sends read data ready several clock ticks before the data is actually valid because of the necessary protocol between MMCX and CCU. MMCX sends the return data (or error) command to the JBox. The JBox responds with send data.

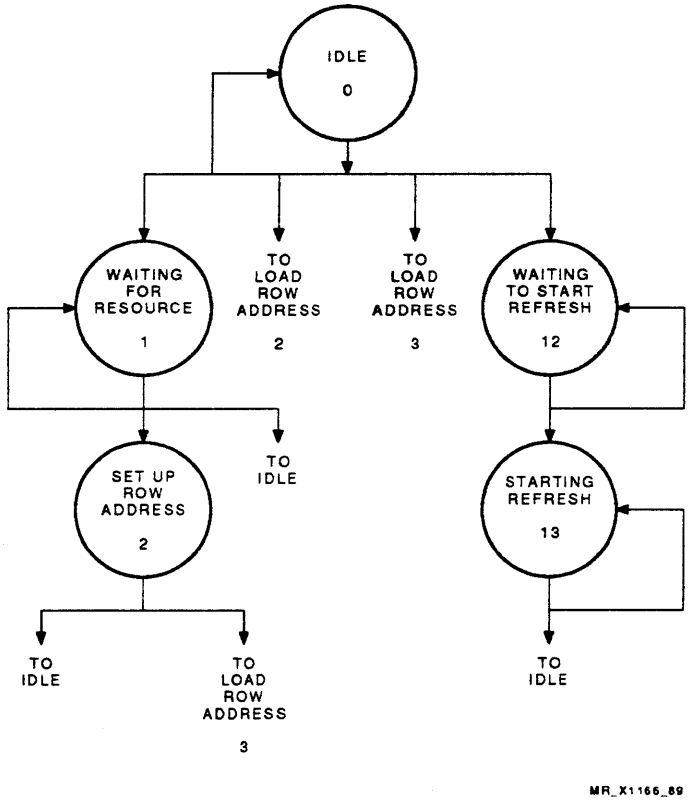


Figure 5–31 Starting the Segment Controller States

5.4.2.2 Loading the Row and Column Addresses

For reads and writes, if the segment is not busy, the segment controller moves from the load row address states to the set up column address states. If the segment is busy, the controller moves from the load row address state to the hold row address state.

The DCA in the memory module uses the address strobe signals from the MMCX MCA to latch the row and address bits from the ADRX MCA.

The MCDX asserts RAS and WE when MMCX sends start. MMCX sends write data ready to MCDX. MCDX uses write data ready to send CAS to MMU.

Figure 5–32 shows the loading row address segment controller states. Figure 5–33 shows the loading of the column address for the segment controller.

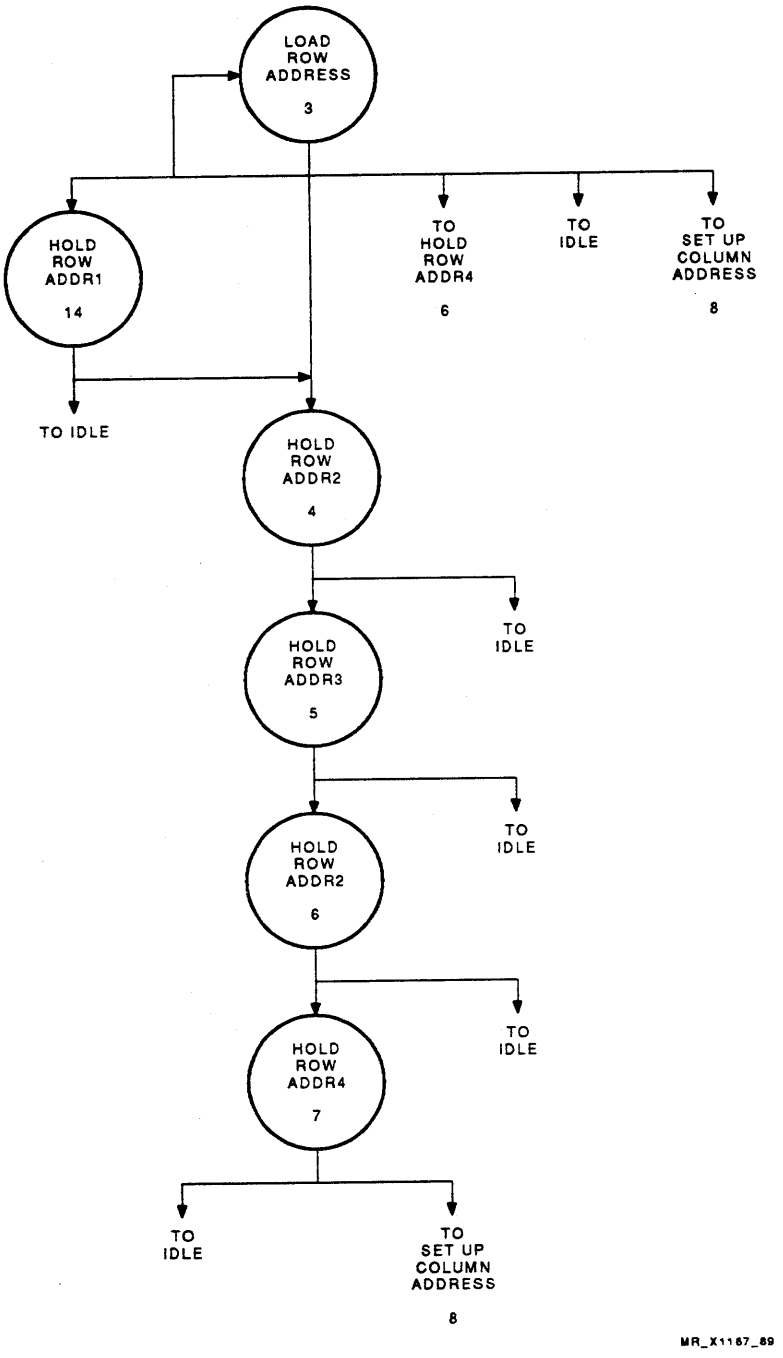


Figure 5–32 Loading the Row Address for the Segment Controller States

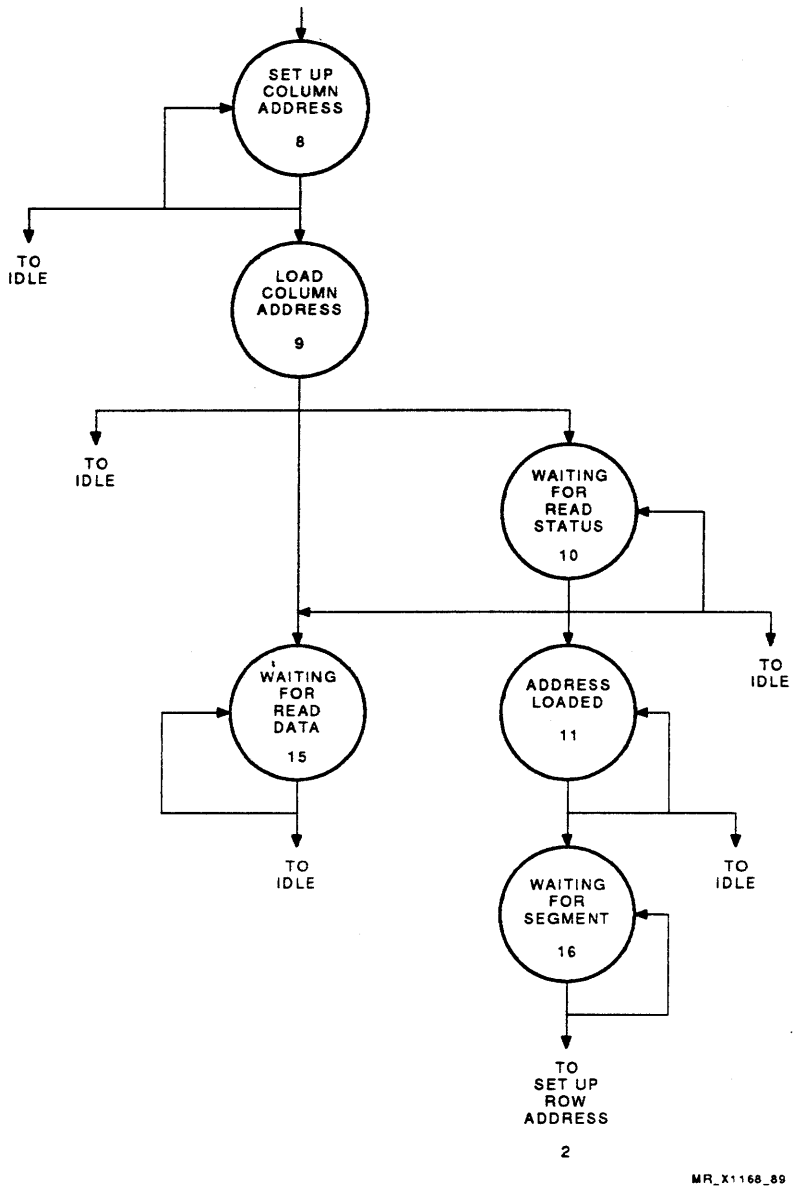


Figure 5–33 Loading the Column Address for the Segment Controller States

5.4.3 Command Latch

The inputs and outputs of the command latch are shown in Figure 5–34. The command latch receives the following internal MMCX inputs:

- Starting address, read select, command latch select, and BIST from the input buffer
- Address select, read command latch, single-step read select, and single-step mode from the command buffer control

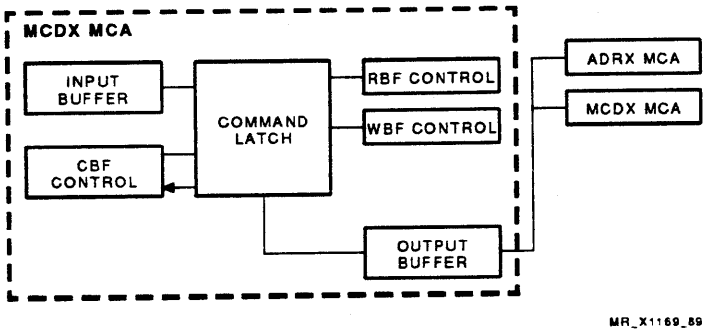


Figure 5–34 Command Latch

The command latch sends the following:

Command to the command buffer control
Read command to the read buffer control
Write command to the write buffer control

Through the output buffer, the command latch sends the following:

Index to the ADRX MCAs
Command and bank address to the MCDX MCA

5.4.4 Read Buffer Control

The inputs and outputs of the read buffer control are shown in Figure 5–35. The read buffer control receives the following internal MMCX inputs:

- Buffer available, send data, read data ready, lower longword read error, high longword read error, single-step request, cycle status, and BIST from the input buffer
- Command buffer status, single-step mode, and read latch OK from the command buffer control
- Command from the command latch
- Read-modify-write status bit from the write buffer control

The read buffer control sends:

- Read latch busy and read buffer single-step read to the command buffer control
- Read-modify-write select and read-modify-write beginning of data bit to the write buffer control

Through the output buffer, the read buffer control sends the following:

- Command, segment, load command, and read OK to the CCU MCU
- Buffer select, merge select, buffer latch, mask select, error log select, read latch hold, beginning of data bit, quadword error address, and ECC enable to the MDPX MCA
- Read latch busy and latch read error to the MCDX MCA
- MAC read select and data output latch enable to MMU

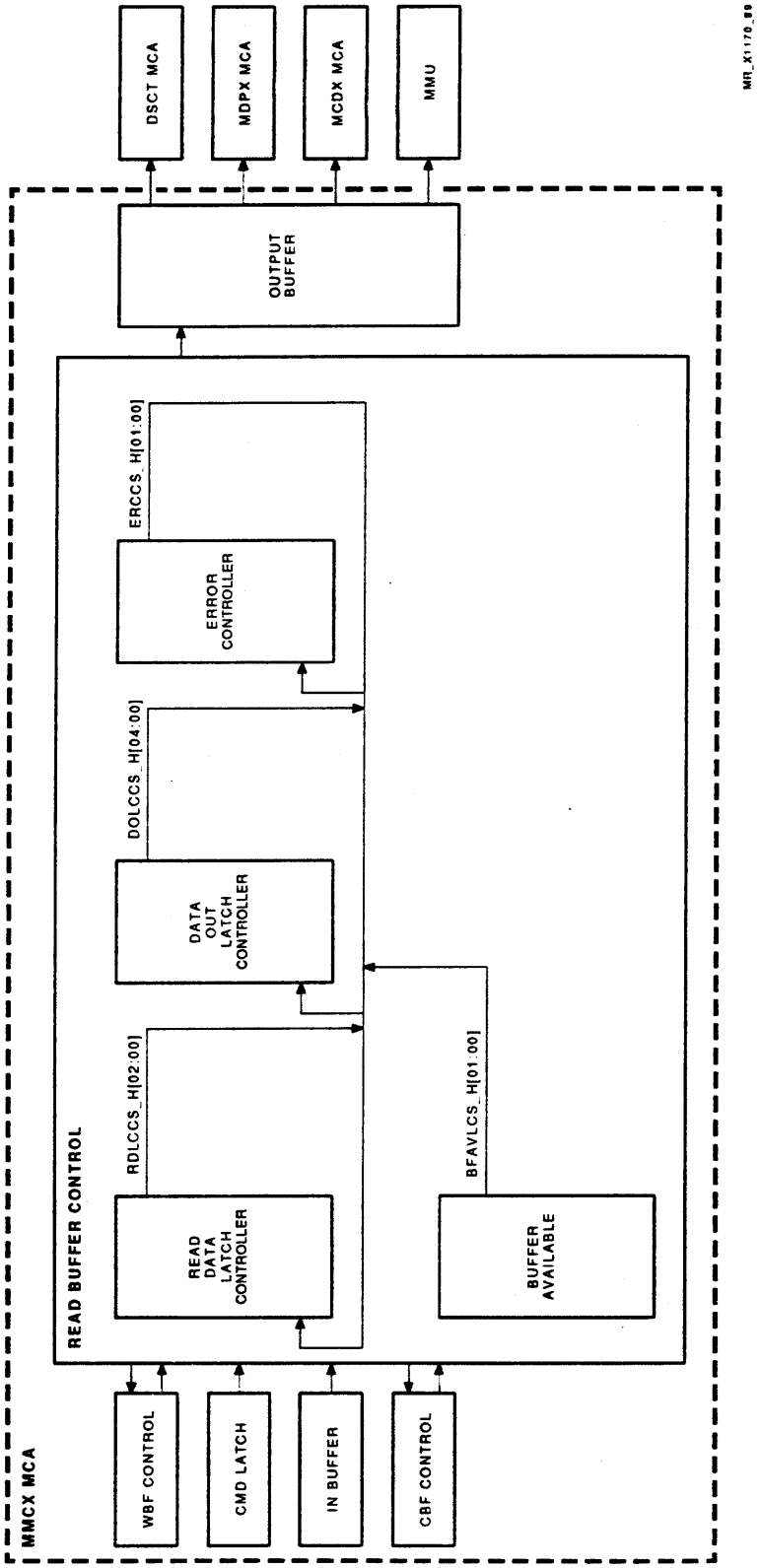


Figure 5–35 Read Buffer Control

5.4.4.1 Read Data Latch Controller

The read data latch controller controls the read data latch (read buffer 1). Figure 5–30 shows the read data latch in the memory module. Figure 5–36 shows the states of the read data latch controller.

For reads, the controller moves from the idle to the command sent state. While in this state, MMCX sends the load command, segment number, and return data (or error) command to the JBox.

For read-modify-write operations, the controller moves from the idle state to the pending read-modify-write state. The data output latch (read buffer 0) loads the read data for the merge operation that takes place in the MDPX MCA. Section 5.9.3 describes a read-modify-write operation.

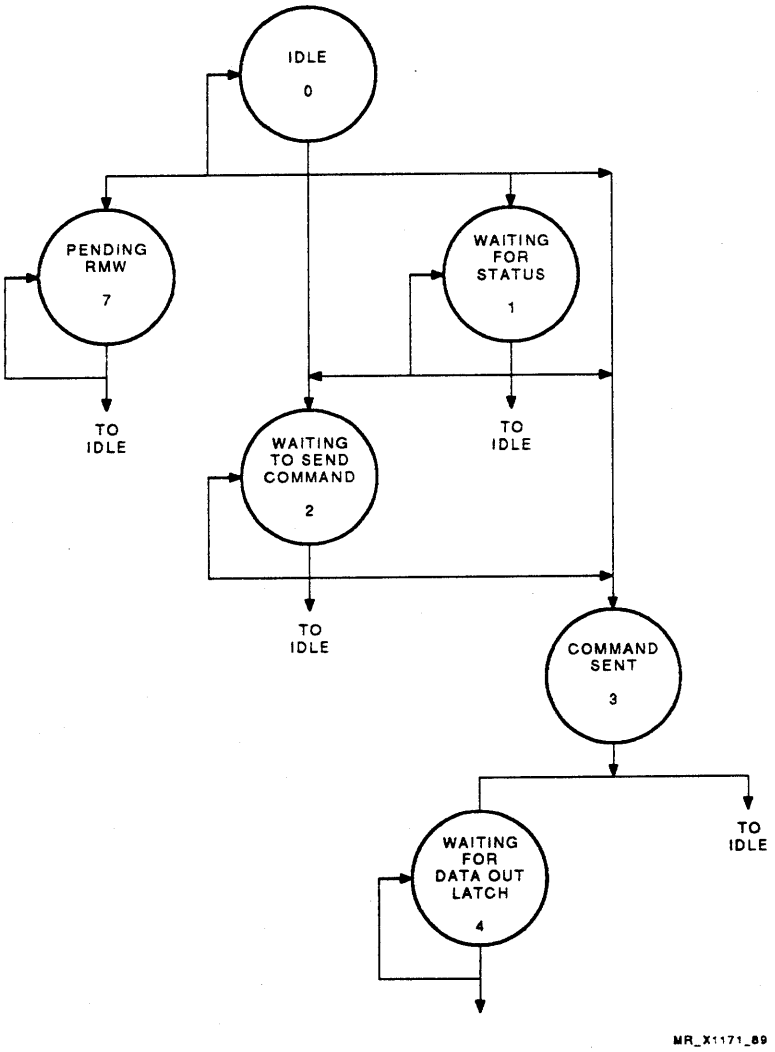


Figure 5–36 Read Buffer Controller States

5.4.4.2 Data Output Latch Controller

The data output latch controller controls the data output latch (read buffer 0) in the memory module. Figure 5–30 shows the data output latch in the memory module. Figure 5–37 shows the states of the data output latch controller. For read operations, the data output latch controller moves from the idle state to the load starting address states. Eight quadwords are read from the DRAMs regardless of the context. The controller returns to idle.

For read-modify-write operations, the data output latch controller moves through the following sequence of states:

1. **Idle state.**
2. **Read-modify-write pending states.**
3. **Load write data states** — Write data is loaded into the data output latch and is sent to MDPX.
4. **Load read data states** — Read data is loaded into the data output latch and is sent to MDPX.
5. **Merge data states** — The read and write data is merged in MDPX.
6. **Idle state.**

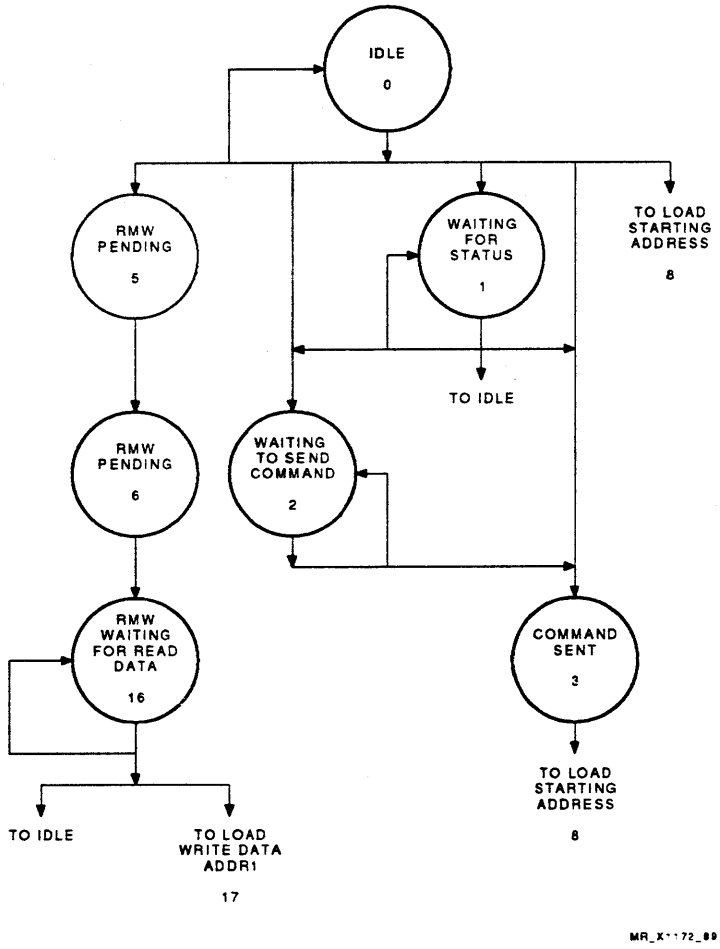


Figure 5–37 (Cont.) Data Output Latch Controller States

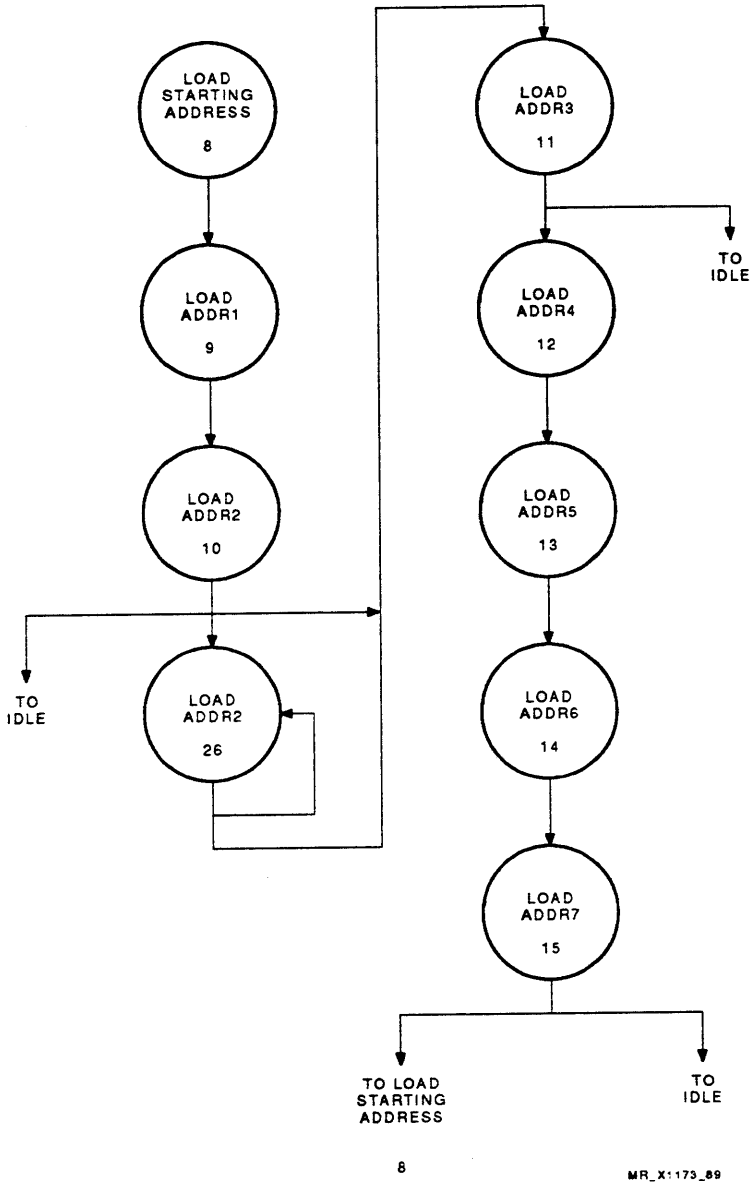


Figure 5–37 (Cont.) Data Output Latch Controller States

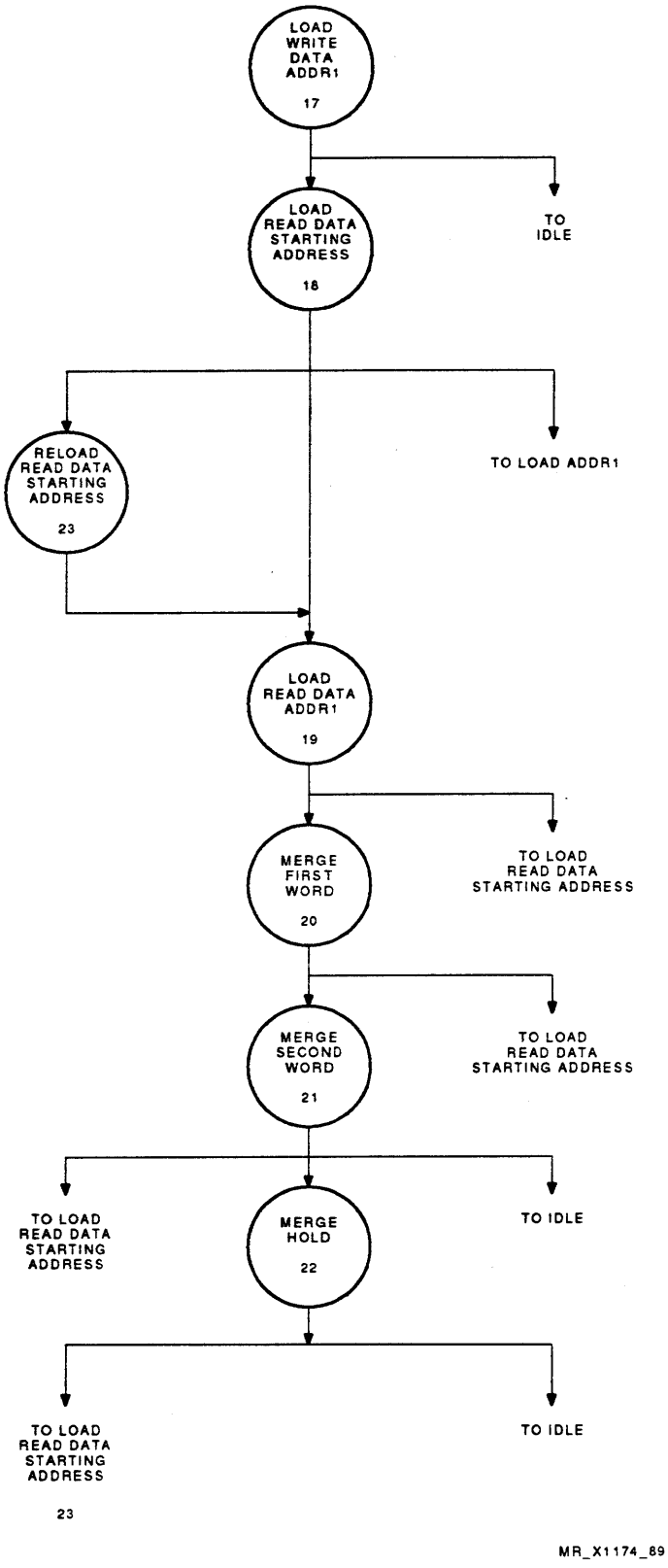


Figure 5–37 Data Output Latch Controller States

5.4.4.3 Error Report Controller

The error report controller controls how the ACU ECC error logic reports an error. The following steps summarize how ACU detects and handles an error:

1. MDPX detects and latches an ECC error as a result of a write data error or a read data error.
2. MMCX sends the error command to the JBox. The JBox sends a send data command to ACU for the error data.
3. The MDPX data error register latches all data errors except for a write parity error. MDPX sends the ECC error data to the JBox. The JBox sends the error data to the ICU transmit buffer and then to SPU.

Figure 5–38 shows the states of the error report controller.

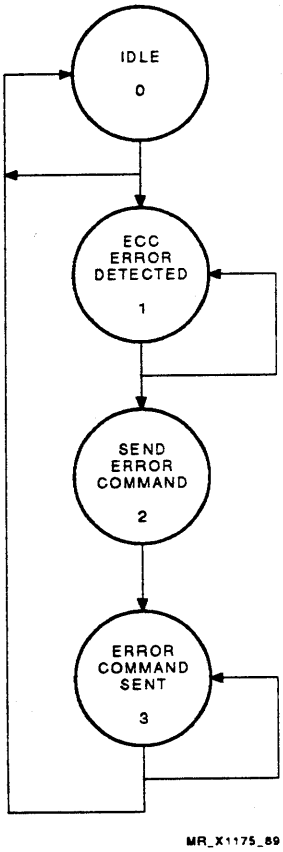


Figure 5–38 Error Report Controller States

5.4.5 Write Buffer Control

The inputs and outputs of the write buffer control are shown in Figure 5–39. The write buffer control receives the following MMCX internal inputs:

- Load command, bank address, beginning of data bit, write done, mask status, and BIST from the input buffer
- Command buffer status from the command buffer control
- Read-modify-write select and read-modify-write beginning of data bit from the read buffer control
- Command from the command latch

The write buffer sends the following:

- Read-modify-write required, data received in the data input latch, and write data ready to the command buffer control
- Read-modify-write required and merge abort to the read buffer control

Through the output buffer, the write buffer sends the following:

- Write OK to the CCU MCU
- CAS mask control, MSKDIRSEG, write buffer strobes, write flip-flop enables, and write select to MMU
- Read-modify-write required, write data ready, low longword mask parity, and high longword mask parity to the MCDX MCA
- Mask latch select and write address parity to the MDPX MCA

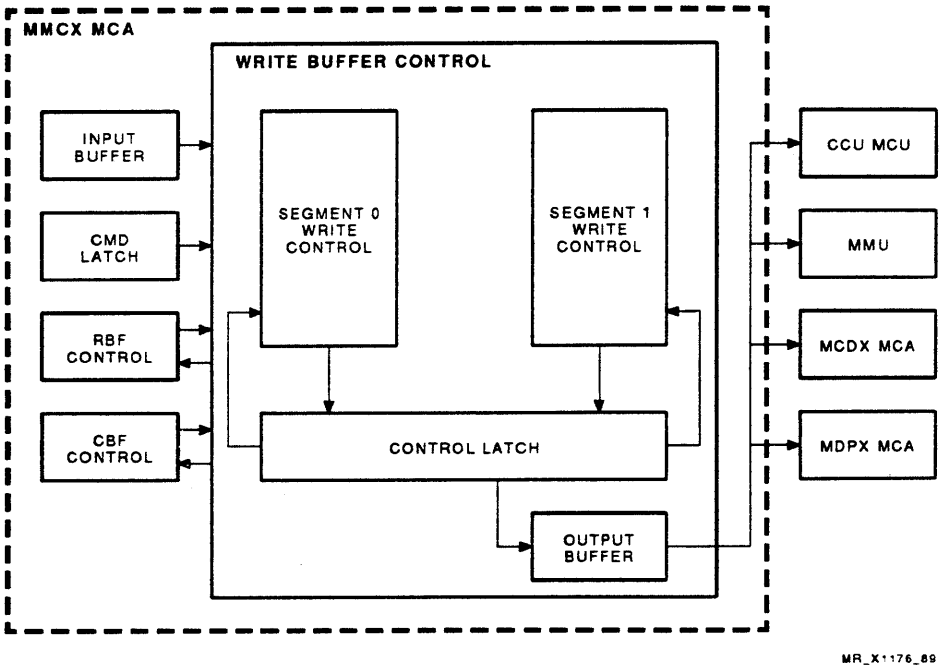


Figure 5–39 Write Buffer Control

5.4.5.1 Segment Data Latch Controller

The segment data latch controller controls the data input latch (write buffer 0) and write buffer 1. Figure 5–30 shows write buffers 0 and 1 in the memory module.

MMCX receives the load command, write command, index field, and length field from the CCU. MMCX looks at the index to determine whether it is a CPU or an I/O device write. If it is a CPU write, the length equals eight quadwords. MMCX monitors the beginning of the data from the data switch to determine when the data transfer starts, counts clock ticks to determine when the data transfer completes, and asserts write data ready.

Figure 5–40 shows the states of the segment data latch controller. For writes, the controller moves through the following sequence of states:

1. **Idle state.**
2. **Write data expected state** — Data switch sends the beginning of the data bit to mark the first valid quadword.
3. **Loading last data quadword state** — The data input latch continues to load quadwords until the last quadword is loaded.
4. **Input data ready state** — MMCX sends write data ready status to MCDX.
5. **Write data ready state** — MMCX sends write select, write strobes, CAS mask control, and write flip-flop enables. The controller remains in this state until MCDX sends write done to MMCX. MMCX sends command buffer available to the JBox.

For read-modify-writes, the controller enters the merge data states.

5.4.5.2 Read-Modify-Write Status Bit

The read-modify-write required status bit remains asserted until MCDX sends write done to the MMCX MCA. Figure 5–41 shows the states of the read-modify-write status bit controller.

5.4.6 Mode Transition Controller

The mode transition controller controls how the SPU switches from step to normal mode during BISTs. Figure 5–42 shows the states of the mode transition controller.

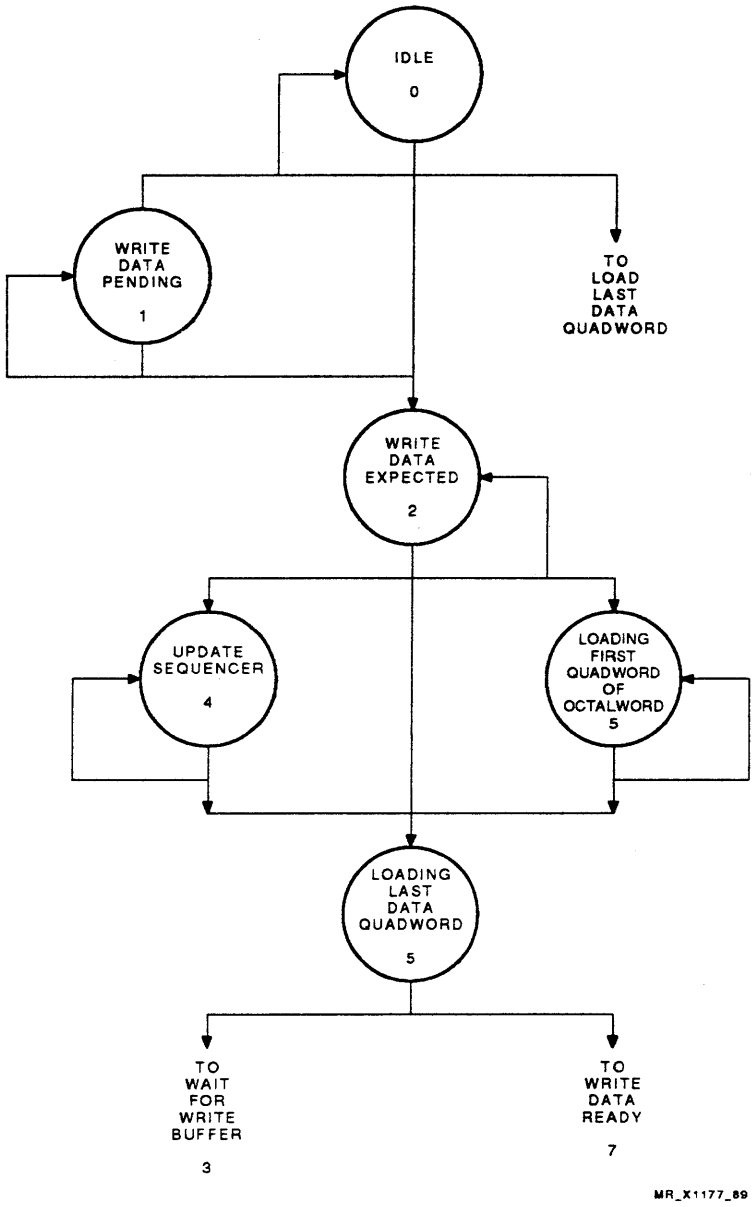


Figure 5–40 Segment Data Latch Controller States

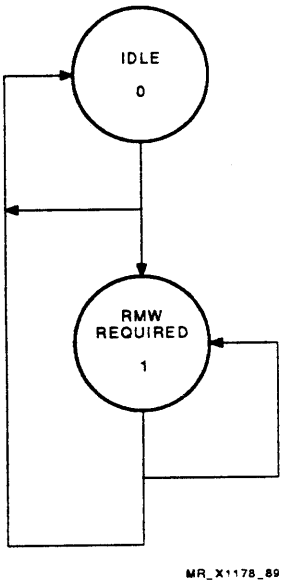


Figure 5–41 Read-Modify-Write Status Bit States

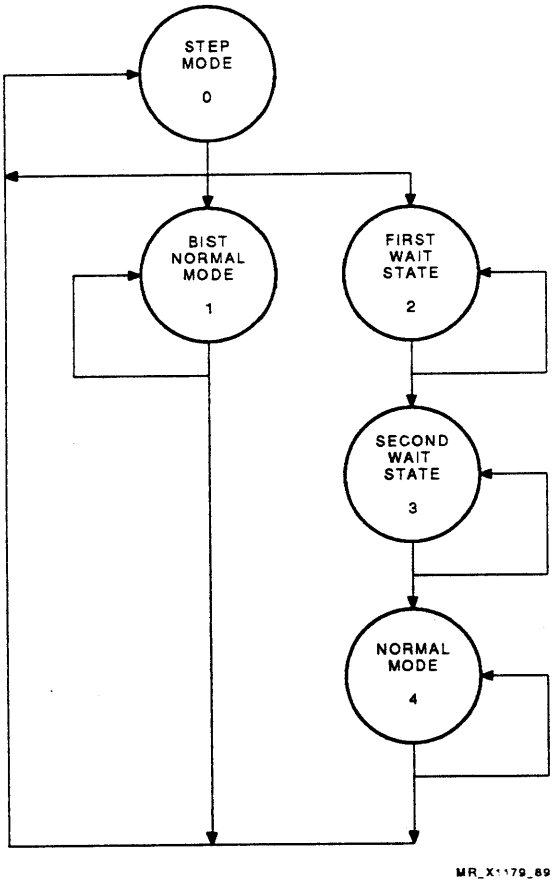


Figure 5–42 Mode Transition Controller States

5.4.7 MMU Interface

This section describes the interface between the MMU and the MMCX MCAs.

5.4.7.1 MMCX-to-MMU Interface

Figure 5–43 shows the MMCX_MMUX_CTL_H[47:00] fields. Table 5–18 lists the control fields and their descriptions.

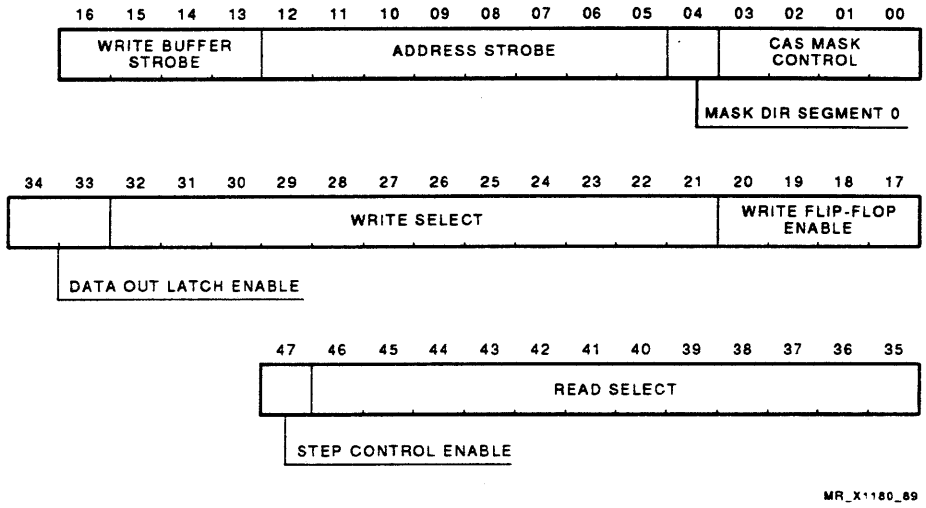


Figure 5–43 MMCX-to-MMU Control Format

Table 5–18 MMCX-to-MMU Control Field Descriptions

Bits	Name	Description
03:00	CAS mask control	This field specifies the chip select mask control signals for the four main array cards. MMCX_MAC_CASMSKCTL_H[03:00] sets the state of the bits in the mask control register. Table 5–19 lists the bits and their descriptions. Table 5–20 lists the the CAS mask control lines for the main array cards, segments, and banks.
04	Mask direct segment	This field specifies the segment to which the current CAS mask bits are directed.
12:05	Address strobe	This field contains the address hold control signals for the four main array cards. Table 5–21 lists the address strobes for the main array cards, segments, and banks.
16:13	Write buffer strobe	This field contains the data latch strobe bits for each of the two data buffers across the four main array cards. The strobe signals strobe the selected data into the buffer. A strobe is sent when write selects are to be loaded or when data and mask bits are to be loaded into the data input or write data latch. Table 5–22 lists the write strobe bits for the main array cards, segments, and banks.

Table 5–18 (Cont.) MMCX-to-MMU Control Field Descriptions

Bits	Name	Description
20:17	Write flip-flop enable	This field contains the write flip-flop enable bits for the write data buffer across four main array cards. These bits enable the strobing of data and mask bits into the write data buffer. Table 5–23 lists the write flip-flop bits for the main array cards, segments, and banks.
32:21	Write select	This field contains the write select bits that select one of the eight input data buffers of write buffer 0. Table 5–24 lists the write select bits for the MACs, segments, and banks.
34:33	Data out latch enable	This field contains the data out latch enable bits that open the data output latches to receive new data. Table 5–25 lists the data output latch enable bits for the MACs, segments, and banks.
46:35	Read select	This field contains the read select bits that select one of eight output data buffers in read buffer 0 for transmission to the data path. Table 5–26 lists the read select bits for the MACs, segments, and banks.
47	Step control enable	This field contains the step control enable bit that switches the four MACs to single-step mode. It must be asserted before executing any single-step or burst-mode operations. While it is asserted, the DRAMs are controlled by the MACs local controller and normal DRAM control lines are interpreted as follows: MMCX_MAC_RAS_L[03:00] — Strobe mode commands into the DCA command buffer; MMCX_MAC_CAS_L[01:00] and MMCX_MAC_WE_L[01:00] — The CAS and write lines are encoded with the step mode command. When the bit is reset, any cycle in progress is completed by the local DRAM controller and control then returns to the normal system timing.

Table 5–19 MMCX_MAC_CASMSKCTL_H[03:00]

Bits	Description
00	Reset all mask latch locations.
01	Set selected mask latch location, reset all other locations.
10	No change to any location.
11	Set selected mask latch location, no change to other locations.

Table 5–20 CAS Mask Control Field Descriptions

CAS Mask Control Lines	MAC	Segment	Bank
01:00	0, 1	0, 1	0, 1
03:02	2, 3	0, 1	0, 1

Table 5–21 Address Strobe Field Descriptions

Address Strobe	MAC	Segment	Bank
00	0	0	0, 1
01	0	1	0, 1
02	1	0	0, 1
03	1	1	0, 1
04	2	0	0, 1
05	2	1	0, 1
06	3	0	0, 1
07	3	1	0, 1

Table 5–22 Write Strobe Field Descriptions

Write Strobe Bit	MAC	Segment	Bank
00	0	0, 1	0, 1
01	1	0, 1	0, 1
02	2	0, 1	0, 1
03	3	0, 1	0, 1

Table 5–23 Write Flip-Flop Enable Field Descriptions

Write Flip-Flop Enable	MAC	Segment	Bank
00	0	0, 1	0, 1
01	1	0, 1	0, 1
02	2	0, 1	0, 1
03	3	0, 1	0, 1

Table 5–24 Write Select Field Descriptions

Write Select	MAC	Segment	Bank
02:00	0	0, 1	0, 1
05:03	1	0, 1	0, 1
08:06	2	0, 1	0, 1
11:09	3	0, 1	0, 1

Table 5–25 Data Output Latch Enable

Data Output Latch Enable	MAC	Segment	Bank
00	0, 1	0, 1	0, 1
01	2, 3	0, 1	0, 1

Table 5–26 Read Select Field Descriptions

Read Select	MAC	Segment	Bank
02:00	0	0, 1	0, 1
05:03	1	0, 1	0, 1
08:06	2	0, 1	0, 1
11:09	3	0, 1	0, 1

5.4.7.2 MMU-to-MMCX Interface

MMUX sends status information to MMCX MCA. Figure 5–44 shows the MMU-to-MMCX status fields. Table 5–27 lists the status bits and their descriptions.

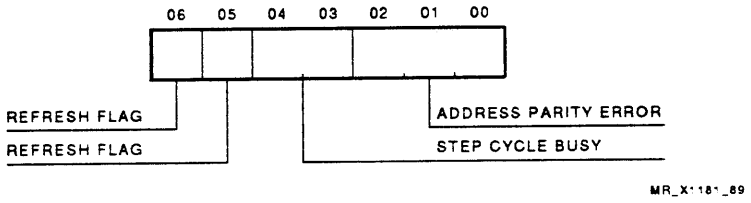


Figure 5–44 MMU-to-MMCX Status Format

Table 5–27 MMU-to-MMCX Status Field Descriptions

Bits	Name	Description
03:00	Address parity error	This field indicates that an address parity error has been detected on MAC. Each of these signals is received from a different MAC.
04	Step cycle busy	This field indicates that a single-step or burst-mode cycle is in progress. No new request can be sent to the MACs while this signal is asserted.
06:05	Refresh flag (differential)	This field indicates that a refresh is due.

5.4.8 MCDX MCA Interface

This section describes the interface between MCDX and the MMCX MCAs.

5.4.8.1 MCDX-to-MMCX Interface

Figure 5–45 shows the control fields. Table 5–28 lists the bits and their descriptions.

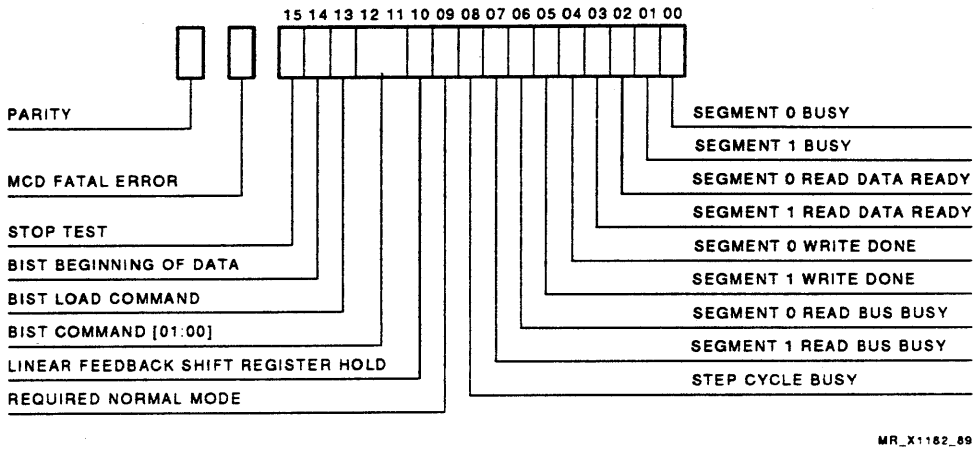


Figure 5–45 MCDX-to-MMCX Control Format

Table 5–28 MCDX-to-MMCX Field Descriptions

Signal	Field	Description
MCDX_MMCX_ STATUS_H[00]	SG0BUSY_H Segment 0 busy	The assertion of this line indicates that segment 0 is busy. No new command is initiated to the segment while this line is asserted.
MCDX_MMCX_ STATUS_H[01]	SG1BUSY_H Segment 1 busy	The assertion of this line indicates that segment 1 is busy. No new command is initiated to the segment while this line is asserted.
MCDX_MMCX_ STATUS_H[02]	SG0RDDRDY_H Segment 0 read data ready	The assertion of this line indicates that segment 0 read data will be latched into the read buffer in two clock periods.
MCDX_MMCX_ STATUS_H[03]	SG1RDDRDY_H Segment 1 read data ready	The assertion of this line indicates that segment 1 read data is latched into the read buffer in two clock periods.
MCDX_MMCX_ STATUS_H[04]	SG0WTDONE_H Segment 0 write done	The assertion of this line indicates that the segment 0 data in the write buffer may be overwritten.
MCDX_MMCX_ STATUS_H[05]	SG1WTDONE_H Segment 1 write done	The assertion of this line indicates that the segment 1 data in the write buffer may be overwritten.
MCDX_MMCX_ STATUS_H[06]	SG0RBSBSY_H Segment 0 read bus busy	The assertion of this line indicates that segment 0 controller has asserted CAS and is using the output bus to load read data into the read data latch.

Table 5–28 (Cont.) MCDX-to-MMCX Field Descriptions

Signal	Field	Description
MCDX_MMCX_ STATUS_H[07]	SG1RBSBSY_H Segment 1 read bus busy	The assertion of this line indicates that segment 1 controller has asserted CAS and is using the output bus to load read data into the read data latch.
MCDX_MMCX_ STATUS_H[08]	STEPCYCBSY_H Step cycle busy	The assertion of this line indicates that the single-step controller in DCA is busy. No new request can be sent to the MACs while this signal is asserted. This signal is not valid and must be ignored in normal mode.
MCDX_MMCX_ STATUS_H[09]	REQNRMODE_H Required normal mode	The assertion of this line indicates to MMCX that the subsequent BIST commands should be executed in normal mode. When it is not asserted, BIST commands are executed in step mode. It is enabled only during BIST.
MCDX_MMCX_ STATUS_H[10]	LFSRHLD_H Linear feedback shift register hold	The negation of this line in BIST enables the MAC's address B strobe 0. In BIST, this causes DCA to select the next test address. It is ignored except in BIST normal mode.
MCDX_MMCX_ STATUS_H[12:11]	BISTCMD_H[01:00] BIST command [01:00]	The lines are encoded with the commands to be executed during self-test. They are valid only during self-test. They are encoded as follows:

1	0	Function
0	0	Read
0	1	Write/read
1	0	Write
1	1	Write pass

MCDX_MMCX_ STATUS_H[13]	BISTLDCMD_H BIST load command	When this line is asserted during self-test, the command encoded on the STMCMD lines is executed. It is ignored except during self-test.
MCDX_MMCX_ STATUS_H[14]	BISTBOD_H BIST beginning of data	The assertion of this line simulates the beginning of data during some BIST tests.
MCDX_MMCX_ STATUS_H[15]	STOPTEST_H Stop test	The assertion of this line indicates completion of a BIST test. It may be due to an error or to the normal completion of the test.
MCDX_MMCX_ FATALERR_L	MCDFTLERR_H MCD fatal error	The assertion of this line indicates that a fatal error has been detected in MCD.
MCDX_MMCX_ STATUS_PAR_H	Parity	The assertion of this line indicates parity across the signals received from MCD. This parity bit includes the fatal error line.

5.4.8.2 MMCX-to-MCDX Interface

Figure 5–46 shows the control fields. Table 5–29 lists the bits and their descriptions.

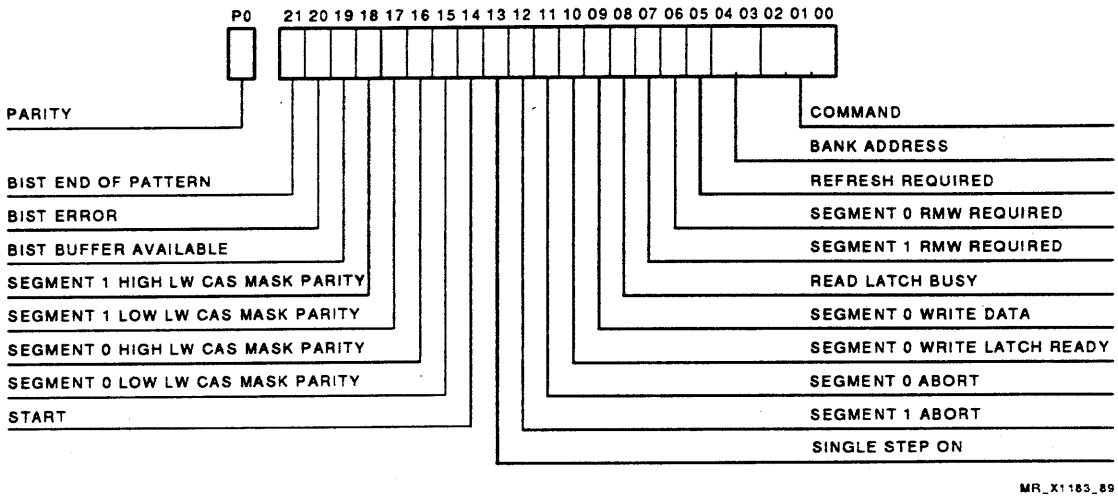


Figure 5–46 MMCX-to-MCDX Control Format

Table 5–29 MMCX-to-MCDX Field Descriptions

Signal	Field	Description
MMCX_MCDX_ CTL_H[02:00]	CMD_H[02:00] Command	These bits are encoded with the type of cycle to be executed. The encoding in step mode is different from that in normal mode. The two types of encodings are as follows:

Normal Mode

2	1	0	Function
0	0	0	Read
0	0	1	Write read
0	1	0	Masked write
0	1	1	Write pass

Step Mode

2	1	0	Function
0	0	0	Memory read
0	0	1	Memory write read
0	1	0	Memory masked write
0	1	1	Write pass
1	0	0	EEPROM read
1	1	0	EEPROM write

Table 5–29 (Cont.) MMCX-to-MCDX Field Descriptions

Signal	Field	Description
MMCX_MCDX_ CTL_H[04:03]	BANKADDR_ H[01:00] Bank address	These bits contain the encoded bank address of the command to be executed. Bit 1 indicates one of two segments and bit 0 indicates one of two banks within that segment.
MMCX_MCDX_ CTL_H[05]	RFSHRQRD_H Refresh required	This line is asserted when a refresh is due. No new memory command is sent to a segment after the assertion of this signal until a segment busy cycle (assertion and negation) has been detected for that segment. It is negated when both segments have completed their refresh.
MMCX_MCDX_ CTL_H[06]	SG0RMWRQRD_H Segment 0 RMW required	The assertion of this line indicates that the current segment 0 write command must be converted to a read-modify-write. It is asserted simultaneously with or before SG0WTDRDY. It is negated after SG0WTDONE is received.
MMCX_MCDX_ CTL_H[07]	SG1RMWRQRD_H Segment 1 RMW required	The assertion of this line indicates that the current segment 1 write command must be converted to a read-modify-write. It is asserted simultaneously with or before SG1WTDRDY. It is negated after SG1WTDONE is received.
MMCX_MCDX_ CTL_H[08]	RDLATBSY_H Read latch busy	The assertion of this line indicates that the read latch is busy.
MMCX_MCDX_ CTL_H[09]	SG0WTDRDY_H Segment 0 write data	The assertion of this line indicates that the data for the current segment 0 write is in the write data latch.
MMCX_MCDX_ CTL_H[10]	SG1WTDRDY_H Segment 0 write latch ready	The assertion of this line indicates that the data for the current segment 1 write is in the write data latch.
MMCX_MCDX_ CTL_H[11]	SG0ABRT_H Segment 0 abort	The assertion of this line indicates that the received status for the current segment 0 command is an abort. When abort is received, the cycle should be terminated as quickly as possible.
MMCX_MCDX_ CTL_H[12]	SG1ABRT_H Segment 1 abort	The assertion of this line indicates that the received status for the current segment 1 command is an abort. When abort is received, the cycle should be terminated as quickly as possible.
MMCX_MCDX_ CTL_H[13]	SNGLSTPON_H Single stop on	The assertion of this line indicates that the DRAM controller should switch to step mode as soon as any cycles in progress have been completed. No new cycles are initiated in normal mode. When this line is negated, the DRAM controller should switch immediately to normal mode.
MMCX_MCDX_ CTL_H[14]	START_H Start	The assertion of this line indicates that the command encoded on the command lines should be executed at the address on the address lines. The selected segment should be determined from the MSB of the bank address.

Table 5–29 (Cont.) MMCX-to-MCDX Field Descriptions

Signal	Field	Description
MMCX_MCDX_ CTL_H[15]	SG0LLWCASMSKPAR_ H Segment 0 low LW CAS mask parity	This line reflects the parity of the CAS mask bits contained in the last block of data written to the segment 0 lower longword.
MMCX_MCDX_ CTL_H[16]	SG0HLWCASMSKPAR_ H Segment 0 high LW CAS mask parity	This line reflects the parity of the CAS mask bits contained in the last block of data written to the segment 0 higher longword.
MMCX_MCDX_ CTL_H[17]	SG1LLWCASMSKPAR_ H Segment 1 low LW CAS mask parity	This line reflects the parity of the CAS mask bits contained in the last block of data written to the segment 1 lower longword.
MMCX_MCDX_ CTL_H[18]	SG1HLWCASMSKPAR_ H Segment 1 high LW CAS mask parity	This line reflects the parity of the CAS mask bits contained in the last block of data written to the segment 1 higher longword.
MMCX_MCDX_ CTL_H[19]	BISTBUFAVL_H BIST buffer available	The assertion of this line indicates that the segment 0 command has finished and that its command latch is now available to receive a new command.
MMCX_MCDX_ CTL_H[20]	BISTERR_H BIST error	The assertion of this line indicates that a data error was detected in the read data that has just passed through MDP.
MMCX_MCDX_ CTL_H[21]	BISTEOP_H BIST end of pattern	The assertion of this line indicates that the pseudo-random data pattern generator in MDPX has reached its last pattern. It is used only in BIST.
MMCX_MCDX_ CTL_PAR_H	PARITY_H Parity	This line is the parity across all the lines sent from MMCX to MCD.

5.4.9 MDPX MCA Interface

This section describes the interface between MDPX and the MMCX MCAs.

5.4.9.1 MMCX-to-MDPX Interface

Figure 5–47 shows the control fields. Table 5–30 lists the bits and their descriptions.

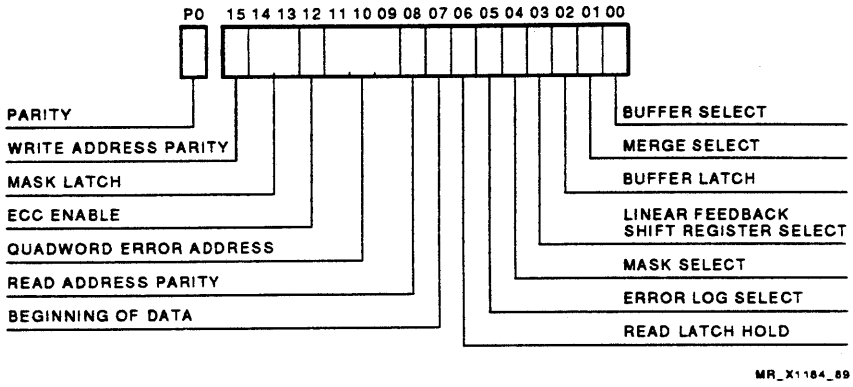


Figure 5–47 MMCX-to-MDPX Control Format

Table 5–30 MMCX-to-MDPX Control Field Descriptions

Signal	Field	Description
MMCX_MDPA_CTL_H[00]	BUFSEL_H[00] Buffer select	This is the multiplexer select signal that controls whether mask data is sourced from the first or second QW. Assertion of the signal selects the second QW.
MMCX_MDPA_CTL_H[01]	MRGSEL_H[00] Merge select	This signal enables the latched byte mask to control whether bytes of data are sourced from I/O data buffers or from data read that is from memory during read-modify-write operations.
MMCX_MDPA_CTL_H[02]	BUFLAT_H[00] Buffer latch	This signal is used during merge operations. It latches the write data to be merged into the I/O data buffer. As the read data passes through MDP, the bytes to be written are selected according to the mask that has previously been latched and merged with nonselected bytes of read data. As soon as the merged data exits from MDPX, this signal may be negated.
MMCX_MDPA_CTL_H[03]	LFSRSEL_H[00] Linear feedback shift register select	This signal switches the source of write data from ECC data latches to pseudo-random pattern generator data latches for test purposes (B latch source).
MMCX_MDPA_CTL_H[04]	MSKSEL_H[00] Mask select	This is the multiplexer select signal that controls whether the data to be latched into the data buffer latch is to be sourced from the first or second LW of input data. Bits [01:00] control the upper and lower longwords of the first I/O QW buffer, respectively. Bits [03:02] control the upper and lower longwords of the second I/O QW buffer, respectively.
MMCX_MDPA_CTL_H[05]	ERRLOGSEL_H[00] Error log select	This signal switches the source of read data to the error log (B latch source).
MMCX_MDPA_CTL_H[06]	RDDATHLD_H[00] Read latch hold	The data output latch control signal temporarily holds read data being output to the crossbar in data output latches (B latch source).

Table 5–30 (Cont.) MMCX-to-MDPX Control Field Descriptions

Signal	Field	Description
MMCX_MDPA_CTL_H[07]	BOD_H[00] Beginning of data	This signal is asserted when the first data word of returning read data is transferred to the memory data path.
MMCX_MDPA_CTL_H[08]	RDADRPAR_H[00] Read address parity	This is the parity bit generated on address bits [05:03]. It is supplied to each MDPX as each read data longword is received from the MACs. These three bits determine the relative position of the longword in the read data buffer.
MMCX_MDPA_CTL_H[11:09]	QWERRADR_H[02:00] Quadword error address	These lines carry the address of the quadword that is currently being checked for correctness by MDP. It is logged by MDPX if an ECC error is detected. It can then be read by accessing the ECC error log.
MMCX_MDPA_CTL_H[12]	ECCENB_H[00] ECC enable	This is the static ECC enable signal (B latch source).
MMCX_MDPA_CTL_H[14:13]	MSKLAT_H[01:00] Mask latch	The data output latch control signals temporarily hold I/O data from crossbar in data buffer latches in MDPX (B latch source). See the following table for usage:
MMCX_MDPA_CTL_H[15]	WRTADRPAR_H[00] Write address parity	The parity bit generated on address bits [05:03] is supplied to each MDPX along with the write data longword. These three bits determine the relative position of the longword in the write data buffer.
MMCX_MDPA_CTL_PAR_H	PARITY_H[00] Parity	This parity bit is generated across the control signals from MMCX to MDP.
MMCX_MDPB_CTL_H[00]	BUFSEL_H[00] Buffer select	This is the multiplexer select signal that controls whether mask data is sourced from the first or second QW. Assertion of the signal selects the second QW.
MMCX_MDPB_CTL_H[01]	MRGSEL_H[00] Merge select	This signal enables the latched byte mask to control whether bytes of data are sourced from I/O data buffers or from data that is read from memory during read-modify-write operations.

Signal	Bytes	LW	Segment Number
MSKLAT[00]	0–3	LLW	0
MSKLAT[01]	4–7	HLW	0
MSKLAT[02]	0–3	LLW	1
MSKLAT[03]	4–7	HLW	1

Table 5–30 (Cont.) MMCX-to-MDPX Control Field Descriptions

Signal	Field	Description
MMCX_MDPB_CTL_ H[02]	BUFLAT_H[00] Buffer latch	This signal is used during merge operations. It latches the write data to be merged into the I/O data buffer. As the read data passes through MDP, the bytes to be written are selected according to the mask that has previously been latched and merged with nonselected bytes of read data. As soon as the merged data exits from MDPX, this signal may be negated.
MMCX_MDPB_CTL_ H[03]	LFSRSEL_H[00] Linear feedback shift register select	This signal switches the source of write data from ECC data latches to the pseudo-random pattern generator data latches for test purposes (B latch source).
MMCX_MDPB_CTL_ H[04]	MSKSEL_H[00] Mask select	This is the multiplexer select signal that controls whether the data to be latched into the data buffer latch is to be sourced from the first or second LW of input data. Bits [01:00] control the upper and lower longwords of the first I/O QW buffer, respectively. Bits [03:02] control the upper and lower longwords of the second I/O QW buffer, respectively.
MMCX_MDPB_CTL_ H[05]	ERRLOGSEL_H[00] Error log select	This signal switches source of read data to the error log (B latch source).
MMCX_MDPB_CTL_ H[06]	RDDATHLD_H[00] Read latch hold	The data output latch control signal temporarily holds read data being output to the crossbar in the data output latches (B latch source).
MMCX_MDPB_CTL_ H[07]	BOD_H[00] Beginning of data	This signal is asserted when the first data word of returning read data is transferred to the memory data path.
MMCX_MDPB_CTL_ H[08]	RDADRPAR_H[00] Read address parity	The parity bit generated on address bits [05:03] is supplied to each MDPX as each read data longword is received from the MACs. The three bits determine the relative position of the longword in the read data buffer.
MMCX_MDPB_CTL_ H[11:09]	QWERRADR_ H[02:00] Quadword error address	These lines carry the address of the quadword that is currently being checked for correctness by the MDP. It is logged by MDPX if an ECC error is detected. It can then be read by accessing the ECC error log.
MMCX_MDPB_CTL_ H[12]	ECCENB_H[00] ECC enable	This is the static ECC enable signal (B latch source).

Table 5–30 (Cont.) MMCX-to-MDPX Control Field Descriptions

Signal	Field	Description
MMCX_MDPB_CTL_ H[14:13]	MSKLAT_H[01:00] Mask latch	The data output latch control signals temporarily hold I/O data from crossbar in data buffer latches in MDPX (B latch source). See the following table for usage:

Signal	Bytes	LW	Segment Number
MSKLAT[00]	0–3	LLW	0
MSKLAT[01]	4–7	HLW	0
MSKLAT[02]	0–3	LLW	1
MSKLAT[03]	4–7	HLW	1

MMCX_MDPB_CTL_ H[15]	WRTADRPAR_H[00] Write address parity	The parity bit generated on address bits [05:03] is supplied to each MDPX along with the write data longword. These three bits determine the relative position of the longword in the write data buffer.
MMCX_MDPB_CTL_ PAR_H	PARITY_H[01] Parity	This parity bit is generated across the control signals from MMCX to MDP.

5.4.9.2 MDPX-to-MMCX Interface

Figure 5–48 shows the control fields. Table 5–31 lists the bits and their descriptions.

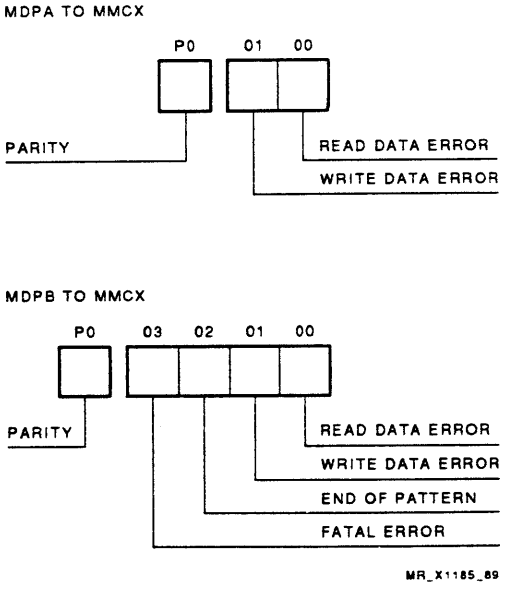


Figure 5–48 MDPX-to-MMCX Control Format

Table 5–31 MDPX-to-MMCX Control Field Descriptions

Signal	Field	Description
MDPA_MMCX_ STATUS_H[00]	RDATERR_H Read data error	This signal indicates that a data (nonfatal) error has been detected during the transfer of read data through MDPX on DAX.
MDPA_MMCX_ STATUS_H][01]	WDATERR_H Write data error	This signal indicates that a data (nonfatal) error has been detected during the transfer of write data through MDPX on DBX.
MDPA_MMCX_ STATUS_PAR_H	Parity	This parity bit is generated across the control signals from MDPX to MMC and includes the fatal error line.
MDPB_MMCX_ STATUS_H[00]	RDATERR_H Read data error	This signal indicates that a data (nonfatal) error has been detected during the transfer of read data through either of the MDPX MCAs.
MDPB_MMCX_ STATUS_H][01]	WDATERR_H Write data error	This signal indicates that a data (nonfatal) error has been detected during the transfer of write data through either of the MDPX MCAs.
MDPB_MMCX_ STATUS_H[02]	EOP_H End of pattern	The assertion of this line indicates that the pseudo-random data pattern generator in the MDPX has reached its last pattern. It is used only in BIST. This signal is received only from MDPX on the same MCU as MMCX.
MDPB_MMCX_ FATALERR_L	FTLERR_L Fatal error	This signal indicates that a control or other type of fatal error has been detected in one of the MDPX MCAs. The signal is received only from MDPX on the same MCU as MMCX.
MDPB_MMCX_ STATUS_PAR_H	Parity	This parity bit is generated across the control signals from MDPX to MMC and includes the fatal error line.

5.4.10 CCU MCU Interface

This section describes the interface between CCU and the MMCX MCAs.

5.4.10.1 CCU-to-MMCX Interface

Figure 5–49 shows the control fields. Table 5–32 lists the bits and their descriptions.

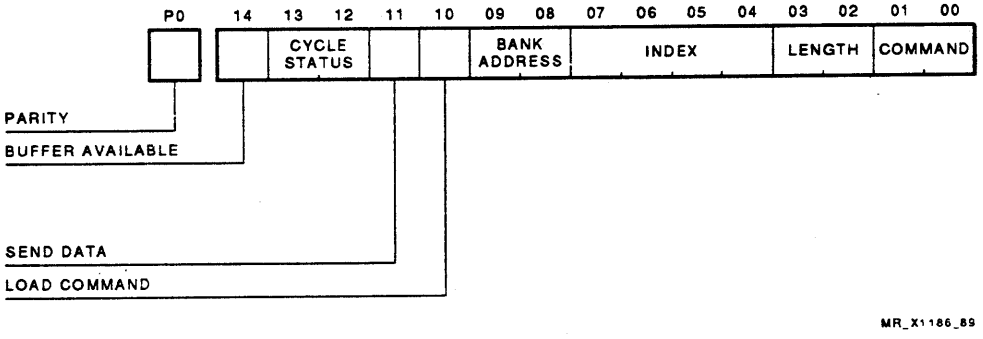


Figure 5–49 CCU-to-MMCX Control Format

Table 5–32 CCU-to-MMCX Field Descriptions

Signal	Field	Description
CCU_MMCX_CMD_ H[01:00]	CMD_H[01:00] Command	These bits contain the encoded operation. The following table defines the supported types of operations:

1	0	Operation
0	0	Read
0	1	Write/read
1	0	Masked write
1	1	Write pass

The memory system supports the following types of cycles. Decoding the CMD lines uniquely determines which of these operations is to be performed. The supported operations and their sizes are as follows:

Type	Size CPU	I/O
Read	8 QW	–
Read	–	1, 2, 4 QW
Masked write	8 QW	–
Masked write	–	1, 2 QW
Write/read	8 QW of write data received from a CPU and 8 QW of read data returned to a different CPU or 1, 2, or 4 QW of read data returned to an I/O device.	
Write/pass	All 8 QW of write data received from a CPU and returned to a different CPU or 1, 2, or 4 QW of the received 8 QW returned to an I/O device.	

Table 5–32 (Cont.) CCU-to-MMCX Field Descriptions

Signal	Field	Description
CCU_MMCX_CMD_ H[03:02]	Length	These bits determine the size of the current I/O operation in quadwords and are valid only during I/O transactions. The following table defines the coding of these bits:

Reads 1	0	Size	Writes 1	0	Size
0	0	4 QW	0	0	4 QW
0	1	8 QW	0	1	8 QW
1	0	1 QW	1	0	1 QW
1	1	2 QW	1	1	2 QW

CCU_MMCX_CMD_ H[07:04]	Index	These lines identify the command buffer and corresponding CPU or I/O port. They are encoded as follows:

3 2 1 0	Source (Port)	Command Buffer
0 0 0 0	CPU0	A
0 0 0 1	CPU0	B
0 0 1 0	CPU0	C
0 0 1 1	CPU1	A
0 1 0 0	CPU1	B
0 1 0 1	CPU1	C
0 1 1 0	I/O0	A
0 1 1 1	I/O0	B
1 0 0 0	CPU2	A
1 0 0 1	CPU2	B
1 0 1 0	CPU2	C
1 0 1 1	CPU3	A
1 1 0 0	CPU3	B
1 1 0 1	CPU3	C
1 1 1 0	I/O1	A
1 1 1 1	I/O1	B

SCU uses the index bits to distinguish between CPU and I/O commands.

CCU_MMCX_CMD_ H[09:08]	Bank address	These lines contain the address bits that are decoded to determine the memory bank selected.

Table 5–32 (Cont.) CCU-to-MMCX Field Descriptions

Signal	Field	Description
CCU_MMCX_CMD_H[10]	LDCMD_H Load command	This signal indicates that a new command and address are ready to be loaded into one of the two memory command buffers.
CCU_MMCX_CMD_H[11]	Send data	This signal indicates the crossbar is ready to receive data from the MDPs.
CCU_MMCX_CMD_H[13:12]	CYCLESTAT_H[01:00] Cycle status	These bits contain information on how the read cycle should be completed based on the results of the consistency check. The following table defines the results for read cycles:

Read Cycles

1	0	Action
0	0	NOP
0	1	Read OK
1	0	Read cancel
1	1	Reserved

CCU_MMCX_CMD_H[14]	BUFAVAIL_H Buffer available	This line is asserted for one clock cycle whenever a command is retired. If two MMCX_CTLB_CMD_Hs are sent before a CTLB_MMCX_BUFSTAT_H is received, the CCU buffer is considered full and no more CMDs are sent until a BUFAVAIL is received.
CCU_MMCX_CMD_PAR_H	Parity	This line is a parity bit generated on the received command, length, index, bank address, LDCMD, send data, status, and BUFAVAIL fields received from CCU.

A brief explanation of the memory operations follows:

- **Read** — A read cycle returns the requested number of quadwords with an index that identifies the requester. A read may be halted by sending read cancel on the cycle status lines. Read data is not returned to the JBox until the JBox sends read OK on the cycle status lines.
- **Masked write** — If the write is eight quadwords in length, indicating that it has been initiated by a CPU, each longword that has its mask bit asserted is written. If the write is one or two quadwords in length, indicating that it is initiated by an I/O device, each byte that has its mask bit asserted is written. The contents of the unwritten bytes must be read from memory and merged with the write data. New ECC check bits are generated for the merged longwords, and then the merged longwords and their associated ECC check bits, are written to memory.
- **Write read** — A read from either a CPU or an I/O device may be determined to have write partial status in a cache by the consistency check. In this case, the initial read is canceled, and a write read command with the block of cache data is issued to memory. The data is written to memory, and the merged data is returned to the requesting device.

- **Write pass** — A read from either a CPU or an I/O device may be determined to have write full status in a cache by the consistency check. In this case, the initial read is canceled, and a write read command with the block of cache data is issued to memory. The full block of data is written to memory and returned to the requesting device.

5.4.10.2 MMCX-to-CCU Interface

Figure 5–50 shows the control fields. Table 5–33 lists the bits and their descriptions.

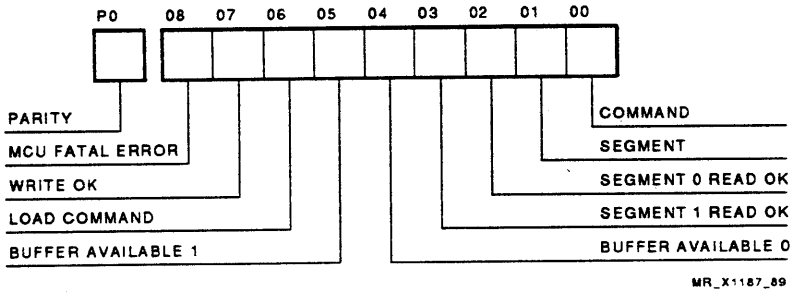


Figure 5–50 MMCX-to-CCU Control Format

Table 5–33 MMCX-to-CCU Field Descriptions

Signal	Field	Description
MMCX_CCU_CMD_ H[00]	CMD_H Command	This bit is encoded as follows: **0 Function** 0 Return ECC error data 1 Return read data
MMCX_CCU_CMD_ H[01]	SEG_H Segment	Assertion of this signal indicates that the LDCMD currently being transmitted to CCU is for a segment 1 command.
MMCX_CCU_CMD_ H[02]	SG0RDOK_H Segment 0 read OK	This signal indicates that the read data from the latest segment 0 read command has passed through MDPX and that no error has been detected. It indicates to CCU that the address of the just-completed read command is not required for error reporting. It is asserted for one clock period.
MMCX_CCU_CMD_ H[03]	SG1RDOK_H Segment 1 read OK	This signal indicates that the read data from the latest segment 1 read command has passed through MDPX and that no error has been detected. It indicates to CCU that the address of the just-completed read command is not required for error reporting. It is asserted for one clock period.
MMCX_CCU_CMD_ H[04]	BUFAVAIL0_H Buffer available 0	This line indicates the availability of the CCU memory command segment 0 buffer register. This signal is asserted for one clock period when the segment 0 buffer is available to receive a new command.

Table 5–33 (Cont.) MMCX-to-CCU Field Descriptions

Signal	Field	Description
MMCX_CCU_CMD_ H[05]	BUFAVAIL1_H Buffer available 1	This line indicates the availability of the CCU memory command segment 1 buffer register. This signal is asserted for one clock period when the segment 1 buffer is available to receive a new command.
MMCX_CCU_CMD_ H[06]	LDCMD_H Load command	This signal is asserted for one clock period when memory has read data ready to be returned.
MMCX_CCU_CMD_ H[07]	WRTOK_H Write OK	This signal indicates that the write data has been received without error. It is asserted for one clock period.
MMCX_CCU_CMD_ H[08]	MCUFTLERR_H MCU fatal error	This signal is asserted when a fatal error is detected on the DBX MCU. It is just the OR of the fatal error signal received from each MCA on the MCU.
MMCX_CCU_CMD_ PAR_H	Parity	This line is a parity bit on the full set of signals transmitted from MMCX to CCU. It is valid every clock cycle.

5.4.11 Tag MCU Interface

This section describes the interface between the tag MCU and the MMCX MCAs.

5.4.11.1 ADRX-to-MMCX Interface

Memory address decoding is taken care of by the JBox. For every command sent to MMCX, the JBox sends an index. If the requested memory segment is ready, MMCX transmits the index plus a row and column select bit to the ADRX chips in the JBox. The ADRX chips use the index to select the row and column address. ADRX sends the address to MMU through the address lines.

The ADRX row and column multiplexer select logic, which is controlled by the MMCX MCA, determines whether the lines have row or column addresses on them. The ADRX MCAs have two independent multiplexers, one for MMU0 and one for MMU1.

The ADRX chips source 12 row and column addresses, which are demultiplexed into 12 row plus 12 column addresses on the MACs. This supports the 16-Mbit chips (2**24 = 16M).

Figure 5–51 shows the control fields. Table 5–34 lists the bits and their descriptions.

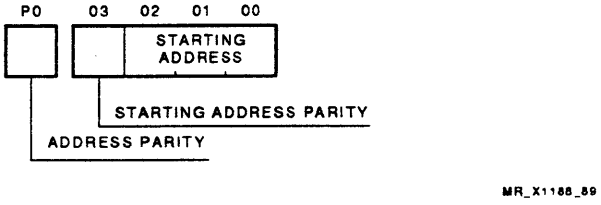


Figure 5–51 ADRX-to-MMCX Control Format

Table 5–34 ADRX-to-MMCX Field Descriptions

Signal	Field	Description
ADRX_MMCX_ CMD_H[02:00]	STADDR_H[02:00] Starting address	These bits determine the starting quadword address of the current operation. These bits are valid for I/O as well as CPU operations. They should be received with a load command.
ADRX_MMCX_ CMD_H[03]	STADDRPAR_H Starting address parity	This signal indicates parity on the preceding three starting address bits. It is received simultaneously with the starting address bits.
ADRX_MMCX_ CMDPAR_H[00]	ADDRPAR_H Address parity	This signal is always forced to zero.

5.4.11.2 MMCX-to-ADRX Interface

Figure 5–52 shows the control fields. Table 5–35 lists the bits and their descriptions.

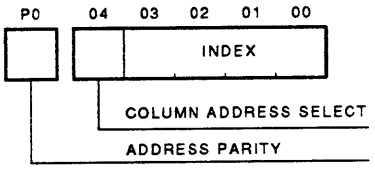


Figure 5–52 MMCX-to-ADRX Control Format

Table 5–35 MMCX-to-ADRX Field Descriptions

Signal	Field	Description
MMCX_ADRX_CTL_ H[03:00]	Index	These lines contain the index to the command whose address is currently required by the memory controller. They are used to select the appropriate memory address.
MMCX_ADRX_CTL_ H[04]	COLADRSEL_H Column address select	The assertion of this line causes the MAC address lines to be switched from row to column addresses.
MMCX_ADRX_CTL_ PAR_H[00]	ADRPAR_H Address parity	This line is a parity bit on the index field and the column address select signal.

5.4.12 DSXX MCA Interface

Table 5–36 lists the bits and their descriptions.

Table 5–36 DSXX-to-MMCX Field Descriptions

Signal	Field	Description
DSAX_MMCX_ STATUS_H[01:00]	MSKSTAT_H[01:00]	This signal is received from DSXX and indicates certain information about the byte mask received with the current data word. The information is encoded as follows:
DSAX_MMCX_ STATUS_PAR_H	MSKSTAT_PARITY_ H[00]	This signal is transmitted with the mask status bits to indicate parity.
DSB0_MMCX_STATUS_H[02:00]		
DSB0_MMCX_ STATUS_H[01:00]	MSKSTAT_H[03:02]	This signal is the same as the mask status above.
DSB0_MMCX_ STATUS_H[02]	BOD_H	This signal indicates that the first word of write data has been transmitted from DSXX to MDPX.
DSB0_MMCX_ STATUS_PAR_H	MSKSTAT_PARITY_ H[01]	This signal is transmitted with the above mask status bits to indicate parity.
DSB0_MMCX_ FATALERR_L	DSBFTLERR_L[00]	This signal indicates that a fatal error has been detected in DSB0, which is on the same MCU as MMCX.
DSB1_MMCX_STATUS_H		
DSB1_MMCX_ FATALERR_L	DSBFTLERR_L[01]	This signal indicates that a fatal error has been detected in DSB1, which is on the same MCU as MMCX.
DSB2_MMCX_STATUS_H		
DSB2_MMCX_ FATALERR_L	DSBFTLERR_L[02]	This signal indicates that a fatal error has been detected in DSB2, which is on the same MCU as MMCX.

1	0	Number of 1s in Mask
0	0	0
0	1	1, 2, or 3
1	0	Reserved
1	1	4

5.4.13 JDAX MCA Interface

Table 5–37 lists the bits and their descriptions.

Table 5–37 JDAX-to-MMCX Control Field Descriptions

Signal	Description
JDAX_MMCX_FATALERR_ L[00]	Indicates that a fatal error has been detected in JDAX.
JDAX_MMCX_ERRATTEN_ L[00]	Indicates that a nonfatal error has been detected in JDAX.

5.4.14 JDBX MCA Interface

Table 5–38 lists the bits and their descriptions.

Table 5–38 JDBX-to-MMCX Control Field Descriptions

Signal	Description
JDBX_MMCX_FATALERR_ L[00]	Indicates that a fatal error has been detected in JDBX.

5.4.15 Service Processor Unit Interface

Table 5–39 lists the bits and their descriptions.

Table 5–39 SPU Control Field Descriptions

Signal	Field	Description
SPU_MMCX_CTL_ H[00]	REQSTEPCTL_H, L	Assertion of this differential signal indicates that the service processor wants the memory to go to step mode. The memory must be assured of a number of clock cycles after the assertion of this signal. During this transition period, all pending transactions must be completed.
SPU_MMCX_ REQSTEPCTL_ H, L	–	Assertion of this differential signal indicates that the service processor wants the memory to go to step mode. The memory must be assured of a number of clock cycles after the assertion of this signal. During this transition period, all pending transactions must be completed.

5.5 MCDX MCA

The MCDX MCA controls the memory DRAMs (Figure 5–53). The MCD0 MCA, located on the DB0 MCU, supports MMU0 memory modules. The MCD1 MCA, located on the DB1 MCU, supports MMU1 memory modules. The MMCX receives command information from the JBox and, after decoding the information, sends commands, bank and segment addresses, and start to MCDX. The MCDX decodes the segment and the command (read, refresh, write, write read, and write pass) for the segment.

For step mode, MMCX sends single-step on, and the command (read, write, write read and write pass). MCDX decodes single-step on, and the command.

To read the data from memory and to write data into memory, MCDX sends row address strobe (RAS), column address strobe (CAS), and write enable (WE) control signals to access the data in the DRAMs on the memory module. MCDX sends read bus control signals, such as bypass, to control the flow of data on the read bus in the memory module. MCDX sends read latch enable to load the data from the read bus into the read latch. To coordinate memory events with SCU events, MCDX sends status information to MMCX.

5.5.1 Generating RAS, CAS, and WE

For each access to memory, the memory module activates 160 DRAMs on each memory module. A total of 640 DRAMs are activated on all four memory modules. The two segments operate independently and share a common data path called the read bus. The address lines are not common between segments. This organization permits two-way interleaving. Both segments can be accessed simultaneously.

Each segment has eight unique CAS signals. The WE signal determines if a write operation is permitted. The following steps summarize a write operation to bank 0 in which the MCDX MCA does the following:

1. Applies row address to all DRAMs in segment 0.
2. Sends RAS 0 to strobes in the row address.
3. Applies column address to all DRAMs in segment 0.
4. Applies WE to all DRAMs.
5. Sends CAS 0 through 7, depending on the data mask bits sent with the data. Data is written into the DRAMs where RAS, CAS, and WE are applied.
6. Deasserts RAS, CAS, and WE.

On the first module, the MCDX MCA sends each CAS line to 20 DRAMs per bank. (The 20 DRAMs represent a 20-bit slice of the data.)

On the second module, the same CAS signal is sent to the same 20 DRAMs. Each CAS line controls 40 DRAMs (longword), ECC check bits, and a mark bit. MCDX controls individual CAS lines only during write operations. During read operations, all eight CAS lines are logically tied together. MCDX sends the same RAS signal to the same 160 bits on all four memory modules. MCDX also sends the WE signal to the same 640 DRAMs on all four modules.

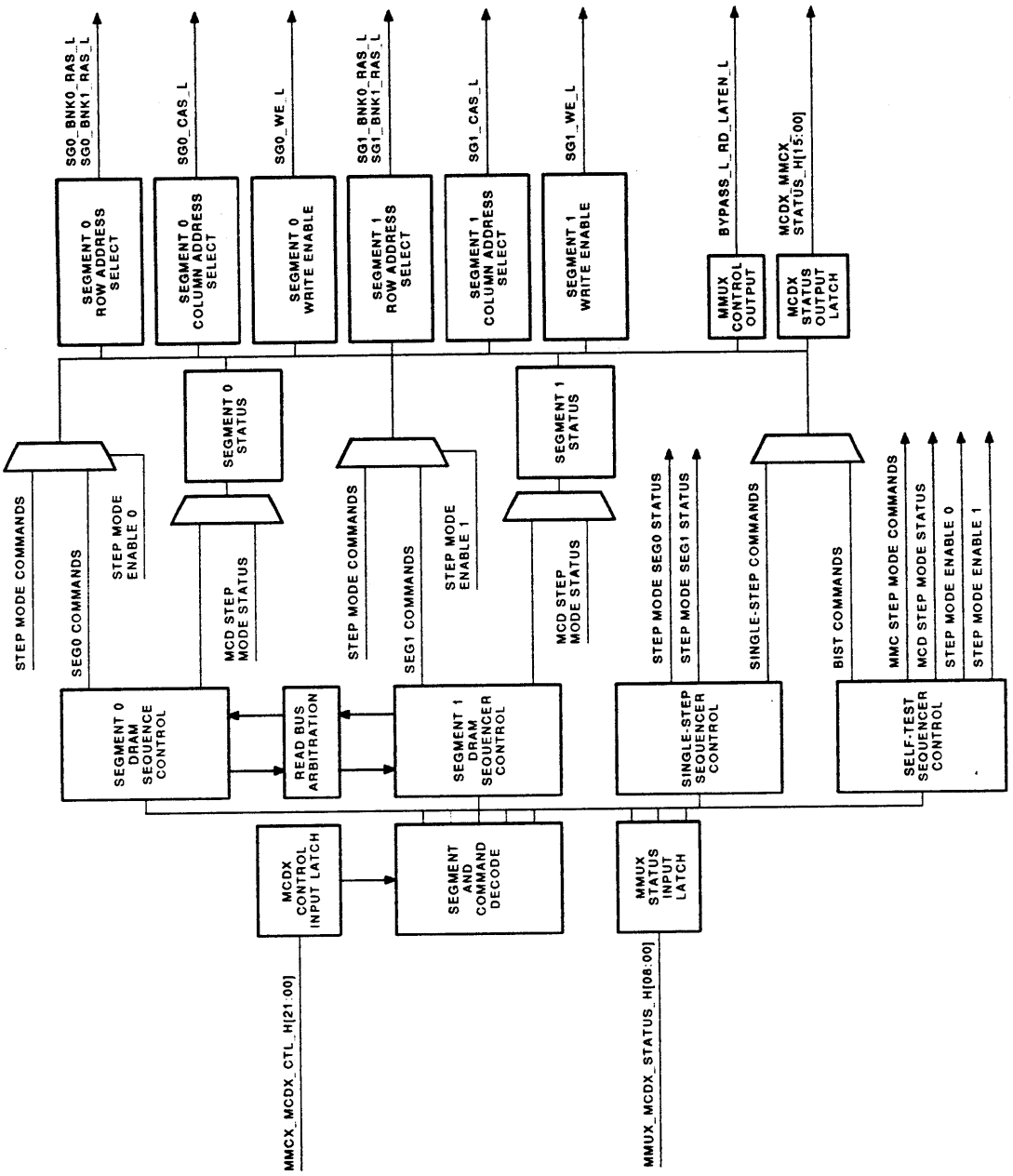


Figure 5–53 MCDX MCA Block Diagram

5.5.2 MCDX MCA Controllers

The MCDX MCA contains the following controllers:

- **Segment 0 DRAM sequence controller** — For read, write, read-modify-write, and refresh operations, the MCDX MCA uses the segment 0 DRAM sequence controller to generate the RAS, CAS, and WE control signals and to monitor the status of the read and write path in the memory module for segment 0. The controller receives the commands, read bus acknowledge, read-modify-write status bit, read latch busy, refresh required, and write data ready from the MCDX internal logic. The controller generates RAS, CAS, WE, segment busy, read bus busy, ready data ready, read latch enable, write done, and the current state of the DRAM controller.
- **Segment 1 DRAM sequence controller** — For read, write, and read-modify-write operations, the MCDX MCA uses the segment 1 DRAM sequence controller to generate RAS, CAS, and WE control signals and to monitor the status of the read and write path in the memory module for segment 1. The controller receives and generates similar signals for segment 1 as the segment 0 DRAM sequence controller.
- **Single-step sequence controller** — For single-step operations, the MCDX MCA uses the single-step sequence controller to generate the DCA commands and to monitor the status of the read and write path in the memory module during the single-step operation. The controller receives read latch busy, read-modify-write status bit, commands, step cycle busy, write data ready, and bank address from the MCDX internal logic. The controller generates single-step bypass, read latch enable, read data ready, write done, segment busy, command strobe signals, and DCA commands.
- **Self-test (BIST) sequence controller** — For self-test operations, the MCDX MCA uses the self-test sequence controller to generate the BIST and DCA commands and to monitor the status of the read and write path in the memory module during the DRAM data path test and DRAM control and address test. The controller receives BIST buffer available, BIST end of pattern, BIST error, and step cycle busy from the internal MCDX MCA logic. The controller generates linear feedback shift register hold, request read-modify-write mode, stop test, BIST beginning of data bit, command strobe, BIST command, and DCA commands.

MCDX receives the following:

- Control signals MMCX_MCDX_CONTROL_H[21:00] from the MMCX MCA. Figure 5–46 shows the control signals from MMCX. Table 5–29 lists the control signals and their descriptions.
- Status signals MMUX_MCDX_STATUS_H[08:00] from the MMUX. The MMUX sends segment 0 and 1 control parity bits to the MCDX MCA.

MCDX sends the following DRAM control signals to each segment in MMU:

- **Two RAS (row address strobe)** — Selects one bank of 160 bits within a segment.
- **Eight CAS (column address strobe)** — For a read operation, all eight CAS lines are asserted, allowing all 160 bits to be read. The DRAM data path is structured to allow eight transfers of 20 bits to unload the data from the data output latch. If a write is in progress, SCU sends mask bits to control assertion or deassertion of the appropriate CAS lines for correct data capture into the DRAMs. The mask bits can mask out selected bytes and prevent them from being written into the DRAMs.
- **One WE (write enable)** — Selects a read or write operation.

- **Bypass read latch enable** — Enables loading the read data latch with I/O write data for merging during byte writes or CPU (cache block) having written full status.

The MCDX MCA sends status information to the MMCX MCA. MCDX sends read bus busy and step cycle busy, normal mode or BIST mode, and hold LFSR data to the MMCX MCA. For BIST testing, MCDX sends the BIST command to MMCX. Figure 5–45 and Table 5–28 show the MCDX status information that the MCDX MCA sends to the MMCX MCA.

5.5.3 DRAM Sequencing

The DRAM sequence controller generates states for read, write, write read, write pass, and refresh operations. This section describes when the RAS, CAS, and WE control signals are generated during these operations. Section 5.9 describes the memory operations.

5.5.3.1 Read States

Figure 5–54 shows the read states and corresponding RAS and CAS signals for a read operation. Section 5.9.1 describes a read operation.

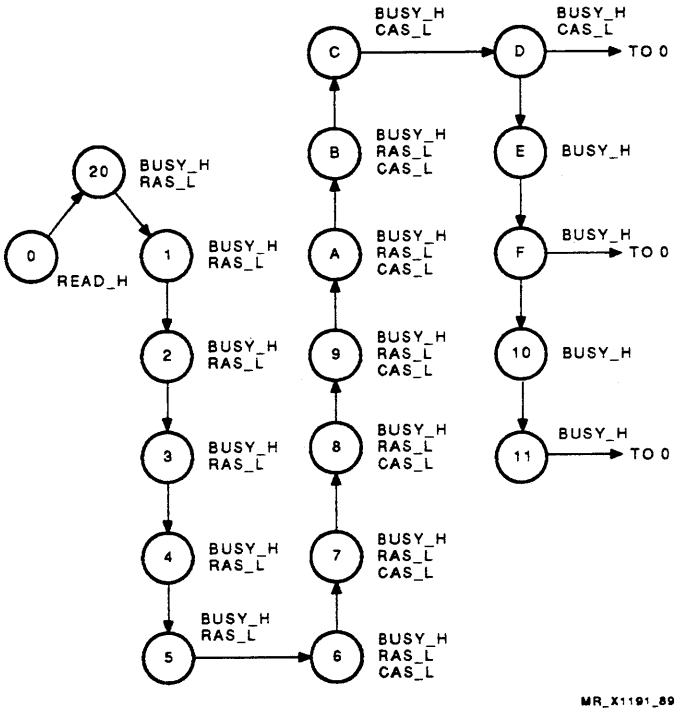
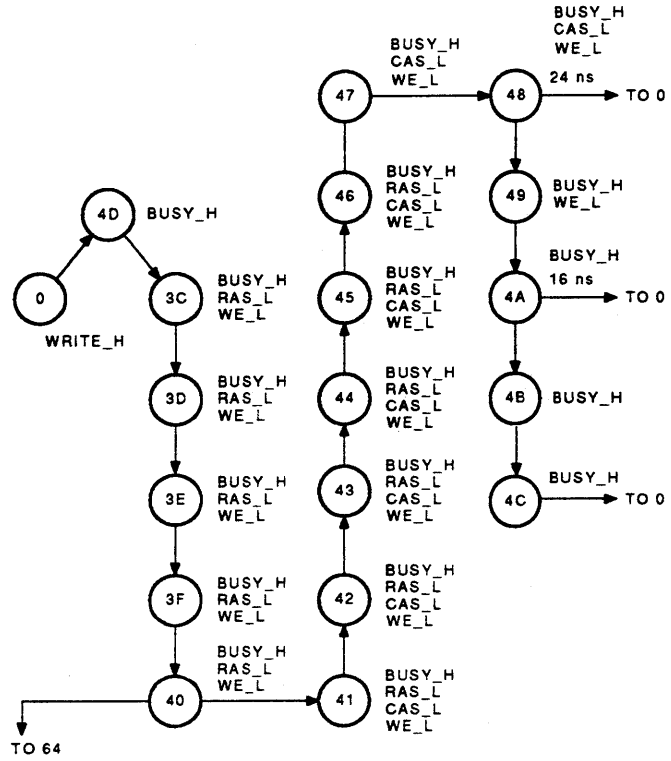


Figure 5–54 Read States

5.5.3.2 Write States

Figure 5–55 shows the write states and corresponding RAS, CAS, and WE signals for a write operation. Section 5.9.2 describes a write operation.



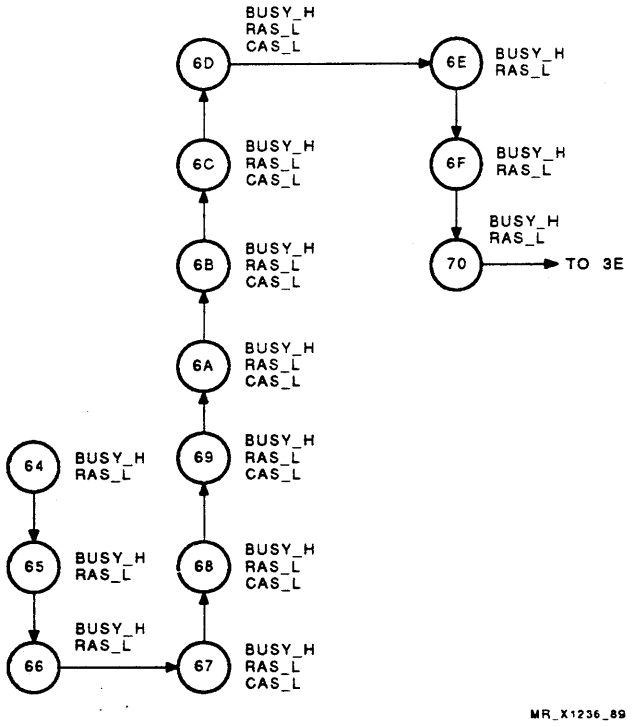


Figure 5–55 Write States

5.5.3.3 Write Pass States

Figure 5–56 shows the write pass states and corresponding RAS, CAS, and WE signals for a write pass operation. Section 5.9.5 describes a write pass operation.

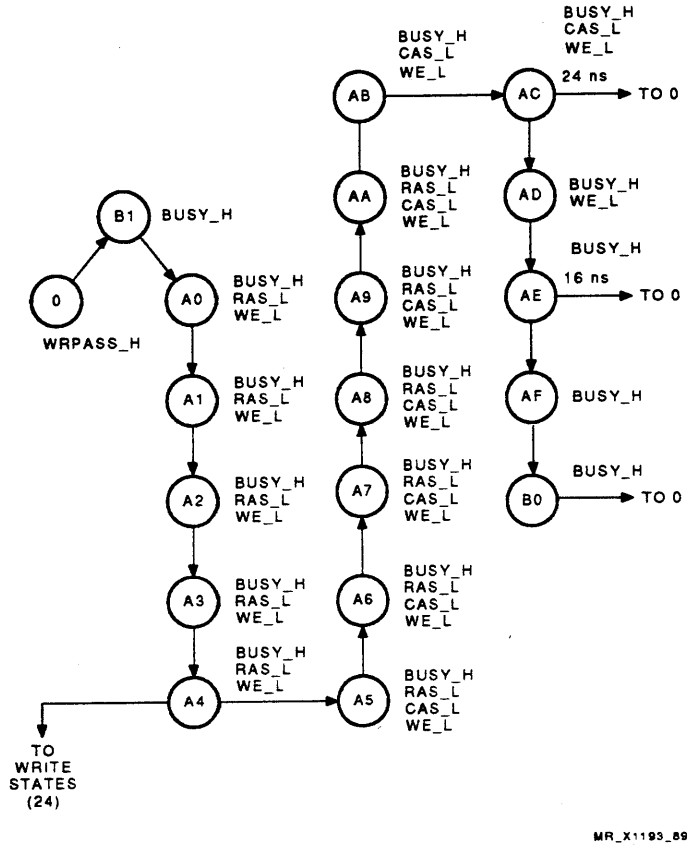


Figure 5–56 Write Pass States

5.5.3.4 Write Read States

Figure 5–57 shows the write read states and corresponding RAS, CAS, and WE signals for a write read operation. Section 5.9.4 describes a write read operation.

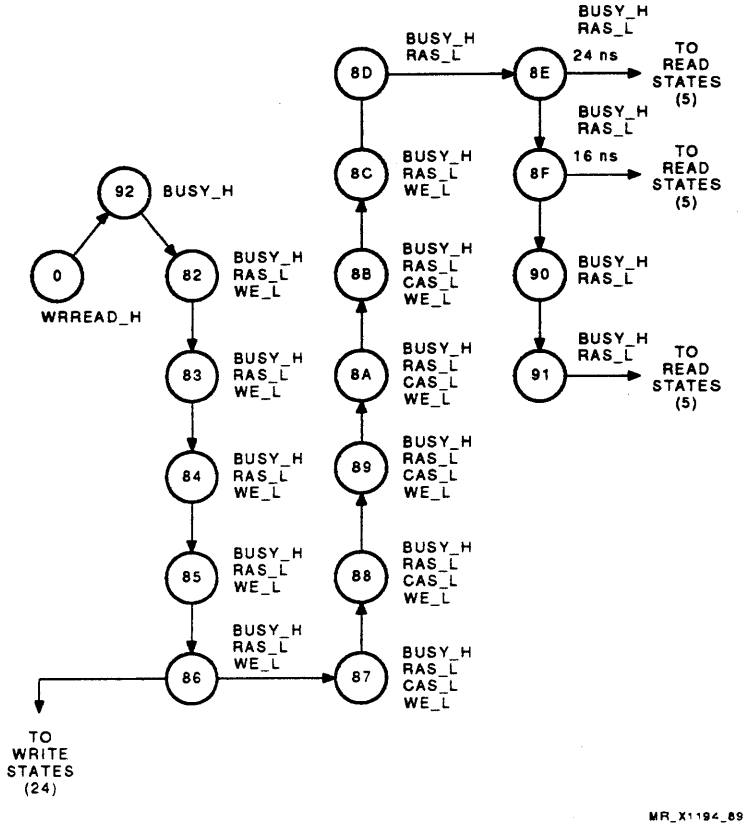


Figure 5–57 Write Read States

5.5.3.5 Refresh States

Figure 5–58 shows the refresh states and corresponding RAS and CAS signals for the refresh operation. Section 5.9.6 describes a refresh operation.

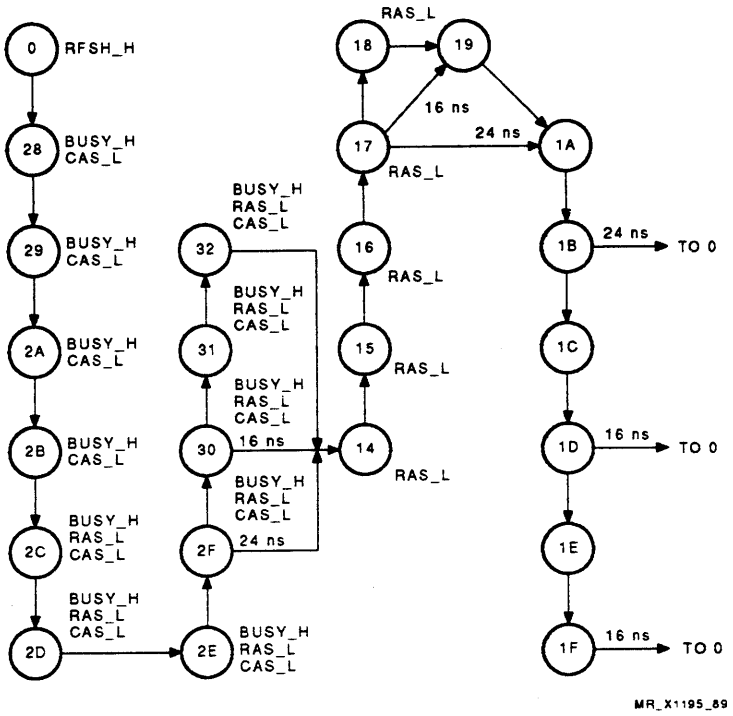


Figure 5–58 Refresh States

5.6 MDPX MCA

The MDPX MCA provides the data path for memory data to and from the data switch (Figure 5–59). The MDPX MCA sends JBox write data to the memory modules and receives JBox read data from the memory modules. The memory modules have an internal data path. Figure 5–30 shows the memory data path in the memory module. The MDPX MCA contains three data paths (read, write, and merge) and performs error checking and correction on data sent to and from the memory modules. This MCA contains BIST logic to support module testing.

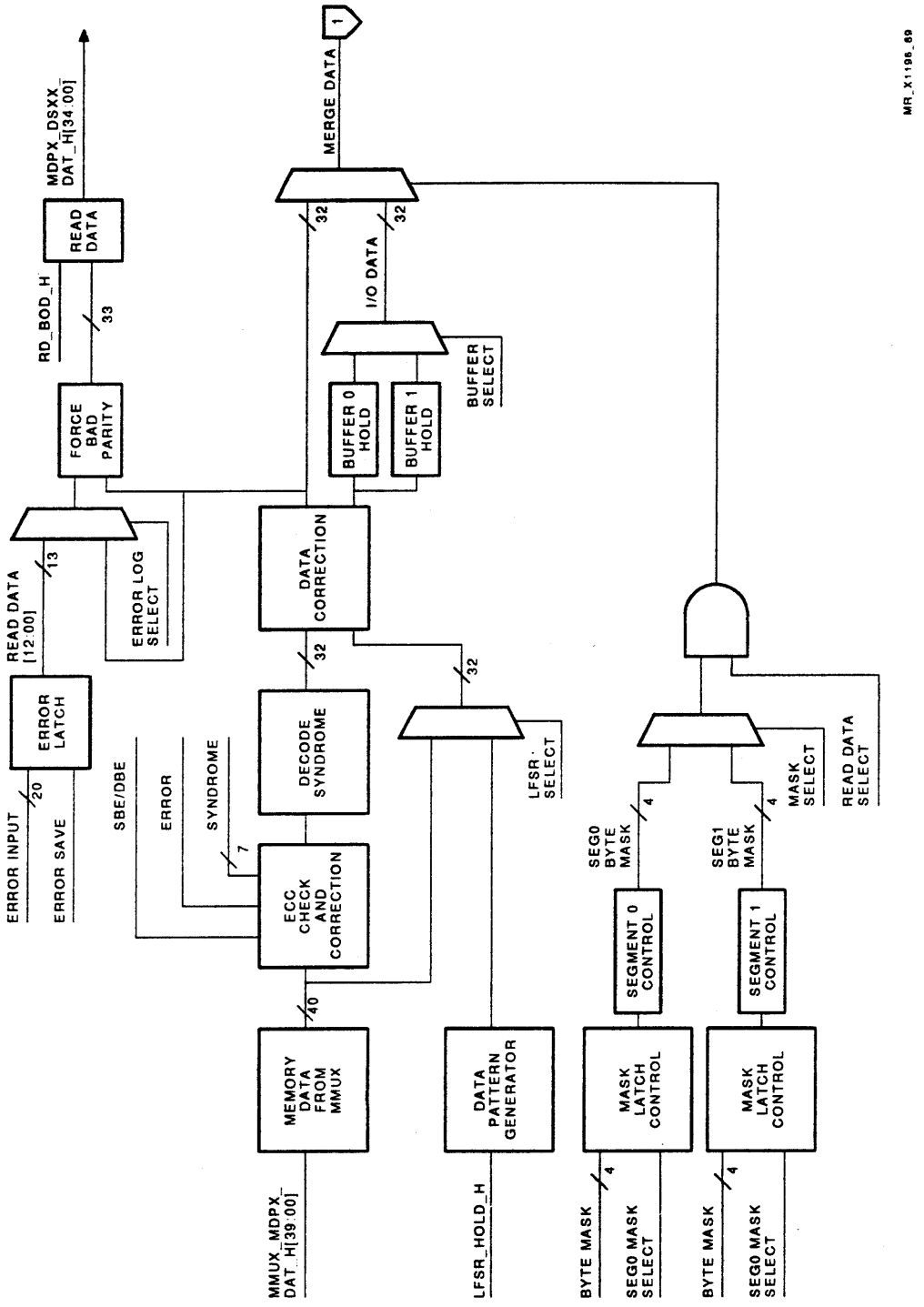


Figure 5–59 (Cont.) MDPX MCA Block Diagram

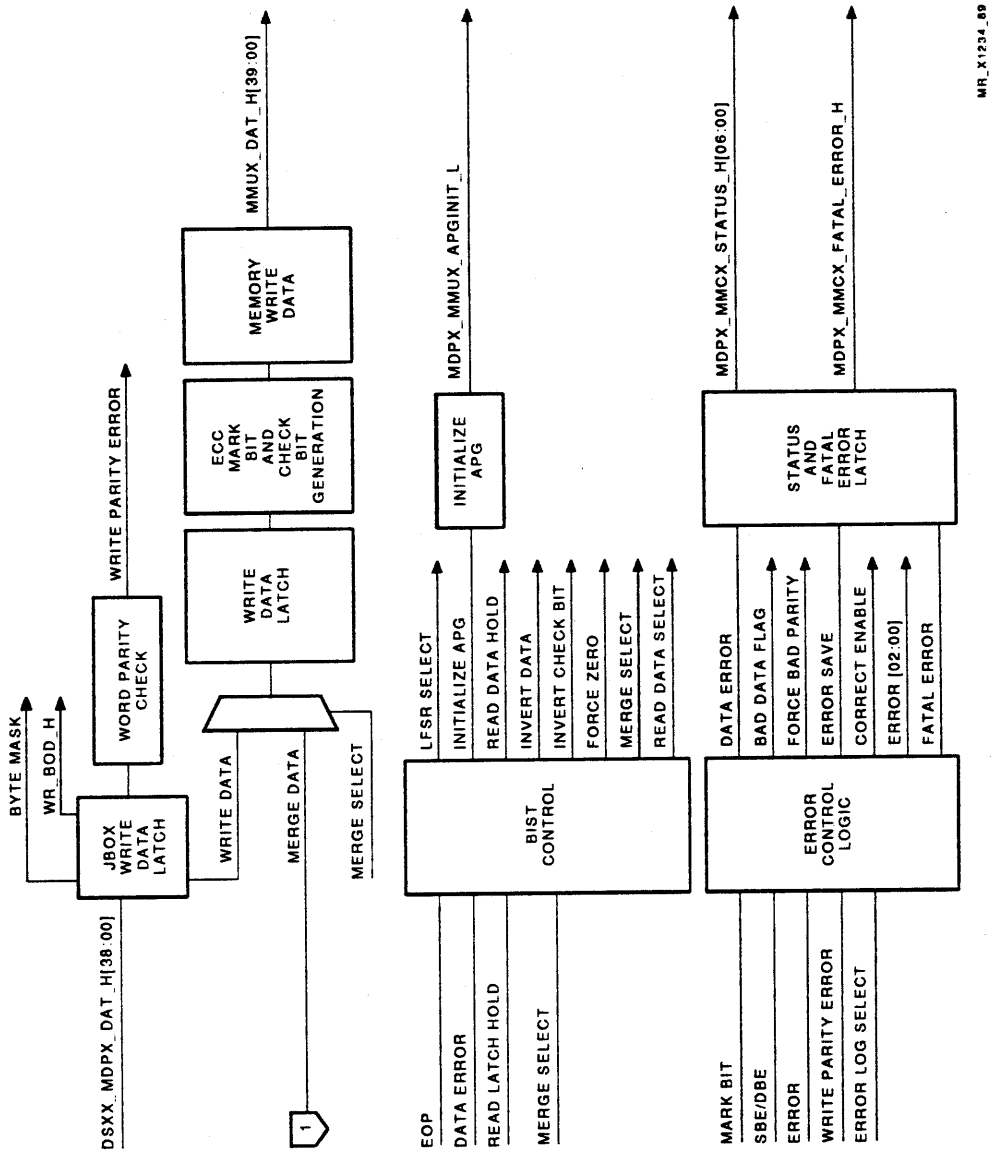


Figure 5–59 MDPX MCA Block Diagram

For write operations, MDPX receives DSXX_MDPX_DAT_H[38:00] from the data switch. Bits [38:37] are parity bits, [36:33] contain the mask data, [32] contains the write beginning of data signal, and [31:00] contain the data. MDPX latches the data from the data switch, appends check bits and a mark bit to the data, and sends MDPX_MMUX_ DAT_H[39:00] to memory. Bit [39] contains the mark bit, [38:32] contain the check bits, and [31:00] contain the data.

During write operations, two MDPX MCAs receive data. Each MDPX operates on a longword and provides ECC check bits on write data. MDPX passes the data to MMU, which stores the data in the data input latch (write buffer 0). Figure 5–60 shows the MDPX MCAs receiving data from the DSXX MCAs and sending the data to MMU.

On write operations, data passes through MDPX and is stored in MMU. Figure 5–61 shows the MDPX MCA write path. 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 a pair of MDPX MCAs until the entire block of data has been transferred.

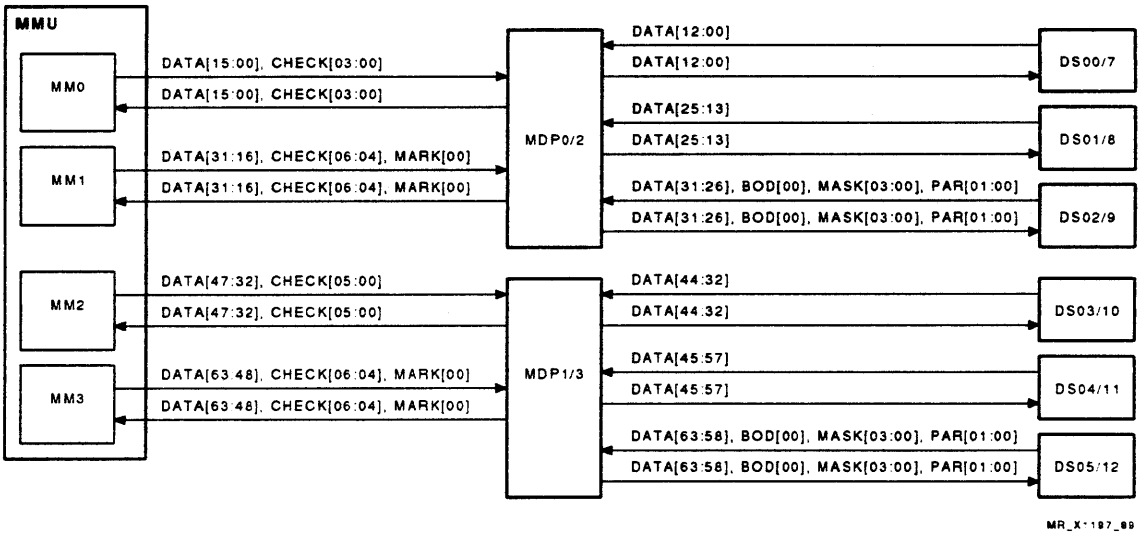


Figure 5–60 MDPX MCAs Receiving Data from the DSXX MCAs

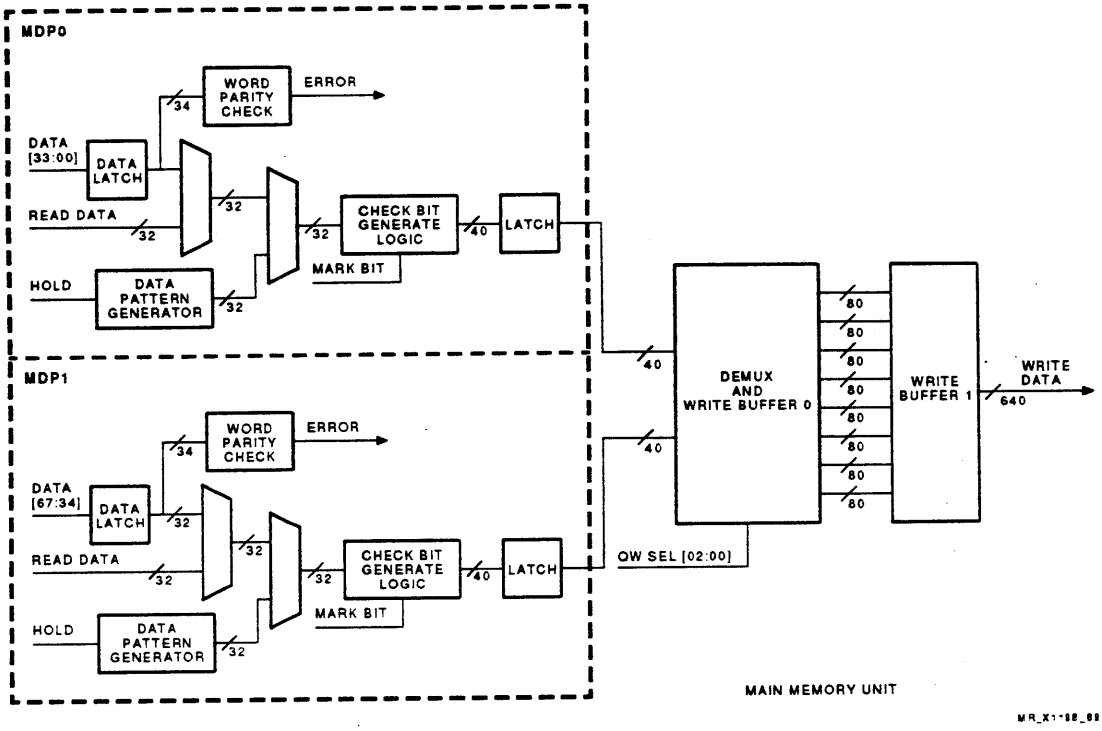


Figure 5–61 MDPX MCA Write Data Path

For read operations, MDPX receives MMUX_MDPX_DAT_H[39:00] from memory. Bit [39] contains the mark bit, [38:32] contain the received check bits, and [31:00] contain the data. The MDPX latches the data, checks and corrects single-bit errors, detects double-bit errors, detects write data errors, and sends MDPX_DSXX_DAT_H[34:00] to the data switch. Bit [34] contains the high longword parity bit, [33] contains the low longword parity bit, [32] contains the beginning of data signal, and [31:00] contain the data.

During read operations, data is received by the MDPX MCAs from MMU. Figure 5–62 shows the MDPX MCA read path. MDPX decodes the seven check bits, correcting single-bit errors and detecting double-bit errors. Data is then passed to the JBox.

MMU contains all data buffering needed, storing 128 bytes of write data in two 64-byte write buffers and 128 bytes of read data in two 64-byte read buffers.

For read-modify-write operations, MDPX receives the write data from the data switch and sends the data to memory. The data moves from the write buffers to the read bus and read buffers. MDPX receives the write data as if it were read data. MDPX holds this data, which is merged with read data in a holding buffer. MDPX receives the read data, performs the ECC checking and correcting, merges the read data with the write data in the holding buffer (regenerating ECC check bits), and sends the data as write data to memory.

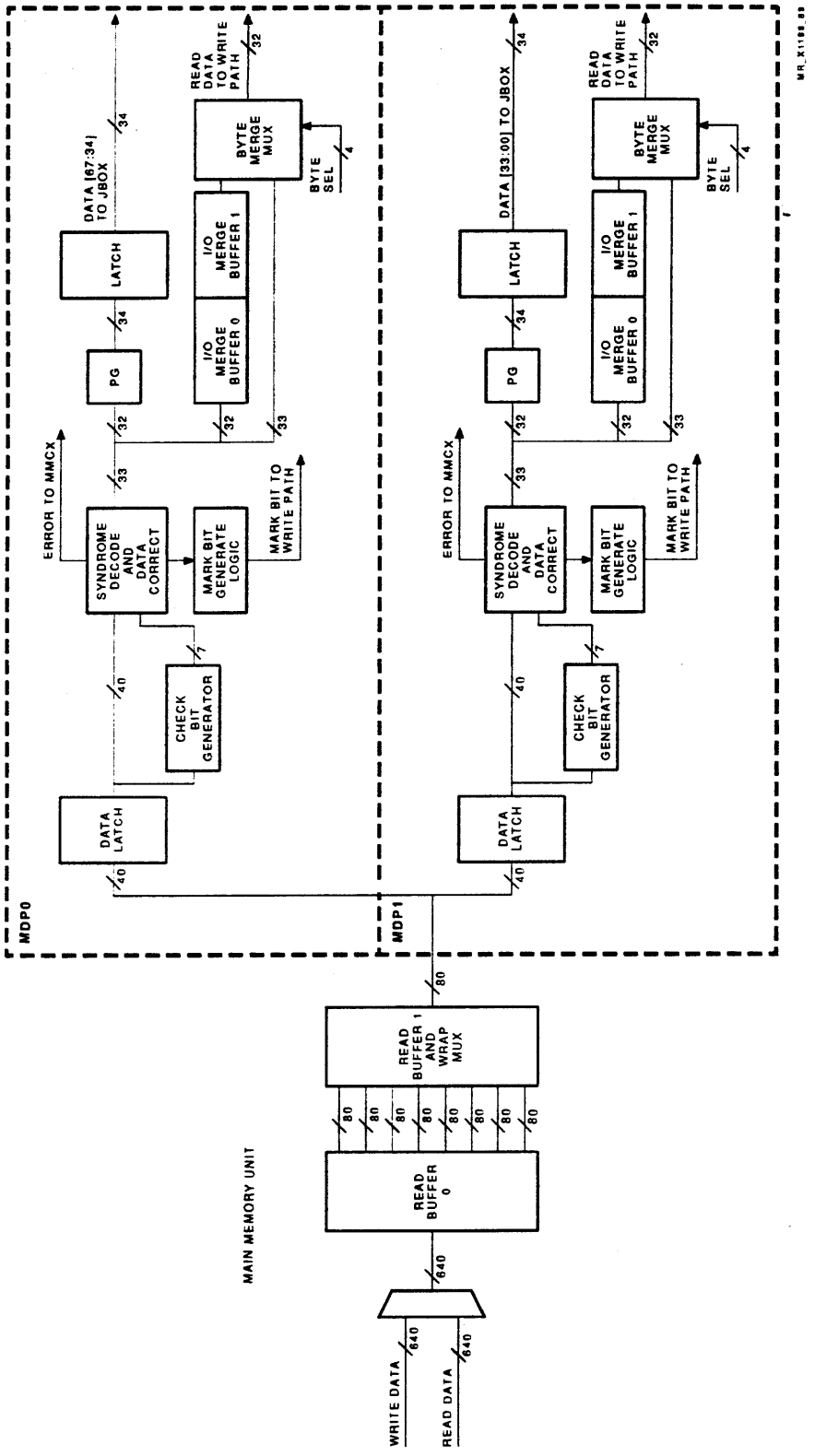


Figure 5–62 MDPX MCA Read Data Path

For write read operations, MDPX receives data from the data switch and sends the data to memory where the data (selected by the write strobes) is merged with the data in the DRAMs. MDPX receives the merged data from memory and sends the data to the data switch.

For write pass operations, MDPX receives data from the data switch and sends the data to memory. While the data is written into the DRAMs, the same data, using the read bus as a bypass path around the DRAMs, is loaded into the read buffer. MDPX receives the data from memory and sends the data to the data switch.

5.6.1 LFSR — Data Pattern Generator

LFSR in the MDPX is a data pattern generator that generates and applies a pattern to the data path during three BIST tests: DDP, data path, and DRAM. Section 5.10 describes the BIST operations.

The memory module DCA also contains an LFSR that is used as an address pattern generator. The DCA LFSR has direct access to the address path and generates addresses only in the DRAM test that exercises every address. Scan loads the addresses for other tests into the ADRX MCAs, which send the row and column address bits to MMU. (During normal operations, the tag MCU's ADRX MCA generates the address bits.)

Each LFSR cycles through every address before repeating the addresses. Both LFSRs are initialized to zero and are then forced to their initial address. The contents of the LFSRs are incremented so that every address is tested.

The MDPX LFSR is synchronized to the DCA LFSR, in that the data pattern written equals the address or another pattern derived from the address. SPU can examine the contents of only the MDPX LFSR by scanning and determine which address failed, if the stop on error bit is set.

The DDP and data path test uses the MDPX LFSR to generate random data patterns. The LFSR hold signal is deasserted during the tests so that the data pattern is continuously changing. The DRAM test uses both the MDPX LFSR and the DCA LFSR. The following steps summarize how the DRAM test uses the MDPX LFSR and DCA LFSR to test 1-Mbit DRAMs:

1. The SPU uses a scan to load 80000 into the MDPX LFSR. The 80000 is the last pattern for the 1-MByte DRAM.
2. The MCDX BIST controller deasserts the LFSR hold signal for one clock tick. Whenever the BIST controller deasserts the LFSR hold signal, DCA receives either an APGINIT from MDPX (if the last pattern is asserted) or an APG increment signal from MMCX (if the last pattern is not asserted). MCDX sends the LFSR hold signal to MMCX, which forwards it to MDPX.
3. MDPX asserts the last pattern signal (because the last pattern is loaded into LFSR) and asserts APGINIT. MDPX uses the last pattern signal to set the next data pattern generator to 100000, in which the lower 20 bits define the address as 0.
4. DCA receives APGINIT, which clears the DCA LFSR. (APGINIT also sets a bit in a flip-flop, which later forces a 1 into LFSR.)
5. Both LFSRs are effectively at 0.
6. The MCDX BIST controller initiates a write to address 0.
7. When the write is completed, the BIST controller again deasserts the LFSR hold signal for one clock tick.

8. MMCX receives the LFSR hold again and sends it to the MDPX MCA.
9. MDPX uses LFSR hold to advance LFSR to its next pattern, 200001.
10. The MMCX MCA sends an APG increment over the address strobe line to DCA. The DCA now has 00001, and the LFSRs are again equal. Address 1 is written and the process is repeated.
11. The next time that the BIST controller deasserts LFSR hold, the existing pattern shifts left one bit and becomes 0002. After 2**20-1 deassertions of LFSR hold, LFSR returns to the last pattern (80000), and having passed through all possible addresses except 0.

For 4- and 16-Mbit DRAMs, the LFSR contents are 22 and 24 bits long.

5.6.2 ECC Initialization

Three bits can be set during scan initialization to control ECC logic operation. These bits are independent of the BIST operation and do not have any effect on the force bad data logic that is used during a write and a read-modify-write operation.

5.7 ACU-to-JBox Interface

The JBox sends command and status to the MMCX MCA, data to the MDPX MCAs, and row and column addresses to the memory module's DRAM control and address (DCA) chips. For more details on the ACU-to-JBox interface, see Chapter 2.

Each ACU has two input command buffers and an output command buffer in the JBox. To communicate with the JBox, MMC0 can load commands (return data read or error command) into one of two input ACU0 command buffers and MMC1 can load commands into one of two ACU1 input command buffers.

To communicate with the ACU, the JBox loads commands into the memory command output buffer (in CTLB) and sends the commands (read, write, write read, and write pass) to the MMCX.

5.8 Modes of Operation — Timing

The memory module has the four following modes of operation:

- **Normal mode** — In this mode, the SCU controls DRAM and the DRAM data paths. The SCU also generates refresh cycles.
- **Step mode** — In this mode, the SCU sets up write data, transfers read data, and sends commands to the DRAM command and address gate array (DCA). The DCA executes the command using an on-board, 10-MHz oscillator as a clock. DCA performs the refresh cycles. In this mode, accesses to the field service register (FSREG) can be made.
- **Standby mode** — In this mode, using a 10-MHz oscillator as a clock, the standby logic continually refreshes the DRAMs. No DRAM accesses are allowed in standby mode.
- **Address pattern generation mode** — In this mode, DCA redefines control lines from the SCU. The lines control the address pattern generator (APG). The SCU logic performs all the refresh and normal cycles. The memory system self-test uses this mode.

5.8.1 Normal Mode

In normal mode, the DRAM control signals (RAS, CAS, and WE) are derived from the system A and B clocks. This allows optimum memory performance. These signals can be generated in increments of 16 ns and are closely coordinated with system events such as load commands for reads and the arrival of data for writes. DRAM control signals cannot be generated this way in step mode.

5.8.2 Step Mode

During system debug, the system can be placed into a single-step mode of operation in which the system can be stepped through a program or hardware operation. Because the system clocks may be off or running at very reduced speeds, normal memory functions cannot be maintained. The MAC modules are placed in a mode in which they maintain memory refresh and perform memory accesses for the system.

The purpose of step mode is primarily to support burst clocking. Control is passed to the MMU. The MMU does not use the system clocks and is not affected by their stopping and starting. The step mode is only used for debug, system maintenance, and to access standby mode. While in step mode, the system clock must be at a cycle time greater than 21 ns before any read operations can take place.

In step mode, the system clocks can stop anytime. If the clock stops in the middle of a memory cycle with RAS and CAS asserted for a long time, data can be lost or the DRAMs damaged. In step mode, the DRAM control signals are derived from a free-running oscillator on the memory modules.

5.8.2.1 Entering Step Mode from Normal Mode

The following steps summarize entering step mode from normal mode:

1. SPU sends REQSTEPCTL to MMCX.
2. MMCX inhibits DRAM control commands from MCDX.
3. MMCX waits for not busy from MCDX.
4. MMCX transmits STEPCTLEN to each DCA. (See Figure 5–63.)
5. Each DCA starts the step mode controller.
6. When each DCA sends STEPCTLOK to SPU, the system is in step mode.

In step mode, the MCDX DRAM segment control is disabled. MMCX transmits the memory command to MCDX. The MCDX translates the command into a series of DCA step mode commands. DCA latches the DRAM control signals as a command.

DCA uses the RAS lines to select the bank of DRAMs and start the command cycle. CAS and WE lines carry a 4-bit command to DCA. DCA stores the command in the DCA command buffer. Figure 5–64 shows the 4-bit control field. Table 5–40 lists the values for and descriptions of the control field.

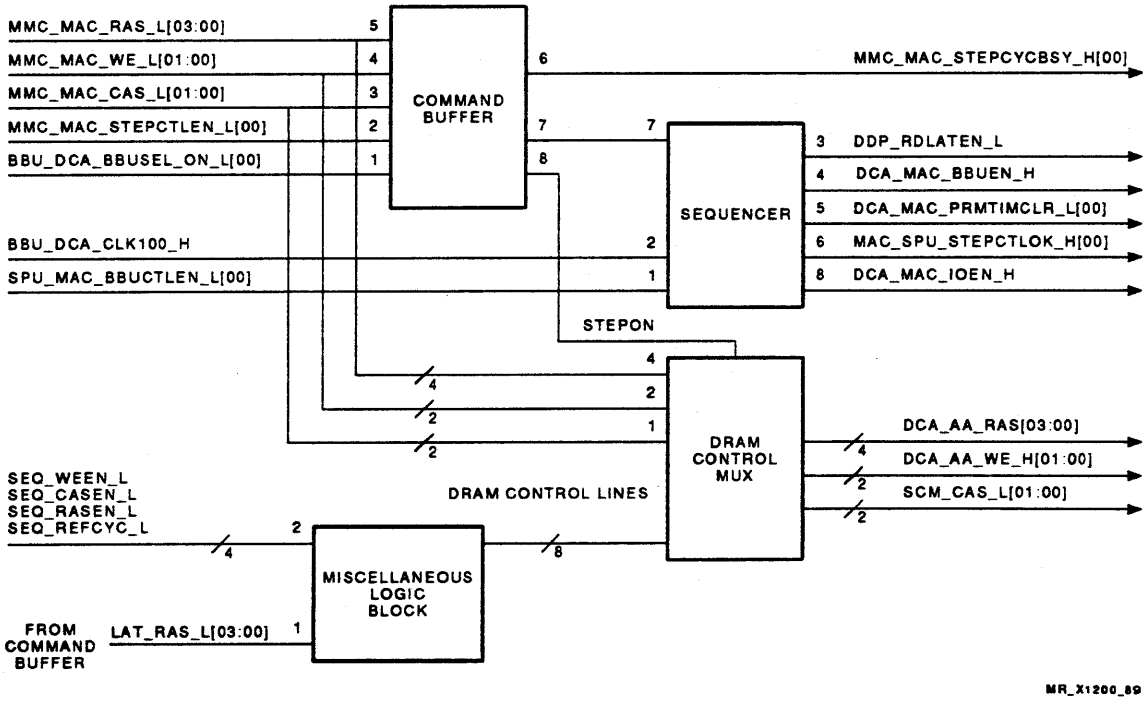


Figure 5–63 Step Mode Logic on the Memory Module

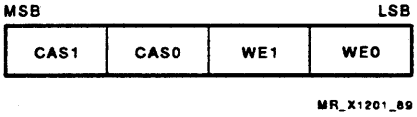


Figure 5–64 Step Mode Command Signals

Table 5–40 Step Mode Commands

Value	Name	Description
0000	–	Reserved.
0001	Self-test 16 enable	This command performs two functions. One function is to configure the APG to 24 bits. This implies that the memory module is configured with 16-Mbit DRAMs. The second function puts DCA into APG mode.
0010	Self-test 4 enable	This command performs two functions. One function is to configure the APG to 22 bits. This implies that the memory module is configured with 4-Mbit DRAMs. The second function puts DCA into APG mode.
0011	Self-test 1 enable	This command performs two functions. One function configures the APG to 20 bits. This implies that the memory module is configured with 1-Mbit DRAMs. The second function puts DCA into APG mode.
0100	–	Reserved.
0101	–	Reserved.
0110	–	Reserved.
0111	–	Reserved.
1000	EEPROM write	This command instructs DCA to perform a byte write of EEPROM at the address defined by the contents of the segment 0 address latch. Data [07:00] from the 160-bit write buffer 1 contains the data to be written.
1001	EEPROM read	This command instructs DCA to perform a read of EEPROM at the location pointed to by the segment 0 address latch. The DCA loads the data (Figure 5–67) into the 160-bit DDP read buffer 0.
1010	Memory write	DCA performs a write to the DRAMs. This command requires that the data be preloaded into the DDP write buffer 1, the row address be preloaded into the segment 0 address latch, and the column address be preloaded into the segment 1 address latch. DCA signals completion of this command by deasserting step cycle busy.
1011	Memory read	DCA performs a read of the DRAMs. This command requires that the row address be preloaded into the segment 0 address latch and the column address be preloaded into the segment 1 address latch. DCA signals completion of this command by deasserting step cycle busy, at which time the read data is available in the DDP read buffer 0.
1011–1111	–	Reserved.

5.8.2.2 Exiting Step Mode

The following steps summarize how the memory system exits step mode and returns to normal mode:

1. SPU deasserts SPU_MMC_REQSTEPCTL_H and sends the signal to MMCX.
2. MMCX receives the deassertion of SPU_MMC_REQSTEPCTL_H and deasserts MMC_MAC_STEPCTLEN_H to the DCAs.
3. Each DCA monitors MMC_MAC_STEPCTLEN_H and, on negation, exits step mode after any memory cycle in progress is complete.
4. Each DCA continues to assert MAC_MMC_STEPCYCBSY_H while any memory cycles are in progress.
5. MMCX monitors MAC_MMC_STEPCYCBSY_H from each DCA and resumes control immediately after MAC_MMC_STEPCYCBSY_H is deasserted.

5.8.3 Standby Mode

During a scan operation or system power loss, the memory modules go into standby operation. Standby operation continues to refresh the DRAMs, ensuring that the contents of memory are not compromised. The standby logic provides RAS, CAS, and WE to the DRAMs during the power loss or scan operation. Standby mode is usually entered using a handshake sequence initiated by SPU.

The memory subsystem cycles into step mode before it can enter standby mode. With one exception, this is always the case. During the DRAM BIST, DCA receives one of three self-test enable step mode commands. The command puts DCA into a state in which it cannot accept any more step mode commands. The contents of the address pattern generator in the DCA is placed into the address path. (Normally, DCA strobes the address bits coming from the ADRX MCAs onto the address lines.) This state between step and normal is called the APG mode.

When DCA is in APG mode, SPU can initiate standby mode by asserting STBYCTLEN. This activity is useful during BIST and keeps DCA in APG mode so that the APG contents are not affected. Therefore, when using a stop on error function, the test can be restarted where it left off.

5.8.3.1 Initiating Standby Operation

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 SPU_MAC_STBYCTLEN_L to the memory subsystem. The memory subsystem switches into standby operation if no DRAM cycles are in progress. Otherwise, the memory subsystem enters standby mode after completing the DRAM cycle.

Table 5–41 lists the SPU handshaking signals and their descriptions.

Table 5–41 SPU Handshaking

Signal	Description
SPU_MAC_CTLINIT_L	DCA performs a complete initialization of all its circuitry.
SPU_MMC_REQSTEPCTL_H	MMCX completes any memory cycle in progress and brings all memory cycles to a halt. MMCX sends MMC_MAC_STEPCTLEN_H to DCA. The negation of SPU_MMC_REQSTEPCTL_H results in the negation of MMC_MAC_STEPCTLEN_H, sent to the DCA.
MMC_MAC_STEPCTLEN_H	The control logic on each memory module switches to a local DRAM control mode. The DRAM control lines between MMCX and the DCAs change function to become command lines. DCA completes any DRAM cycles in progress when MMC_MAC_STEPCTLEN_H is negated. MMCX engages in normal system timing upon the negation of MAC_MMC_STEPCYCBSY_H.
MAC_SPU_STEPCTLOK_H	This signal is asserted after all memory modules have switched into step mode. This signal deasserts under two conditions. One, if the memory modules are switched into standby mode from step mode. Two, if the memory modules are switched into normal mode from step mode.
MAC_MMC_STEPCYCBSY_H	The DCA is executing a DRAM timing cycle.
SPU_MAC_STBYCTLEN_L	The memory module enters standby operation. This signal can be asserted only during step mode. In standby mode, the memory modules ignore all external signals except this one. The negation of this signal switches the memory modules out of standby.

5.8.3.2 Exiting Standby

After power is restored, the memory modules remain in standby mode until after initialization. Each memory module is initialized by the SPU with SPU_MMC_CTLINIT_L signal. As part of the initialization, the SPU sets conditions so that MMCX is in step mode. After scan operations to MMCX are complete, the SPU negates SPU_MAC_STBYCTLEN_L to the memory modules. At this point, each memory module responds with MAC_SPU_STEPCTLOK_H. Each memory module is then in step mode.

5.8.4 APG Mode

Address pattern generation (APG) mode supports memory self-tests and can be entered only through the step mode. A step mode command (SLFTESTxEN) is issued on the CAS and WE lines. This command enables DCA to enter APG mode and to select the AP sequence to test the DRAMs. The address strobe lines increment the address generator and select between row and column addresses. For more detail, see Sections 5.6.1 and 5.8.3.

5.8.5 Switching from One Mode to Another

The SPU can switch the memory subsystem from one mode to another mode. However, the mode switching must be done in a specific order to preserve data integrity in the DRAMs. The switch operations are as follows:

Standby to step
Step to standby
Step to normal
Normal to step
Step to APG
APG to standby

Figure 5–65 shows the mode switching logic in the memory module.

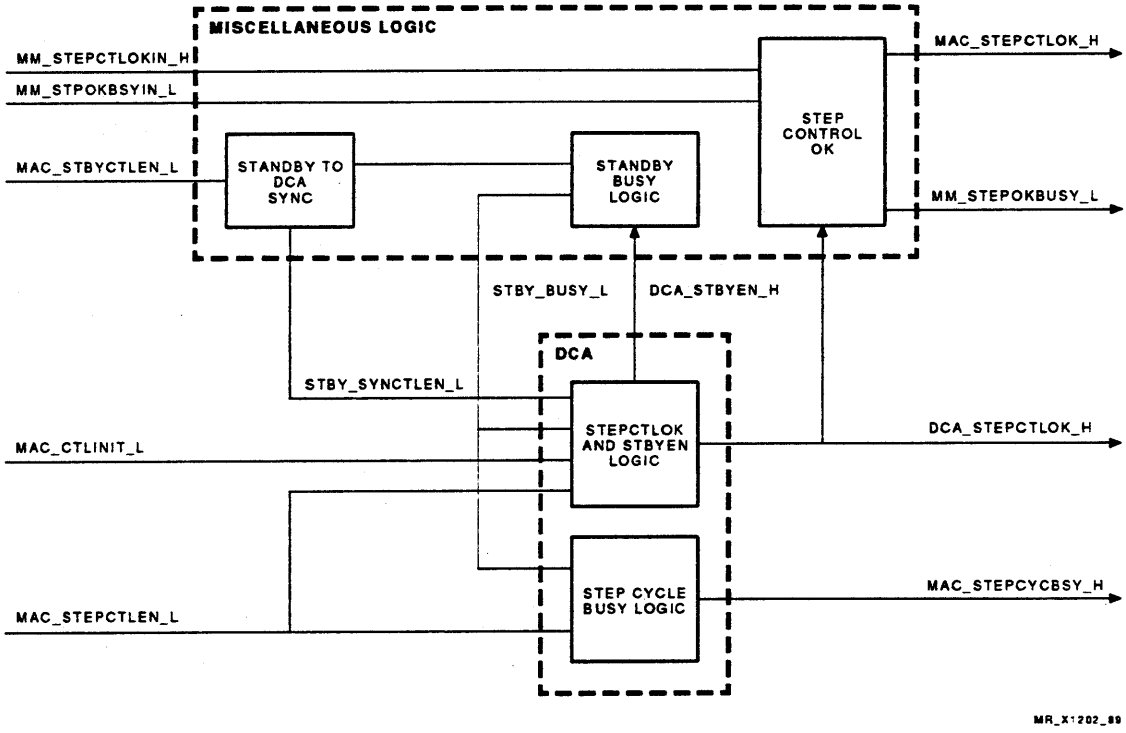


Figure 5–65 Mode Switching Logic in the Memory Module

From power-on to normal system operation, the mode switching order is as follows:

1. Standby to step
2. Step to normal

From normal system operation to standby (battery backup or scanning), the mode switching order is as follows:

1. Normal to step
2. Step to standby

From normal system operation to an operation that stops the system clocks, the mode can switch from normal to step.

From an operation that allows system clocks to stop to a normal system operation, the mode can switch from step to normal.

The mode switching operations do not apply during the BIST operation. During the BIST operation, the MCDX BIST controller directs the mode switching functions normally executed by the SPU. The linear feedback shift register in the MDPX MCA supplies the test data during the DRAM test, and the linear feedback shift register in the DCA supplies the address.

During the BIST operation, the mode can switch as follows:

1. From standby to step to APG
2. From APG to standby

5.9 Memory Operations

This section describes the memory operations performed by the memory subsystem. These operations are as follows:

Read
Write
Read-modify-write
Write read
Write pass
Refresh
EEPROM read and write

5.9.1 Read Operation

When a read data cycle executes, MMU reads 640 DRAMs and loads the data into read buffer 0. Figure 5–30 shows the read path in the memory module. The DRAMs load data into read buffer 0, and the memory module transfers the data to read buffer 1. This allows a second read operation from the other segment to continue. Read buffer 1 feeds into a 8:1 selector used to wrap data 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 transfer of the 80-bit data word is from four memory modules to two MDPs. Each MDPX operates on 40 bits, performing single-bit error correction and double-bit error detection as necessary. Each MDPX then transmits a longword per clock cycle to the JBox using the control/command interface between the JBox and MMCX. A summary of the sequence is as follows:

1. The JBox logic transfers command information to the MMCX segment command buffer.

   The MMCX and MCDX do the following:

   a. Decode command and index information and determine if the data is for a CPU or I/O operation.

   b. Decode bank select.

   c. Strobe the address from JBox into MMU.

   d. Perform DRAM cycle timing through DCA.

   e. Latch data from the DRAMs into MMU read buffer 0.

   f. Transfer data to MMU read buffer 1.

   g. Complete the DRAM timing sequence.

   h. Transmit the return data read command to the JBox.

   i. Receive the send data command from the JBox.

   j. Transfer each quadword through the MDPX MCAs.

2. The MDPX MCA under MMCX control does the following:

   a. Receives one quadword per clock cycle from MMU.

   b. Checks for errors and corrects if necessary (single bit).

   c. Generates word parity.

   d. Sends one quadword per clock cycle to the data switch in the JBox.

3. The MMCX updates segment command buffer available status for the JBox.

The read data transfers from MMU to MDPX use free-running STRAM clocks. Consequently, MMU constantly returns data to MDPX. However, the data is not always valid. When read data is requested from memory, a read cycle is initiated. A block of 64 bytes is always accessed from memory, regardless of the length of the transfer. When the data becomes valid at the DRAM outputs, it is latched into the read data buffer 1. The data is then transferred to the data output latch when the latch becomes available. MMCX sends one, two, four, or eight valid read selects to MMU, depending on the length.

As the read data enters MDPX, MMCX sends a signal called ECCENB_H to MDPX to enable the comparison of expected check bits with received check bits for error detection. This signal is asserted only for the length of the data transfer. The read data from MMU constantly flows from MMU to MDPX, and the beginning of data bit is appended only to the first quadword of a valid data transfer. Error detection is enabled only for valid data quadwords.

5.9.1.1 Wrap on Read Sequence

Wrap on read sequences can occur for I/O and CPU read requests. A wrap on read is defined as an octaword or hexword operation (for I/O), and a cache block operation (for CPU) in which the quadwords are returned in a specific pattern. The specifically addressed quadword is returned first, independent of alignment. The remaining quadwords are returned in a specific pattern.

If the index value indicates that the data is for an I/O request, MMCX uses the length field [03:02] of CCU_MMCX_CMD_H[14:00] to determine the number of quadwords to return. If the index value indicates that the data is for a CPU request, the length is eight quadwords. MMCX uses the index value and three starting bits from the ADRX MCA and sends the appropriate sequence of read select values to the data output latch in the memory module. MDPX receives the quadwords and ECC check bits in the correct order and sends the quadwords and word parity to the data switch. The data switch sends the data to the CPU or ICU.

XMI protocol requires that all octaword and hexword reads, both normal and interlocked, be delivered in a specific wrapped order. Table 5–42 shows the quadword, octaword, and hexword boundaries on which I/O can request data. A hexword read is made up of two octaword reads, and the addressed octaword read data are returned first. Within each of the octawords, the wrapping order is the same; the quadwords in the second octaword are in the same order as the quadwords in the first.

Table 5–42 I/O Boundaries

Quadword	Octaword	Hexword
0	–	–
1	0	–
2	–	–
3	1	0
4	–	–
5	2	–
6	–	–
7	3	1

For I/O requests, the wrap order for read data follows the XMI bus specification. Tables 5–43 and 5–44 list the possible wrap sequences for each cycle.

The CPU request to memory is for eight quadwords of data, and the requested quadword is returned in the first cycle. The possible CPU wrap sequences are listed in Table 5–45.

Table 5–43 I/O Cycles for Wrap on Read (Quadwords 0, 1, 2, and 3)

1st	2nd	3rd	4th
0	1	–	–
1	0	–	–
2	3	–	–
3	2	–	–
0	1	2	3
1	0	3	2
2	3	0	1
3	2	1	0

Table 5–44 I/O Cycles for Wrap on Read (Quadwords 4, 5, 6, and 7)

1st	2nd	3rd	4th
4	5	–	–
5	4	–	–
6	7	–	–
7	6	–	–
4	5	6	7
5	4	6	7
5	4	7	6
6	7	4	5
7	6	5	4

Table 5–45 CPU Cycles for Wrap on Reads

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

5.9.2 Write Operation

The memory subsystem can write from 1 to 16 longwords. The JBox transfers data in 8-byte wide (quadword) increments. ACU splits the data path into two 4-byte (longword) paths. Each 4-byte path feeds into an MDPX.

The check bit generation logic appends a 7-bit ECC code to the 32-bit data path. The fortieth bit (mark bit), used in conjunction with the ECC code, is also appended to the data. ACU sends 80 bits (40 bits from each MDPX) to MMU.

MMU loads the 80 bits into the data input latch (write buffer 0) through a 1:8 demultiplexer. Figure 5–30 shows the write path in the memory module. The encoding 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, MMCX sets 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, allows CAS to be applied to the set of 40 DRAMs that enables writing.

The following steps summarize a write operation:

1. The JBox transfers command information to the selected segment command buffer in MMCX.
2. The JBox logic transfers data to the memory subsystem as follows:
   a. Data to the MDPX MCAs
   b. Mask and BOD to MMCX
   c. Beginning of data bit set during first and fourth transfer
3. The MMCX and MCD do the following:
   a. Decode command information.
   b. Decode index to determine CPU or I/O operation.
   c. Decode bank select information.
   d. Strobe the address into the selected segment.
   e. Initiate DRAM cycle timing.
4. Two MDPs under MMCX control do the following:
   a. Receive one quadword per clock cycle.
   b. Check longword parity on each quadword transfer.
   c. Generate seven check bits for each longword.
   d. Send one quadword per clock cycle to MMU.

5. MMU, under MMCX and MCDX control, does the following:
   a. Receives 80 data bits per clock cycle.
   b. Loads data into write buffer 0 through a 1:8 demultiplexer.
   c. Transfers data to write buffer 1.
   d. Transfers data from write buffer 1 to the DRAMs.
   e. Completes the DRAM timing sequence.
   f. Updates data and segment command buffer status for the JBox.

MMCX does not forward the length field to either MMU or the MDPX MCAs. The length is used in MMCX by the read data controller and write data controller state machines. The length field determines how long certain control signals from MMCX to MMU and MDPX should be asserted.

During writes, MDPX has no knowledge of the length of the data transfer. Whatever data enters MDPX from the data switch emerges from MDPX two clock ticks later with the appropriate check bits appended. It does not matter if the data is valid write data. If it is valid write data, MMCX sends the appropriate control signals to MMU to load the data into the data buffer. The length is conveyed to the MMU by the number of write buffer strobe (WRTBSTROBE) pulses. A write buffer strobe accompanies each quadword and loads the quadword into the data input latch (write buffer 0). After the memory module loads the last quadword into write buffer 0, MMCX sends one write buffer strobe pulse and transfers the data from the data input latch (write buffer 0) into the write data buffer (write buffer 1). When the data is in the write data buffer, MMCX concurrently transmits the write flip-flop enable (WRTFFEN_L) with the write select lines.

5.9.2.1 Loading the CAS Mask Register

MMCX receives the following two byte mask bits per longword from the data switch:

DSB2_MMCX_CTL_H[01:00] from the DBX MCU (upper longword)
DSA2_MMCX_CTL_H[01:00] from the DAX MCU (lower longword)

MDPX receives the full byte mask bits [36:33] with the data DSXX_MDPX_DAT_H[38:00]. The MDPX on DAX receives the lower longword, and the MDPX on DBX receives the upper longword. Each segment has a mask bit register in the MDPX MCA that holds the mask bits until the data is merged with the read data from memory.

The three following mask registers exist in the memory subsystem:

CAS mask build register
CAS mask operation register 0
CAS mask operation register 1

Figure 5–66 shows how the CAS mask register is loaded.

The CAS mask build register receives the mask bits with the data. This register is analogous to the data input latch (write buffer 0) in the DDPs. The CAS mask operation registers, 0 and 1, receive the 8-bit mask field from the build register in parallel. The contents of CAS mask operation registers 0 and 1 remain constant until the CASs have been deasserted. The CAS mask operation registers are analogous to the write data buffer 1 in the DDPs.

As the data is received a quadword at a time, the write selects are incremented and sent to the CAS mask build register. The CAS mask build register is clocked by a series of write strobes as the data is received and loaded into the data input latch. The write strobes arriving at the build register sample the write select and CAS mask control signals and cause the appropriate modification of the contents of the CAS mask build register.

Write flip-flop enable is asserted concurrently with write select for the last quadword in the data transfer and is stored in the build register. The next write strobe transfers the eight bits in the build register to one of the CAS mask operation registers. (MSKDIRSEG0 determines which of the CAS mask operation registers gets loaded.) The mask operation register gates CAS to enable the corresponding longwords to be written.

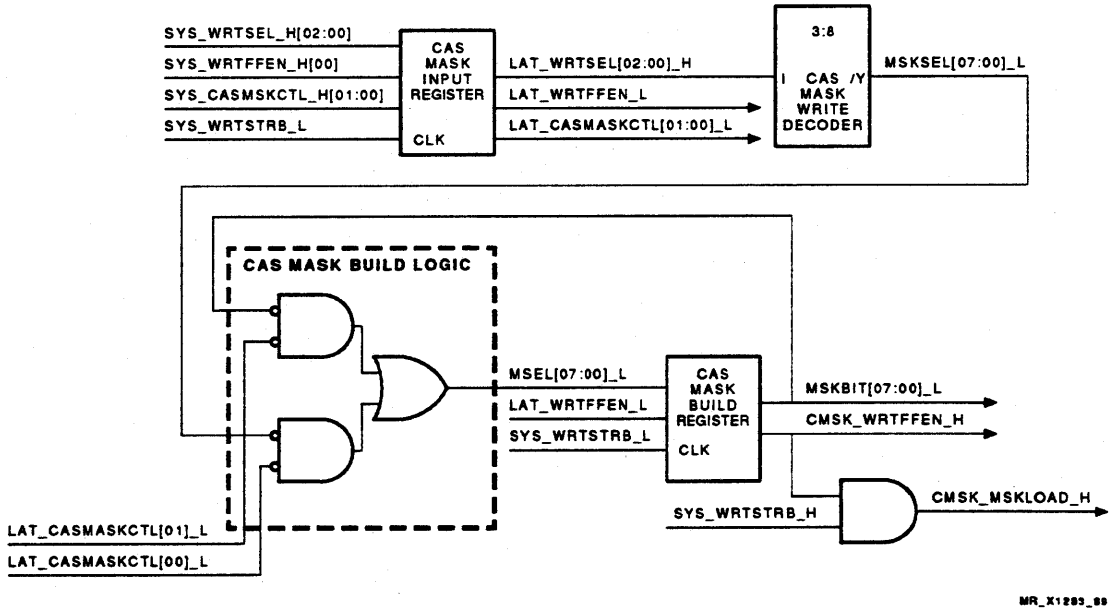


Figure 5–66 Loading the CAS Mask Register

5.9.3 Read-Modify-Write Operation

A read-modify-write data operation begins as a write command to the memory subsystem. MMCX decodes the index to determine whether the write operation is for I/O or CPU. Because an I/O write can be a byte, MMCX examines the mask bits and determines whether a byte write (partial longword) must be written. If so, MMCX executes a read-modify-write cycle.

During the initial stages of the read-modify-write operation, the memory module transfers the I/O write data through the data input latch (write buffer 0), write data latch (write buffer 1), and DRAM bypass (read bus) path, into the read data latch (read buffer 1) and then the data output latch (read buffer 0). The memory module transfers the data from read buffer 0, through the wrap multiplexer and into the I/O merge buffer in the MDPX MCAs. The I/O merge buffer holds the I/O write data until the bytes needed to generate ECC check bits can be read from the DRAMs.

When read data from the DRAMs is ready, it is loaded into read buffer 1 and is moved to the data output latch (read buffer 0). Figure 5–30 shows read and write paths in the memory module. One or two quadwords are transferred from read buffer 1, through the wrap multiplexer and into the MDPX for byte merging with the I/O write data.

The MDPX MCA combines the I/O write data bytes with the read data bytes, generating new check and mark bits to be appended to the data. The merged data is now ready to be loaded into write buffer 0 in the memory module.

The following steps summarize a read-modify-write operation:

1. The JBox logic transfers command information to the segment command buffer in MMC.
2. MMCX and MCD do the following:
   a. Decode LDCMD_H and CMD_H[01:00] as a write masked command.
   b. Decode BANKADDR_H[01:00], in which bit 1 specifies the segment command buffer to load and bit 0 specifies the bank.
   c. Strobe the address from the ADRX MCAs into MMU (row, then column).
   d. Receive and decode cycle status bits from the JBox.
   e. Receive and decode DSB2_MMCX_CTL_H[01:00] (upper word from DBX MCU) and DSA2_MMCX_CTL_H[01:00] (lower word from DAX MCU) mask bits from the data switches.
   f. Initiates DRAM read cycle timing.
3. The MDPX, under MMCX control, passes I/O write data through to the MMU.

4. The MMU, under MMCX and MCDX control, does the following:
   a. Receives I/O write data through a 1:8 demultiplexer and into write buffer 0.
   b. Moves data to write buffer 1.
   c. Moves data through the DRAM bypass to read buffer 0.
   d. Moves data to read buffer 1.
   e. Moves data through the wrap multiplexer to the MDPX I/O merge buffer.
   f. Loads read data from the DRAMs into read buffer 0.
   g. Moves read data from read buffer 0 to read buffer 1.
   h. Transfers the read data through the wrap multiplexer to MDPX for byte merging.
5. The MDPX, under MMCX control, does the following:
   a. Receives read data from the MMU.
   b. Checks for errors and corrects if necessary.
   c. Marks data bad if necessary.
   d. Merges the read data bytes with the I/O write data bytes in the I/O merge buffer to generate a valid longword.
   e. Generates seven check bits for each longword and sets the mark bit.
   f. Transfers data to the MMU (one or two quadwords).
6. The memory unit, under MMCX and MCDX control, does the following:
   a. Receives one data quadword per clock cycle.
   b. Loads each quadword into write buffer 0 through a 1:8 demultiplexer.
   c. Moves the write data to write buffer 1.
   d. At the correct DRAM timing, loads all valid longwords in write buffer 1 into the DRAMs.
7. MMCX and MCD do the following:
   a. Complete the DRAM timing sequence.
   b. Update data and segment command buffer status for the JBox.

5.9.4 Write Read Operation

A write-read data operation is a mixed operation. Data is first written to a location and then read from the same location. During the write cycle of the operation, not all 640 DRAMs are written. Invalidated longwords from CPU are not written. During the read phase of the operation, all of the 640 DRAMs are read.

The following steps summarize a write read operation:

1. The JBox transfers command information to the selected segment command buffer in the MMC.
2. The JBox transfers data to the memory subsystem as follows:
   a. Passes data to MDPX.
   b. Passes mask information to MMC.
3. MMCX and MCD do the following:
   a. Decode write read command information.
   b. Decode index.
   c. Decode bank select (0 or 1).
   d. Strobe an address into the selected memory segment.
   e. Initiate DRAM cycle timing.
4. The MDPX, under MMCX control, does the following:
   a. Receives one quadword per clock cycle.
   b. Checks longword parity on each quadword transfer.
   c. Generates seven check bits for each longword.
   d. Sends one quadword per clock cycle to the MMU.
5. The MMU, under MMCX and MCDX control, does the following:
   a. Receives 80 data bits per clock cycle.
   b. Loads data into the write data 0 buffer through a 1:8 demultiplexer. (Figure 5–30 shows the read and write path in the memory module.) For each valid longword transferred, MMCX sets a bit in the CAS mask register.
   c. Moves data to write buffer 1.
   d. Loads DRAM data from write buffer 1 into the DRAMs.
   e. Completes the write part of the DRAM cycle.
   f. Begins the read part of the DRAM cycle.
   g. Makes DRAM read data available (from the DRAMs).
   h. Loads read data into read buffer 0.

6. The MMCX and MCD do the following:
   a. Complete the DRAM timing sequence.
   b. Transmit a return data read command to the JBox.
   c. Receive a send data command from the JBox.
   d. Transfer each quadword through the MDPX MCAs to the JBox.
7. The MDPX, under MMCX control, does the following:
   a. Receives one quadword per clock cycle from the MMU.
   b. Checks for errors and corrects if necessary (single bit).
   c. Generates longword parity.
   d. Sends one quadword per clock cycle to the JBox.
8. The MMCX updates segment command buffer status.

5.9.5 Write Pass Operation

A write-pass data operation uses the same write timing as a normal write cycle. Immediately after the write buffers are loaded, the memory module uses the DRAM bypass path and sends the data directly to the read buffers. Figure 5–30 shows the bypass path in the memory module. During the write cycle, all data is valid and all 640 DRAMs are written. The memory module unloads the data in the read buffers the same way as in a read operation.

The following steps summarize a write pass operation:

1. The JBox transfers command information to the selected segment command buffer in MMCX.
2. The JBox transfers the following to the memory subsystem:
   a. Data switch data to the MDPX MCAs
   b. Data switch mask data to MMCX
3. The MMCX and MCD do the following:
   a. Decode command information.
   b. Decode index.
   c. Decode bank select information.
   d. Strobe the address into the selected memory segment.
   e. Initiate DRAM cycle timing.
4. The MDPX, under MMCX control, does the following:
   a. Receives one quadword per clock cycle.
   b. Checks longword parity on each quadword transfer.
   c. Generates seven check bits for each longword.
   d. Sends one quadword per clock cycle to the MMU.

5. The MMU, under MMCX and MCDX control, does the following:
   a. Receives data 80 bits per clock cycle.
   b. Loads data into write buffer 0.
   c. For each valid longword loaded, MMCX sets a bit in the CAS mask register.
   d. Moves data to write buffer 1.
   e. Transfers data through the bypass path to read buffer 0.
   f. At the correct time, loads DRAM data from write buffer 1 into the DRAMs.
6. The MMCX and MCD do the following:
   a. Complete the DRAM timing sequence.
   b. Transmit a return data read command to the JBox.
   c. Receive a send data command from JBox.
   d. Transfer each quadword through the MDPX MCAs to the JBox.
7. The MDPX, under MMCX control, does the following:
   a. Receives one quadword per clock cycle from the MMU.
   b. Generates word parity.
   c. Sends one quadword per clock cycle to the JBox.
8. The MMCX updates segment command buffer status.

5.9.6 Refresh Operation

Three sources of logic provide DRAM refresh support: SCU, DRAM control and address (DCA) gate array, and standby. The mode of operation determines the source of the refresh logic. The standby logic generates the refresh interval and sends the interval to the SCU and DCA. During powerfail or scan operation, standby refreshes are used.

The memory modules initiate the refresh operations. A refresh signal originates on each of the memory modules. MMCX receives one of these signals and uses it as a command request for refresh. Upon completion of active cycles, MMCX executes a refresh cycle to all 2560 DRAMs in MMU. The refresh logic uses the CAS and RAS control signals and does the data and address paths.

5.9.7 EEPROM Operations

Each memory module has an EEPROM that stores information about the memory module. Figure 5–67 shows the EEPROM data.

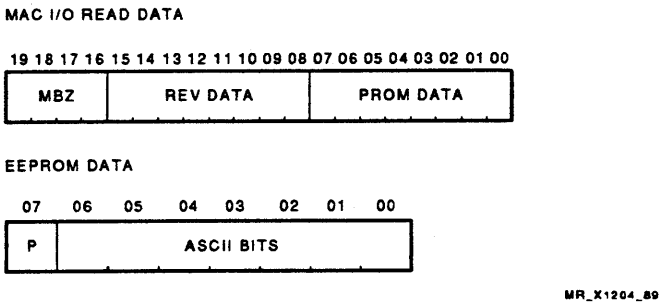


Figure 5–67 EEPROM Data Format

The storage in the EEPROM is divided into two areas of equal size. Manufacturing has read and write access to both storage areas. SPU has read access to all locations and can write to only the lower half addresses. When initiating the DRAM test, SPU reads the EEPROM, collects DRAM size information, and passes the information to the MCDX MCA.

The SPU performs EEPROM read and write operations at initialization or power-down. The following steps summarize reading and writing to the EEPROM:

1. Sequences the memory into standby timing.
2. Stops system clocks to SCU.
3. Sets EEPROM operation bit in MMCX through scan. With the bit set, MMCX interprets all I/O requests as EEPROM reads or writes.
4. Turns on system clocks.
5. Reads or writes the EEPROM through the I/O channel.

To turn off access to the EEPROM, the SPU performs the following steps:

1. Stops the system clocks to SCU.
2. Clears the EEPROM operation bit in MMCX using the scan.
3. Turns on the system clocks.
4. Sequences the memory into normal timing.

5.10 Memory Module Testing

The memory subsystem can self-test its memory modules. This self-test uses BIST logic located in the memory module DCA and the ACU's MDPX, MCDX, and MMCX MCAs. The SPU initiates the self-test, and the ACU provides the means to test the memory modules. The testing of the DRAMs emulate the random activity normally seen during system operation. Random data patterns are written to the DRAMs at randomly generated addresses.

The self-test logic is as follows:

- **BIST controller** — The controller is located in the MCDX MCA. This logic issues commands to MMCX and receives status. MMCX places the BIST commands on the CAS and WE lines.
- **BIST command** — The command interface is located in the MMCX MCA. The commands are identical to the commands MMCX receives from the JBox. MMCX acts on them in an identical manner.
- **BIST control logic and data pattern generator** — The BIST control logic and data pattern generator are located in the MDPX MCA. The data pattern generator is described in more detail in Section 5.6.1.
- **BIST command decode and address pattern generator** — The command decode and address pattern generator is located in the DCA on each memory module. Figure 5–68 shows the address pattern generator in the memory module. The address pattern generator is described in more detail in Section 5.6.1.

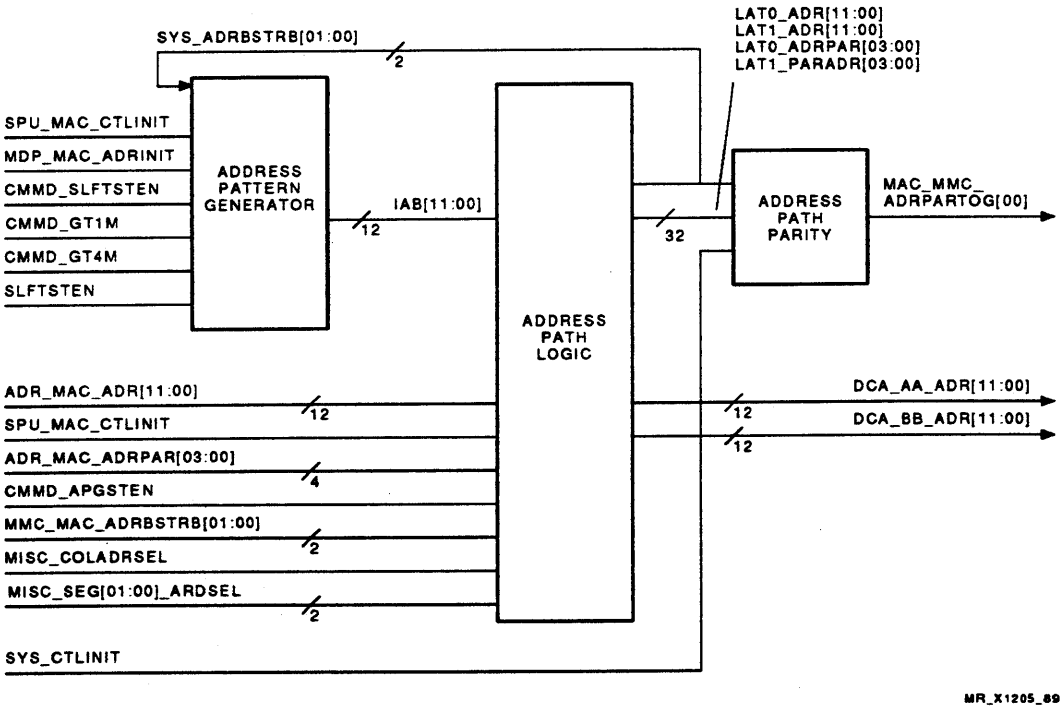


Figure 5–68 Address Pattern Generator in the Memory Module

5.10.1 BIST Controller

The BIST controller is located in the MCDX MCA. The controller sends the following:

- Self-test mode commands (using STM_CMD_H[02:00]) to MMCX. The MMCX executes the BIST command when MCDX sends a load command. Table 5–46 lists the BIST commands.
- Step mode commands to DCA. The DCA switches the address pattern generator into APG mode. Each step mode command listed in Table 5–40 determines the length (count) for the APG contents.

Table 5–46 BIST Self-Test Commands

Value	Description
00	Read
01	Write read
10	Write
11	Write pass

Table 5–47 BIST Step Mode Commands

Name	Description
SLFTST1EN	Self-test 1-Mbit DRAMs enable
SLFTST4EN	Self-test 1-Mbit DRAMs enable
SLFTST16EN	Self-test 1-Mbit DRAMs enable

5.10.2 BIST Data

MDPX provides the data patterns through a 32-bit data pattern generator (also called LFSR). The data patterns are random and no two data patterns written to a memory bank are the same. The lower order bits of the data pattern generator are identical to the contents of the address pattern generator in DCA. This allows the address pattern to be scannable in MDPX. The MDPX provides data manipulation during the DRAM test.

5.10.3 BIST Address

Addresses are provided by one of the following:

ADRX MCAs in the tag MCU
Address pattern generator in the DCA

The DRAM test uses the address pattern generator in DCA. All other tests rely on an address scanned into the ADRX MCAs during test initialization. The MMCX and MDPX BIST logic controls the address pattern generator in DCA and sends the following signals:

- **MMCX_MAC_ADRBSTROBE0_L** — DCA clocks the address pattern generator to the next test address. The system mode function of this signal is to strobe the address from the JBox into the segment 0 address latch.
- **MMC_MAC_ADRBSTROBE1_L** — DCA toggles between row and column of the generated test address. The system mode function of this signal is to strobe the address from the JBox into the segment 1 address latch.
- **MDP_MAC_ADRINIT_L** — DCA initializes the address pattern generator to its starting address. This signal has no function during system mode.

For more detail, see Section 5.6.1.

5.10.4 BIST Mode Switching Order

During the BIST operation, the memory switches between standby-to-step, and step-to-standby modes. Switching between modes is different in BIST than in normal mode because the MCD BIST controller takes over mode switching functions normally executed by the SPU.

5.10.4.1 Standby-to-Step Mode

In step mode, the service processor should not switch to standby until signaled by the MCD BIST controller. The SPU ignores the signal MAC_SPU_STEPCTLOK_H from the MMU when the memory is in step mode. When the MCD BIST controller puts the memory into normal mode, MAC_SPU_STEPCTLOK_H is deasserted.

5.10.4.2 Step-to-Standby Mode

The MCD BIST controller signals a mode switch to standby by sending a stop signal to MMCX. The MMCX pulls the attention line on the CDCX MCA. The SPU switches the memory into standby mode by asserting SPU_MAC_STBYCTLEN_L.

5.10.5 BIST Registers

The self-tests use the following BIST registers:

ADRX address latches
MCD BIST registers
MMCX BIST registers
MDPX BIST registers

5.10.5.1 ADRX Address Latches

During DCA tests, a known-good address is required. The four ADRX MCAs located in the tag MCU supplies the address. Each ADRX supplies three address bits, plus one parity bit. The four ADRX supply a 12-bit address (Table 5–48). Table 5–49 lists the address bits that are dependent on memory size.

Table 5–48 ADRX 12-Bit Address

ADRX Bit	Description
ADR0 BIST [02:00]	Row and column MMU address bits [02:00]
ADR0 BIST [03]	Parity on MMU address bits [02:00]
ADR1 BIST [02:00]	Row and column MMU address bits [05:03]
ADR1 BIST [03]	Parity on MMU address bits [05:03]
ADR2 BIST [02:00]	Row and column MMU address bits [08:06]
ADR2 BIST [03]	Parity on MMU address bits [08:06]
ADR3 BIST [02:00]	Row and column MMU address bits [11:09]
ADR3 BIST [03]	Parity on MMU address bits [11:09]

Table 5–49 MMU Size

MMU Size	Row/Column Address Bits	Not Used
256 Mbytes	09:00	11:10
1 Gbytes	10:00	11
4 Gbytes	11:00	–

5.10.5.2 MCD BIST Registers

The MCD BIST register can be written to and read by the SPU through scan. It is used only during BIST operations, which is signaled by the BIST enable bit being set. Figure 5–69 shows the MCD BIST register. Table 5–50 lists the fields and their descriptions.

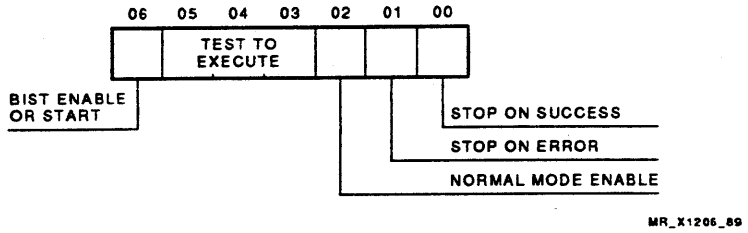


Figure 5–69 MCD BIST Control Register

Table 5–50 MCD BIST Register Field Descriptions

Bit	Name	Description
00	Stop on success	This signal is active when both MMCX and MDPX do not detect an error. MCD halts the test when no error is detected.
01	Stop on error	MMCX provides a BIST error signal to MCD. This error signal is active when MMCX or MDPX detects an error. MCD halts the test whenever a BIST or MCD error is detected. Sampling for errors is done at the end of each memory cycle. Some flexibility is also available. Parity error detectors can be individually disabled using scan initialization.
02	Normal mode enable (NME)	This function allows the DCA test to be used more than once. When this bit is set the DRAM cycles are executed in normal mode. When not set, the DRAM cycles are executed in step mode.
05:03	Test to execute	This field identifies the tests that MCD is to execute. Table 5–51 lists the tests that can be executed.
06	BIST enable or start	This bit engages the BIST controller in the operation defined by the MCD BIST register.

Table 5–51 MCD BIST [05:03]

Value	Test
000	DDP
001	DCA
010	Data path
011	Reserved
100	Reserved
101	16-Mbit DRAM
110	4-Mbit DRAM
111	1-Mbit DRAM

Figure 5–70 shows the MCD EOP BIST register. Table 5–52 lists the fields and their descriptions.

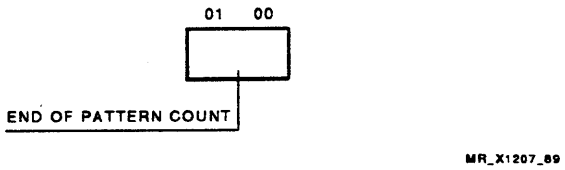


Figure 5–70 MCD EOP BIST Register

Table 5–52 MCD EOP BIST Register Field Descriptions

Bit	Name	Description
MCD EOP [01:00]	End of pattern (EOP) count	End of pattern count is read-only. These two bits indicate the state of the data produced by MDPX.

5.10.5.3 MMC BIST Register

The MMC BIST register can be written to and read by only the SPU using a scan during BIST operation, which is signaled by the BIST enable bit being set. Figure 5–71 shows the MMC BIST register. Table 5–53 lists the fields and their descriptions.

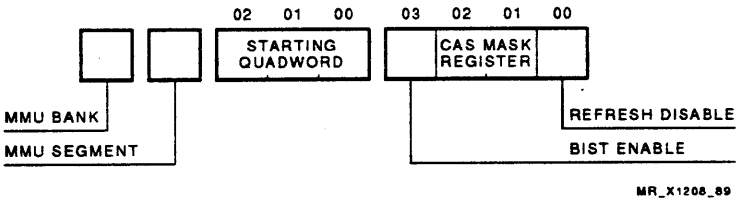


Figure 5–71 MMC BIST Register

Table 5–53 MMC BIST Register Field Descriptions

Value	Name	Description
00	Refresh disable	This bit disables refresh.
02:01	CAS mask register	This field defines how MMC sets up the CAS mask register in DCA during the test. This field controls all write operations that occur. Table 5–54 lists the values and their descriptions.
03	BIST enable	This bit substitutes SPU_MMC_REQSTEPCTL_H with MCD_MMC_STMRQSMCTL_H from MCD and the JBox command and status field with the MCD command and status field.
MMC STQW [02:00]	Starting quadword	This field sets the starting quadword. At the start of each BIST test, this field equals zero.
MMC SEG	MMU segment	This bit defines the memory segment.
MMU BANK	MMU bank	This bit defines the memory bank.

Table 5–54 CAS Mask Register Field Descriptions

Value	Description
00	Sets CAS mask = 00000000
01	Sets CAS mask = 10000000
10	Reserved
11	Sets CAS mask = 11111111

5.10.5.4 MDP BIST Registers

The MDPX MCA has three BIST registers: the MDP BIST register, linear feedback shift register, and data input register.

- **MDP BIST register** — The MDP BIST register can be written to and read by only the SPU using a scan. It is used only during BIST operations, which are signaled by the BIST enable bit being set. Figure 5–72 shows the MDP BIST register. Table 5–55 lists the fields and their descriptions.
- **MDP linear feedback shift register** — The linear feedback shift register, MDP LFSR [31:00], holds the BIST data pattern. The lower 24 bits equal the address pattern in the DCA address pattern generator.
- **MDP input data register** — The input data register, MDP INDAT [39:00], contains the error data received by the MDPX MCA that caused a SBE or DBE.

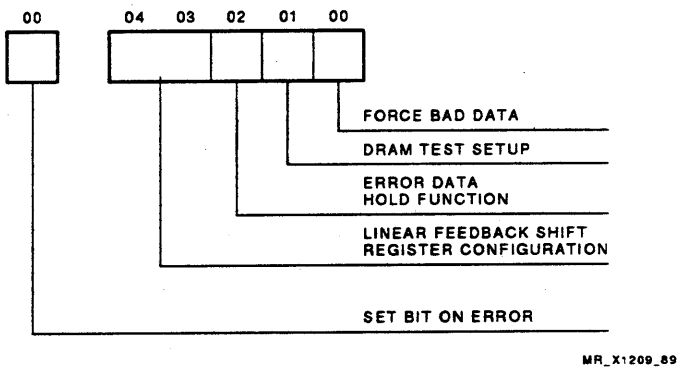


Figure 5–72 MDP BIST Register

Table 5–55 MDP BIST Register Field Descriptions

Bit	Name	Description
00	Force bad data	This bit is used by MDPX during the DCA CAS mask test. This bit enables writing error data into MMU, depending on the state of the CAS mask bits.
01	DRAM test setup	This bit is used by MDPX during the DRAM test to switch read data into the data path after the first phase. During the first phase, LFSR supplies the data.
02	Error data hold function	This field locks the input data register in MDP at the occurrence of the first ECC error. The ECC register holds the data transferred from MMU that caused the error. This bit is initialized by a scan. By setting this bit, the data in error is held for later scan. This function is used during the DDP test.
04:03	Linear feedback shift register configuration	The MDP LFSR is 32 bits wide. During pattern generation, the lower 24 bits equal the address pattern produced by LFSR in DCA. The DCA LFSR uses 20, 22, or 24 bits, depending on the size of the DRAMs. In this way, the address pattern used during DRAM testing can be scanned out. Table 5–56 lists the values for the LFSR configuration.
MDP SBOE [00]	Set bit on error	This bit is used only during the isolation routine and functions to modify bit 0 or the read data. When set, MDP forces data bit 0 to a 1 if an ECC error is detected.

Table 5–56 LFSR Configuration

Value	Configuration
00	1-Mbit DRAM patterns
01	4-Mbit DRAM patterns
10	16-Mbit DRAM patterns

5.10.6 BIST Tests

The BISTs ensure the correct operation of the MMU. The BIST tests include the following:

DDP
DRAM
Data path
DCA control parity
DCA CAS mask
DCA DRAM control

Each test begins with a wake-up sequence and ends with a stop sequence, as follows:

- **Wake-up sequence** — This sequence puts the memory system into step mode. The SPU does the following:

  1. Initializes the system.
  2. Turns on system clocks.
  3. Takes memory out of standby.

  For BIST tests, the memory system starts out in step mode.

- **Stop sequence** — This sequence follows the BIST test. The following steps summarize the stop sequence:

  1. BIST test completes.
  2. Memory transitions into the step mode. MCD deasserts request normal mode to MMC (MMC in step mode).
  3. MCD asserts attention on CDCX chip and calls for the SPU.
  4. SPU places memory into standby and asserts standby control enable to MMU.
  5. SPU scans the data.

5.10.6.1 DDP Test

This test checks for opens, stuck at, and shorts on the data path. It does this by toggling every data bit high and low. It does require the DRAMs to have been tested. This test routes the data around the DRAMs using the DRAM bypass path.

The MDPX LFSR supplies the data patterns for this test. Each pattern moves through the normal write path (the write path for JBox write data). ECC check bits are appended to the write data, and the quadword address parity is put in place of the mark bit.

The data moves through the normal read path in the MDPX. If the MDPX detects an error, the data is held in the MDPX input read buffer and testing stops. At this point, scan is required to retrieve the data and error information.

If no errors are detected, MDPX generates an EOP signal. This signal marks the last pattern and forces data inversion as the patterns repeat. The process is repeated except that the original data pattern is inverted before being processed through ECC. The process is repeated one more time so that the ECC check bits are the opposite of what they were for the previous set of patterns.

5.10.6.2 DRAM Test

DRAM testing is done one bank at a time. Each bank must be initialized before testing. As there are two segments with two banks per segment, this test must be initialized through scan and executed four times. DRAM testing uses the pseudo random pattern generators (LFSR) located in MDP and DCA.

The DRAM test performs the following activities:

- Writes a data pattern to all addresses.
- Reads the data pattern, checks the data, and writes a complimented data pattern.
- Reads the data pattern, checks the data, and writes. a modified data pattern that generates complimented check bits.
- Reads the data pattern, checks the data, and writes zeros.

5.10.6.3 Data Path Test

This test runs four times, once for each bank. The data path test ensures that the DRAMs can correctly perform a write read operation. In addition, the test checks the data lines by using many different random data patterns.

The test uses the MDP LFSR. The patterns from LFSR are routed to the DRAM array and out (through the write read command) to the MDP for error checking. When the LFSR reaches the last pattern, testing stops.

5.10.6.4 DCA Control Parity

This test runs twice, once for each segment. This DCA control parity test checks the control parity logic in DCA and checks the DCA CAS mask parity generation logic. Each DCA generates a parity signal made up of the received RAS, CAS, WE, FFEN (write flip-flop enables), and the bits in the segment CAS mask register received from MMC. MCDX receives and checks the control parity from each DCA.

5.10.6.5 DCA CAS Mask Test

This test runs twice, once for each segment. The DCA CAS mask test checks the CAS MASK register. The CAS mask for the selected segment is set to all 0s. This test disables writing to DRAM locations, attempts to write forced bad data into DRAM locations, and reads back the DRAM locations, looking for errors. An error indicates that the CAS mask function failed.

5.10.6.6 DCA DRAM Control Test

This test runs twice in each bank. The first pass uses an LFSR pattern having all 1s. The second pass uses an LFSR pattern having all 0s. The test writes all 1s and reads the data back, writes all 0s and reads the data back, and ensures that DCA provides the correct control signals to the DRAMs.

6
I/O Control Unit

The SCU can have two I/O control units (ICUs), ICU0 and ICU1. ICU0, located on the DA0 and DB0 MCUs, consists of the JDAX, JDBX, JDCX, and IRCX MCAs and supports the XJA0 and XJA1 modules and the service processor unit (SPU). ICU1, located on the DA1 and DB1 MCUs, consists of the JDAX, JDBX, JDCX, and IRCX MCAs and supports the XJA2 and XJA3 modules.

6.1 Overview

The ICUs, shown in Figure 6–1, exchange commands, addresses, and data with the following:

- **SPU** — Using a cable, ICU0 sends register read and write, ECC, and I/O read and write commands to, and receives them from, SPU. The SPU also sends DMA read and write requests to ICU. In addition, the ICU receives and arbitrates powerfail, keep-alive, halt CPU, console terminal, and console storage device interrupts from the SPU.
- **Four XJA modules** — Using two JXDI cables, the two ICUs send register read and write commands to, and receive them from, the XJA0, XJA1, XJA2, and XJA3 modules. The XJAs also send DMA read and write requests to the ICUs. In addition, the ICUs also receive and arbitrate interrupts for recoverable and nonrecoverable XMI errors, and fatal XJA and XMI errors.
- **JBox** — Using the logical interface on the SCU planar module, the ICU loads its receive buffers with commands, addresses, and data from the XJAs or SPU. The ICU then sends the commands to the CCU MCU (port controller), the addresses to the tag MCU (I/O address receive latches), and the data to the data switches on the DA0/DB0 and DA1/DB1 MCUs.

  The ICU loads its transmit buffers with commands from the CCU command latch, addresses from the I/O address receive latches, and data from the data switch. The ICU then sends commands, addresses, and data to either the XJA or SPU. The IRCX MCA sends and receives data and addresses to and from the CCU MCU and DSXX MCAs.
- **Four CPUs** — Using a single pair of differential cables to connect the SCU planar module to each CPU planar module, the ICU provides the interrupt interface and arbitrates I/O (XJA) interrupts, SPU (console) interrupts, and interprocessor interrupts. The ICU determines which interrupt has the highest interrupt priority level (IPL) and sends an interrupt code to the EBox. The EBox executes an interrupt service routine and handles the interrupt (if the interrupt has a higher IPL than the current one).

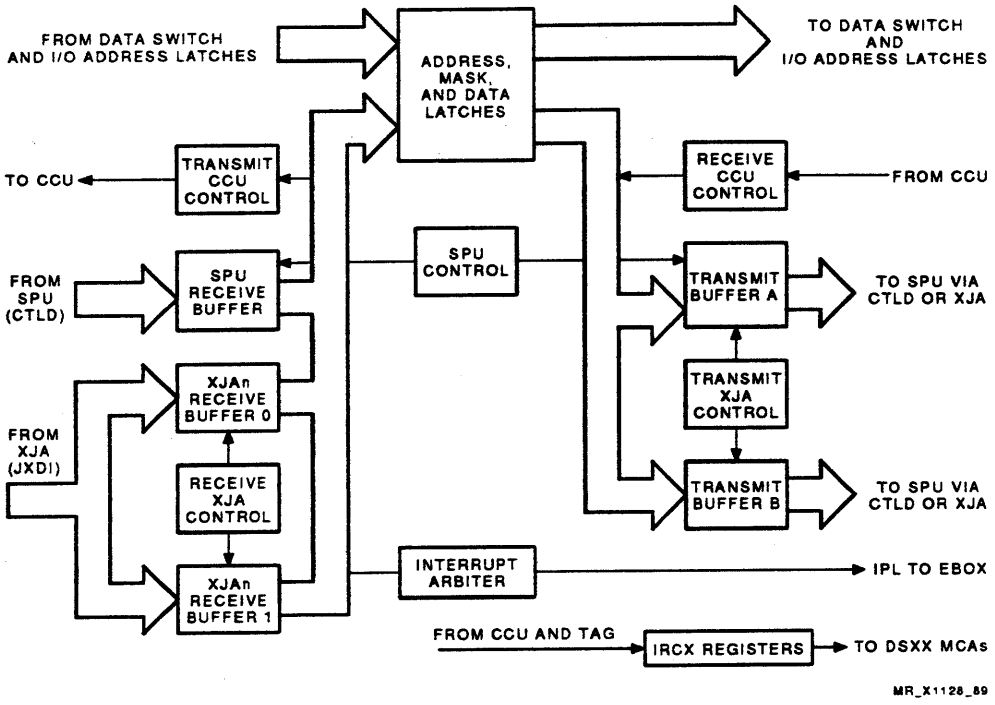


Figure 6–1 ICU Block Diagram

6.2 I/O Subsystem — Physical Description

The I/O subsystem consists of the following components:

- ICU
- XJA module
- JXDI bus
- XMI bus
- XMI devices
- SPU

Figure 6–2 shows the I/O subsystem.

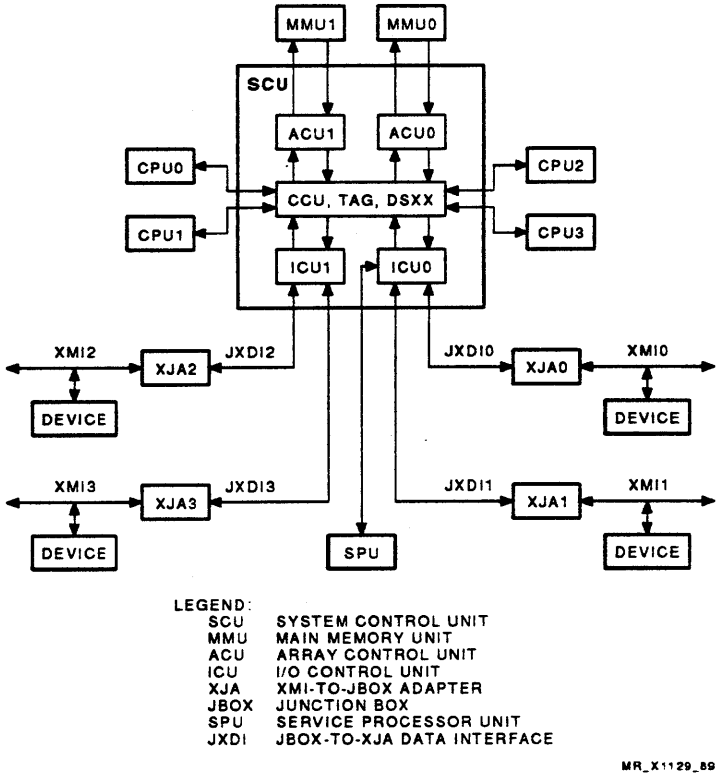


Figure 6–2 I/O Subsystem

6.2.1 I/O Control Unit

The ICU resides on the SCU planar module and consists of the following MCAs:

- **JDCX** — This MCA contains the SPU, CCU, and XJA control. JDC0 (ICU0) is located on DA0 MCU and JDC1 (ICU1) is located on DA1 MCU.
- **JDAX** — This MCA contains the SPU and XJA receive buffers. JDA0 and JDA1 are located on DA0 and DB0 MCUs. JDA2 and JDA3 are located on the DA1 and DB1 MCUs.
- **JDBX** — This MCA contains the transmit buffers that transmit command, address, and data for either XJA0, XJA1, or the SPU. JDB0 and JDB1 (ICU0) are located on DA0 and DB0 MCUs. JDB2 and JDB3 (ICU1) are located on the DA1 and DB1 MCUs.
- **IRCX** — This MCA contains the central system interrupt arbiter. IRC0 (ICU0) is located on the DA0 MCU and IRC1 (ICU1) is located on the DA1 MCU.

Figure 6–3 shows the SCU planar module and its MCAs, MCUs, STRAMs, and cable connections.

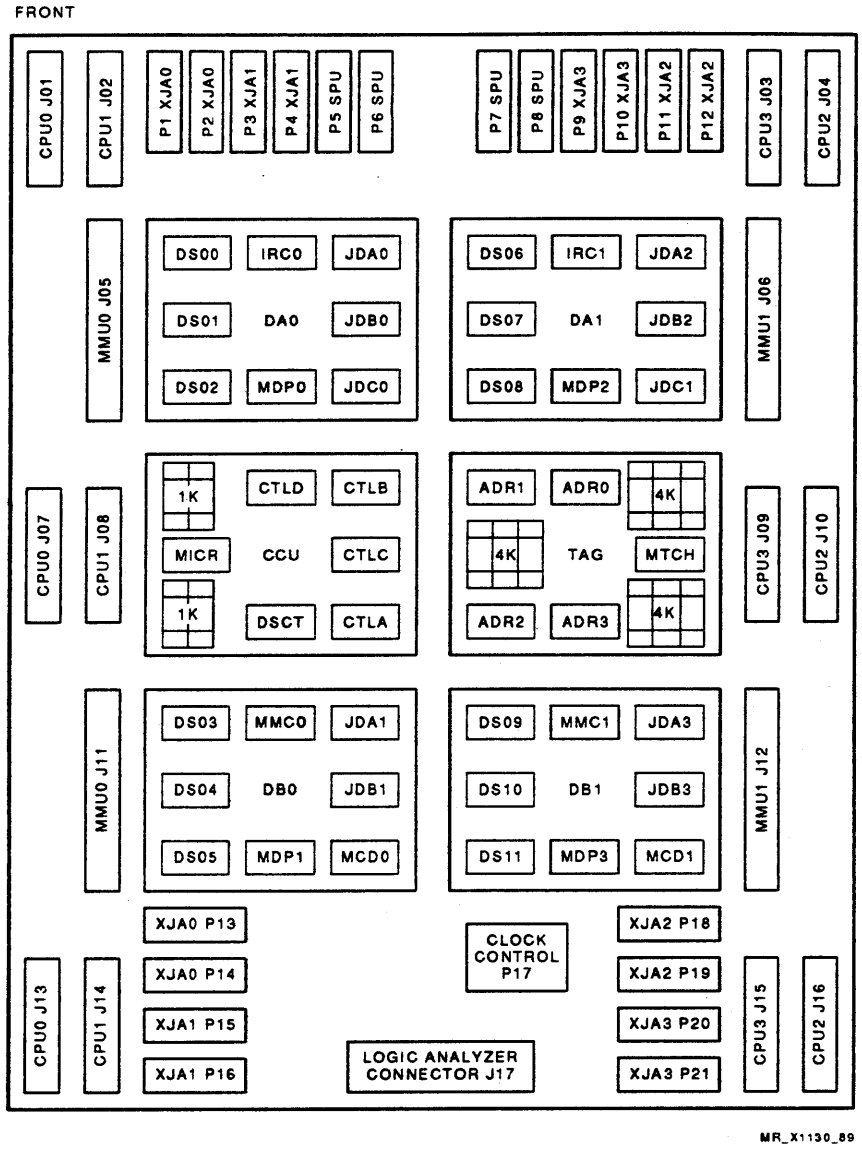


Figure 6–3 SCU Planar Module

6.2.2 XJA Module

The XMI-to-JBox adapter (XJA) module and ICU provide an information path between the JBox and the I/O devices connected to the XMI bus.

Figure 6–4 shows the XJA MCAs. The XJA is implemented on an extended T-series module and conforms to XMI specifications. The XJA module plugs into the XMI card cage. The JBox-to-XJA data interconnect (JXDI), located in the rear of the cabinet, consists of four cables and runs from the slot in which the XJA resides (slot 8 next to the CCARD module) to the SCU planar module. Figure 6–3 shows the XJA0, XJA1, XJA2, and XJA3 connectors on the SCU planar module.

The XJA communicates with the ICU using the following three types of transactions:

- **DMA transactions** — DMA transactions are reads, writes, read locks, or write unlocks. DMA transactions can be a quadword or octaword in length.
- **CPU transactions** — CPU transactions access the I/O portion of the VAX physical address space. CPU transactions are a longword in length, and the XJA can accept only a single CPU transaction at a time.
- **Interrupt transactions** — Interrupt transactions notify the operating system of recoverable and nonrecoverable XMI errors or of fatal XMI and XJA errors.

For more details, see Section 6.9.

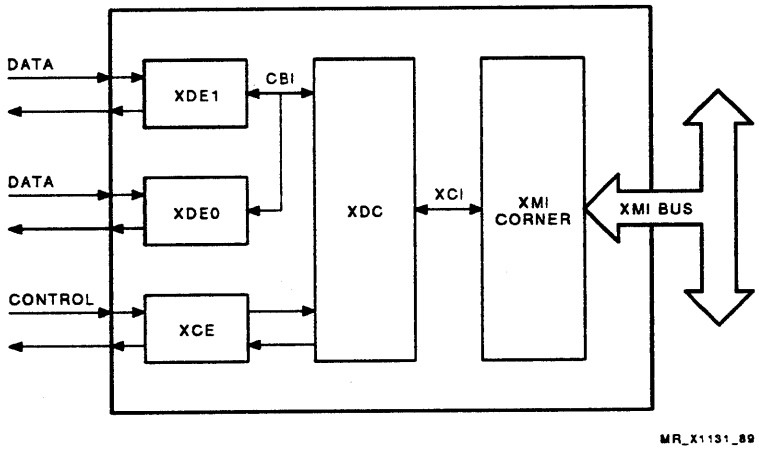


Figure 6–4 XJA Module MCAs

6.2.3 JXDI

The JXDI is a ten-foot cable that connects the ICU and XJA. Most signals are unidirectional and differential. Figure 6–5 shows the JXDI data, handshake, and clock interface. The ICU and XJA contain the JXDI transmitter and receiver logic.

The JXDI is, in general, symmetrical, in that the the ICU and XJA send and receive the same data and handshake signals.

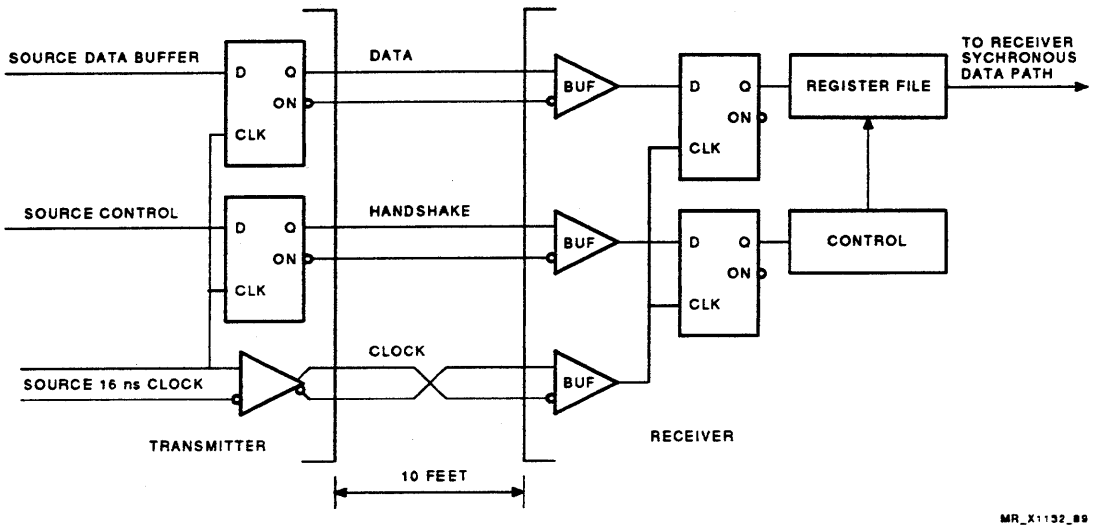


Figure 6–5 JXDI

6.2.4 XMI Bus

The XMI bus is a synchronous, 64-bit wide, multidrop, pended bus with a cycle time of 64 ns. (In a pended bus, the nodes do not hold up the bus while waiting for a response.) The XMI bus is a limited length bus with centralized arbitration, which can support multiple processors, multiple memory subsystems, up to eight I/O adapters, and a 40-bit physical address. Operating at 64 ns, the XMI bus has a bandwidth of 125 Mbytes.

The XMI bus consists of a card cage, transceivers, an arbitration chip, protocol, and signal integrity error checking. The XMI card cage supports the electrical environment of the bus and backplane, housing the logic (on the XJA modules) that implements the bus protocol.

The XMI bus allows several transactions to be in progress at once and provides a highly efficient use of bus bandwidth. The XMI bus multiplexes data and address lines, which allows arbitration and data transfers to occur simultaneously.

The XMI bus supports quadword and octaword reads and writes to memory. In addition, the bus supports longword read and write operations to I/O space. These longword operations can implement the byte and word modes required by certain I/O devices.

6.2.5 SPU

The service processor unit (SPU) is based on a BI MicroVAX system installed in a single VAXBI card cage. The processor consists of a service processor module (SPM), 2 Mbytes of ECC memory, KFBTA (AIO — disk controller), DEBNT (AIE-NI/TK50 controller), and a scan control module (SCM).

The SPM contains the SPU-to-JBox adapter (SJA) MCA. Two cables, one for the logical interface and the other for the scan interface, plug onto the front edge of the SPM and SCM modules at one end, and the SCU planar module at the other end. Figure 6–3 shows the SPU connectors on the SCU planar module.

6.2.6 Data Transfers (Packets)

The XJA, XMI bus, and SPU communicate using packets. Figure 6–6 shows the interconnects and buses that use packets. XJA modules communicate with the ICU by means of packets. JXDI packets contain command, length, IPL, address, mask, and data information.

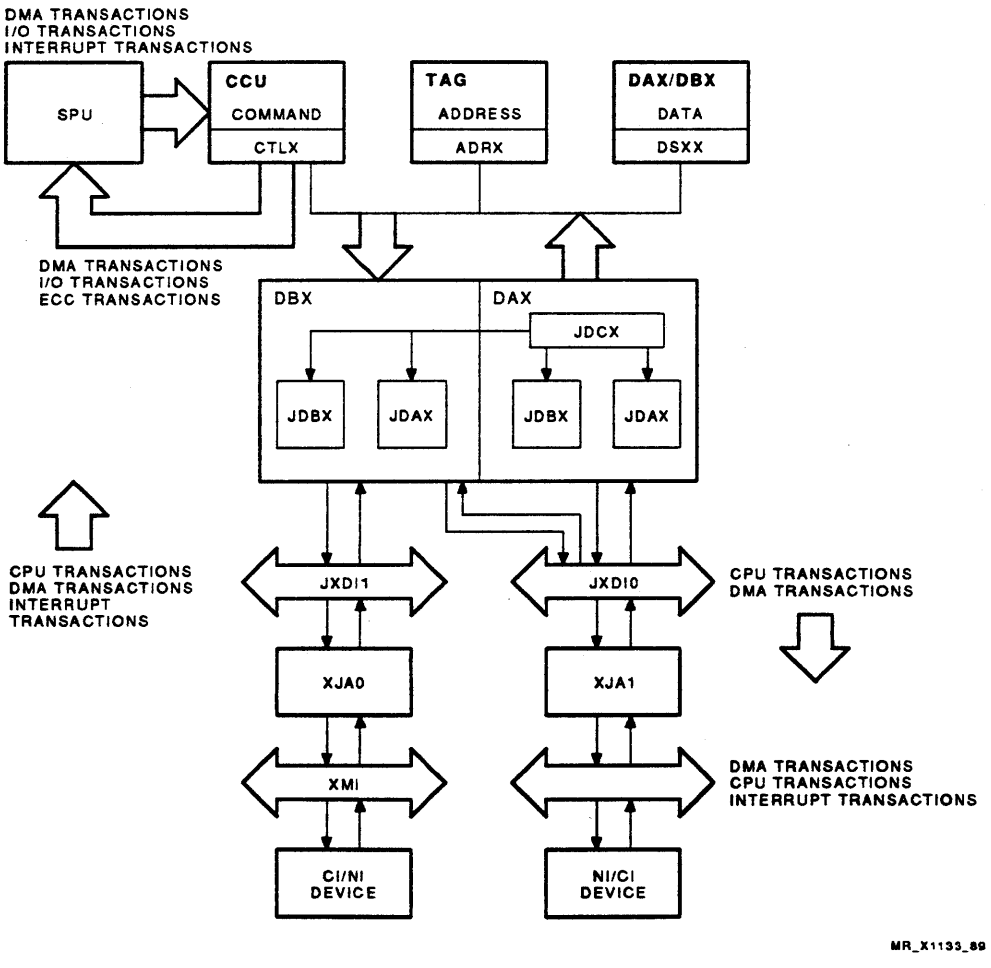


Figure 6–6 Packets

XMI devices communicate with XJA and transfer large amounts of data by the use of packets. XMI packets contain synchronizing information, sending and destination node addresses, packet type and length, data, and error checking information.

The SPU communicates with the ICU by means of packets. See Section 6.10 for more detail. SPU packets contain command, length, CPU ID, address, mask, and data information. See Section 6.6 for more detail.

6.3 ICU — Functional Description

This section provides the functional description of the ICU.

6.3.1 JDCX MCA

JDC0, located on the DA0 MCU, controls transmissions to and from the SPU and the IRCX MCA, CCU MCU, and XJA0 and XJA1 modules. JDC1, located on the DA1 MCU, controls transmissions to and from the CCU MCU, and XJA2 and XJA3 modules.

Figure 6–7 shows how major control areas in the JDCX MCA interface with the SCU MCUs. The JDCX MCA controls loading and unloading of buffers in the JDAX and JDBX MCAs, and also provides the command interface with the CCU. JDCX sends commands to CCU for arbitration and receives commands to be sent to either SPU, XJA0/XJA1, or XJA2/XJA3.

The JDC0 MCA contains the following areas of control:

- Receive XJA0 — Receive from XJA0
- Receive XJA1 — Receive from XJA1
- Transmit XJA0 — Transmit to XJA0
- Transmit XJA1 — Transmit to XJA1
- Receive CCU — Receive from CCU MCU
- Transmit CCU — Transmit to CCU MCU
- SPU Control — Receive from and transmit to SPU

The JDC1 MCA contains the following areas of control:

- Receive XJA2 — Receive from XJA2
- Receive XJA3 — Receive from XJA3
- Transmit XJA2 — Transmit to XJA2
- Transmit XJA3 — Transmit to XJA3
- Receive CCU — Receive from CCU MCU
- Transmit CCU — Transmit to CCU MCU

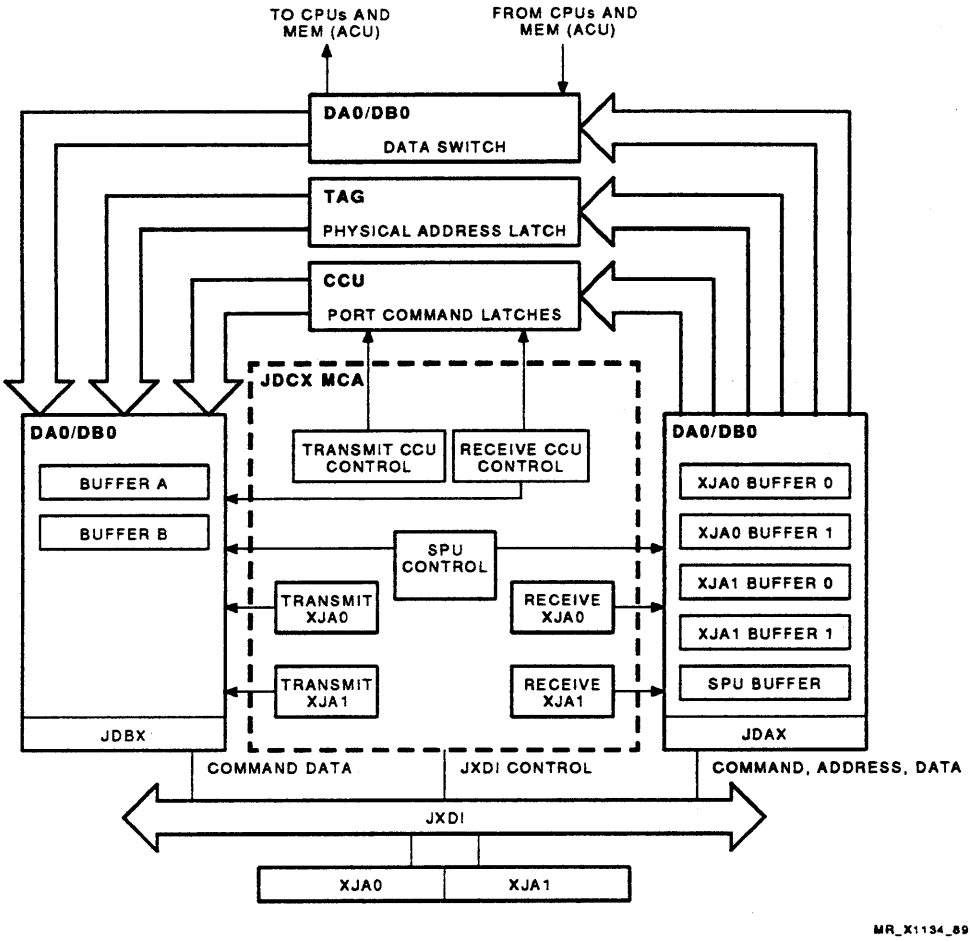


Figure 6–7 JDCX MCA Control Areas

Figure 6–8 shows how the JDCX MCA interfaces with the CTLA, CTLB, and CTLC MCAs on the CCU MCU. Chapter 2, JBox Port Arbitration, details these interfaces.

JDCX receives the JXDI handshaking signals, which include command available, buffer empty, retry, acknowledge, and the XJA fatal signals.

JDCX sends JDCX_CCU_DAX_FATAL_L to the CCU. The following error conditions can initiate a DAX fatal error:

- XJA retry (if retry mode is 2)
- Transmit XJA0, XJA1 fatal error
- XJA0, XJA1 handshaking parity error
- IRCX fatal error
- CCU-to-JDCX parity error
- SPU fatal error

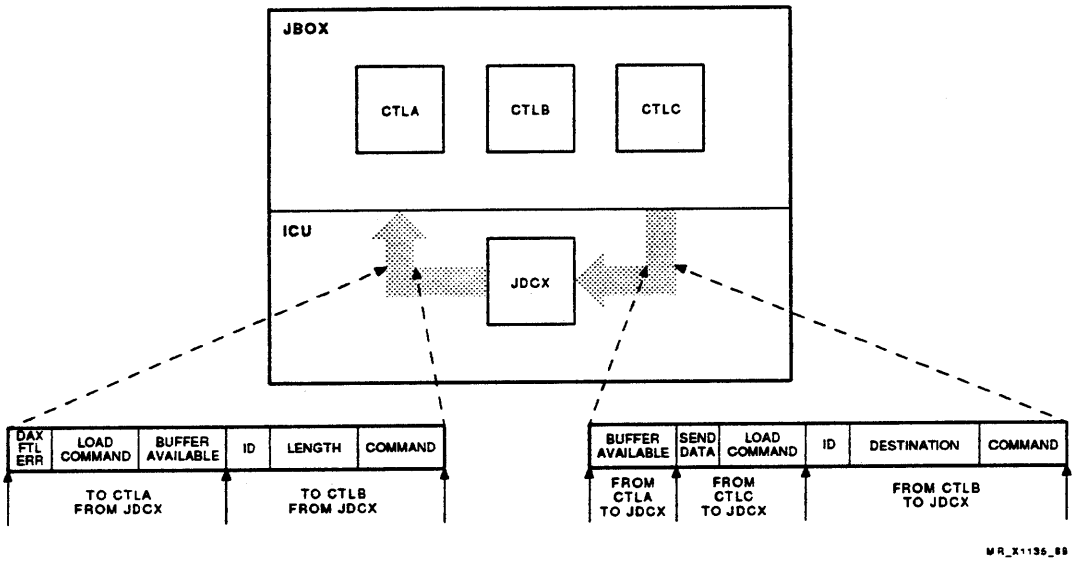


Figure 6–8 ICU-to-CCU Interface

6.3.1.1 Receive from XJA Control

The JDCX has two receive XJA control areas, one for each XJA. Each receive XJA receives the following signals:

- **JDAX_JDCX_XJA0_CMD_H[06:00]** — The XJA command [03:00], length [05:04], and ID [06] from the receive buffer in JDAX.
- **XJA0_CMDAVAIL_H** — The load command bit from the XJA handshaking signals.
- **JDAX XJA parity error** — JDAX has detected a parity error.
- **ICU_XJA_XFERACK_H** — ICU has received a packet successfully.
- **ICU_XJA_BUFEMPTED_H** — ICU has emptied one buffer.
- **ICU_XJA_XFERRETRY_H** — ICU has been unsuccessful in receiving a packet.

Figure 6–9 shows the receive XJA logic.

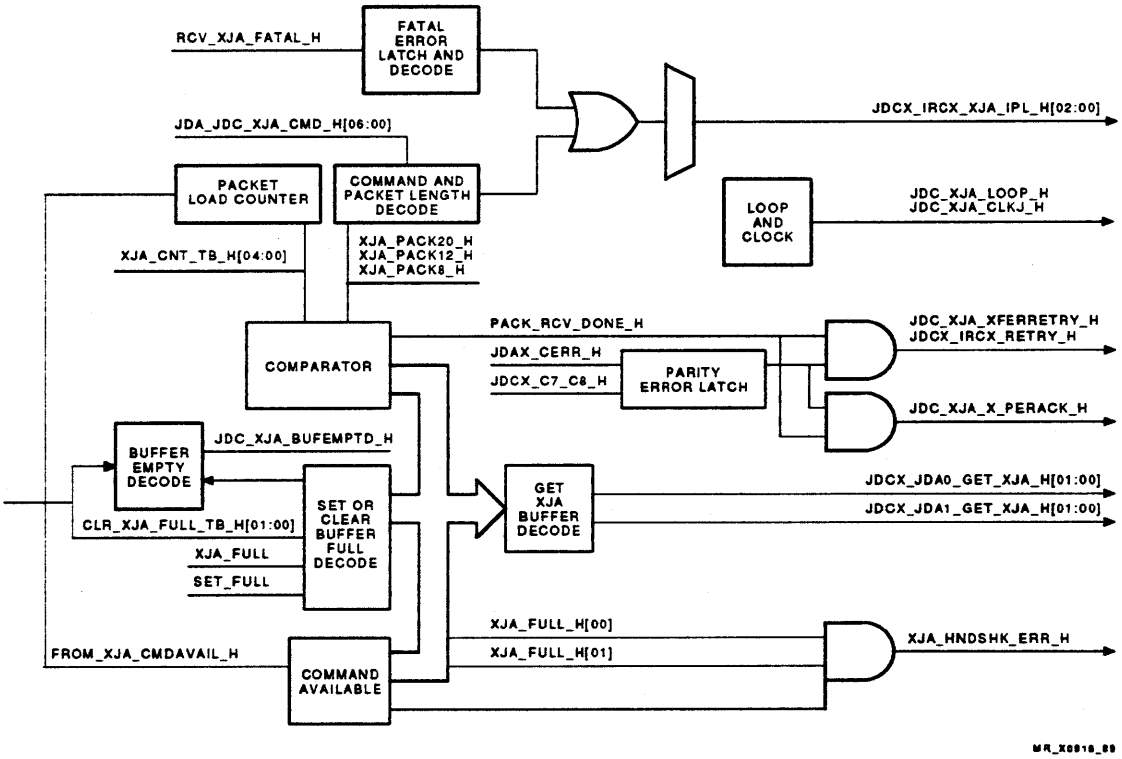


Figure 6–9 Receive from XJA Logic

Each receive XJA control sends the following signals:

- JDCX_JDA0_GET_XJAn_H[01:00] and JDCX_JDA1_GET_XJAn_H[01:00] are sent to JDA0 and JDA1, respectively.

  XJA sends command available to the receive XJA control in the JDCX MCA. Command available and the current buffer status determine JDC_JDA_GET_XJA_H[01:00], which JDCX sends to the JDAX MCA to select the XJA buffer, 0 or 1, to be loaded. The packet load counter begins to count and continues until PACK_RCV_DONE_H. The JDCX MCA resets JDC_JDA_GET_XJA_H[01:00].

- XJAn_FULL_TB_H[01:00] are sent to the XMIT CCU control logic.

  JDCX sets the buffer status as full when the JDAX MCA loads the packets into the receive buffer. JDCX sends retry to XJA, if JDAX detects a parity error. JDCX sends acknowledge to XJA if JDAX does not detect a parity error.

  JDCX tracks the number of buffers available for each XJA. Each XJA sends command available when one or both buffers are empty. JDCX detects a command available error when both receive buffers are full and XJA sends command available.

- XJA handshaking, retry, acknowledge, buffer empty, error, and loop are sent to the transmit CCU control logic. Figure 6–10 shows the buffer empty, acknowledge, and load command signals. The following steps summarize how SCU and XJA communicate using the load command, acknowledge, and buffer empty signals:

  ❶ Device unloads its command buffer and sends command over the XMI bus to the XJA.

  ❷ XJA loads its command buffer and responds with acknowledge.

  ❸ XJA sends load command followed by command to the ICU.

  ❹ ICU sends acknowledge to XJA.

  ❺ XJA sends buffer empty to the device to signify that the XJA can accept another command.

  ❻ ICU sends load command followed by the command to the command buffer in the CCU.

  ❼ ICU sends buffer empty to the XJA to signify that the ICU can accept another command.

  ❽ CCU arbitrates and the command is executed.

  ❾ CCU retires command when operation completes.

  ❿ CCU sends buffer available to the ICU. For write operations, CCU sends a send data signal to the ICU to unload the buffer holding the data.

- JDCX_IRCX_XJAn_IPL_H[02:00], JDCX_IRCX_SPUT_INT_H[07:00], and retry mode (JDCX_IRCX_RETRY_H[01:00]) are sent to the IRCX MCA. JDCX decodes either an XJA or SPU interrupt and sends IPL to the IRCX MCA.

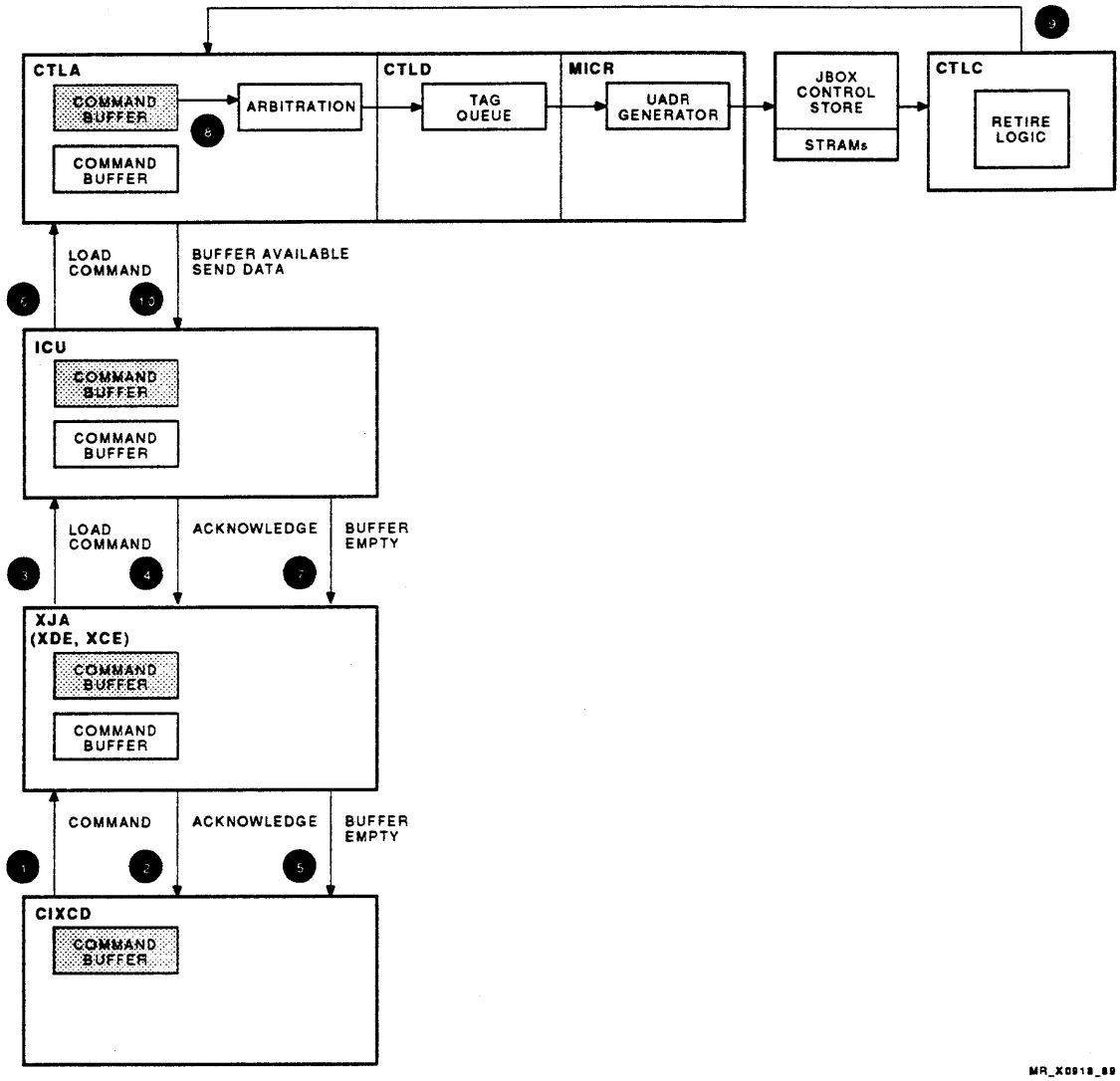


Figure 6–10 Buffer Empty, Acknowledge, and Load Command

6.3.1.2 Transmit to CCU Control

The transmit CCU control contains a CCU buffer counter that determines how many command buffers are available for the ICU port. The ICU port has two input command buffers, A and B, which send commands to port arbitration in the CCU. The ICU can send XJA or SPU commands to the input command buffers.

As JDCX sends the data from the receive buffer to CCU, transmit CCU tracks the three following major ICU-to-CCU states:

- **T1** — JDCX starts a CCU MCU sequence and sends a buffer select command to JDAX.
- **T2** — JDCX decodes the command in the selected buffer.
- **T3** — JDCX sends a load command to CCU, updates the least recently used status of the XJA buffers, sets XJA_BUF_BUSY_H for commands involving data transfers, and clears XJA_BUF_FULL_H for nondata commands.

The transmit CCU control receives the following:

- XJA0, XJA1 buffer 0 and 1 full status, 64 bytes, and sequence bit from the receive XJA logic
- Clear XMID field from the receive CCU for a DMA read command
- Read and write IRCX registers from the CCU MCU
- I/O tag [06:00] (ID field) from JDAX MCA
- Command [06:00] from JDAX MCA (XJA command [03:00], length [05:04])
- Number of ICU port command buffers (1 or 2) available from the CCU MCU
- The send data signal from the CCU MCU

Figure 6–11 shows the transmit-to-CCU logic.

The transmit CCU control sends the following commands:

- JDC_JDA_TRX_SEL_H[02:00] to JDAX to select one of five receive buffers.
- A send data command to IRCX for a CPU read IRCX register request. The IRCX MCA sends the data to the data switch.
- A send data command to JDAX to send buffer data to the data switch.
- CLR_XJA0_FULL_TB_H[01:00], CLR_XJA1_FULL_TB_H[01:00], and CLEAR_SPU_BUF_H to JDAX to clear buffer full status.

  The JDCX uses JDCX_BUF_TRXSEL_H[02:00] to send the data to the CCU MCU. The JDCX clears the XJA buffer full status using JDCX_BUF_TRXSEL_H[02:00] to identify the buffer and SEND_BUF_SEL_H[02:00] to determine when the buffer is empty. JDCX clears buffer full status for SPU, XJA0 buffers 0 and 1, and XJA1 buffers 0 and 1. JDCX_BUF_TRXSEL_H[02:00] determines the type of request: SPU, XJA0, or XJA1.

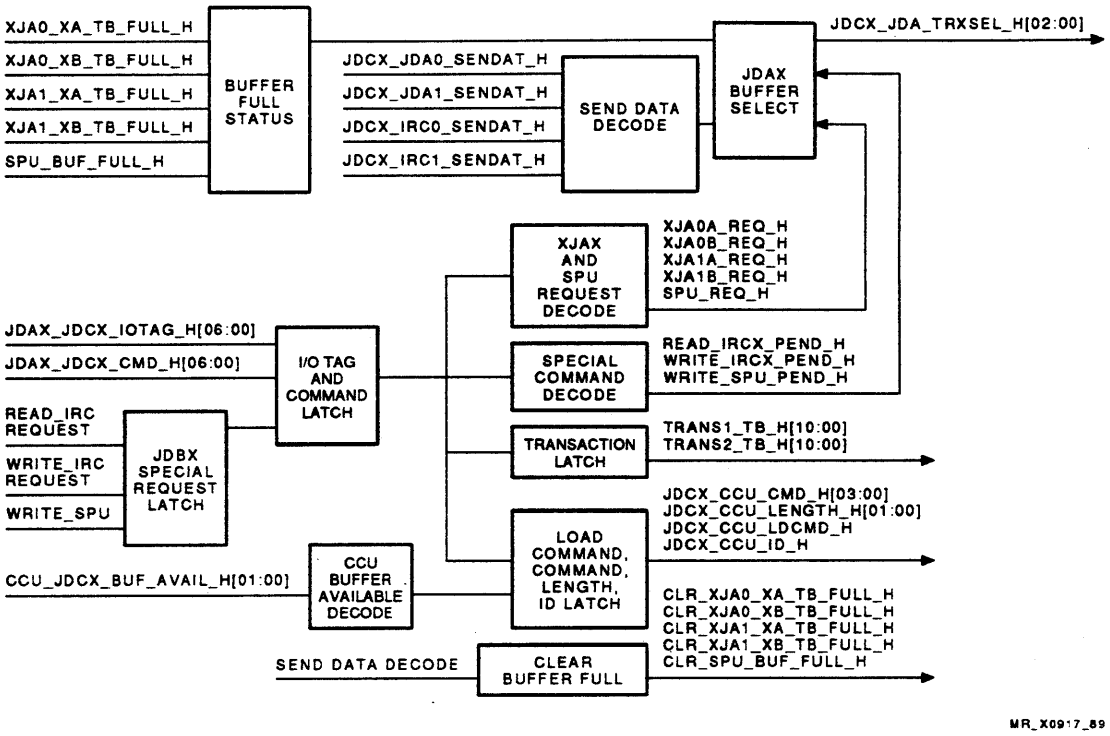


Figure 6–11 Transmit-to-CCU Logic

- JDCX_CCU_CMD_H[03:00] to CCU MCU.
- JDCX_CCU_LENGTH_H[01:00] to CCU MCU.
- JDCX_CCU_LDCMD_H to CCU MCU.
- JDCX_CCU_ID_H to CCU MCU to specify which ID field in the JDCX MCA to use. XMIT CCU has two transaction latches that hold TRANS1 and TRANS2 for two read commands.
- TRANS1, TRANS2 [10:00] to the RCV CCU control logic. Table 6–1 lists the fields and descriptions for TRANSn [10:00].

Table 6–1 TRANS [10:00]

Bit	Name	Description
05:00	XMI ID	XMI ID of DMA read command from the XJA
07:06	Length	The length field of the DMA read data return command: 00 = Hexword (32 bytes, reserved) 01 = Block (64 bytes, reserved) 10 = Quadword (8 bytes) 11 = Octaword (16 bytes)
09:08	DMA read data return	The command field of the DMA read data return command: 00 = XJA0 (DA0), XJA2 (DA1) 01 = XJA1 (DA0), XJA3 (DA1) 1x = SPU
10	Valid	Valid entry in the transaction latch

6.3.1.3 Receive from CCU Control

The receive CCU receives the following commands:

- TRANS1 [10:00] or TRANS2 [10:00] from transmit CCU logic
- CCU_JDCX_CMD_H[03:00] from CCU (CCU-to-ICU command)
- CCU_JDCX_IOSEL_H[01:00] from CCU:

  00 = XJA0 (ICU0) or XJA 2 (ICU1)
  01 = XJA1 (ICU0) or XJA3 (ICU1)
  10 = IRCX (ICU0)
  11 = SPU (ICU0)

- CCU_JDCX_ID_H from CCU to specify which ID field to use, TRANS1 or TRANS2, for a DMA read command
- CCU_JDCX_LDCMD_H from CCU to load the command
- DSW_JDB_BOD_H from CCU to mark the beginning of data
- Clear SPU buffer A, B from SPU control
- Clear XJA0 buffer A, B from the transmit XJA0 and clear XJA1 buffer A, B from the transmit XJA1 control

Figure 6–12 shows the receive from CCU logic.

The receive CCU sends the following commands:

- JDC_JDB_RCV_SEL_H[01:00] to JDBX to select the transmit buffer, A or B, to load for SPU, XJA0, or XJA1
- JDCX_JDB_CMD_H[05:00] to JDBX to specify the XJA or SPU command
- JDCX_JDB_IOTAG_H[05:00] to JDBX for ID field information

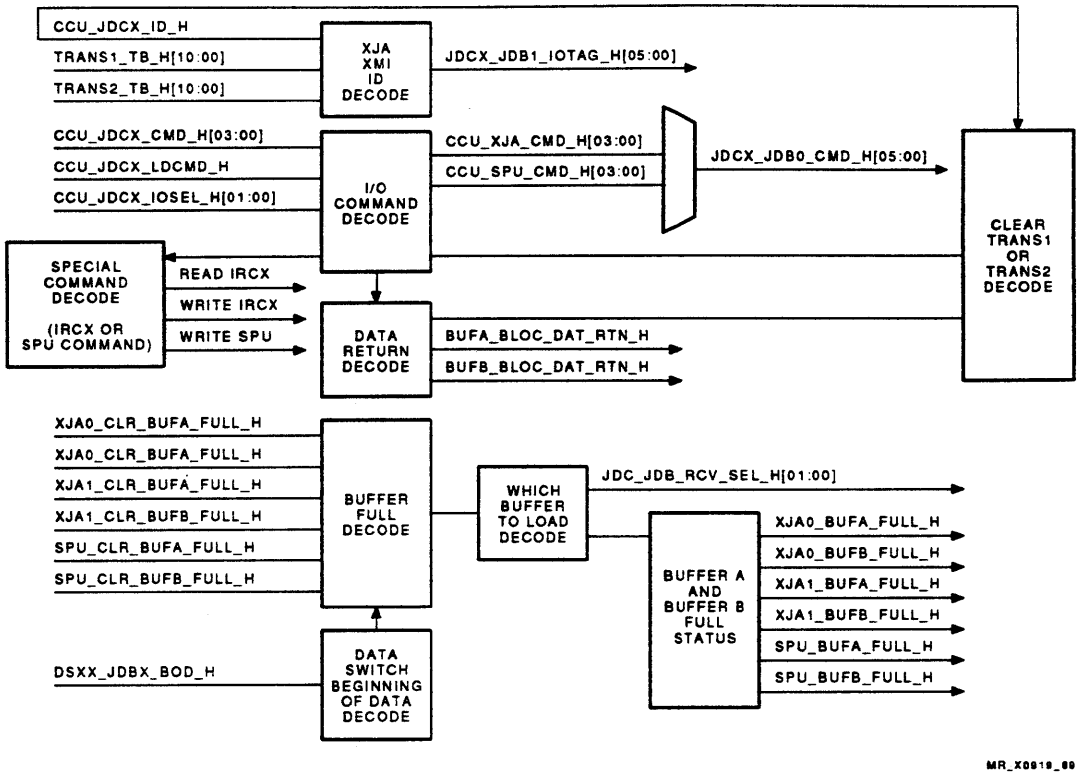


Figure 6–12 Receive from CCU Logic

- JDCX_CCU_BUF_AVAIL_H to notify CCU that the ICU receive command buffer is available
- Clear TRANS1, TRANS2 to XMIT CCU
- Read and write IRCX register request to XMIT CCU
- Write SPU register complete to transmit CCU
- Buffer full, command, and block data return to SPU control and transmit XJA control

As the CCU MCU sends command and data to the buffers in the ICU, the receive from CCU logic control tracks the following two major CCU-to-ICU states:

- **T0** — During this state, JDCX receives load command, command, and destination from CCU.
- **T1** — During this state, JDCX does the following:
  - — Decodes the CCU command as XJA or SPU.
  - — For a DMA read data return, sends command and ID field to JDBX.
  - — Determines the buffer full status for XJA0, XJA1, and SPU (loading the buffers).
  - — Sends JDC_JDB_RCV_SEL_H[01:00] to JDBX. The DSXX MCAs send DSXX_ JDBX_BOD_H.

6.3.1.4 Transmit to XJA Control

The transmit XJA control receives the following:

- Buffer full, command, and block data return from RCV CCU
- Acknowledge, retry, and buffer empty from XJA0, XJA1

Figure 6–13 shows the transmit-to-XJA logic. (The A1 and A2 state machines control transmit buffer A. The B1 and B2 state machines control transmit buffer B. These state machines are reserved for hexword DMA read operations.)

The transmit XJA control sends the following commands:

- JDC_JDB_SEND_XJAn_H[01:00] to JDBX to enable the data to the differential XJA_DAT_H[07:00]. Each transmit buffer in JDBX, A and B, has a state machine that controls loading, unloading, and sending data.
- JDC_JDB_XMIT_BUFn_H[02:00] to JDBX to select a transmit buffer, A or B, and send the data to SPU, XJA0, or XJA1.
- JDCX_XJAn_CMDAVAIL_H to XJA.
- Clear XJA buffer A, B to receive CCU control.

6.3.1.5 SPU Control

The SPU control receives the following signals:

1. CTLD_JDCX_REQUEST_H, CTLD_JDCX_BUFFULL_H, and CTLD_JDCX_ERR_H (handshaking signals) from CTLD MCA
2. JDA0_JDCX_SPU_CMD_H[03:00] (command) and JDA1_JDCX_SPU_ID_H[03:00] CPU ID from JDAX
3. Clear SPU buffer from transmit CCU control
4. Buffer full status from receive CCU control
5. SPU buffer A and B commands (JDCX_JDB0_SPU_CMD_H[03:00]) from receive CCU control, which are loaded into a transmit buffer in JDBX and sent to the SPU

Figure 6–14 shows the SPU control logic.

The SPU control sends:

- JDCX_CTLD_BUF_GRANT_H (grant), XMIT_SPU_FRAME_TA_H (frame), and JDCX_CTLD_SPU_ERR_H (error) to CTLD, which sends the handshaking signals to SPU.
- JDC_JDA_GET_SPU_H to the JDAX MCA.
- JDCX_IRCX_SPU_INT_H[07:00] to the IRCX MCA, which decodes the interrupt command, determines the type of interrupt, sets the corresponding IPL, and sends a request to the arbiter. Table 6–2 lists the interrupt types and IPLs.
- JDC_JDB_SEND_SPU_H[02:00] to JDBX. Table 6–3 lists the bits and their descriptions.
- Clear buffer A and B full status to receive CCU control.

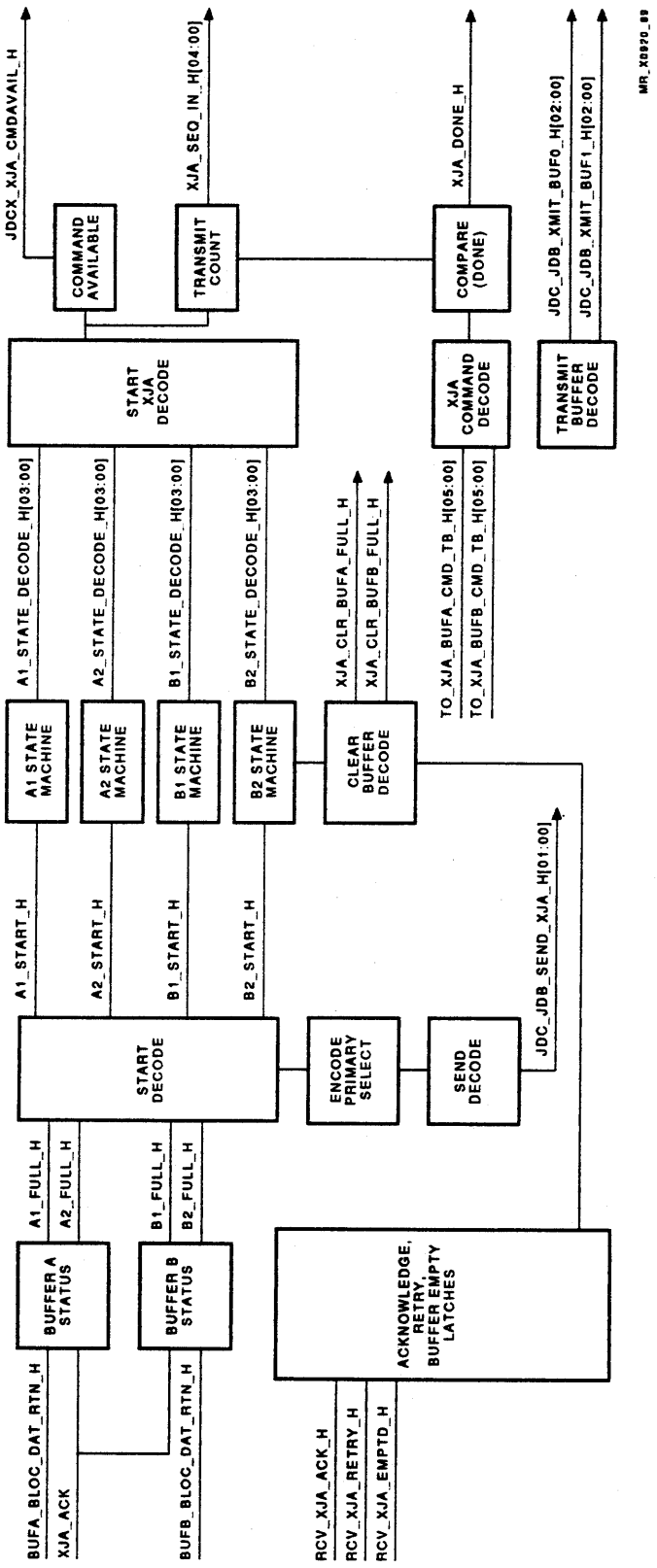


Figure 6–13 Transmit-to-XJA Logic

- XJA ON from the scan latches to the differential receiver for XJA0 and XJA1 to enable the handshaking signals error, command available, retry, and acknowledge to be sent to XJA0 and XJA1.
- JDCX_CTLD_SPU_CLKJ_H to SPU to synchronize transfers to and from the JBox.

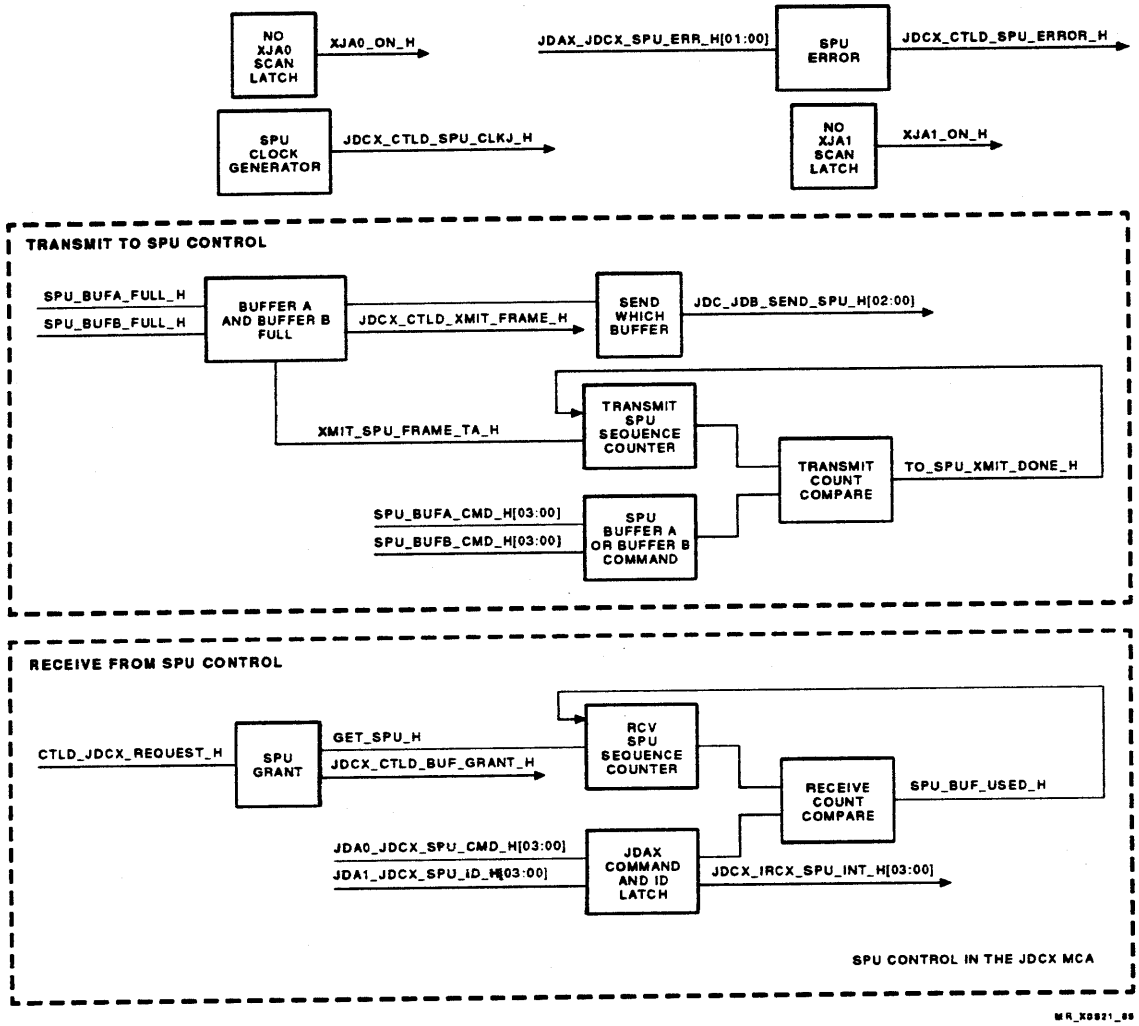


Figure 6–14 SPU Control Logic

Table 6–2 SPU IPLs

Interrupt	IPL (Hex)
Halt	20
Spare	20
Spare	1E
Powerfail	1E
Keep alive	16
Terminal receive	14
Terminal transmit	14
Console block storage receive	17
Console block storage transmit	17

Table 6–3 SEND SPU [02:00]

Bit	Name	Description
00	Buffer B	JDBX transmits buffer B.
01	ECC	JDBX loads the ECC SPU command, ECC address, and syndrome data into the buffer.
02	Go	JDBX unloads the buffer and sends XMIT_SPU_FRAME_TA_H with the data.

6.3.2 JDAX MCA

JDA0, located on the DA0 MCU, and JDA1, located on the DB0 MCU, support XJA0, XJA1, and SPU by providing two receive buffers for each XJA and one receive buffer for SPU, for a total of five receive buffers, as follows:

XJA0 buffer 0
XJA0 buffer 1
XJA1 buffer 0
XJA1 buffer 1
SPU buffer

JDA2, located on the DA1 MCU, and JDA3, located on the DB1 MCU, support XJA2 and XJA3 by providing two receive buffers for each XJA, for a total of four receive buffers, as follows:

XJA2 buffer 0
XJA2 buffer 1
XJA3 buffer 0
XJA3 buffer 1

Each JDAX MCA has a front receiver and receives XJAn_JDA_DATA_H[07:00]. For example, JDA0 receives XJA0_JDA_DATA_H[07:00] and JDA1 receives XJA0_JDA_DATA_H[15:08].

Under JDCX control, JDAX loads the data into a receive buffer. Each buffer is byte-sliced across MCAs and can hold up to four quadwords. For each JXDI cycle, two JDAX MCAs receive and load one byte at a time into a receive buffer. For example, JDA0 receives XJA data bits [07:00] and a parity bit, and JDA1 receives data bits [15:08] and a parity bit. Figure 6–15 shows the byte-slices across MCAs and MCUs for receive buffer 0.

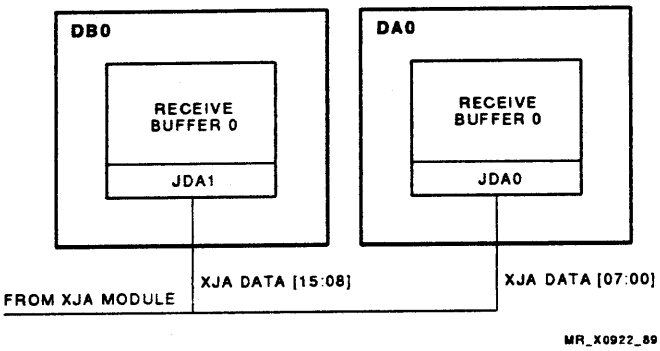


Figure 6–15 Receive Buffer — Byte-Slices

JDA0 and JDA1 each have a counter that increments by one each time a byte of the packet is loaded into the receive buffer. Together, JDA0 and JDA1 load one word of the packet each cycle. Table 6–4 lists the number of cycles, bytes, and data sizes for the JXDI packets.

Table 6–4 JXDI Cycles

Number of Cycles	Number of Bytes	Data Size
20	32	Hexword
12	16	Octaword
8	8	Quadword

Figure 6–16 shows the JDAX MCA block diagram. The JDAX MCA receives the following:

- XJA command, address, and data
- JDC_JDA_TRX_SEL_H[02:00] (buffer select) from the JDCX (Table 6–5)

- Send data from JDCX
- Get XJA and get SPU control signals from JDCX
- SPU command, address, and data from CTLD
- Retry mode [01:00] from SCAN

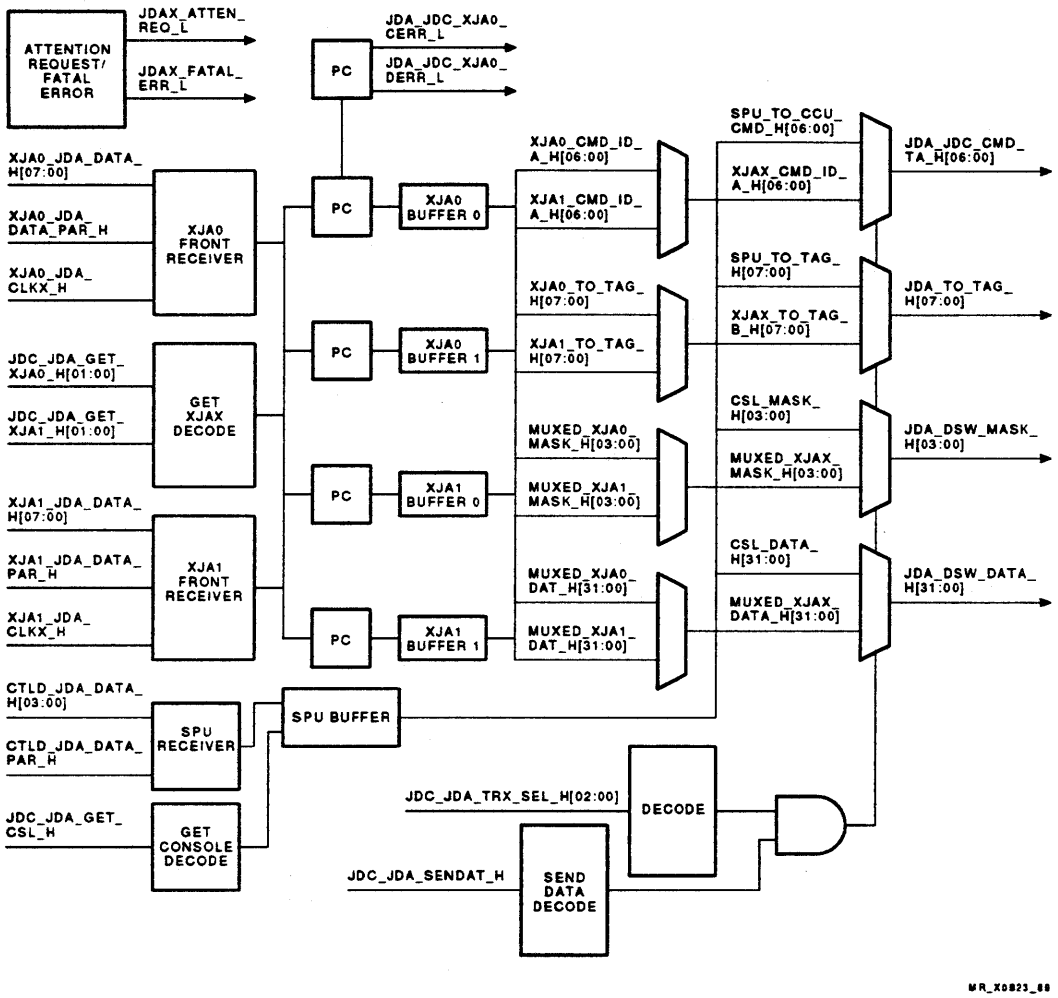


Figure 6–16 JDAX MCA Block Diagram

Table 6–5 JDCX Transmit Buffer Select

Value	Buffer
011	SPU
100	XJA0
101	XJA0
110	XJA1
111	XJA1

Table 6–6 lists each mode and the corresponding MCU attention lines. For more details on the JBox error summary register in the IRCX MCA, see Section 6.14.3.3.

Mode 0 — The ICU does not report the retry by pulsing an attention line. The ICU stores information about the error in scan latches and sets bits in the error summary register in the IRCX MCA.

Mode 1 — The MCU detects the retry error pulses on its attention line. An XJA retry report occurs when SPU determines that the DAX or DBX attention lines, and not the CCU attention line, were pulsed. The ICU stores information about the error in scan latches and sets bits in the error summary register in the IRCX MCA.

Mode 2 — The MCU detects the retry error pulses on its attention line. In addition to DAX or DBX pulsing its attention line, DAX or DBX asserts one of the error lines to CCU. A possible XJA retry report has occurred and the SPU determines if the error register contains retry information by analyzing the DAX or DBX attention lines and the CCU attention line.

The SPU uses scan to determine if the error was due to an XJA retry. The ICU stores the XJAX parity error data and information about the error in scan latches and sets bits in the error summary register in the IRCX MCA.

Table 6–6 Retry Modes

Mode	DAX/DBX Attention	CCU Attention	CCU Error Signal	Enter Code into JBox Error Summary Register
0	N	N	N	Y
1	Y	N	N	Y
2	Y	Y	Y	Y

The JDAX MCA sends the following:

- Physical address to the ADRX MCAs
- XJA0 and XJA1 command to the JDCX MCA
- SPU command to the JDCX MCA
- XJA0, XJA1, and SPU parity errors, JDAX fatal error, and JDAX attention request to JDCX MCA
- Mask, data, and beginning of data to the data switch (DSXX MCAs)

The JDAX MCA has parity checkers for the control and data lines and generates the following error signals when detecting a parity error:

- **CTLD control parity error** — JDAX detects a parity error across the CTLD_JDA_XJA0_INIT, CTLD_JDA_XJA1_INIT using parity lines CTLD_JDA_XJA0, 1 _INIT_PAR_H.
- **JDC control parity error** — JDAX detects a parity error across the JDCA_JDA_GET_XJA0_H[01:00], JDCA_JDA_GET_XJA1_H[01:00], JDC_JDA_TRX_SEL_H[02:00], JDC_JDA_SENDAT_H, and JDC_JDA_GET_CSL_H using JDC_JDA_CTL_PAR_H.
- **JDAX_ATTEN_REQ** — JDAX detects an XJA data error.
- **JDAX_FATAL_ERROR_L** — JDAX detects any of the following errors:

  CTLD control parity error (CTLD_CTRL_PE_H)
  JDC control parity error (JDC_CTRL_PE_H)
  XJAX fatal error (XJAX_FATAL_REQ_L)
  Receive console error (RCV_CSL_ERR_TB_L)

6.3.2.1 XJA Receive Buffers

Each XJA has two receive buffers, 0 and 1. Figure 6–17 shows the receive buffers for XJA0.

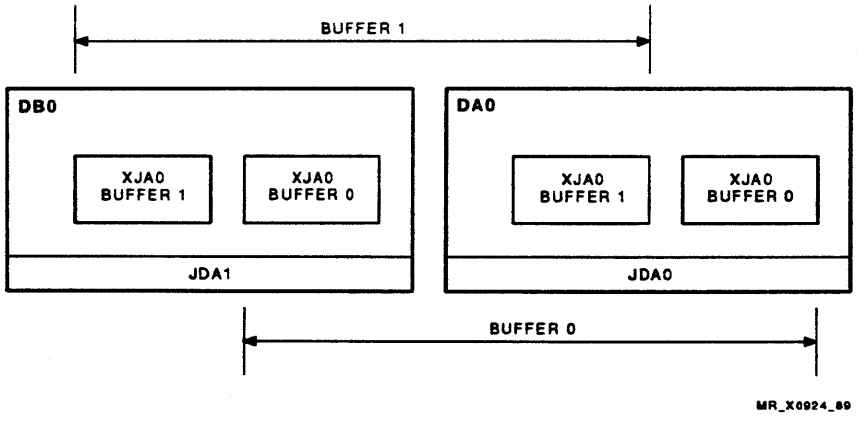


Figure 6–17 XJA0 Receive Buffers

Figure 6–18 shows the receive buffer.

Each XJA buffer receives the following signals:

- CLOCK_A_L and CLOCK_B_L.
- JDC_JDA_GET_XJAn[00] (buffer 0), JDC_JDA_GET_XJAn[01] (buffer 1) from the JDCX receive XJA logic. Buffer full and command available determine which buffer JDAX selects.
- XJAn_CHK_DAT_H[07:00] from the parity checkers.

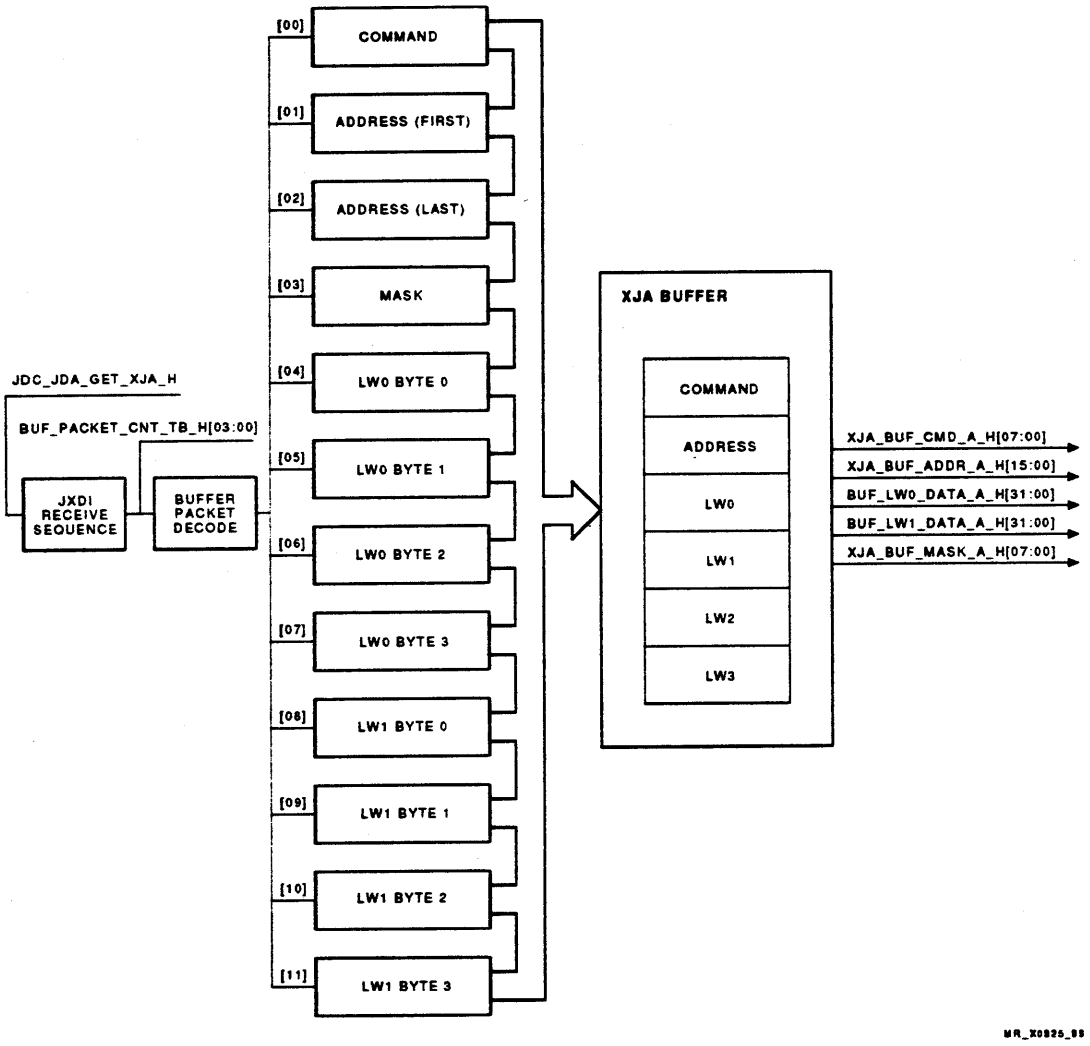


Figure 6–18 XJA Receive Buffer

Each XJA receive buffer sends the following signals:

- XJAnBUFn_CMD_A_H[07:00] to the command buffer selector logic
- XJAnBUFn_ADDR_A_H[15:00] to the address buffer selector logic
- XJAnBUFn_MASK_A_H[07:00] to the mask buffer selector logic
- XJAnBUFn_LW0_DAT_A_H[31:00] to the data selector logic
- XJAnBUFn_LW1_DAT_A_H[31:00] to the data selector logic

6.3.2.2 SPU Receive Buffer

The SPU buffer receives a quadword of data. Figure 6–19 shows the SPU receive buffer.

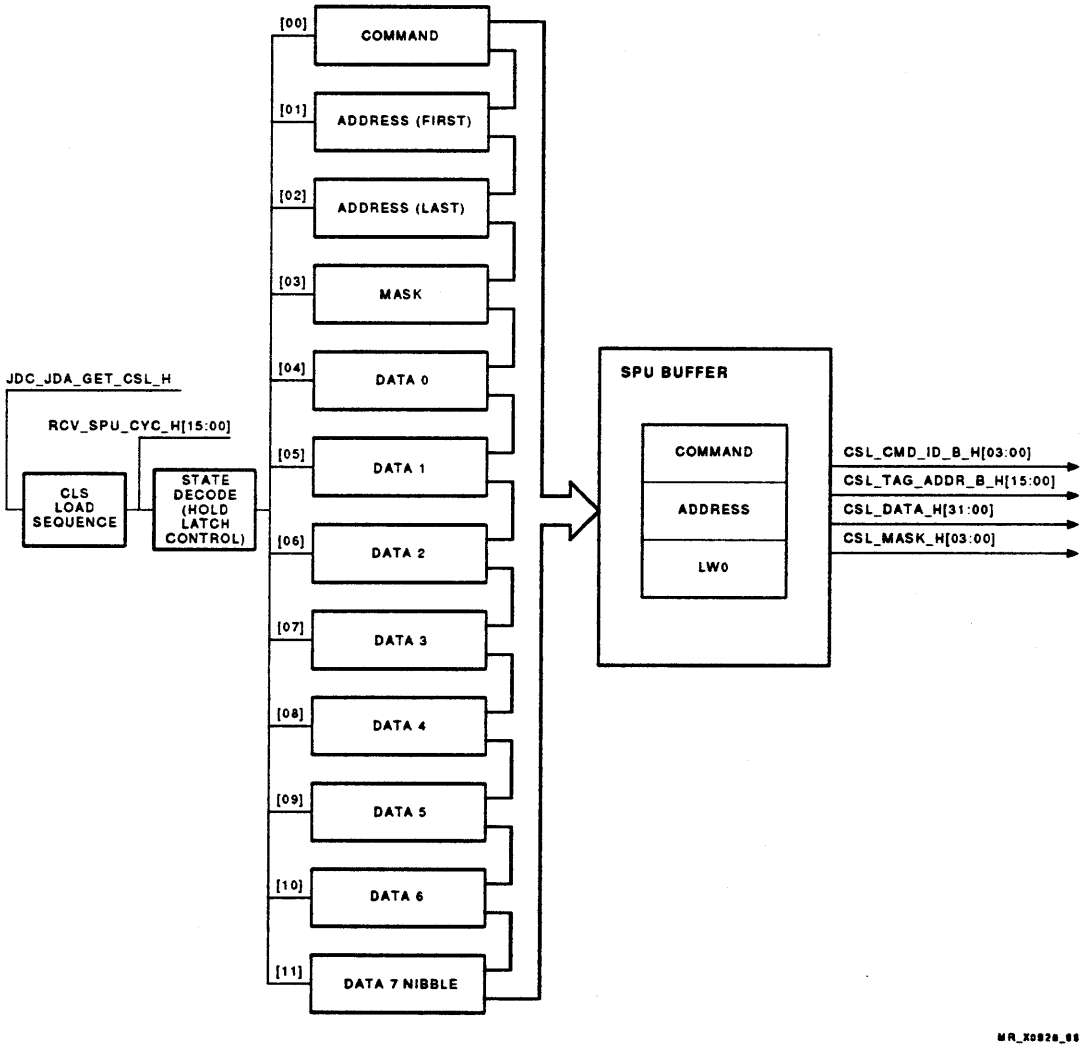


Figure 6–19 SPU Receive Buffer

The SPU buffer receives the following signals:

- CLOCK_A, CLOCK_B
- CTLD_JDA_DATA_H[03:00] and parity from the CTLD SPU interface
- JDC_JDA_GET_CSL_H from the JDC SPU control logic

The SPU buffer sends the following signals:

- CSL_ADDR_B_H[15:00] to the address buffer selector logic
- CSL_MASK_H[03:00] to to the mask buffer selector logic
- CSL_DAT_H[31:00] to the data buffer selector logic
- JDA_JDC_CSL_CMD_TA_H[03:00] to the command buffer selector logic

The JDAX MCA decodes JDC_JDA_TRX_SEL_H[02:00] and selects an XJA or SPU command, address, mask, and data. Table 6–7 lists the JDC_JDA_TRX_SEL_H[02:00] codes and their corresponding buffers selected.

Table 6–7 JDC_JDA_TRX_SEL_H[02:00] Decode

Code	Buffer Selected
0	IRCX
1	Reserved
2	Reserved
3	SPU
4	XJA0 buffer 0
5	XJA0 buffer 1
6	XJA1 buffer 0
7	XJA1 buffer 1

6.3.2.3 Selecting XJA or SPU Command and Address

The command/address buffer selector (Figure 6–20) selects one of the following, and sends the command as XJAX_CMD_ID_A[06:00] to the CCU:

- XJA0BUF0_CMD_A_H[06:00]
- XJA0BUF1_CMD_A_H[06:00]
- XJA1BUF0_CMD_A_H[06:00]
- XJA1BUF1_CMD_A_H[06:00]
- SPU_TO_CMD_H[06:00]

Otherwise, the command/address buffer selector (Figure 6–20) selects one of the following and sends the address in two cycles as XJAn_TO_TAG_H[07:00] or SPU_TO_TAG_H[07:00] to the tag MCU:

- XJA0BUF0_ADDR_A_H[07:00], XJA0BUF0_ADDR_A_H[15:08]
- XJA0BUF1_ADDR_A_H[07:00], XJA0BUF1_ADDR_A_H[15:08]
- XJA1BUF0_ADDR_A_H[07:00], XJA1BUF0_ADDR_A_H[15:08]
- XJA1BUF1_ADDR_A_H[07:00], XJA1BUF1_ADDR_A_H[15:08]
- SPU_TO_TAG_H[07:00]

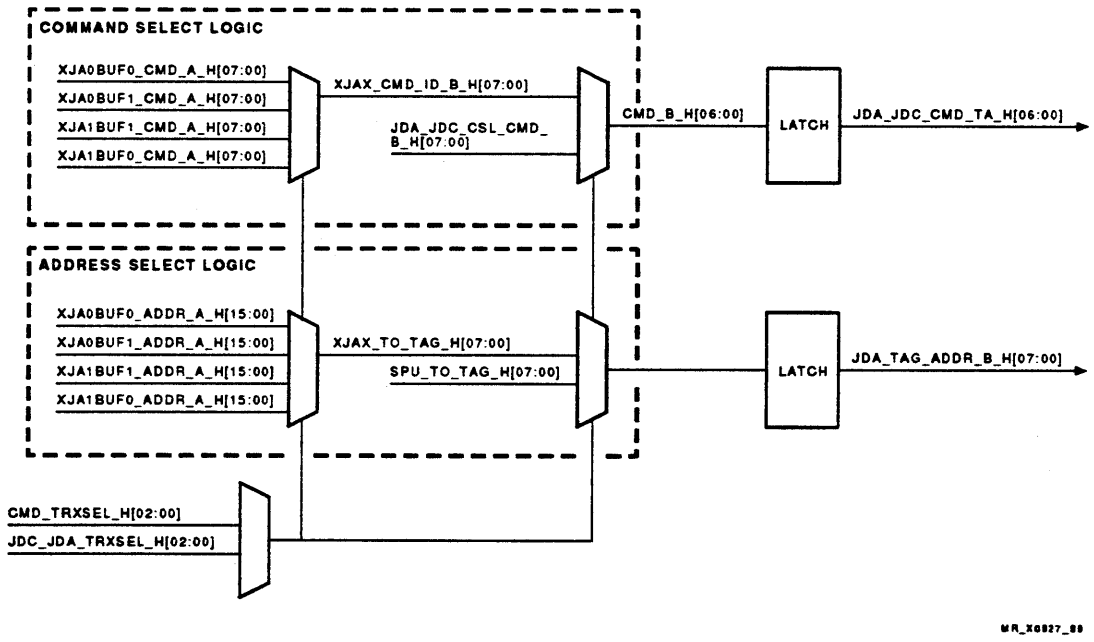


Figure 6–20 Selecting the XJA Command

6.3.2.4 Sending Data

JDC_JDA_SENDAT_B_H generates JDA_DSW_BOD_H (beginning of data transfer to the data switch). Each buffer, XJA0 buffer 0, XJA0 buffer 1, XJA1 buffer 0, and XJA1 buffer 1, sends two longwords, LW0 and LW1, and a parity bit for each longword to the data buffer selector.

JDAX sends the output of the XJA data select (MUXED_XJA0_DAT_H[31:00] and MUXED_XJA1_DAT_H[31:00]) or CSL_DATA_H[31:00] to the data switch as JDA_DSW_DATA_H[31:00]. JDC_JDA_TRX_SEL_H[02:00] selects the buffer and longword. Figure 6–21 shows how the JDAX MCAs send longwords to the DSXX MCAs.

Each buffer, XJA0 buffer 0, XJA0 buffer 1, XJA1 buffer 0, and XJA1 buffer 1, sends two mask field, [07:04] and [03:00], to the mask buffer selector.

JDAX sends the output of the XJA mask select (MUXED_XJA0_MASK_H[03:00] and MUXED_XJA1_MASK_H[03:00]) or CSL_MASK_H[03:00] to the data switch as JDA_DSW_MASK_H[03:00]. JDC_JDA_TRX_SEL_H[02:00] and JDC_JDA_SENDAT_B_H determine which mask the JDAX MCAs select.

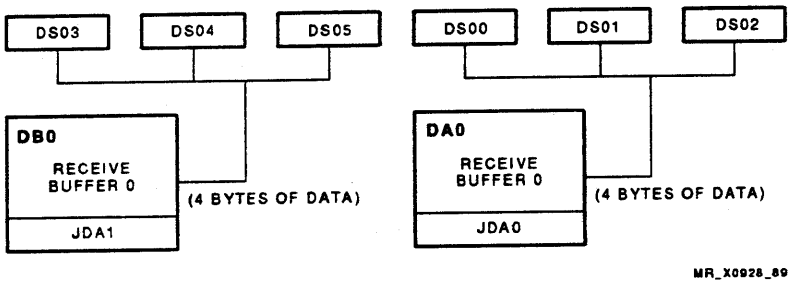


Figure 6–21 JDAX Sending Data to the DSXX MCAs

6.3.2.5 Loading an XJA Buffer

The following steps summarize the loading of XJA0 buffer 0 with a DMA write command, address 2000, and 16 bytes (octaword) of data. In consecutive cycles, XJA0 sends:

1. XJA_CMDAVAIL_H, which JDCX latches and uses to generate JDC_JDA_GET_XJA_H[01:00]. JDCX tracks the status of buffers 0 and 1. Buffer 0 is available.

   JXDI receiver sequencer uses JDC_JDA_GET_XJA_H[01:00] to enable and start the packet counter. BUF_PACKET_INC_H[03:00] becomes BUF_PACKET_DECODE_L[15:00]. Figure 6–18 shows the buffer packet decode logic.

2. BUF_PACKET_DECODE_L[00], which enables JDAX to load the command, DMA write command, and quadword data size.

3. BUF_PACKET_DECODE_L[01] and [02], which enable JDAX to load the address, the first half of address 2000 and then the second half of address 2000.

4. BUF_PACKET_DECODE_L[03], which enables JDAX to load the mask.

5. BUF_PACKET_DECODE_L[04], [05], [06], [07], [08], [09], [10] and [11], which enable JDAX to load bytes 0 through 3 as LW0 [31:00]. Bytes 4 through 7 are loaded into buffer 0 as LW1 [63:32].

6. XJA waits for retry or acknowledge.

7. XJA waits for buffer empty.

6.3.2.6 Unloading the XJA Buffer

The following steps summarize the unloading of XJA0 buffer 0:

1. JDC_JDA_TRX_SEL_H[02:00] generates CMD_TRX_SEL_TA_H[02:00], which selects XJA0_BUF0_CMD_A_H[06:00]. JDAX sends this to JDCX as JDA_JDC_CMD_TA_H[06:00] (Figure 6–16).
2. ADR_MUX_SEL_TB_H[01:00] selects XJA0BUF0_ADDR_A_H[07:00] and XJA0BUF0_ADDR_A_H[15:08]. The address is loaded into the ICU0 address receive buffer in ADRX. ADRX sends physical address bits to MTCH, which addresses the global tag STRAMs and compares the physical address bits stored with the ICU0 address. MTCH sends the match results to MICR. Also stored in the tag STRAMs are the status bits for that address. MICR latches the status bits.
3. The command is arbitrated and placed in the tag queue in CTLD. The MICR MCA takes the entry from the tag queue (queue data contains the command, arbitration index, and results from NPAMM). The MICR MCA uses the results (match and status bits) of the tag lookup and queue data to form a microaddress to send to the control store.
4. The microword sends a fix command to the fixup queue. The fixup queue sends another microword address to a fixup microword to do the following:
   a. Send SENDAT to JDCX. JDC_JDA_SENDAT_B_H generates JDA_DSW_BOD_H (the beginning of data transfer to the data switch). LWO and LW1 are sent to the data switch. The data switch sends the data to MDPX, which sends the data to main memory.
   b. Determine the index (assigned at arbitration) sent to ADRX to identify which I/O address buffer is to be used to drive the row and column lines for the memory arrays.
   c. Send a write command to the memory port.

6.3.2.7 Loading the SPU Buffer

The following steps summarize the loading of the SPU buffer:

1. CTLD receives CTLD_JDA_DATA_H[03:00] and CTLD_JDA_DATA_PAR_H, and sends them to JDAX. The SPU receiver sequencer uses JDC_JDA_GET_CSL_H to enable and start the packet counter. STATE_B_H[06:00] becomes RCV_SPU_CYC_H[15:00] (Figure 6–19).
2. RCV_SPU_CYC_H[00] enables JDAX to load the command, SPU write command, and quadword data size.
3. RCV_SPU_CYC_H[01], [02], [03], and [04] enable JDAX to load the address.
4. RCV_SPU_CYC_H[05] enables JDAX to load the mask. Figure 6–58 shows the SPU DMA packet.
5. RCV_SPU_CYC_H[06], [07], [08], [09], [10], [11], [12], and [13] enable JDAX to load the data into the SPU BUFFER as ARRAY_DAT0_H[31:00] through ARRAY_DAT7_H[31:00]. Figure 6–58 shows the SPU DMA packet.

6.3.2.8 Unloading the SPU Buffer

The following steps summarize the unloading of the SPU buffer:

1. JDC_JDA_GET_CSL_H unloads the SPU buffer. JDC_JDA_TRX_SEL_H[02:00] generates CMD_TRX_SEL_TA_H[02:00], which selects SPU_TO_CCU_CMD_H[06:00] and sends this to JDC as JDA_JDC_CMD_TA_H[06:00] (Figure 6–16).
2. ADR_MUX_SEL_TB_H[01:00] selects and sends SPU_TO_TAG_H[07:00] to the tag MCU (ADRX MCAs I/O receive address latches). ADRX sends physical address bits to MTCH, which addresses the global global tag STRAMs and compares the physical address bits stored with the XJA0 address. MTCH sends the match results to MICR. Also stored in the global tag STRAMs are the status bits for that address. MICR latches the status bits.
3. ICU sends the command to CCU command arbitration. When the command wins arbitration, CCU places an entry onto the tag queue. The MICR MCA takes the entry from the tag queue (queue data contains the command, arbitration index, and results from the NPAMM), uses the results (match and status bits) of the tag lookup and queue data, and forms a microaddress to send to the control store.
4. The microword sends a fix command to the fixup queue. The fixup queue then sends another microword address to a fixup microword to do the following:
   a. Send SENDAT to JDCX. JDC_JDA_SENDAT_B_H generates JDA_DSW_BOD_H (the beginning of data transfer to the data switch). CSL_DATA_H[31:00] are sent to the data switch. The data switch sends the data to MDPX, which sends the data to main memory.
   b. Determine the index (assigned at arbitration) set to ADRX to identify which I/O address buffer is to be used to drive the row and column lines for the memory arrays.
   c. Send a write command to the memory port.

6.3.3 JDBX MCA

JDB0, located on the DA0 MCU, and JDB1, located on the DB0 MCU, support the XJA0, XJA1, and SPU transmit command, address, and data buffers. JDB2, located on the DA1 MCU, and JDB3, located on the DB1 MCU, support XJA2 and XJA3.

Each JDBX MCA contains transmit XJA logic. For example, JDB0 sends JDB_XJA0_DATA_H[07:00] and JDB_XJA1_DATA_H[07:00], and JDB1 sends JDB_XJA0_DATA_H[15:08] and JDB_XJA1_DATA_H[15:08]. Figure 6–22 shows the byte-slices for a transmit buffers command. Figure 6–23 shows the JDBX MCA block diagram.

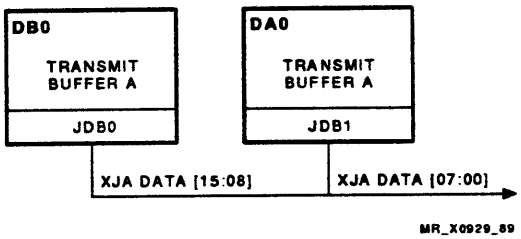


Figure 6–22 JDBX Transmit Buffer — Byte-Slices

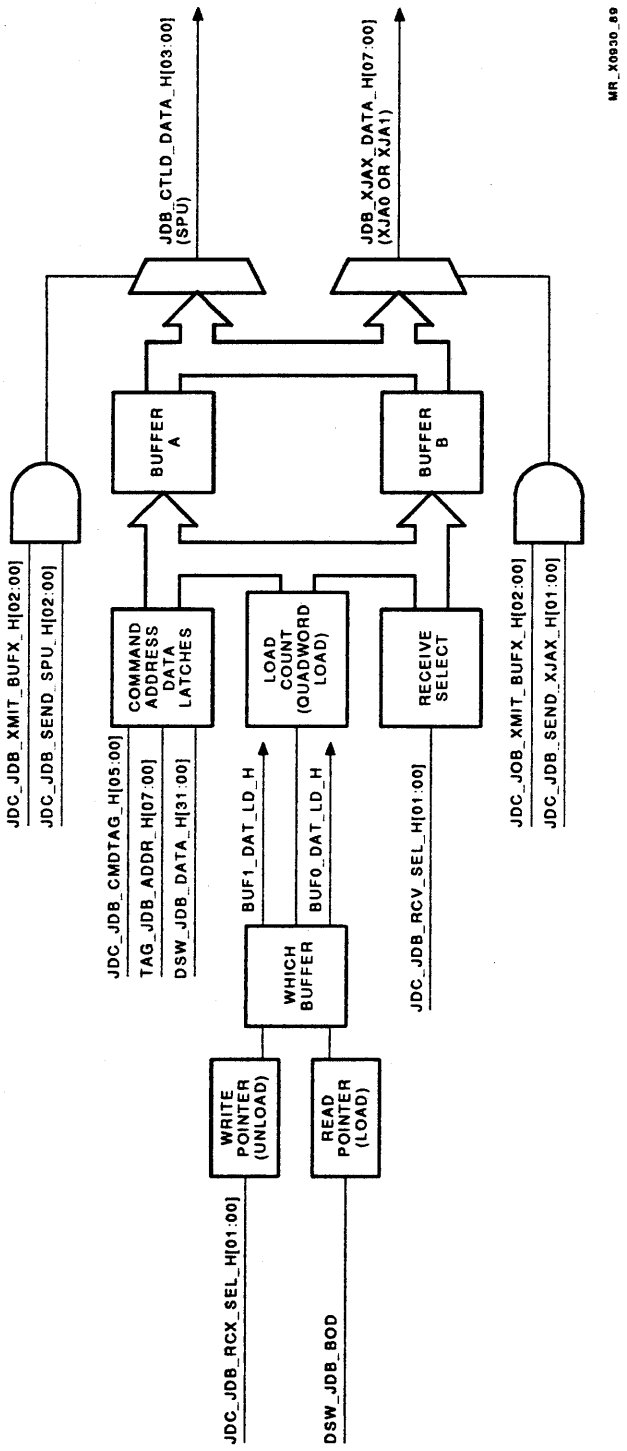


Figure 6–23 JDBX MCA Block Diagram

Unlike the JDAX MCA receive buffers, the two JDBX MCA transmit buffers can be used for sending a command, address, and data to either XJA0/XJA2, XJA1/XJA3, or SPU. The transmit buffers are called buffer A and buffer B. However, signal names and references may refer to buffer A as buffer 0 and buffer B as buffer 1. Figure 6–24 shows the transmit buffer.

The JDBX MCA receives the following:

- JDC_JDB_RCV_SEL_H[01:00] (buffer select) from the JDCX MCA to control the loading of buffers. Figure 6–25 shows how the JDBX MCAs receive longwords from the DSXX MCAs.

  JDC_JDB_RCV_SEL_H[01:00] generates BUF0_CMDTAG_ENA_H, BUF1_CMDTAG_ENA_H, BUF0_BOD_PEND_H, and BUF1_BOD_PEND_H.
- Send data control to enable buffer data onto the differential lines.
- Transmit buffer to control unloading the buffers.
- Data from the DSXX MCAs.
- Physical address from the ADRX MCAs.
- Clocks.

The JDBX MCA sends the following:

- XJA command, address, and data.
- Fatal error to JDCX MCA when detecting any of the following parity errors:
  - — Hold data switch parity error (DSW_PE_H)
  - — JDCX control parity error (JDC_CTL_PE_H)
  - — Tag address parity errors (TAG_JDBX_PE74_H, TAG_JDBX_PE30_H)
  - — SPU command, address, and data.

JDBX has the following four pointers (Figure 6–26):

- **Write pointer** — Indirect pointer, points to one of two buffers for the command with data.
- **Read pointer** — Indirect pointer, points to one of two buffers to write the data corresponding to the command and address already loaded there.
- **Pointer 1** — Points to the buffer for one of two data commands.
- **Pointer 0** — Points to the buffer for one of two data commands.

The CCU sends two commands to JDBX. Each time JDBX receives a command, it uses the write pointer to determine where to write the command and address, with or without data. If CCU sends the command and address without data, and sends a second command before sending the data for the first command, JDBX uses the write pointer to select the other buffer. JDBX loads the second command and its address into the other buffer.

When CCU sends the data (JDBX receives the data in the same order as the read commands), JDBX uses the read pointer to determine where the data is to be written. Pointer 0 points to the buffer for one command, and pointer 1 points to the buffer for the second command.

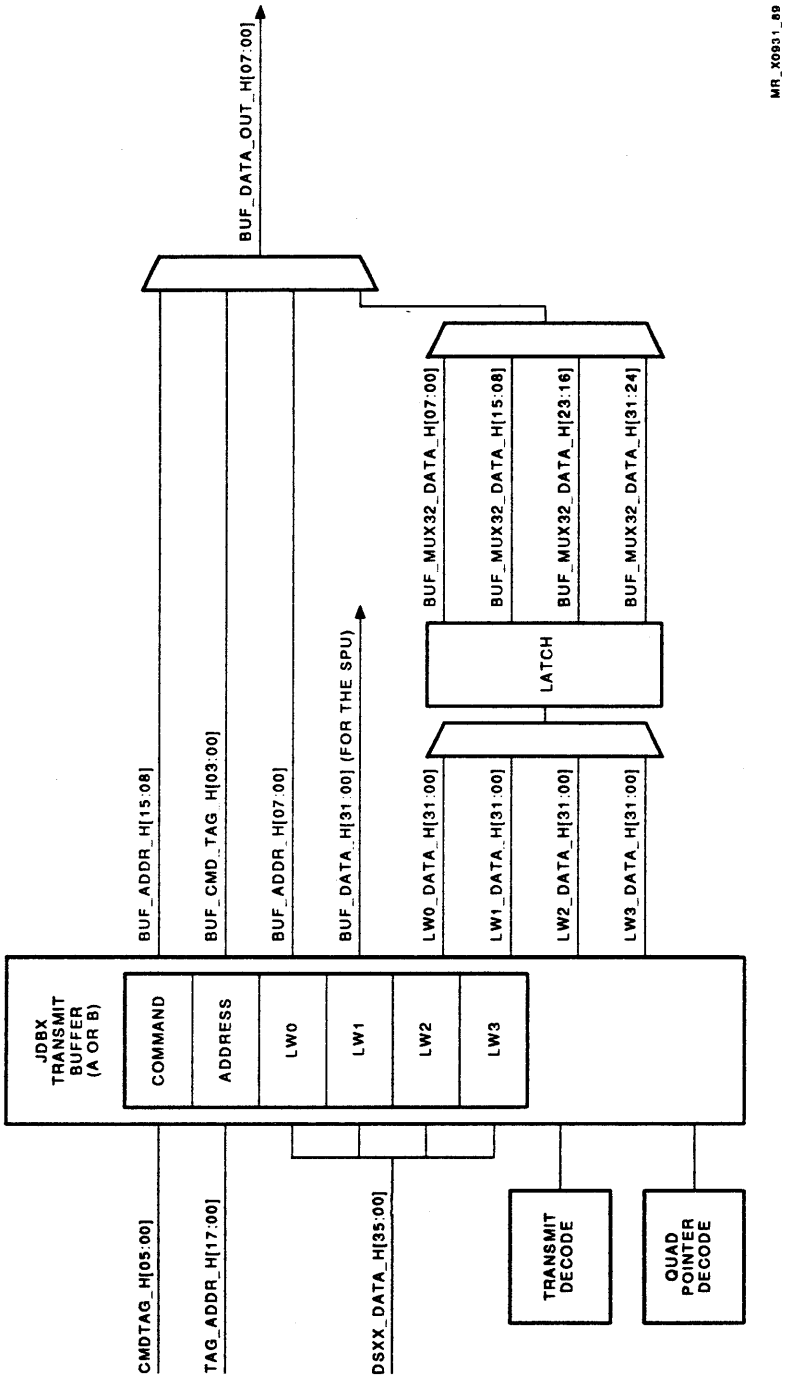


Figure 6–24 JDBX Transmit Buffer

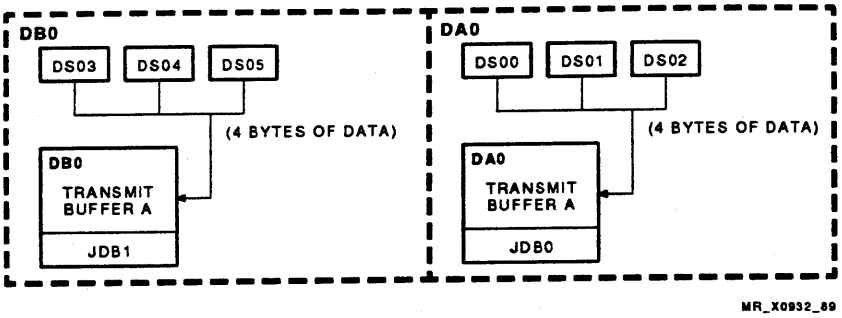


Figure 6–25 Loading Data from the DSXX MCAs

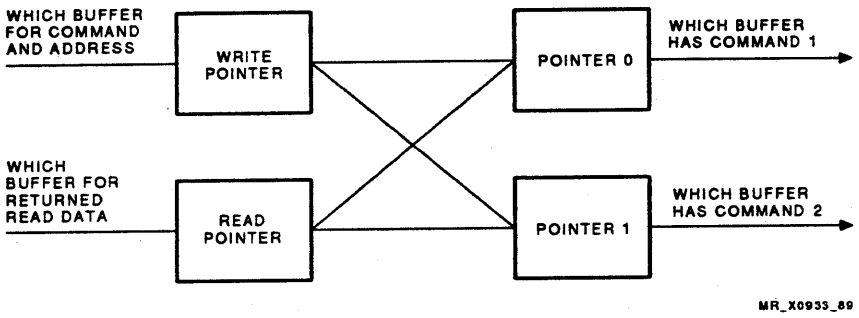


Figure 6–26 Write and Read Pointers

6.3.3.1 Loading a Transmit Buffer for XJA

The pointers are decoded and generate the load buffer 1 and 0 control commands (BUF1_DAT_LD_H, L and BUF0_DAT_LD_H, L). JDCX sends the command JDC_JDB_CMD_TAG_H[05:00]. The data switch sends DSW_JDB_BOD_H and the data, DSW_JDB_DATA_H[35:00]. The ADRX MCAs send TAG_ADDR_H[15:00] to the transmit buffer (Figure 6–23).

The JDBX buffer A and B load counters send the following:

- BUF0_DAT_LD_H starts the load counter for buffer 0. BUF0_LOAD_CNT_TB_H[01:00] is decoded as BUF0_QW_LOAD_L[03:00]. BUF1_DAT_LD_H starts the load counter for buffer 1. BUF1_LOAD_CNT_TB_H[01:00] is decoded as BUF1_QW_LOAD_L[03:00].
- The output, BUF0_DATA_TB_H[07:00] and BUF1_DATA_TB_H[07:00], along with parity, are sent to XJA0 and XJA1 through the differential transceivers.

The following steps summarize loading buffer 0:

1. JDC_JDB_RCV_SEL_H[01:00] controls the loading of the buffer. Each buffer receives the data from the data switch, DSXX_DATA_H[31:00], address from the tag address buffer, TAG_ADDR_H[15:00], and command from JDCX CMDTAG_H[05:00].
2. JDC_JDB_RCV_SEL_H[01:00] starts the load counter, generates BUFn_CMDTAG_ENA_H, L to load the command and tag address, and generates BUFn_QW_LOAD_L[03:00] to load the DSXX_DATA_H[31:00].

6.3.3.2 Unloading a Transmit Buffer for XJA

The following steps summarize unloading buffer 0 and sending the data to XJA0 and XJA1 (Figure 6–23):

1. JDBX decodes JDC_JDB_XMIT_BUF0_H[02:00] and JDC_JDB_XMIT_BUF1_H[02:00], starts the counter, and generates BUF_CAD_SEQ_TB_H[01:00], which selects:
   a. Command
   b. Address
   c. Data
2. JDCX sends command available to XJA. In the next cycle, JDBX begins to send the packet.
3. JDC_JDB_SEND_XJAn_H[01:00] enables the BUF0_DAT_TB_H[07:00] through the differential transceivers to the JDB_XJAn_DATA_H[07:00] lines.
4. JDCX waits for retry or acknowledge.
5. JDCX waits for buffer empty.

6.3.3.3 Unloading a Transmit Buffer for SPU

The following steps summarize unloading the transmit buffer for the SPU (Figure 6–23):

1. JDBX decodes JDC_JDB_SEND_SPU_H[02:00], starts the counter, and generates SPU_BUF1_H, which selects:
   a. Command, TO_SPU_CMD_H[03:00]
   b. Address, TO_SPU_ADDR_H[15:00]
   c. Data, TO_SPU_DATA_H[31:00]
2. JDC_JDB_SEND_SPU_H[01:00] enables the SPU command, address, and data to JDB_CTLD_DATA_H[03:00].

6.4 JBox-to-ICU Interface

Table 6–8 lists the JBox commands and their descriptions. Figure 6–27 shows which commands the JBox sends to XJA, SPU, or IRCX.

Table 6–8 JBox-to-ICU Commands

Code	Command	Description
0	CPU read	JBox sends the CPU read to access registers in the XJAs, SPU, and IRCX MCA. One of four CPUs or SPU can read a register in XJA, SPU, or IRC. CCU_DAX_IOSEL[01:00] indicates where this request is to be sent.
1	–	Reserved.
2	DMA read data return	JBox sends the DMA return read data command in response to a previous DMA read request from XJA or SPU. CCU_DAX_ID comes with the load command so that the JDCX MCA can send the data to the XJA or SPU that made the request.
3	DMA read lock data return	JBox sends the DMA read lock data return command in response to a previous DMA read lock request from an XJA or SPU. CCU_DAX_ID comes with the load command so that JDCX MCA sends the data to the XJA or SPU that made the request.
4	CPU write	JBox sends the CPU write command to write data into registers in XJAs, SPU, and IRCX. It can be longword or byte access. CCU_DAX_IOSEL_H[01:00] indicates to which XJA this request is sent.
5	–	Reserved.
6	SPU read data return	JBox sends the SPU read data return command in response to a previous SPU read I/O register request.
7	Incomplete I/O read	JBox sends the incomplete I/O read command when either IPAMM detects a nonexistent memory condition for an SPU read I/O request (ICU forwards it to SPU) or an XJA responds with read error status to ICU for an SPU read I/O request.
8	Register 0 write to SPU	JBox sends the register 0 write to SPU command when SCU sends memory ECC information to SPU.
9	–	Reserved.
10	DMA read nonexistent memory	JBox sends the DMA read nonexistent memory command when MPAMM detects a nonexistent memory condition for a DMA read request. ICU forwards this command to the requester.
11	–	Reserved.
12	DMA lock denied	JBox sends the DMA lock denied command when the memory data cannot be obtained for a previous DMA read lock command in which the requested data was previously locked.
13–15	–	Reserved.

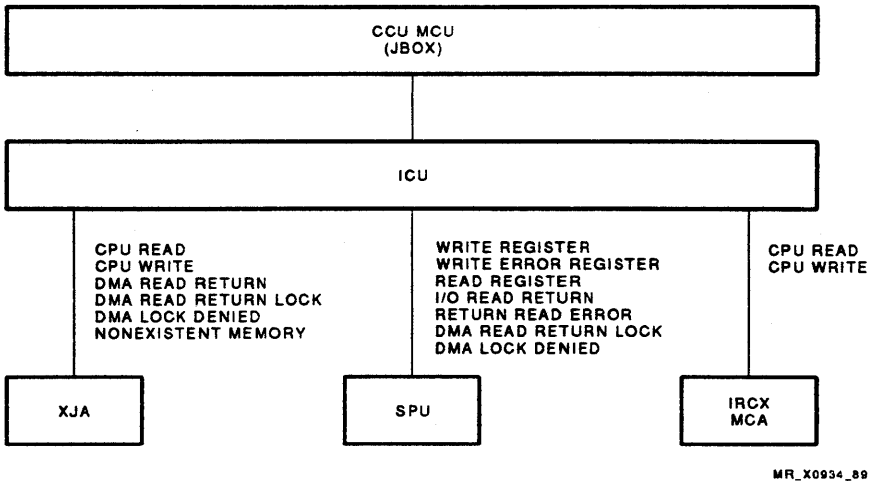


Figure 6–27 CCU Command Summary

6.5 ICU-to-JBox Interface

Table 6–9 lists the ICU-to-JBox commands.

Table 6–9 ICU-to-JBox Commands

Code	Command	Description
0	DMA read	ICU sends the DMA read command to read data from main memory.
1	DMA read lock	ICU sends the DMA read lock command to read and interlock data from main memory.
2	CPU read data return	ICU sends the CPU read data return command in response to a previous CPU read for an SPU, XJA, or IRCX register read. ICU sends the contents of the I/O register to CCU.
3	SPU write unlock	ICU sends the SPU write unlock command to write and unlock a main memory address previously locked by any port.
4	DMA write	ICU sends the DMA write command to write 8 or 16 bytes of data.
5	DMA write unlock	ICU sends the DMA write unlock command to write and unlock a memory address. ICU sends 8 or 16 bytes and unlocks an address requested by a previous DMA read lock request.
6	Incomplete I/O read	ICU sends the incomplete I/O read command if an XJA responds with read error status to ICU for CPU read request. The I/O register data is not obtained for the initial CPU read request command.
7	I/O register response	ICU sends the I/O register response command to inform CCU that the last CPU write request has been completed.
8	SPU read I/O	The ICU sends SPU read I/O command to access I/O registers in the XJAs or IRCX MCA.

Table 6–9 (Cont.) ICU-to-JBox Commands

Code	Command	Description
9	SPU write I/O request	ICU sends the SPU write I/O request to write data into registers in an XJA or IRCX.
10	IRCX return data	ICU sends the IRCX return data command, with the contents of an IRCX register, to CCU in response to a previous CPU read or SPU read I/O register command.
11–15	–	Reserved.

6.6 XJA and ICU Communication Using Packets

An XJA packet is the basic unit of data transfer on JXDI. XJA sends and receives a complete packet (16 bits or 1 word) in each JXDI cycle. The number of cycles depends on the type of command and the data context (length field — quadword or octaword) in word 0.

Table 6–4 lists the number of bytes and cycles for quadword and octaword packets. DMA packets have one command cycle and two address cycles. The mask field determines the number of data cycles. CPU packets have one command cycle, two address cycles, one mask cycle, and four data cycles.

In JXDI cycle 1, the command, length, IPL, sequence, and ID fields are transferred in word 0. In JXDI cycles 2 and 3, the first and then second halves of the address are transferred in words 2 and 3. In JXDI cycle 4, the mask bits are transferred in word 4. In the remaining JXDI cycles, the data is transferred.

6.6.1 JXDI Cycle 1

Figure 6–28 shows the fields sent in the first JXDI cycle. Tables 6–14 and 6–16 list the ICU and XJA commands.

6.6.1.1 Command Field Coding

Figure 6–28 shows the coding of the command field.

6.6.1.2 Length Field Coding

Figure 6–28 shows the coding of the length field. Table 6–10 lists the possible JXDI length codes encoded in the DATA_H[05:04] field in word 0.

Longword-length XMI transactions that reference main memory are translated into quadword transactions. All CPU-type JXDI transactions are longword in length and do not have the length field defined.

WORD	BIT 15 14 13 12 11 10 09 08	BIT 07 06 05 04 03 02 01 00
0	R R ID	R S LEN CMD
1	A [29:26, 05:02]	A [13:10] [09:06]
2	A [33:30, 25:22]	A [21:18] [17:14]
3	F E D C 7 6 5 4	B A 9 8 3 2 1 0
4	BYTE 4	BYTE 0
5	BYTE 5	BYTE 1
6	BYTE 6	BYTE 2
7	BYTE 7	BYTE 3
8	BYTE C	BYTE 8
9	BYTE D	BYTE 9
10	BYTE E	BYTE A
11	BYTE F	BYTE B

R = RESERVED
S = SEQUENCE BIT NOT USED

MR_X0935_89

Figure 6–28 JXDI Cycle 1

Table 6–10 JXDI Length Codes

Code	Size
0 0	Hexword (32 bytes, reserved)
0 1	Block (64 bytes, reserved)
1 0	Quadword (8 bytes)
1 1	Octaword (16 bytes)

6.6.1.3 IPL Field Coding

Figure 6–57 shows the IPL field in the interrupt packet. During interrupt-type JXDI transactions, XJA encodes DATA_H[07:04], as shown in Table 6–11.

Table 6–11 IPL Priority Level

[05:04]	IPL (Hex)
0 0	14
0 1	15
1 0	16
1 1	17

6.6.1.4 ID Field Coding

Figure 6–28 shows the ID field. The DATA_H[13:08] field of word 0 contains a unique ID that identifies the source of the request. ICU receives a DMA command and stores the ID field in the JDCX MCA. JDCX sends a single ID bit to the CCU MCU (CTLB MCA). The CTLB MCA stores the command in the I/O command latch. The CCU MCU returns the ID bit to ICU (JDCX MCA) with the DMA returned data read. ICU can receive two DMA read requests. The ID bit (CCU_ID_H) specifies which of the two DMA read commands corresponds to the returned read data. ICU sends one of two ID fields to the transmit buffer (in the JDBX MCA). Figure 6–29 shows how the ID is passed from XJA to ICU.

During DMA-type JXDI transactions, DATA_H[13:08] contains the value of the XMI_ ID[05:00] field. This field is encoded as follows:

- DATA_H[13:10] identifies the XMI node ID.
- DATA_H[09:08] identifies one of four outstanding commands.

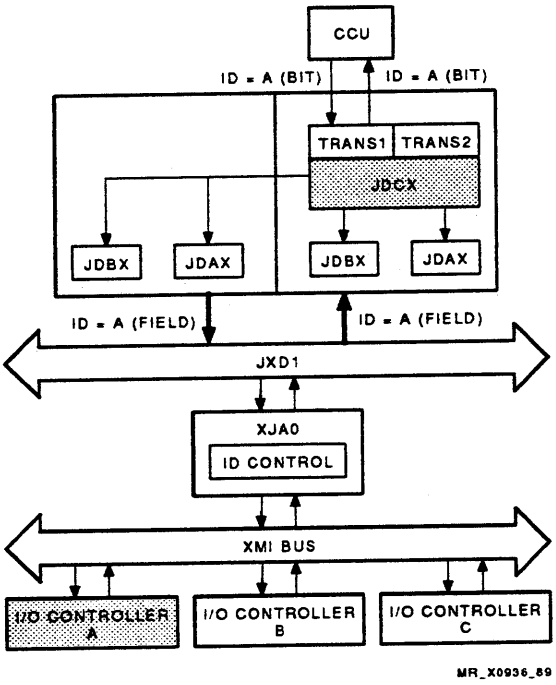


Figure 6–29 ID Field Coding

6.6.2 JXDI Cycles 2 and 3

This section describes cycles 2 and 3 of a JXDI transfer.

6.6.2.1 Address Field Coding

Figure 6–30 shows the address fields.

The JBox supports wrapped read requests on quadword boundaries.

For longword-length transactions, ADDR[01:00] is only significant when dealing with a VAXBI word-mode or byte-mode transaction in I/O space. ADDR[01] is required for word mode, and ADDR[01:00] is required for byte mode.

Quadword and octaword write transfers are quadword-aligned, and the lower bits of the address are ignored.

Reads, however, use the lower three bits of the address when main memory does a wraparound read on a quadword boundary. Wraparound reads allow the SCU to return the requested quadword first, followed by the remaining quadwords in the block.

On longword reads to I/O space, ADDR[01] is used for explicit read word functions on the XMI bus. When a read is directed toward a word-oriented device, ADDR[01] specifies which word is read from the XMI device. If ADDR[01] is set, the high word, D[31:16], is read. If ADDR[01] is zero, the low word, D[15:00], is read.

The data returned on the opposite word of that specified has correct parity, although its data is unspecified. In the case of a longword-oriented device, ADDR[01] is ignored. An address bit and a full longword of data are returned for a read operation.

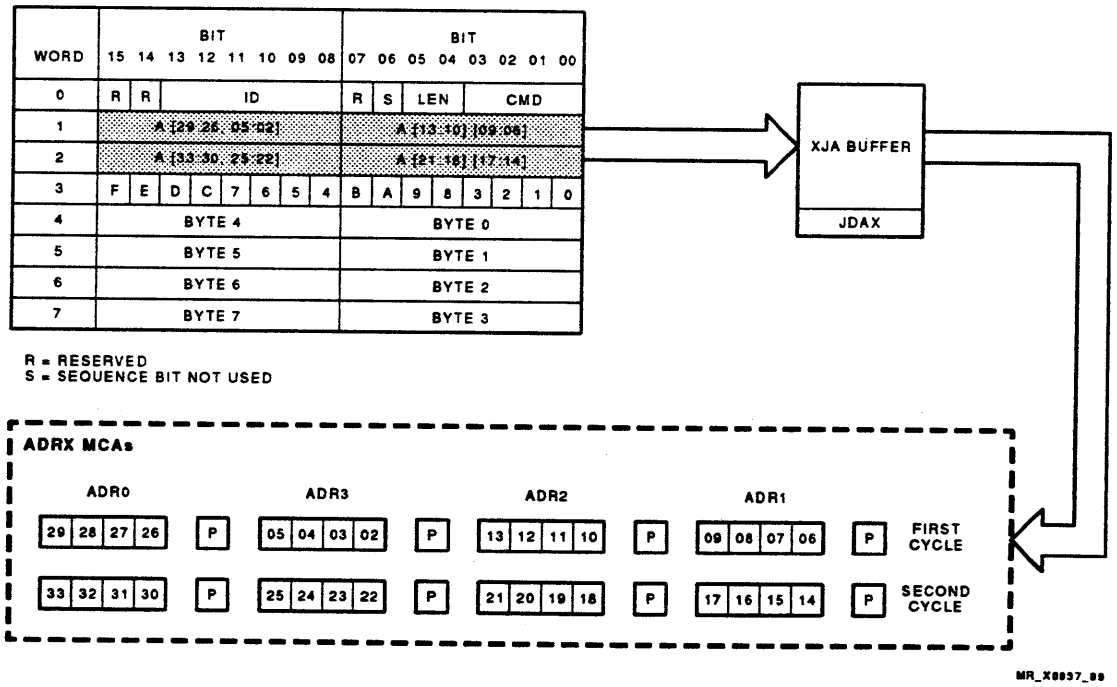


Figure 6–30 Address Field Coding

XJA asserts ADDR[01:00] based on the contents of the mask field in the DATA_H[15:12] field of word 2 of the CPU-type transaction as listed in Table 6–12. The mask field is valid and significant for both CPU write- and read-type transactions.

Table 6–12 Mask Field

Mask Bit	XMI Address [01:00]
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

6.6.3 JXDI Cycle 4

This section describes cycle 4 of a JXDI transfer.

6.6.3.1 Mask Field Coding

Word 3 of a DMA write packet contains the byte mask bits for the write data. Bits [00], [01], [02], and [03] correspond to bytes 0, 1, 2, and 3. Bits [04], [05], [06], and [07] correspond to bytes 8, 9, A, and B. Figure 6–31 shows the mask field. Figure 6–32 shows how the MMCX MCA receives the mask bits from the data switch.

When DMA write transactions are less than 16 bytes in length, the unused mask bits are deasserted. In all packets, the JXDI parity bits reflect the correct parity over the JXDI data fields.

WORD	BIT 15 14 13 12 11 10 09 08	BIT 07 06 05 04 03 02 01 00
0	R R ID	R S LEN CMD
1	A [29:26, 05:02]	A [13:10] [09:06]
2	A [33:30, 25:22]	A [21:18] [17:14]
3	F E D C 7 6 5 4	B A 9 8 3 2 1 0
4	BYTE 4	BYTE 0
5	BYTE 5	BYTE 1
6	BYTE 6	BYTE 2
7	BYTE 7	BYTE 3
8	BYTE C	BYTE 8
9	BYTD D	BYTE 9
10	BYTE E	BYTE A
11	BYTE F	BYTE B

R = RESERVED
S = SEQUENCE BIT NOT USED

MR_X0938_89

Figure 6–31 Mask Field

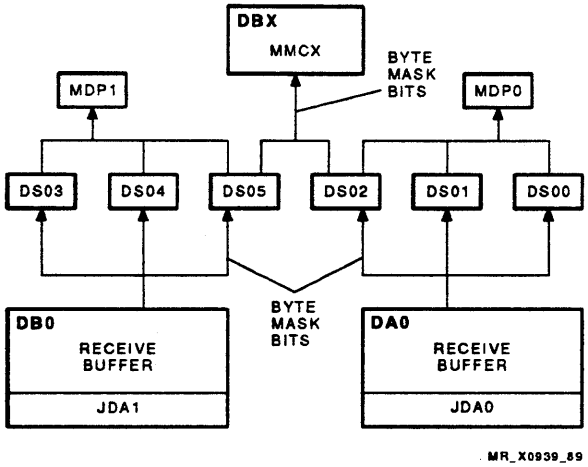


Figure 6–32 Mask Field to MMCX MCA

The DATA_H[15:12] field of word 2 of a CPU write transaction contains the byte mask for the longword of write data in the packet. DATA_H[15] is the mask bit for byte 3, [14] for byte 2, [13] for byte 1, and [12] for byte 0.

6.6.4 JXDI Cycle 5

This section describes cycle 5 of a JXDI transfer.

6.6.4.1 Data Field Coding

Figure 6–33 shows how longword 0 is encoded in the DMA write packet.

WORD	BIT 15 14 13 12 11 10 09 08	BIT 07 06 05 04 03 02 01 00
0	R R ID	R S LEN CMD
1	A [29:26, 05:02]	A [13:10] [09:06]
2	A [33:30, 25:22]	A [21:18] [17:14]
3	F E D C 7 6 5 4	B A 9 8 3 2 1 0
4	BYTE 4	BYTE 0
5	BYTE 5	BYTE 1
6	BYTE 6	BYTE 2
7	BYTE 7	BYTE 3
8	BYTE C	BYTE 8
9	BYTD D	BYTE 9
10	BYTE E	BYTE A
11	BYTE F	BYTE B

R = RESERVED
S = SEQUENCE BIT NOT USED

MR_X0940_89

Figure 6–33 Data Field Coding

Figure 6–34 shows how the quadwords are encoded in the packet.

Figure 6–35 shows how the data is sent to the data switch MCAs.

Figure 6–36 shows how JDAX sends the data to the data switch MCA data-slices. Each DSXX MCA receives slices of the data, beginning of the data bits, mask bits [03:00], and longword parity bits coming from the JDAX MCA.

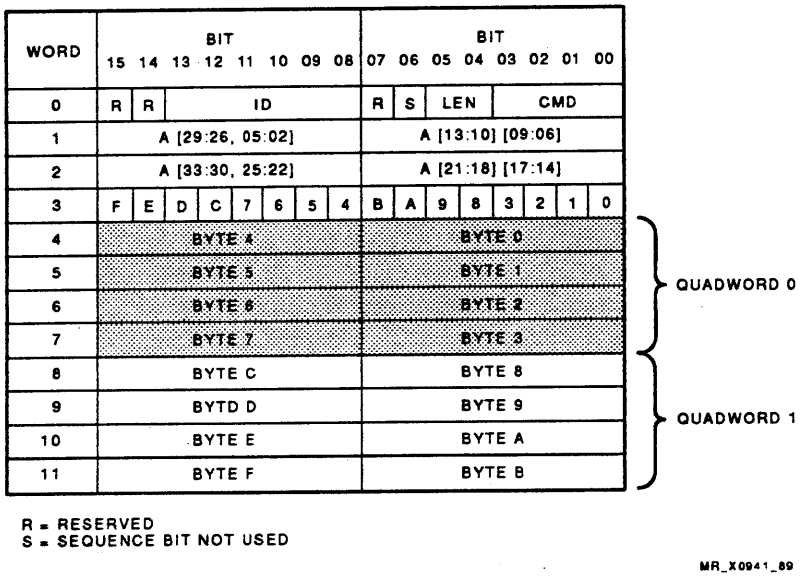


Figure 6–34 Quadword Format

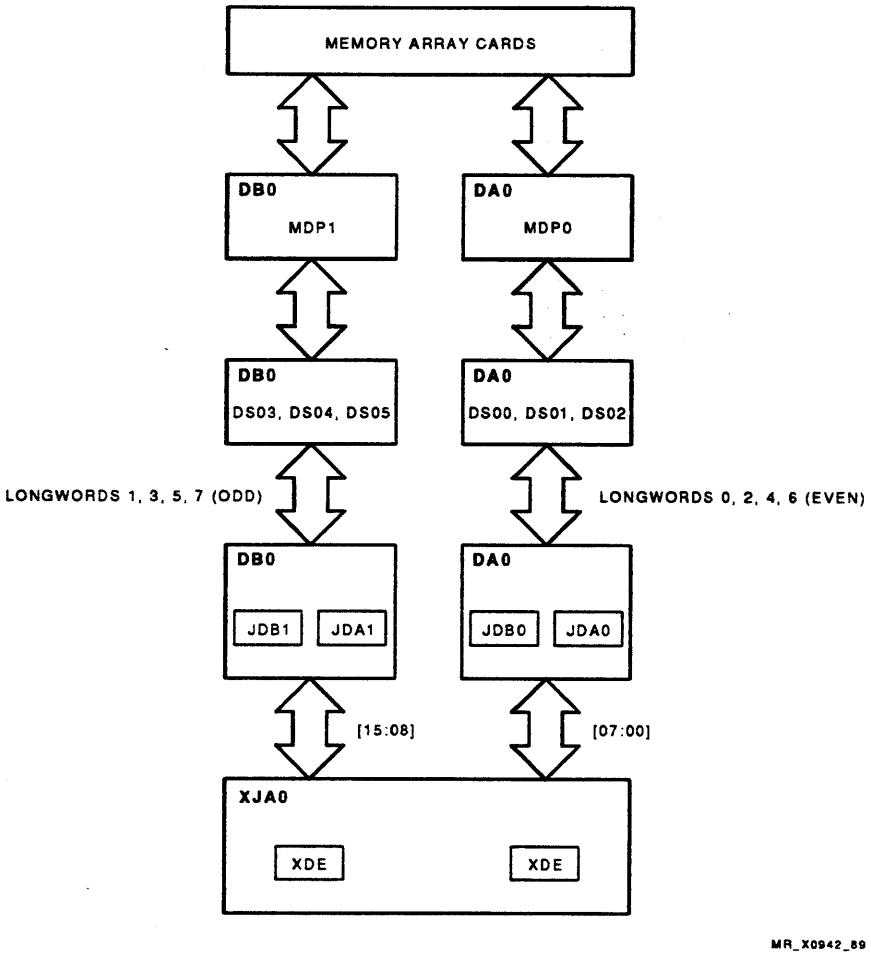


Figure 6–35 Data Path from the Receive Buffer to the Data Switch

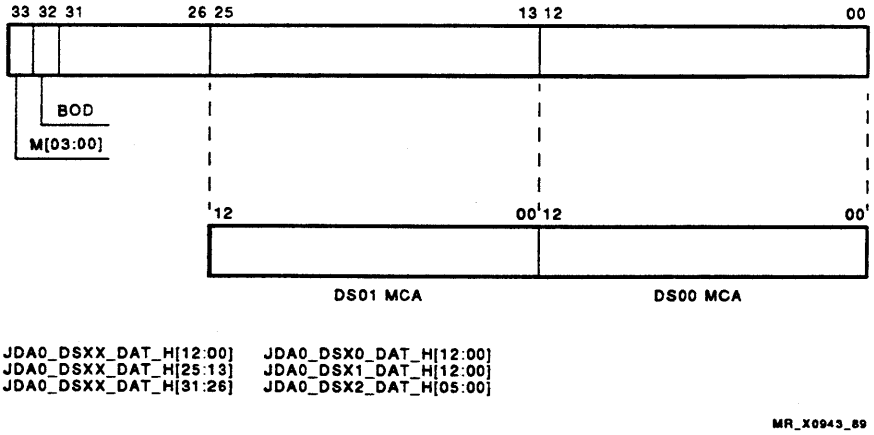


Figure 6–36 Data Path from JDAX to the Data Switch

Figure 6–37 shows how the data switch MCAs send data to JDBX. Each DSXX MCA sends slices of the data, beginning of the data bits, mask bits [03:00], and longword parity bits to the JDBX MCA.

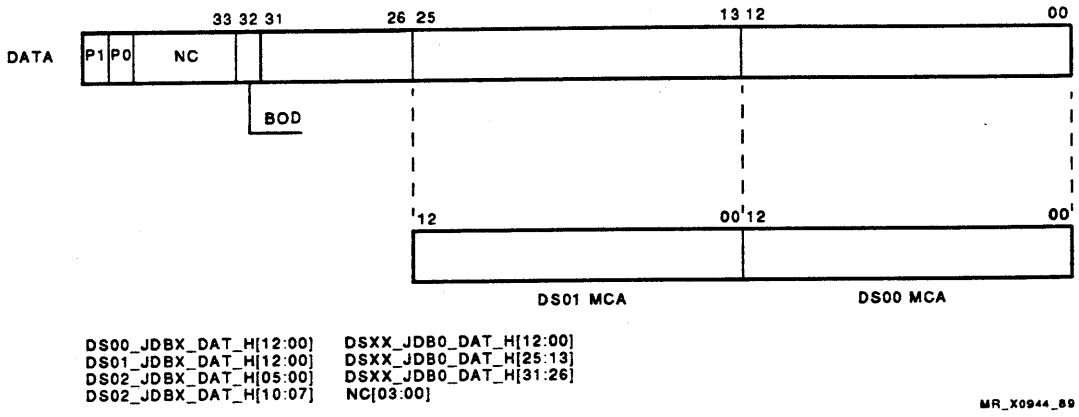


Figure 6–37 Data Path from Data Switch to JDBX

6.7 ICU-to-XJA Interface

Figure 6–38 shows the ICU-to-XJA handshaking signals.

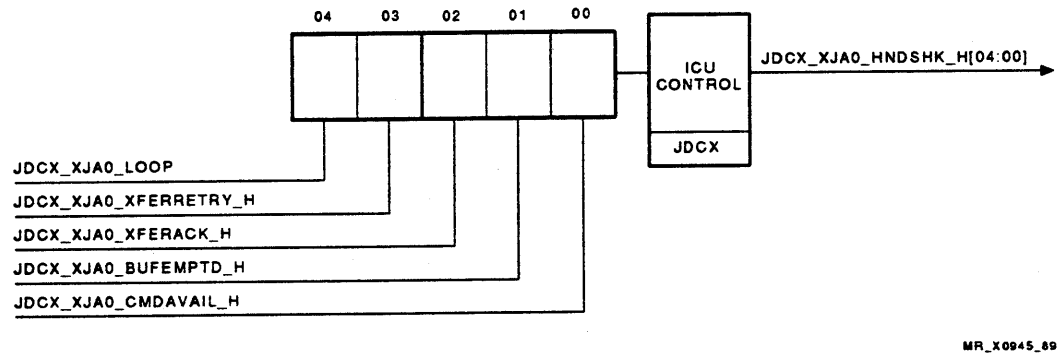


Figure 6–38 ICU-to-XJA Handshaking Signals

6.7.1 Key Signals

Table 6–13 lists the ICU-to-XJA signals.

Table 6–13 ICU-to-XJA Signals

Signal	Description
SPU_XJA_STOPPED_L	This signal is generated by SPU and is sent to XJA through the ICU. It informs XJA of impending clock stoppage. Upon receiving this signal, XJA completes the current JXDI transmission (if any) and does not initiate new transmissions until the signal is negated. XJA_SPU_STOPPED_L informs SPU that XJA is quiet and that clocks can be stopped.
ICU_DAT_H[15:00]	This is the data word transferred from ICU to XJA that carries command, address, and data information.
ICU_PAR_H[01:00]	This field is the odd parity over ICU_DATA_H[15:00]. PAR_H[00] corresponds to DATA_H[07:00], and PAR_H[01] corresponds to DATA_H[15:08]. A parity bit is asserted when the number of bits in the data field is even.
ICU_CMDAVAIL_H	This signal is asserted on the cycle before ICU starts transmitting a transaction packet of information. ICU asserts ICU_CMDAVAIL only if an XJA buffer is available, as indicated by XJA_BUFEMPTD_H.
ICU_XFERACK_H	This signal is asserted following the last transfer to indicate that an XJA-to-ICU transaction packet was received without error. This signal allows XJA to purge the corresponding transmit buffer.
ICU_XFERRETRY_H	This signal is asserted following the last transfer of a transaction. This indicates that a parity error was detected by the ICU receive logic during an XJA-to-ICU transaction and that a retry is required.
ICU_BUFEMPTD_H	This signal indicates that ICU has successfully emptied a JXDI receive buffer by sending the transaction to the JBox. ICU asserts this signal for exactly one JXDI cycle every time it has emptied a JXDI receive buffer.
ICU_CLKJ_H[02:00]	This signal is a system clock, a 50% duty cycle clock signal that ICU sends to XJA for receiving the JXDI data wires.
ICU_LOOP_H	This signal, when asserted, forces XJA to loopback directly all JXDI signals sourced by ICU. For example, ICU_CLKJ_H[02] loops as XJA_CLKX_H[02]. In addition, SPU_RESET_H loops to become XJA_FATALERR_H.

6.7.2 Commands

Table 6–14 lists the possible JXDI command codes encoded in the DATA_H[03:00] field of word 0 of a JXDI transaction.

Table 6–14 ICU-to-XJA Commands

Code	Command	Description
0 0 0 0	CPU read	ICU sends the CPU read command to access registers in the XJAs. One of the four CPUs or the SPU wants to read a register in the XJA IRCX. The CCU_DAX_IOSEL[01:00] indicates where this request is to be sent.
0 0 0 1	–	Reserved.
0 0 1 0	DMA read data return	ICU sends a DMA read data return command in response to a previous DMA read request from an XJA.
0 0 1 1	–	Reserved.
0 1 0 0	CPU write	ICU sends the CPU write command to write to registers in the XJAs. It can be a longword or byte access. The CCU_DAX_IOSEL_H[01:00] indicates to which XJA this request is to be sent.
0 1 0 1	–	Reserved.
0 1 1 0	–	Reserved.
0 1 1 1	–	Reserved.
1 0 0 0	–	Reserved.
1 0 0 1	DMA read lock status	ICU sends the DMA read lock status command to notify the XJA or SPU that memory data was not obtained for a previous DMA read lock request.
1 0 1 0	DMA read error status	ICU sends the DMA read error status command when IPAMM detects a nonexistent memory condition.
1 0 1 1	–	Reserved.
1 1 0 0	–	Reserved.
1 1 0 1	–	Reserved.
1 1 1 0	–	Reserved.
1 1 1 1	–	Reserved.

6.7.3 Transferring a Packet from ICU to XJA

Figure 6–39 shows the sequence of steps in the transfer of a packet from ICU to an XJA.

1. JDCX sends command available to the XCE MCA in the XJA.
2. In the next cycle, JDCX sends word 0 of the packet. JDCX continues to transfer each word of the packet until the transfer is complete. JDCX sends parity in each transfer for parity checking. The parity is sent in the same cycle as the data.
3. XCE in the XJA sends transfer acknowledge upon successful receipt of the packet. XCE sends transfer retry if the XJA was busy transmitting a packet or if a parity error was detected.
4. XJA unloads its receive buffer, sends its contents to an XMI device, and sends buffer empty to ICU.

Figure 6–40 shows the ICU-to-XJA control interface.

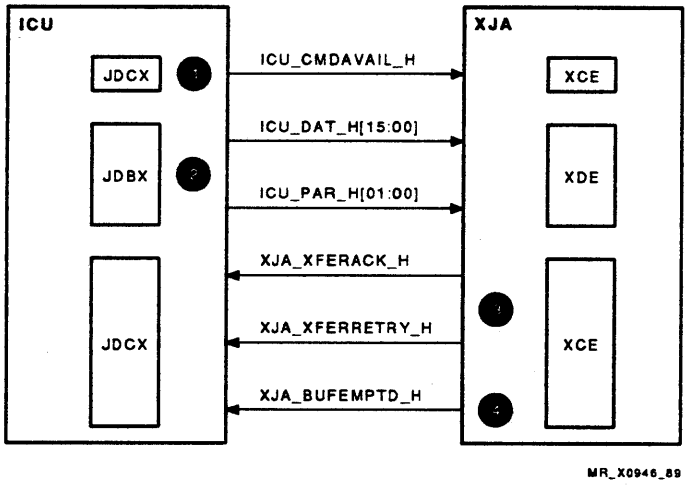


Figure 6–39 Transferring a Packet from the ICU to XJA

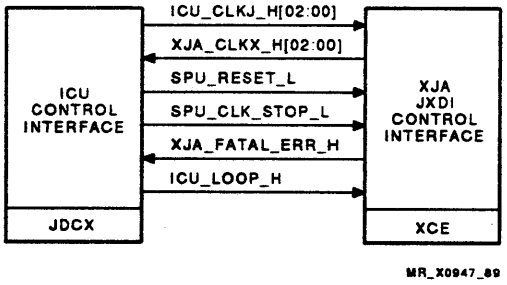


Figure 6–40 ICU-to-XJA Control Interface

6.8 XJA-to-ICU Interface

Figure 6–41 shows the XJA-to-ICU handshaking signals.

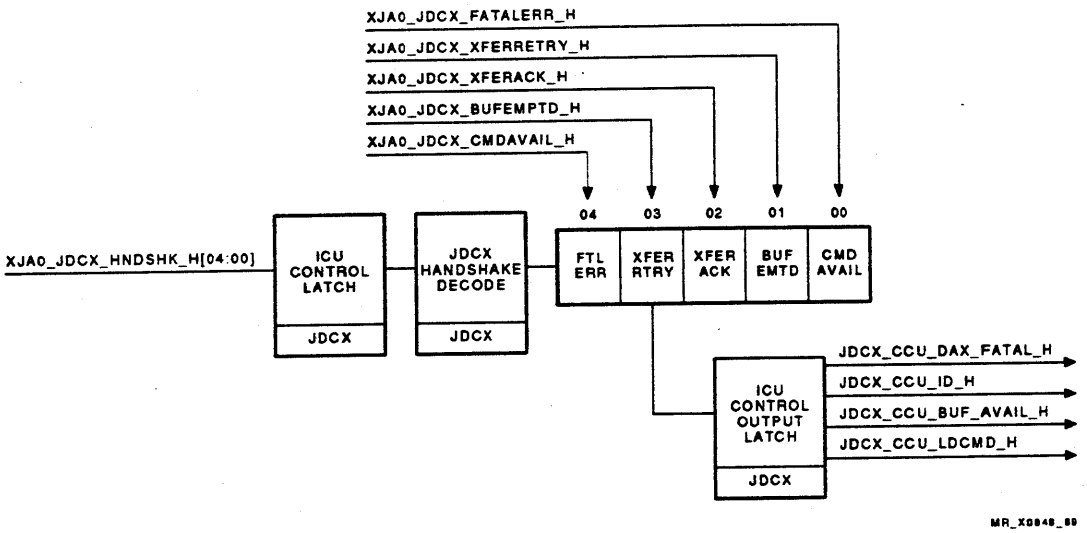


Figure 6–41 XJA-to-ICU Handshaking Signals

6.8.1 Key Signals

Table 6–15 lists the signals from XJA to ICU.

Table 6–15 XJA-to-ICU Signals

Signal	Description
XJA_DAT_H[15:00]	This is the data word transferred from XJA to ICU that carries command, address, and data information.
XJA_PAR_H[01:00]	This field is the odd parity over XJA_DATA_H[15:00]. XJA_PAR_H[00] corresponds to XJA DATA_H[07:00], and XJA_PAR_H[01] corresponds to XJA DATA_H[15:08]. A parity bit is asserted when the number of bits asserted in the data field is even.
XJA_CMDAVAIL_H	This signal is asserted on the cycle before XJA starts transmitting a packet of information to ICU. XJA asserts XJA_CMDAVAIL only if an ICU buffer is available, as indicated by ICU_BUFEMPTD_H.
XJA_XFERACK_H	This signal is asserted following the last transfer to indicate that an ICU-to-XJA transaction packet was received without error. This signal allows ICU to purge the corresponding transmit buffer, if XJA_XFERACK_H is asserted.
XJA_XFERRETRY_H	This signal is asserted following the last transfer of a transaction. This indicates that a parity error was detected by the XJA receive logic during an ICU-to-XJA transaction and that a retry is required.
XJA_BUFEMPTD_H	This signal indicates that XJA has successfully emptied a JXDI receive buffer either by initiating the appropriate XMI transaction or by carrying out the specified internal operation (register write, and so on). XJA asserts this signal for exactly one JXDI cycle every time it empties a JXDI receive buffer.

Table 6–15 (Cont.) XJA-to-ICU Signals

Signal	Description
XJA_SPU_STOPPED_L	This signal is a response to SPU_CLKSTOP_L. XJA sends XJA_SPU_STOPPED_L when the current transaction is completed. SPU does not stop the ICU clocks until XJA_SPU_STOPPED_L is received.
XJA_FATALERR_H	This signal, when asserted, indicates to ICU that this XJA has detected a fatal error and may not be capable of responding to further CPU requests.

6.8.2 Commands

Table 6–16 lists the possible JXDI command codes encoded in the DATA_H[03:00] field of word 0 of a JXDI transaction. Figure 6–42 shows the commands that XJA sends to the CCU and IRCX MCA.

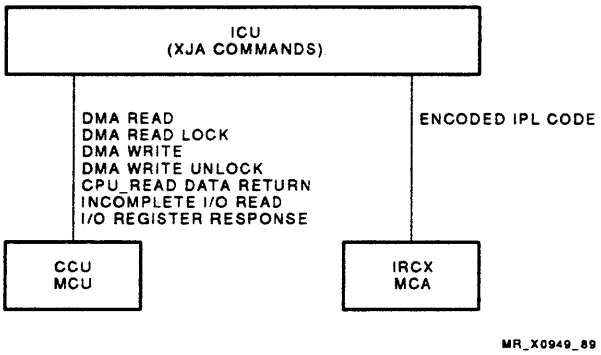


Figure 6–42 XJA Command Summary

Table 6–16 XJA-to-ICU Commands

Code	Command	Description
0 0 0 0	DMA read	XJA sends the DMA read command to read data from main memory.
0 0 0 1	DMA read lock	XJA sends the DMA read lock command to read data and to interlock the address in main memory.
0 0 1 0	CPU read data return	XJA sends the CPU return read data command with the data, in response to a previous CPU read request or SPU read I/O register command.
0 0 1 1	–	Reserved.
0 1 0 0	DMA write	XJA sends the DMA write command to write 8 or 16 bytes to main memory.
0 1 0 1	DMA write unlock request	XJA sends the DMA write unlock request to write and unlock an address in main memory. XJA can send 8 or 16 bytes of data.
0 1 1 0	–	Reserved.
0 1 1 1	–	Reserved.
1 0 0 0	Interrupt request	XJA sends an interrupt request and corresponding IPL to the IRCX MCA. IRCX sends the interrupt to the arbiter. If the interrupt has the highest priority, IRCX sends an interrupt code to the EBox, where the corresponding interrupt service routine is executed.
1 0 0 1	–	Reserved.
1 0 1 0	CPU read error status	XJA sends a CPU read error status command that contains a valid XMI read error, an XMI timeout, or a read XJA register error for an SPU read I/O request, or CCU sends a CPU read error status for an incomplete I/O read.
1 0 1 1	CPU write complete	ICU sends the CPU write complete command to the CCU MCU upon the completion of a CPU write request.
1 1 0 0	–	Reserved.
1 1 0 1	–	Reserved.
1 1 1 0	–	Reserved.
1 1 1 1	–	Reserved.

6.8.3 Transferring a Packet from an XJA to ICU

Figure 6–43 shows the sequence of steps in the transfer of a packet from an XJA to ICU.

1. XCE MCA sends command available to ICU.
2. In the next cycle, XCE in the XJA sends word 0 of the packet. XCE continues to transfer each word of the packet until the packet transfer is complete. XCE sends parity for the parity checking of each data transfer.
3. JDCX sends transfer acknowledge upon successful receipt of the packet. JDCX sends transfer retry if ICU or JBox detects a parity error.
4. JDCX unloads its receive buffer and sends buffer empty to XJA.

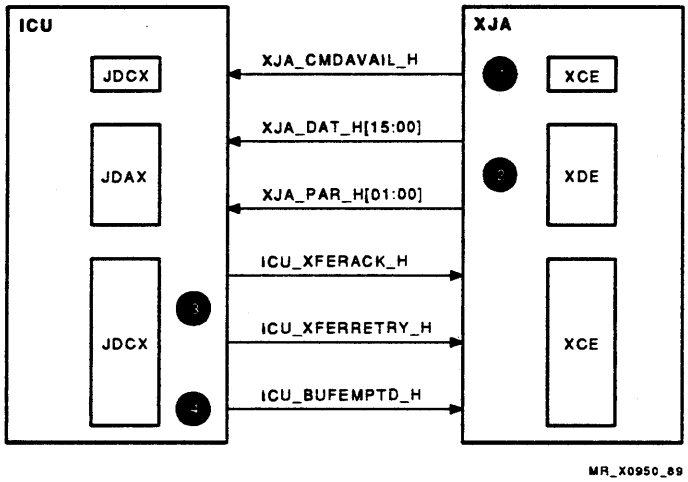


Figure 6–43 Transferring a Packet from XJA to ICU

6.9 XJA Transactions

This section describes the sequences for CPU, DMA, and interrupt transactions.

6.9.1 CPU Read Transaction

The following steps summarize a CPU read operation in which the CPU reads the contents of an I/O register in the XJA module. Figure 6–44 shows the sequence.

❶ EBox sends the I/O register read command, along with the address of the register, to MBox.

❷ MBox determines that the address is for I/O space and sends the I/O register read and the address to JBox (CCU MCU).

❸ CCU sends the CPU read command, the I/O select specifying which XJA, and the address of the register to ICU.

❹ ICU sends the CPU read and address to XJA. Figure 6–45 shows the CPU read packet.

❺ XJA responds with the CPU read data return, address, and contents of the register to ICU. Figure 6–46 shows the CPU read data return packet. If an error occurs, the XJA responds with the CPU read error status. Figure 6–47 shows the CPU read error status packet.

❻ ICU sends the CPU read data return to the CCU MCU.

❼ CCU sends the return I/O register read and the contents of the register to CPU.

❽ MBox receives the command, loads the refill buffer with the data, and sends the data to EBox.

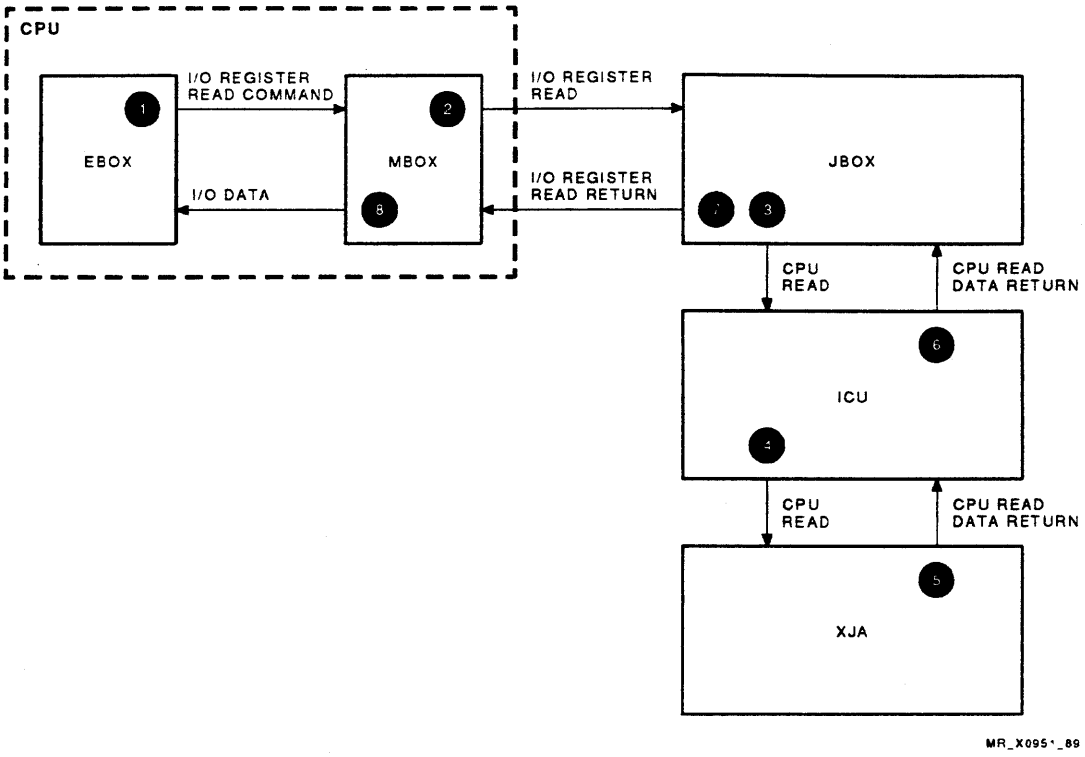


Figure 6–44 CPU Read Operation

WORD	BIT 15 14 13 12 11 10 09 08	BIT 07 06 05 04 03 02 01 00
0	R R ID	R R R R CMD
1	A [29:26, 05:02]	A [13:10] [09:06]
2	MASK* A [25:22]	A [21:18] [17:14]

R = RESERVED
*SPECIFIES ADDRESS BITS [01:00]

MR_X0952_89

Figure 6–45 CPU Read Packet

WORD	BIT 15 14 13 12 11 10 09 08	BIT 07 06 05 04 03 02 01 00
0	R R ID	R R R R CMD
1	RESERVED	RESERVED
2	RESERVED	RESERVED
3	RESERVED	RESERVED
4	RESERVED	BYTE 0
5	RESERVED	BYTE 1
6	RESERVED	BYTE 2
7	RESERVED	BYTE 3

R = RESERVED

MR_X0953_89

Figure 6–46 CPU Read Data Return Packet

WORD	BIT 15 14 13 12 11 10 09 08	BIT 07 06 05 04 03 02 01 00
0	R R ID	R R R R CMD

R = RESERVED

MR_X0954_89

Figure 6–47 CPU Read Error Status Packet

6.9.2 CPU Write Transaction

The following steps summarize a CPU write operation in which CPU writes to an I/O register in the XJA module. Figure 6–48 shows the sequence.

1. EBox sends the I/O register write, address of an XJA register, and data to MBox.
2. MBox sends the I/O register write, address, and data to JBox.
3. CCU sends the CPU write, tag sends the address, and DAX/DBX sends the data to ICU.
4. ICU sends the CPU write, address, and data to XJA. Figure 6–49 shows the CPU write packet.
5. ICU sends the CCU write complete to CCU.

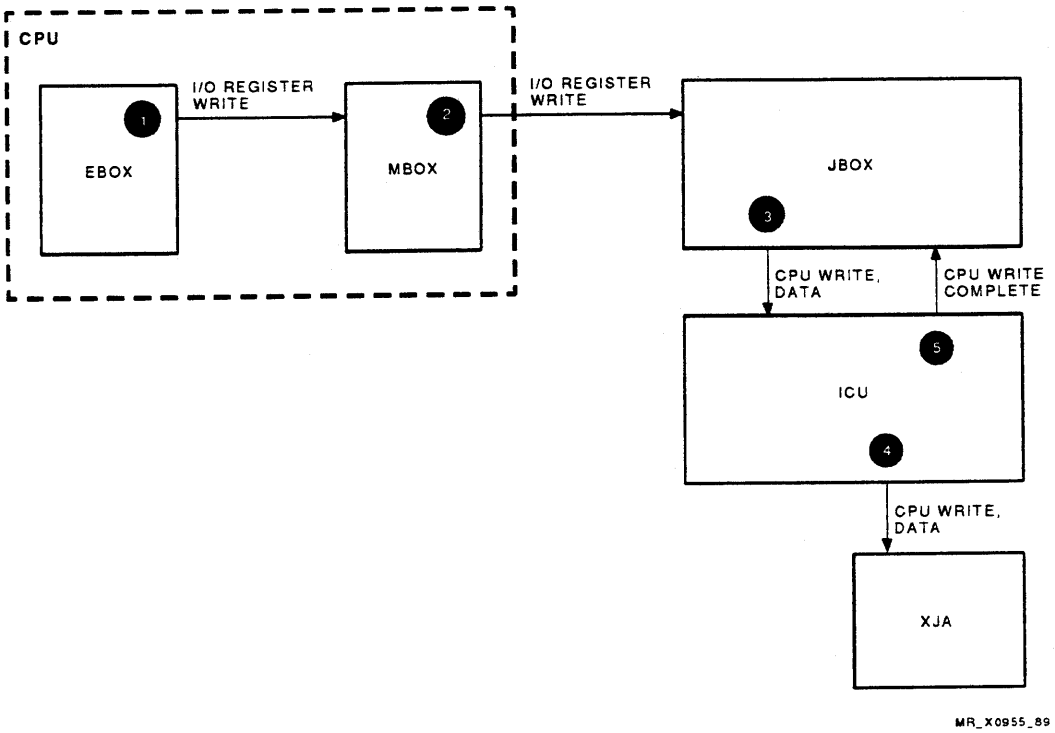


Figure 6–48 CPU Write Operation

WORD	BIT 15 14 13 12 11 10 09 08	BIT 07 06 05 04 03 02 01 00
0	R R ID	R R R R CMD
1	A [29:26, 05:02]	A [13:10] [09:06]
2	MASK A [25:22]	A [21:18] [17:14]
3	0 0 0 0 0 0 0 0	BYTE 0
4	0 0 0 0 0 0 0 0	BYTE 1
5	0 0 0 0 0 0 0 0	BYTE 2
6	0 0 0 0 0 0 0 0	BYTE 3

R = RESERVED

MR_X0956_89

Figure 6–49 CPU Write Packet

6.9.3 DMA Read Transaction

The following steps summarize a DMA read operation in which an XMI device reads data from main memory. The XMI device sends a DMA read command, address, and length to the XJA. Figure 6–50 shows the DMA read sequence.

❶ XJA sends the DMA read, address, and length to ICU. Figure 6–51 shows the DMA read packet.

❷ ICU sends the DMA read and length to the CCU MCU and the address to the tag MCU.

❸ CCU sends the DMA read data return, the tag MCU sends the address, and DAX/DBX sends the data to ICU. CCU sends the DMA read nonexistent memory to ICU if JBox detects a nonexistent memory condition for the address sent with the DMA read command. CCU sends the DMA lock deny to ICU if the address is previously locked by another port.

❹ ICU sends a DMA read data return command and the data to XJA. Figure 6–52 shows the DMA read data return packet.

ICU sends the DMA read nonexistent memory status if the JBox detects a nonexistent memory condition for the address sent with the DMA read command. Figure 6–53 shows the DMA read error status packet.

ICU sends the DMA lock deny status if the JBox determines that the address is locked by another port.

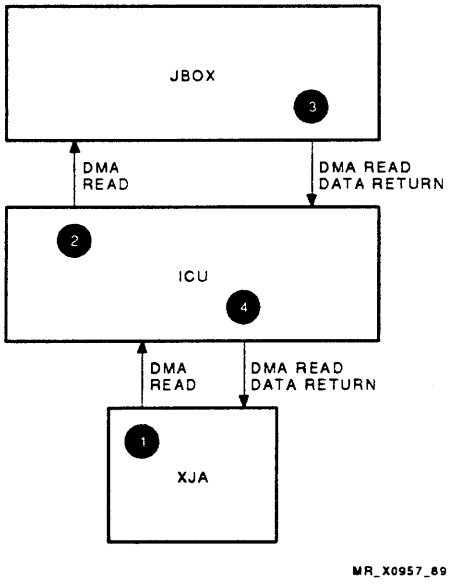


Figure 6–50 DMA Read Operation

WORD	BIT 15 14 13 12 11 10 09 08	BIT 07 06 05 04 03 02 01 00
0	R R ID	R R R R CMD
1	A [29:26, 05:02]	A [13:10] [09:06]
2	MASK* A [25:22]	A [21:18] [17:14]

R = RESERVED
*SPECIFIES ADDRESS BITS [01:00]

MR_X0958_89

Figure 6–51 DMA Read Packet

WORD	BIT 15 14 13 12 11 10 09 08	BIT 07 06 05 04 03 02 01 00
0	R R ID	R S LEN CMD
1	BYTE 4	BYTE 0
2	BYTE 5	BYTE 1
3	BYTE 6	BYTE 2
4	BYTE 7	BYTE 3
5	BYTE C	BYTE 8
6	BYTE D	BYTE 9
7	BYTE E	BYTE A
8	BYTE F	BYTE B

R = RESERVED
S = SEQUENCE BIT NOT USED

MR_X0959_89

Figure 6–52 DMA Read Return Packet

WORD	BIT 15 14 13 12 11 10 09 08	BIT 07 06 05 04 03 02 01 00
0	R R ID	R S R R CMD

R = RESERVED
S = SEQUENCE BIT NOT USED

MR_X0960_89

Figure 6–53 DMA Read Error Packet

6.9.4 DMA Write Transaction

The following steps summarize a DMA write operation in which an XMI device writes data to main memory. The XMI device sends the DMA write, address, and data to XJA. Figure 6–54 shows the sequence for a DMA write operation.

1. XJA sends the DMA write, address, and data to ICU. Figure 6–55 shows the DMA write packet.
2. ICU sends the DMA write to the CCU MCU, the address to the tag MCU, and the data to the DAX/DBX MCUs.

A DMA write unlock operation is similar to a DMA write, except that the previously acquired lock is released for a DMA write command.

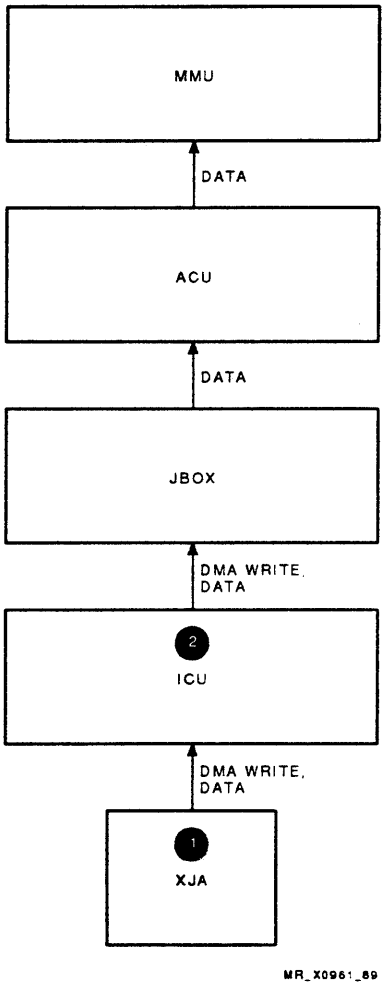


Figure 6–54 DMA Write Operation

WORD	BIT 15 14 13 12 11 10 09 08	BIT 07 06 05 04 03 02 01 00
0	R R ID	R S LEN CMD
1	A [29:26, 05:02]	A [13:10] [09:06]
2	A [33:30, 25:22]	A [21:18] [17:14]
3	F E D C 7 6 5 4	B A 9 8 3 2 1 0
4	BYTE 4	BYTE 0
5	BYTE 5	BYTE 1
6	BYTE 6	BYTE 2
7	BYTE 7	BYTE 3
8	BYTE C	BYTE 8
9	BYTE D	BYTE 9
10	BYTE E	BYTE A
11	BYTE F	BYTE B

R = RESERVED
S = SEQUENCE BIT NOT USED

MR_X0962_89

Figure 6–55 DMA Write Packet

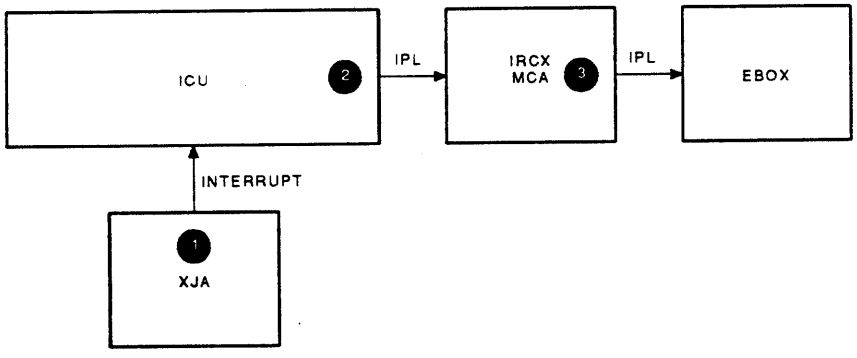
6.9.5 Interrupt Transactions

The following steps summarize an interrupt operation for a vectored interrupt. Figure 6–56 shows the sequence. See Section 6.14 for more details.

❶ XJA sends the interrupt and IPL to ICU. Figure 6–57 shows the interrupt packet.

❷ ICU sends the interrupt and IPL to IRCX.

❸ IRCX sends IPL to the EBox.



MR_X0963_89

Figure 6–56 Interrupt Operation

WORD	BIT 15 14 13 12 11 10 09 08	BIT 07 06 05 04 03 02 01 00
0	RESERVED	R R IPL CMD

R = RESERVED

MR_X0964_89

Figure 6–57 Interrupt Packet

6.10 SPU-to-ICU Communication Using Packets

SPU communicates with ICU using the DMA, ECC, I/O, and interrupt transactions. The transactions use packets in which addresses are quadword-aligned, and parity must be correct for all valid and invalid bytes. Byte wrapping is not supported. The packets are transferred in 14 cycles, 1 byte at a time, and data context cannot be longer than a quadword.

The four kinds of transactions are as follows:

- DMA transactions are reads, writes, read locks, or write unlocks. DMA transactions can be up to a quadword in length.
- I/O transactions access the I/O portion of the VAX physical address space.
- Interrupt transactions notify the operating system of a console terminal receive, console terminal transmit, console storage device receive, console storage device transmit, powerfail, halt CPU, or keep-alive interrupt.
- ECC transactions notify SPU that the error checking and correction logic has detected an error and that SCU is sending the ECC address and the syndrome bits for error reporting and logging.

6.10.1 DMA Packet

Figure 6–58 shows the SPU DMA packet.

The fields of the SPU DMA write packet are as follows:

- **Command field** — This 4-bit field defines the action taken on receipt of the packet. Tables 6–20 and 6–22 list the SPU and ICU DMA commands.
- **Address field** — This field consists of the high 32 bits of a 34-bit address (quadword-aligned). For reads, this field indicates the location to be read. For writes, this field indicates the location to be written. For data packets returned in response to a read request, this field is not used.
- **DMA mask field** — For writes, this 8-bit field indicates which of the 8 bytes of a quadword are to be written. For all other transactions, this field is not used.
- **Data byte field** — For writes or packets returning requested read data, this field contains the quadword of data. This field is not used for read, return read error, or read lock deny messages.

Table 6–17 illustrates the DMA mask field.

	07 04	03 00
00	DON'T CARE	CMD [03:00]
01	ADDR [29:26]	ADDR [13:10]
02	ADDR [05:02]	ADDR [09:06]
03	ADDR [33:30]	ADDR [21:18]
04	ADDR [25:22]	ADDR [17:14]
05	MASK [07:04]	MASK [03:00]
06	DATA [35:32]	DATA [03:00]
07	DATA [39:36]	DATA [07:04]
08	DATA [43:40]	DATA [11:08]
09	DATA [47:44]	DATA [15:12]
10	DATA [51:48]	DATA [19:16]
11	DATA [55:52]	DATA [23:20]
12	DATA [59:56]	DATA [27:24]
13	DATA [63:60]	DATA [31:28]

MR_X0441_89

Figure 6–58 SPU DMA Packet

Table 6–17 DMA Mask

Mask Code	Valid Byte
0	DATA 07:00
1	DATA 15:08
2	DATA 23:16
3	DATA 31:24
4	DATA 39:32
5	DATA 47:40
6	DATA 55:48
7	DATA 63:56

6.10.2 I/O Packet

Figure 6–59 shows the SPU I/O packet.

The fields of the SPU I/O packet are as follows:

- **Command field** — This 4-bit field defines the action taken on receipt of the packet. Tables 6–20 and 6–22 list the SPU and ICU commands.
- **Address field** — (Address [29:02]). For reads, this field indicates the location to be read. For writes, this field indicates the location to be written. For data packets returned in response to a read request, this field is unused.
- **Mask field** — (Address [33:30]). The high four bits of the address provide byte masking for word- or byte-oriented I/O devices.
- **Data byte field** — For writes or packets returning requested read data, this field contains the longword of data. In the case of nonlongword transfers, any masked bytes may be used. For read requests, this field is not used.

Table 6–18 illustrates the address masks.

	07 04	03 00
00	DON'T CARE	CMD [03:00]
01	ADDR [29:26]	ADDR [13:10]
02	ADDR [05:02]	ADDR [09:06]
03	ADDR/MASK [33:30]	ADDR [21:18]
04	ADDR [25:22]	ADDR [17:14]
05	DON'T CARE	DON'T CARE
06	DON'T CARE	DATA [03:00]
07	DON'T CARE	DATA [07:04]
08	DON'T CARE	DATA [11:08]
09	DON'T CARE	DATA [15:12]
10	DON'T CARE	DATA [19:16]
11	DON'T CARE	DATA [23:20]
12	DON'T CARE	DATA [27:24]
13	DON'T CARE	DATA [31:28]

MR_X0442_89

Figure 6–59 SPU I/O Packet

Table 6–18 Address Mask

Mask	Valid Byte
30	DATA 07:00
31	DATA 15:08
32	DATA 23:16
33	DATA 31:24

6.10.3 ECC Packet

Figure 6–60 shows the SPU ECC packet.

The fields of the SPU ECC packet are as follows:

- **Command field** — This 4-bit field defines the action taken on receipt of the packet.
- **ECC address field** — This field consists of the high 32 bits of the 34-bit address in which an ECC error has been detected.
- **ECC syndrome field** — The 32-bit error syndrome for the address where ECC failed. This packet type is only sent by the JBox.

	07 04	03 00
00	DON'T CARE	CMD [03:00]
01	ECC ADDR [29:26]	ECC ADDR [13:10]
02	ECC ADDR [05:02]	ECC ADDR [09:06]
03	ECC ADDR [33:30]	ECC ADDR [21:18]
04	ECC ADDR [25:22]	ECC ADDR [17:14]
05	DON'T CARE	DON'T CARE
06	ECC ADDR [29:26]	ECC ADDR [13:10]
07	ECC ADDR [05:02]	ECC ADDR [09:06]
08	ECC ADDR [33:30]	ECC ADDR [21:18]
09	ECC ADDR [25:22]	ECC ADDR [17:14]
10	MDP1 ECC SYN [07:04]	MDP0 ECC SYN [07:04]
11	MDP1 ECC SYN [03:00]	MDP0 ECC SYN [03:00]
12	MDP1 ECC SYN [15:12]	MDP0 ECC SYN [15:12]
13	MDP1 ECC SYN [11:08]	MDP0 ECC SYN [11:08]

MR_X0443_89

Figure 6–60 SPU ECC Packet

6.10.4 Interrupt Packet

SPU sends seven types of interrupts to the CPU in response to any of the following events:

- Interrupt terminal receive
- Interrupt terminal transmit
- Interrupt console receive
- Interrupt console transmit
- Powerfail
- Console halt
- Keep alive

Figure 6–61 shows the SPU interrupt packet.

	07 04	03 00
00	ID [03:00]	CMD [03:00]
01	DON'T CARE	DON'T CARE
02	DON'T CARE	DON'T CARE
03	DON'T CARE	DON'T CARE
04	DON'T CARE	DON'T CARE
05	DON'T CARE	DON'T CARE
06	DON'T CARE	DON'T CARE
07	DON'T CARE	DON'T CARE
08	DON'T CARE	DON'T CARE
09	DON'T CARE	DON'T CARE
10	DON'T CARE	DON'T CARE
11	DON'T CARE	DON'T CARE
12	DON'T CARE	DON'T CARE
13	DON'T CARE	DON'T CARE

MR_X0444_89

Figure 6–61 SPU Interrupt Packet

The SPU uses two receive registers, RXCS and RXDB, to initiate console terminal receive interrupts. The SPU uses two transmit registers, TXCS and TXDB, to initiate console terminal transmit interrupts.

The SPU uses two receive registers, RXFCT and RXPRM, to initiate console storage receive interrupts. The SPU uses two transmit registers, TXFCT and TXPRM, to initiate console storage transmit interrupts.

The fields of the SPU interrupt packet are as follows:

- **Command field** — This 4-bit field defines the action taken upon receipt of the packet. This packet type is only sent by the SPU.
- **ID field** — This 4-bit field indicates which of the four CPUs to interrupt. Table 6–19 lists the ID bits and corresponding CPUs.

Table 6–19 CPU IDs

ID Bit	CPU
0	CPU0
1	CPU1
2	CPU2
3	CPU3

6.11 ICU-to-SPU Interface

Figure 6–62 shows the ICU-to-SPU handshaking signals.

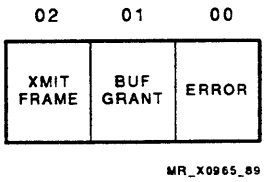


Figure 6–62 ICU-to-SPU Handshaking Signals

6.11.1 Commands

Table 6–20 lists the ICU-to-SPU commands.

Table 6–20 ICU-to-SPU Commands

Command	Code	JBox Action
0000	Read register	Sends the read register command to read a console register that resides physically in the console subsystem. The JBox can only have a single read request outstanding at a given time. The I/O packet is used.
0001	Write register	Sends the write register command to write a console register that resides physically in the console subsystem. The I/O packet is used.
0010	Return DMA read	Sends the return DMA read command to deliver read data that was requested by a previous read request referencing memory space. The DMA packet is used.
0011	Return I/O read	Sends the return I/O read command to deliver read data that was requested by a previous read request referencing I/O space. The I/O packet is used.
0100	Return read error	Sends the return read error command to notify SPU that read data, requested by a previous read request (I/O or memory space), encountered an error condition. This may be due to PAMMs detecting nonexistent memory, memory detecting a double-bit error, or the JBox detecting a fatal XJA or memory error. The DMA packet is used.
0101	Write register 0 write error register	Sends the write register 0 command to report an ECC incident involving a memory access requested by SPU. The data block includes the address of the error and an error syndrome (total of eight bytes). The ECC packet is used.
0110	Read lock denied	Sends the read lock denied command to notify SPU that a read lock request referencing memory space encountered an existing lock. The requested data will not be returned. The DMA packet is used.

6.11.2 Transferring a Packet from the JBox (ICU) to the SPU

All transfers from ICU to SPU are synchronous with JDCX_CTLD_SPU_CLKJ_H. Before sending a packet, the JBox monitors the buffer full status. When buffer full is deasserted, the JBox asserts the transmit frame and places the first byte of the packet on DATA_IN[07:00] and parity on PAR_IN[00] on the rising edge of the JBOX_CLK_H.

On the falling edge of JBOX_CLK_H, the SPU samples the transmit frame. If it is asserted, the SPU clocks in the data and the parity bit. On each rising edge of the clock, the JBox (receiving data from ICU) places a new data byte and parity bit on DATA_IN and PAR_IN[00]. On each falling edge of the clock, the SPU reads in the data and the parity bit.

The JBox deasserts transmit frame on the fifteenth rising edge of JBOX_CLK_H. The state of the DATA_IN and PAR_IN lines is ignored. The SPU samples the transmit frame on the next falling edge of the clock. On the next rising edge of the JBOX_CLK_H, the SPU asserts buffer full, and if a parity error has been detected, also asserts PAR_ERR_OUT_H. On the next falling edge of the clock, the JBox samples PAR_ERR_OUT_H to verify that the transfer was successful. The SPU deasserts PAR_ERR_OUT_H on the first rising edge of the JBOX_CLK_H, after the SPU has emptied its receive buffer (minimum assertion is one clock period).

6.12 SPU-to-ICU Interface

Figure 6–63 shows the SPU-to-ICU handshaking signals.

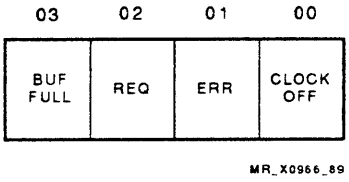


Figure 6–63 SPU-to-ICU Handshaking Signals

6.12.1 Key Signals

Table 6–21 lists the SPU-to-ICU signals.

Table 6–21 SPU-to-ICU Signals

Signal	Description
SPU_RESET_H	A single-ended TTL level signal sourced by the service processor.
SPU_CLKSTOP_H	A single-ended TTL level signal sourced by the service processor. It warns XJA of impending SCU clock stops. Upon receiving this signal, XJA finishes the current JXDI transmit transaction, if there is one, but does not transmit new ones to ICU, even if signaled to retry through ICU_XJA_XFERRETRY_H.

6.12.2 Commands

Table 6–22 lists the SPU-to-ICU commands. Figure 6–64 shows the commands the SPU sends to CCU, XJA, and IRCX.

Table 6–22 SPU-to-ICU Commands

Code	Command	SPU Action
0000	DMA read	Sends the DMA read command and a quadword-aligned address to read a valid memory space address. The SPU can have only a single read request outstanding at a given time. The DMA packet is used.
0001	DMA write	Sends the DMA write command to write to a valid memory address. The DMA packet is used.
0010	DMA read lock	Sends the DMA read lock command to read lock a valid memory space. The SPU can have only a single read request outstanding at a give time. The DMA packet is used.
0011	DMA write unlock	Sends the DMA write unlock command to a valid memory address. This must match a previous DMA read lock request. The DMA packet is used.
0100	I/O read	Sends the I/O read command to read a valid I/O space address. The SPU can have only a single read request outstanding at a given time. The I/O packet is used.
0101	I/O write	Sends the I/O write command to a valid I/O address. The I/O packet is used.
0110	Register return read	Sends the register return read command to deliver read data requested by a previous read register request. The I/O packet is used.
0111	Interrupt terminal receive (TRX)	Sends the interrupt terminal receive command to interrupt the operating system due to console terminal receive. SPU can select which CPU to interrupt using the ID field. The interrupt packet is used.
1000	Interrupt terminal transmit (TTX)	Sends the interrupt terminal transmit command to interrupt the operating system due to console terminal transmit. SPU can select which CPU to interrupt using the ID field. The interrupt packet is used.
1001	Interrupt console block storage receive (SRX)	Sends the interrupt console block storage receive command to interrupt the operating system due to console block storage receive. SPU can select which CPU to interrupt using the ID field. The interrupt packet is used.
1010	Interrupt console block storage transmit (STX)	Sends the interrupt console block storage transmit command to interrupt the operating system due to console block storage transmit. SPU can select which CPU to interrupt using the ID field. The interrupt packet is used.
1011	Interrupt powerfail	Sends the interrupt powerfail command to interrupt the operating systems due to an impending power failure. SPU can select which CPU to interrupt using the ID field. The interrupt packet is used.
1100	Console halt	Sends the console halt command to interrupt the operating system in order to halt one or more of the CPUs. SPU can select which CPU to interrupt using the ID field. The interrupt packet is used.

Table 6–22 (Cont.) SPU-to-ICU Commands

Code	Command	SPU Action
1101	Keep alive	Sends the keep-alive command to interrupt the operating system to prevent a keep-alive timeout. SPU can select which CPU to interrupt using the ID field. The EBox helps the console determine when the CPU is in a hung state. The interrupt packet is used.
1110	–	Spare.
1111	–	Spare.

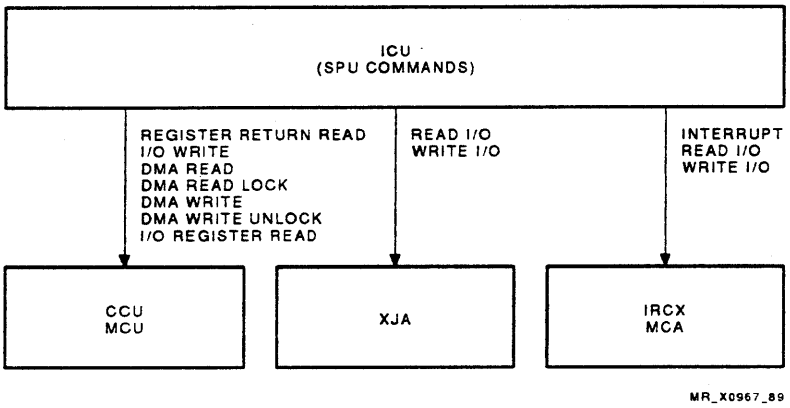


Figure 6–64 SPU Command Summary

6.12.3 Transferring a Packet from the SPU to the JBox (ICU)

All transfers from the SPU to the JBox are synchronous with JDCX_CTLD_SPU_CLKJ_H. The SPU sends all packets in 14 cycles (1 byte at a time). When the SPU sends a packet, it asserts the buffer request on the rising edge of JBOX_CLK_H. The JBox samples the buffer request on the falling edge of JBOX_CLK_H. If it is asserted, the JBox asserts JDCX_CTLD_BUF_GRANT_H on the first rising edge of JBOX_CLK_H after a buffer is available.

The SPU samples the buffer grant on the falling edge of JBOX_CLK_H. If it is asserted, the SPU deasserts buffer request on the next rising edge of JBOX_CLK_H and places the first data byte on DATA_OUT_H and parity on PAR_OUT[00]_H. The JBox sees buffer request deasserted on the falling edge of JBOX_CLK_H and reads in the data and parity bits on DATA_OUT and PAR_OUT.

On the next rising edge of JBOX_CLK_H, the JBox deasserts buffer grant. On the same rising edge, the SPU clocks the next data byte and its parity bit. On the next 12 rising edges of JBOX_CLK_H, the SPU sends the remaining data bytes and parity bit. On the falling edges, the JBox reads the data and parity bits.

On the fifteenth rising edge of JBOX_CLK_H after the JBox deasserts the buffer grant, if a parity error is detected, the JBox asserts PAR_ERR_IN_H and deasserts PAR_ERR_IN_H on the next rising edge of the clock. The SPU samples PAR_ERR_IN_H on the falling edge of JBOX_CLK_H to verify that the transfer was successful.

6.13 SPU Transactions

This section describes DMA, I/O, ECC, and interrupt transactions. CPU read and write SPU register commands are also described. For more details on SPU packets, see Section 6.10.

6.13.1 CPU Read Transaction

The following steps summarize a CPU read in which the EBox reads the contents of an SPU register. Figure 6–65 shows the sequence.

1. EBox sends the I/O read register and the address of an SPU register to the MBox.
2. MBox sends the I/O read register and the address to the JBox.
3. CCU MCU sends the CPU read, and the tag MCU sends the address to ICU.
4. ICU sends the read register and the address to SPU.
5. SPU sends the register return read to ICU.
6. ICU sends the CPU read data return to CCU.
7. JBox sends the I/O register return and data to the MBox.
8. MBox loads the data into the refill buffer and sends it to the EBox.

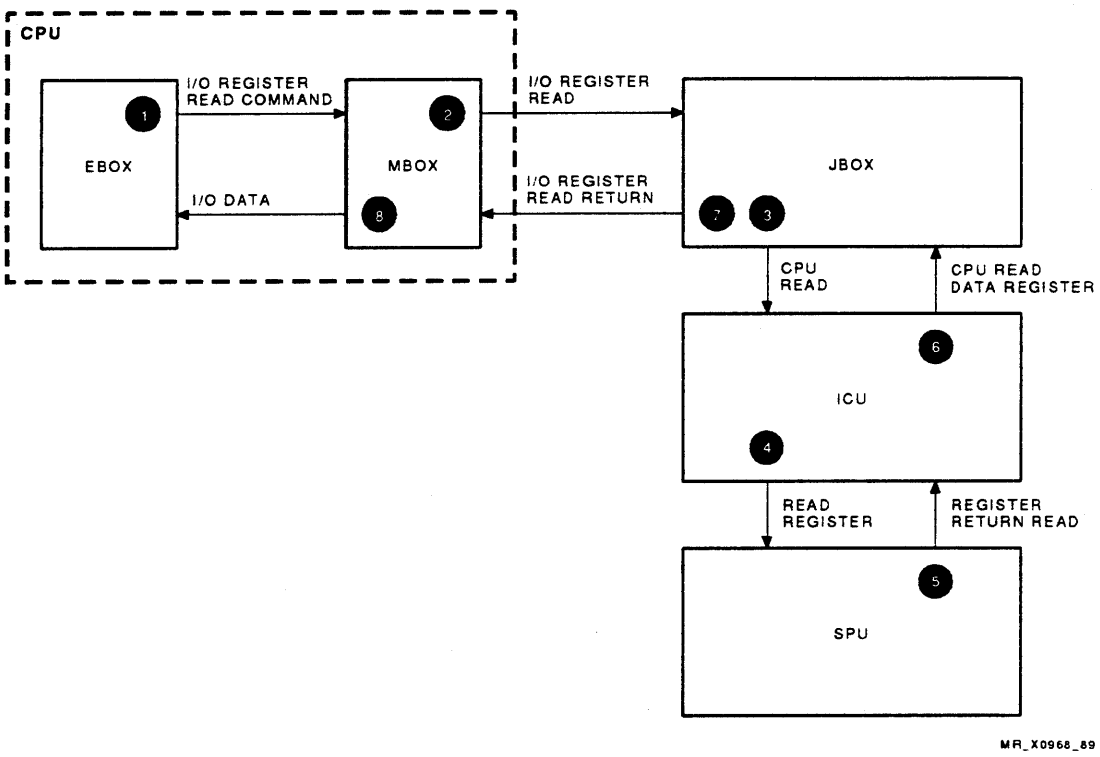


Figure 6–65 CPU Read SPU Register Operation

6.13.2 CPU Write Transaction

Figure 6–66 shows the sequence of steps for a CPU write in which the EBox writes to an SPU register.

1. EBox sends the I/O register write, the address of an SPU register, and data to MBox.
2. MBox sends the I/O register write, address, and data to JBox.
3. CCU MCU sends the CPU write, the tag MCU sends address, and the DAX/DBX MCUs send the data to ICU.
4. ICU sends the register write, address, and data to SPU.
5. ICU sends the write complete to CCU.

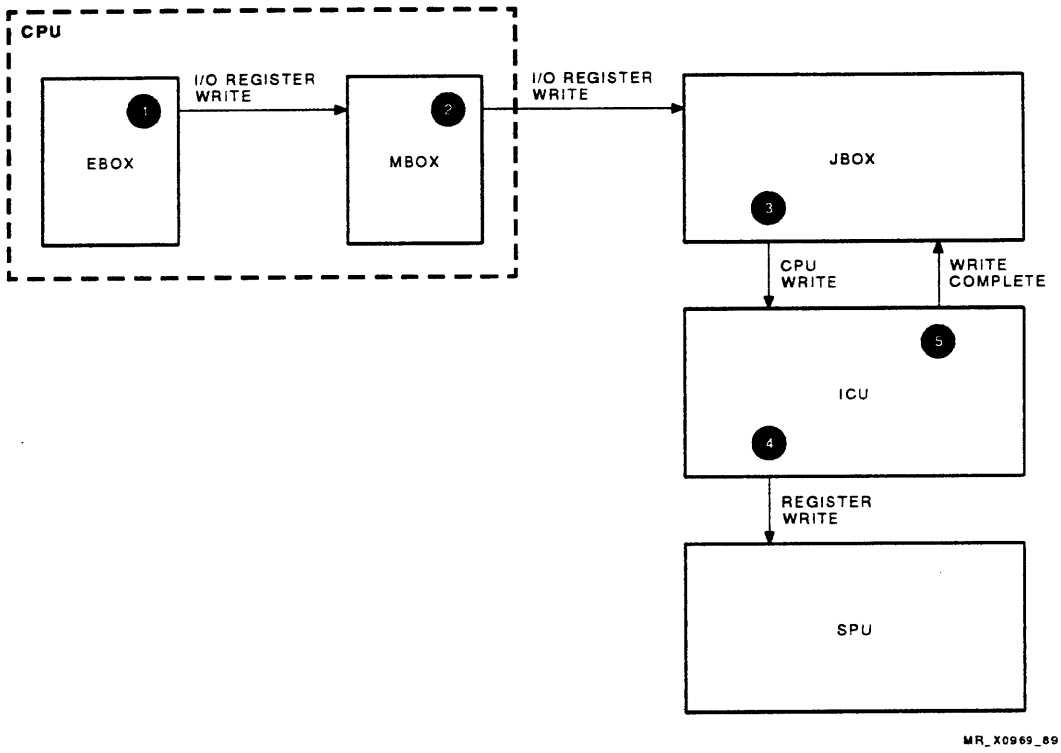


Figure 6–66 CPU Write to an SPU Register

6.13.3 I/O Read Transaction

Figure 6–67 shows the sequence of steps in an I/O read operation in which SPU reads the contents of an XJA register.

1. SPU sends the I/O read and the address of the XJA register to ICU.
2. ICU sends the SPU read I/O to the CCU MCU and the address to the tag MCU.
3. CCU sends the SPU read I/O, and the tag MCU sends the address to ICU.
4. ICU sends the CPU read and address to XJA.
5. XJA sends the CPU read data return with the data to ICU.
6. ICU sends the CPU read data return with the data to CCU.
7. CCU sends the SPU read data return, the tag MCU sends the address, and the DBX/DAX MCUs send data to ICU.
8. ICU sends the return I/O read, address, and data to SPU.

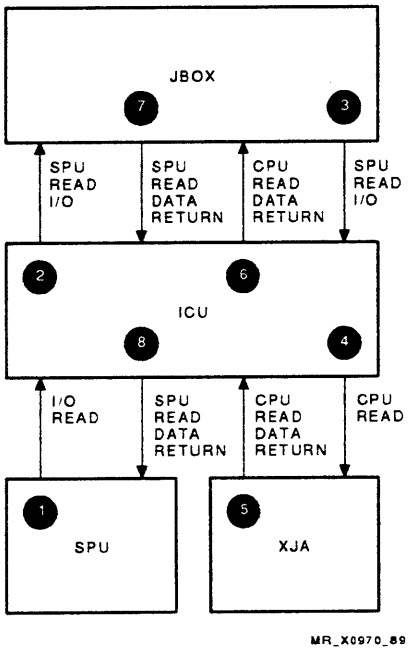


Figure 6–67 SPU I/O Read Operation

Figure 6–68 shows the sequence of steps in an I/O read operation in which SPU reads the contents of an IRCX register.

1. SPU sends the I/O read and the address of the IRCX register to ICU.
2. ICU sends the SPU read I/O to the CCU MCU and the address to the tag MCU.
3. CCU sends the SPU read I/O, and the tag MCU sends the address to ICU.
4. ICU sends the CPU read and address to the IRCX MCA.
5. IRCX sends the CPU read data return to the JBox.
6. CCU sends the SPU read data return, the tag MCU sends the address, and the DBX/DAX MCUs send the data to ICU.
7. ICU sends the return I/O read, address, and data to SPU.

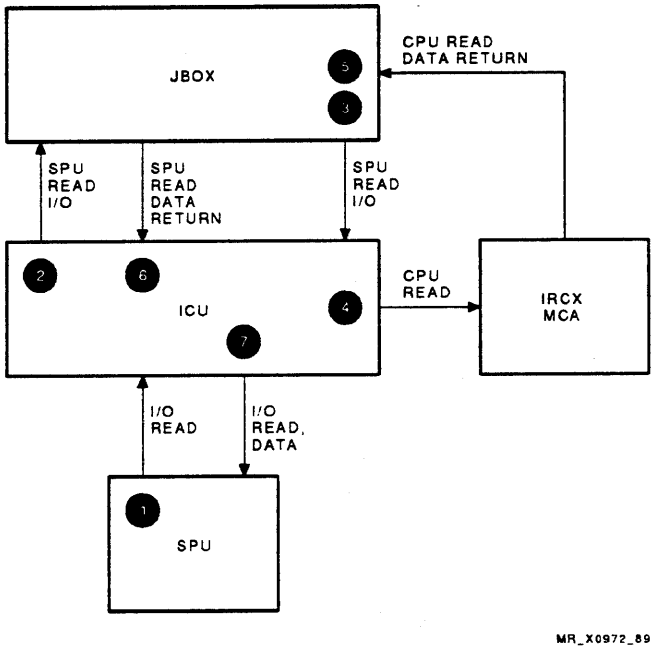


Figure 6–68 SPU Read IRCX Register

6.13.4 I/O Write Transaction

Figure 6–69 shows the sequence of steps in an I/O write operation in which SPU writes data to an XJA register.

1. SPU sends the I/O write, the address of the XJA register, and the data to ICU.
2. ICU sends the CPU write to the JBox. JBox sends the CPU write to the CCU MCU, the address to the tag MCU, and the data to the DAX/DBX MCUs.
3. CCU sends the CPU write, the tag MCU sends the address, and the DAX/DBX MCUs send the data to ICU.
4. ICU sends the CPU write, address, and data to XJA.
5. ICU sends the write complete to CCU.

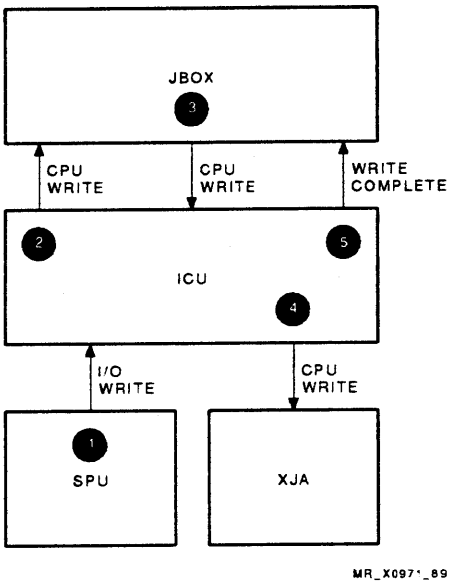


Figure 6–69 SPU I/O Write Operation

Figure 6–70 shows the sequence of steps in an I/O write operation in which SPU writes data to an IRCX register.

1. SPU sends the I/O write, the address of the IRCX register, and the data to ICU.
2. ICU sends the CPU write to the CCU MCU, the address to the tag MCU, and the data to the DAX/DBX MCUs.
3. CCU sends the CPU write, the tag MCU sends the address, and the DAX/DBX MCUs send the data to ICU.
4. ICU sends the CPU write, address, and data to IRCX.
5. ICU sends the write complete to CCU.

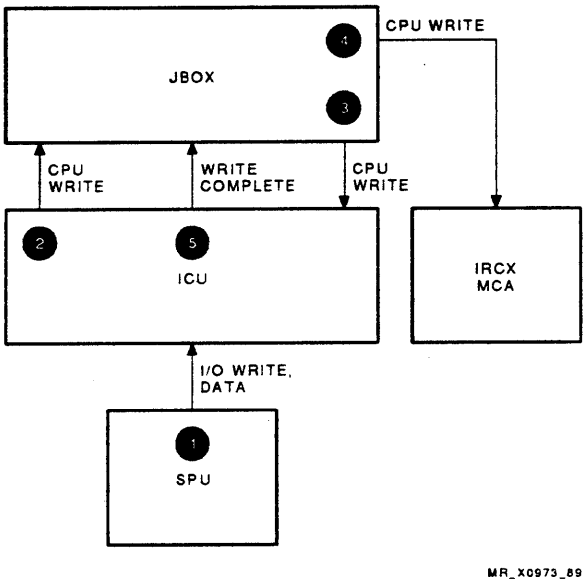


Figure 6–70 SPU Write IRCX Register

6.13.5 DMA Read Transaction

Figure 6–71 shows the sequence of steps a DMA read operation in which SPU reads data from main memory.

1. SPU sends the DMA read and address to ICU.
2. ICU sends the DMA read to the CCU MCU and the address to the tag MCU.
3. CCU sends the SPU read data return, the tag MCU sends the address, and the DAX/DBX MCUs send the data to ICU. CCU sends the return read error, if the JBox detects a nonexistent memory condition for the address sent with the DMA read. CCU sends the read lock denied, if the JBox detects an address that is locked by another port.
4. ICU sends return the DMA read, address, and data to SPU. ICU sends the return read error, if the JBox detects a nonexistent memory condition for the address sent with the DMA read. ICU sends the read lock deny, if the address is locked by another port.

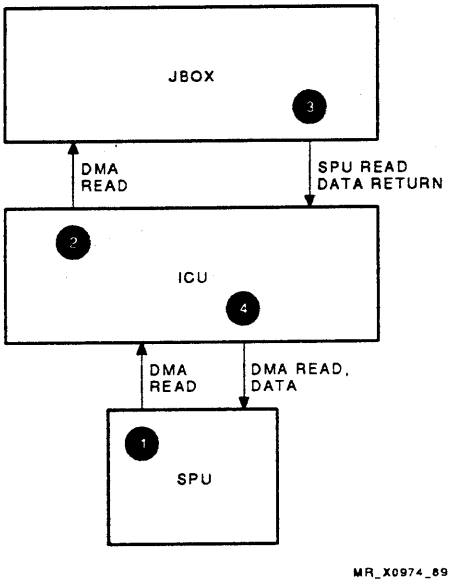


Figure 6–71 SPU DMA Read Operation

6.13.6 DMA Write Transaction

Figure 6–72 shows the sequence of steps in a DMA write operation in which SPU writes data to main memory.

❶ SPU sends the DMA write, address, and data to ICU.

❷ ICU sends the DMA write to the CCU MCU, the address to the tag MCU, and the data to the DAX/DBX MCUs.

The DMA write unlock is similar to a DMA write, except that the address lock is released.

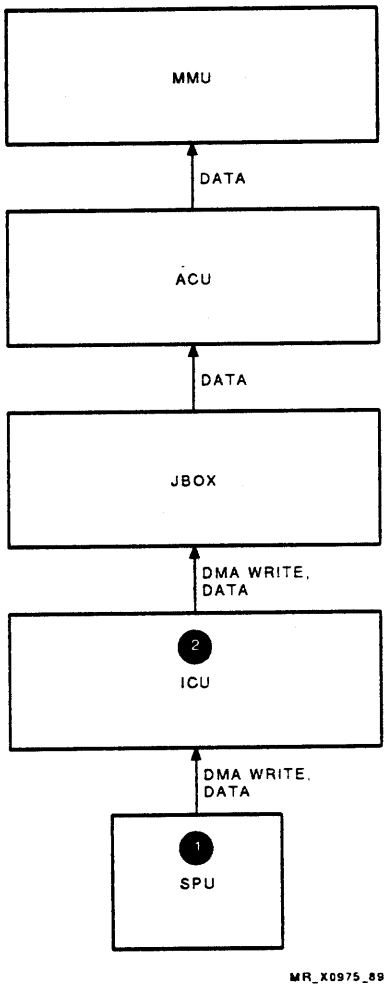


Figure 6–72 SPU DMA Write Operation

6.13.7 Interrupt Transactions

Figure 6–73 shows the sequence of steps in an interrupt operation. See Section 6.14 for more details.

❶ SPU sends the interrupt and the CPU ID to ICU.

❷ JDCX decodes the interrupt and the CPU ID and sends IPL to the IRCX MCA.

❸ IRCX MCA sends IPL to the EBox.

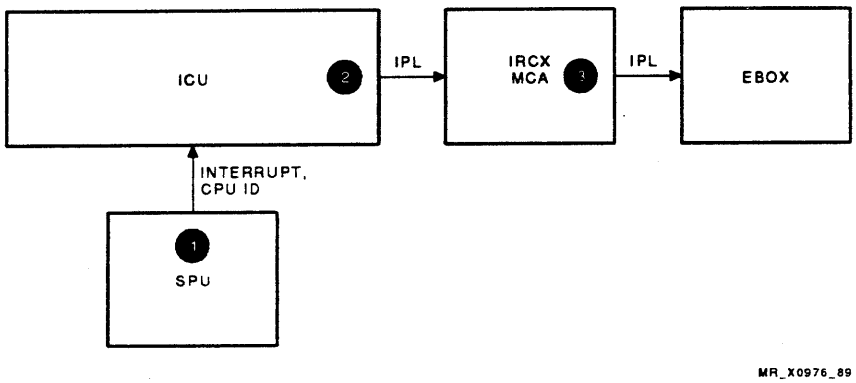


Figure 6–73 SPU Interrupt Operation

6.13.8 ECC Transactions

The MDPX MCAs in the array control unit (ACU) perform error checking and correction for data read from main memory. For successful reads from memory, the ACU sets read OK0 or read OK1 (for segments 0 and 1, respectively) in the JBox-memory interface. If the ACU does not set the read OK bit, the JBox decodes read OK0 and read OK1, determines that an ECC error has occurred, and notifies the SPU for error reporting and logging.

When an ECC error occurs, the CCU MCU sends a write register 0 command, the tag MCU sends the ECC address, and the MDPX MCA sends the syndrome bits to the ICU. Figure 6–74 shows the sequence of steps in a JBox ECC operation.

❶ The CCU MCU sends the write register 0, the tag MCU sends ECC address, and the MDPX MCA sends syndrome bits to ICU.

❷ ICU sends the write register 0, address, and syndrome bits to SPU.

❸ ICU sends the write complete to CCU.

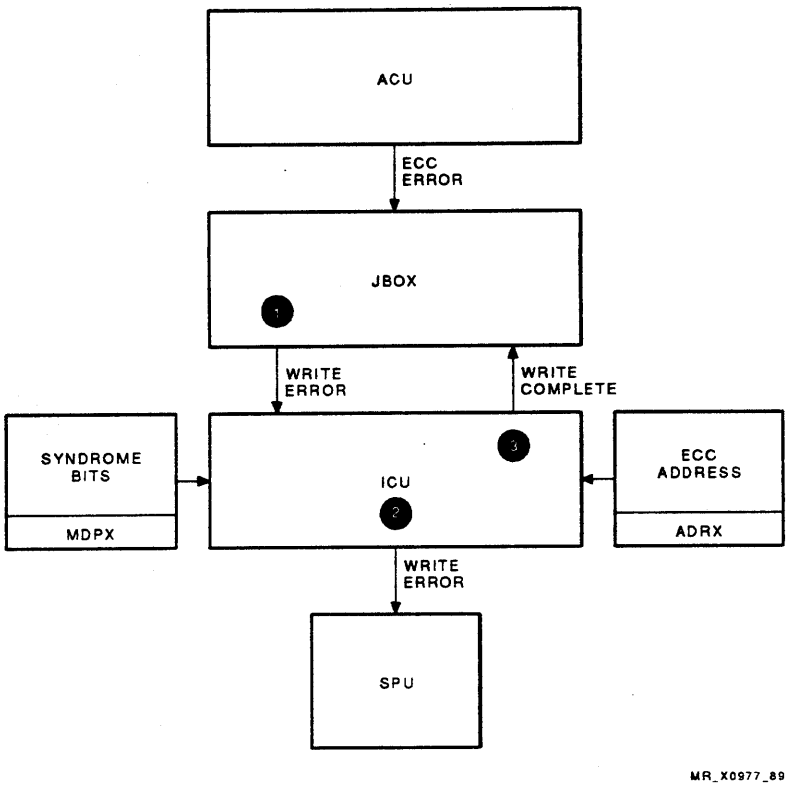


Figure 6–74 ECC Operation

6.14 Interrupts

The IRCX MCA receives, decodes, and prioritizes hardware interrupts. The priority associated with an interrupt is called its interrupt priority level (IPL). Interrupt requests come from the SPU, XMI devices, XJAs, and CPUs.

The CPU has 31 priority levels, divided into 15 software levels, 01 through 0F(hex), and 16 hardware levels, 10 through 1F(hex). IPLs from 10 through 17(hex) are reserved for devices and controllers. IPLs from 18 through 1F(hex) are reserved for urgent conditions. User applications, system calls, and system services interrupt at IPL 0.

Interrupt levels with higher numbers have higher priority. The IRCX MCA contains an interrupt arbiter mapping mechanism that determines which interrupt has the highest priority. IRCX sends the interrupt with the highest priority to the EBox. If IRCX sends an interrupt request with an IPL higher than that of the current EBox IPL, the interrupt is handled immediately. IRCX sends the IPLs in descending order. IRCX latches and holds interrupt requests having lower IPLs.

Figure 6–75 shows the hardware interrupt arbiter in the IRCX MCA.

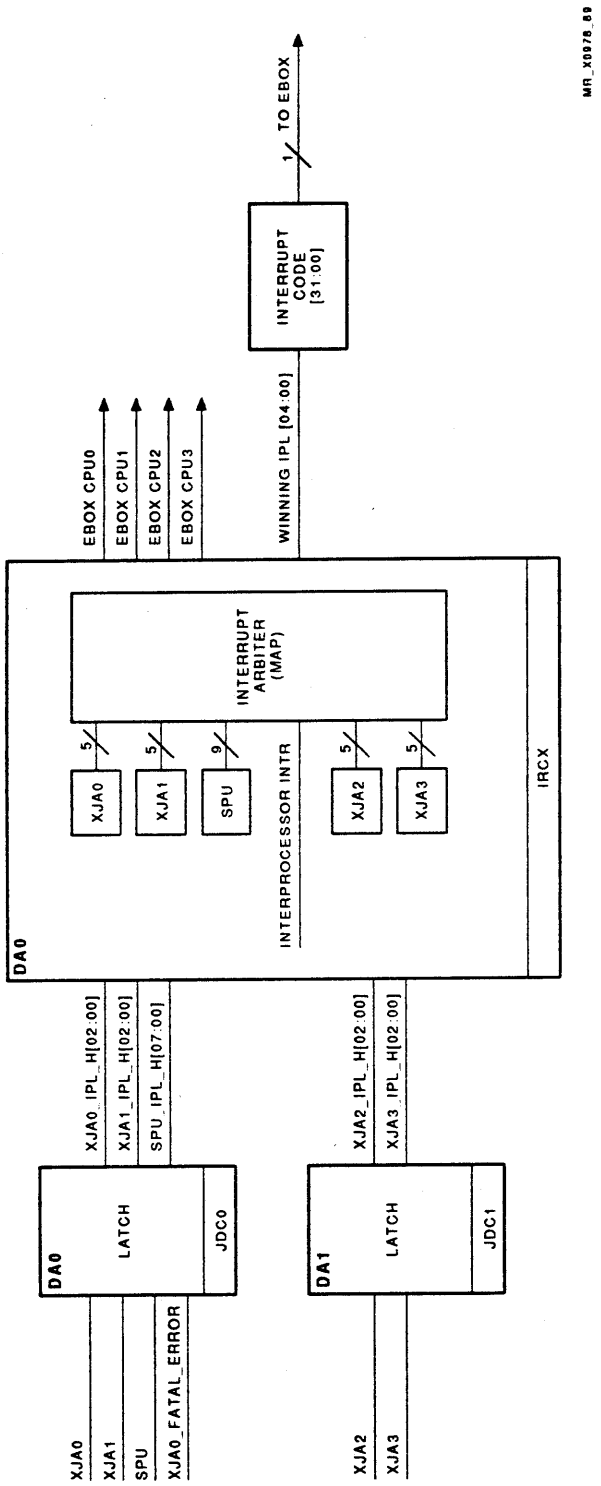


Figure 6–75 Interrupt Arbiter Inputs and Outputs

6.14.1 Interrupt Code

The IRCX MCA sends the IPL that has won arbitration to the EBox (the IPL with the highest priority wins arbitration). IRCX uses a single differential signal between the SCU and EBox. This signal serially transmits the start bit, followed in subsequent cycles by the interrupt code (containing the IPL), parity, and stop bit. Table 6–23 lists the IPLs and their descriptions. The stream of bits on this single signal uses the interrupt protocol shown in Table 6–24.

The start bit for the next interrupt may happen at cycle 8. The quiescent state of the signal is 0.

Table 6–23 Interrupt Priority Level Assignments

IPL (Hex)	Interrupt
00	Reserved for software
20	Console halt
20	Console spare
20	JBox hardware error
1E	Console spare
1E	Powerfail
16	CPU keep-alive/interval timer
16	CPU interrupt
14	Console receive
14	Console transmit
17	Console block storage receive
17	Console block storage transmit
1D	XJA0 fatal error
1D	XJA1 fatal error
1D	XJA2 fatal error
1D	XJA3 fatal error
14	XJA0 I/O
14	XJA1 I/O
14	XJA2 I/O
14	XJA3 I/O
15	XJA0 I/O
15	XJA1 I/O
15	XJA2 I/O
15	XJA3 I/O
16	XJA0 I/O
16	XJA1 I/O
16	XJA2 I/O
16	XJA3 I/O

Table 6–23 (Cont.) Interrupt Priority Level Assignments

IPL (Hex)	Interrupt
17	XJA0 I/O
17	XJA1 I/O
17	XJA2 I/O
17	XJA3 I/O

Table 6–24 Interrupt Protocol

Cycle	Protocol
0	Start bit = 1
1	Interrupt code bit 4
2	Interrupt code bit 3
3	Interrupt code bit 2
4	Interrupt code bit 1
5	Interrupt code bit 0
6	Odd parity across interrupt code bits [04:00]
7	Stop bit = 0

6.14.1.1 EBox Handling Keep Alive

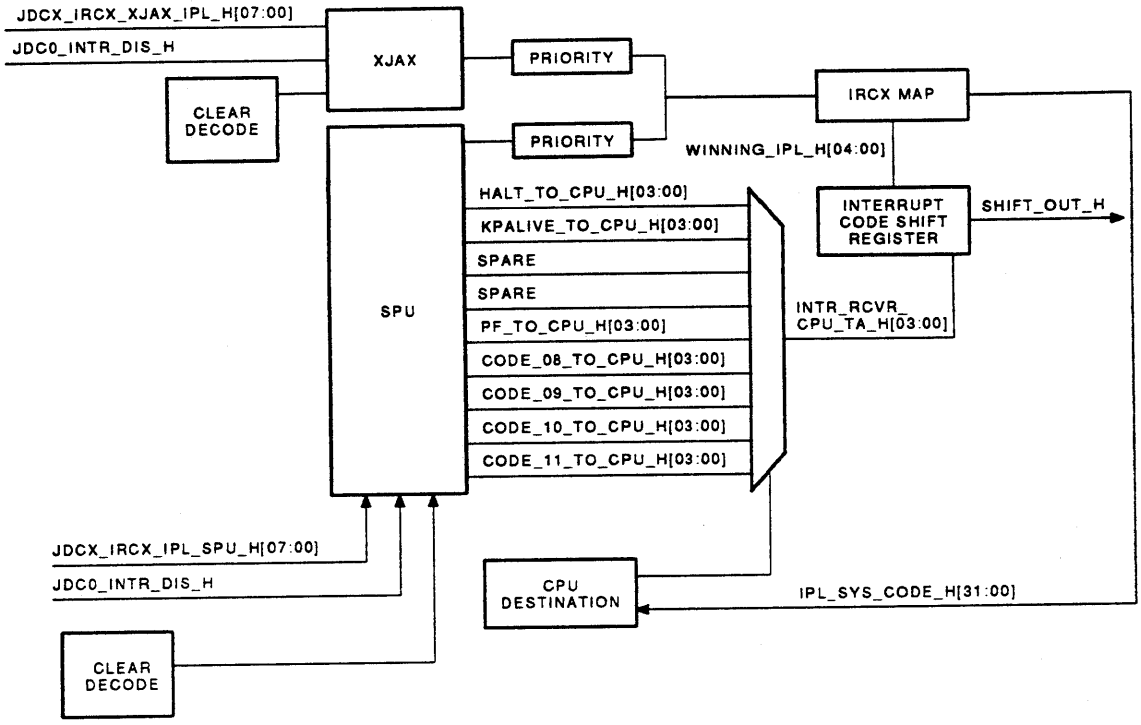
The EBox helps the console determine whether the CPU is in a hung state. The console initiates a keep-alive sequence by sending an interrupt packet to ICU. The ICU sends the winning interrupt code to the EBox. The EBox receives the serial code, passes the interrupt to the microsequencer logic, and decodes the keep-alive request.

The EBox holds the request until the current macroinstruction is completed. When the next macroinstruction is complete, the EBox generates EBOX_KEEP_ALIVE_H (or QUEUE_INSTRUCTION_DONE_H and A_ KEEP_ALIVE_H) to interrupt the console. If the console fails to receive a keep-alive interrupt within the timeout period, a hung CPU condition has occurred. The timeout period is long enough to allow the EBox to complete lengthy macroinstructions.

6.14.2 IRCX MCA

IRC0, located on the DA0 MCU, contains three registers: the CPU configuration register, interprocessor interrupt register, and error summary register. IRC0 interfaces to the data switch (DSXX MCAs). The IRCX MCA sends data to and receives data from the DSXX MCAs during SPU or CPU register reads and writes to IRCX registers.

The IRC0 MCA contains the interrupt logic for I/O-to-CPU interrupts from XJA0, XJA1, and SPU, and the inter-CPU interrupts. The IRC1 MCA, located on DA1, supports I/O-to-CPU interrupts from XJA2 and XJA3. Figure 6–76 shows the IRCX MCA block diagram.



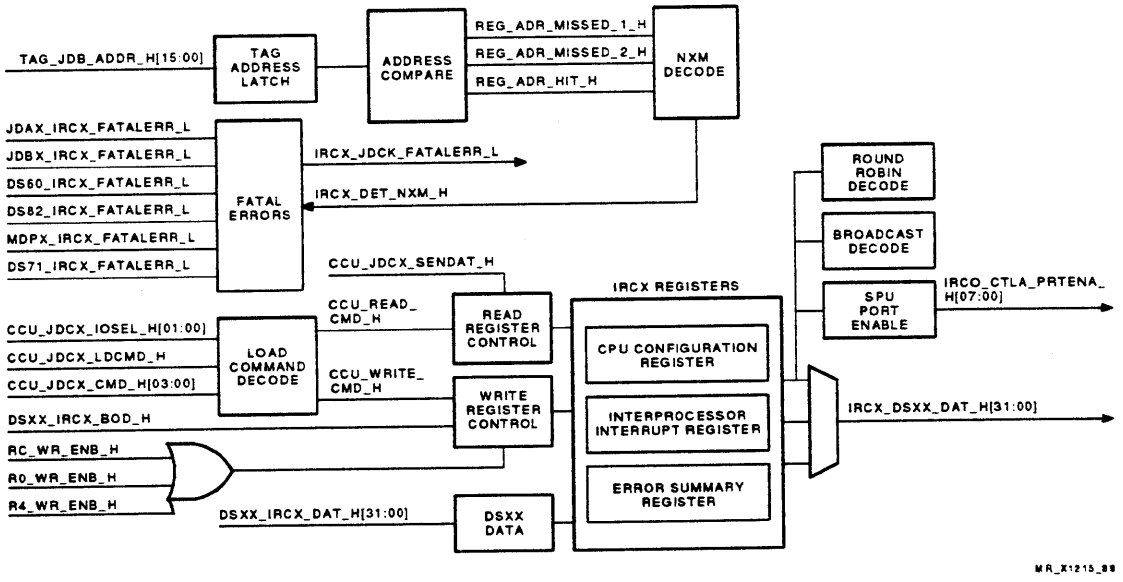


Figure 6–76 IRCX Block Diagram

The IRCX MCA receives the following:

- The load command, command, and I/O select from JDCX MCA
- Register addresses from the ADRX MCAs
- Fatal errors from the JDAX, JDBX, DSXX, and MDPX MCAs
- Data (and beginning of data signal) from the DSXX MCAs
- IPL and interrupt commands from JDCX MCA
- The send data signal from JDCX MCA

The IRCX MCA sends the following:

- Port enables to the CTLA MCA
- Register read data to the DSXX MCAs
- Winning IPL code to the EBox
- Fatal error to JDCX MCA

6.14.3 IRCX Registers

The IRCX MCA has three registers. IRCX has two interrupt registers, the CPU configuration register and the interprocessor interrupt register, and one error register, the JBox error summary register.

6.14.3.1 CPU Configuration Register

Figure 6–77 shows the CPU configuration register. Table 6–25 lists the fields and their descriptions.

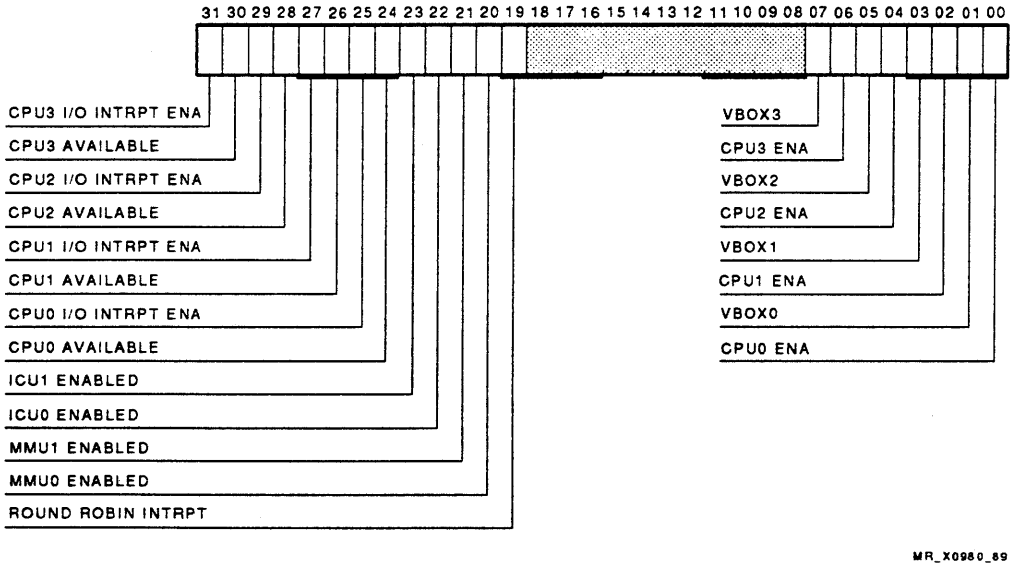


Figure 6–77 CPU Configuration Register

Table 6–25 CPU Configuration Register Field Descriptions

Bit	Description
00	CPU0 enable
01	VBox0 available for CPU0
02	CPU1 enable
03	VBox1 available for CPU1
04	CPU2 enable
05	VBox2 available for CPU2
06	CPU3 enable
07	VBox3 available for CPU3
18:08	Reserved
19	Round-robin interrupt
20	MMU0 enable
21	MMU1 enable
22	ICU0 enable
23	ICU1 enable
24	CPU0 available
25	CPU0 I/O interrupt enable
26	CPU1 available
27	CPU1 I/O interrupt enable
28	CPU2 available
29	CPU2 I/O interrupt enable
30	CPU3 available
31	CPU3 I/O interrupt enable

For XJA interrupts, if [19] (round-robin interrupt) is set, IRCX evenly distributes interrupts to the available CPUs (as defined by [30], [28], [26], and [24]) and can be interrupted (defined by [31], [29], [27], and [25]). If [19], is cleared, IRCX broadcasts interrupts.

For SPU interrupts, if [19] is set and the interrupt packet has CPU_ID_H[03:00] cleared, no interrupts are sent to the CPUs.

For SPU interrupts, CPU_ID_H[03:00] indicates which CPUs IRCX interrupts for this request.

6.14.3.2 Interprocessor Interrupt Register

Figure 6–78 shows the interprocessor interrupt register. Table 6–26 lists the fields. A write request to this register initiates an interprocessor interrupt to the CPUs coded in [03:00]. If these bits are cleared, no CPU receives an interrupt.

If multiple CPUs are writing to the interprocessor interrupt register before arbitration, IRCX delivers interprocessor interrupts to the CPUs with bits set. For example, CPU0 writes [03] and CPU1 writes [02], IRCX sends an interprocessor interrupt to CPU2 and CPU3 when the interprocessor interrupt register wins arbitration.

The CPUs and SPU can read this register. If the contents of this register are nonzero, the arbiter has a pending interrupt. IRCX delivers the interprocessor interrupt to the destination CPUs and clears the register.

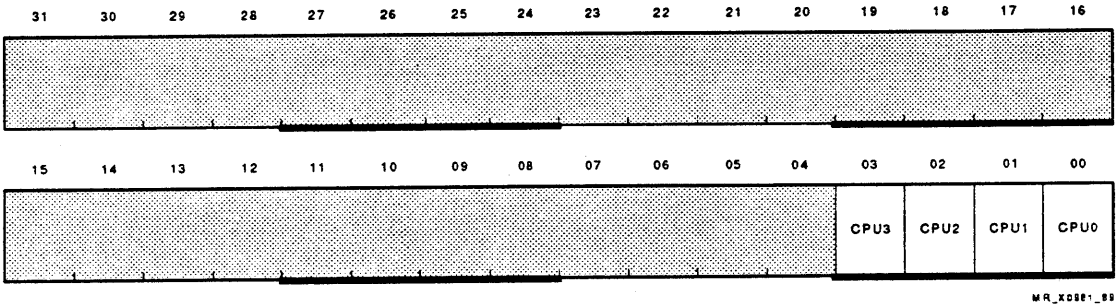


Figure 6–78 Interprocessor Interrupt Register

Table 6–26 Interprocessor Interrupt Register Fields

Bit	Name
00	CPU0
01	CPU1
02	CPU2
03	CPU3
31:04	Reserved

6.14.3.3 Error Summary Register

The SPU and CPU can read to and write from this register. Each of the four XJAs has a retry bit in the register. ICU detects bad parity, sends retry to XJA, and sets the corresponding bit in the register. Figure 6–79 shows the error summary register. Table 6–27 lists the fields.

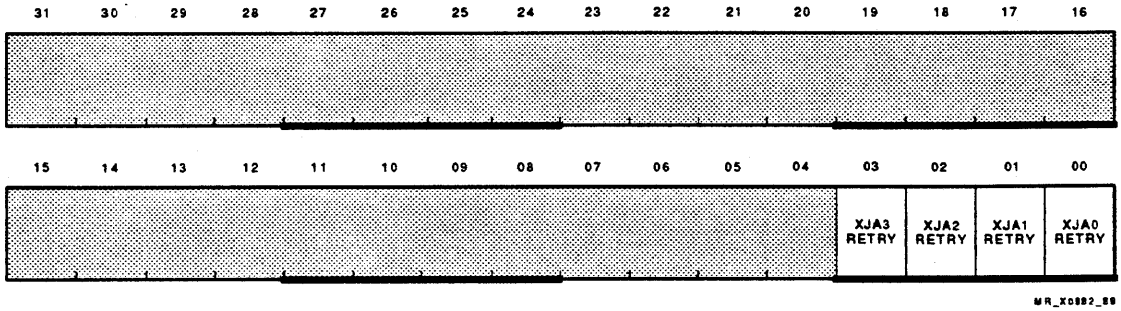


Figure 6–79 Error Summary Register

Table 6–27 Error Summary Register Fields

Bit	Name
00	XJA0 retry
01	XJA1 retry
02	XJA2 retry
03	XJA3 retry
31:04	Reserved

6.14.4 XJA Interrupts

The XJA error handling implementation centers on its error status registers (stored in the XJA module). XJA detects and recovers from most transient bus transmission errors on both JXDI and XMI bus.

XJA sends two types of interrupts, vectored interrupts at IPL 14 through 17(hex) and fatal interrupts at IPL 1D(hex). Figure 6–80 shows the two types of XJA interrupts.

XJA reports errors from which it recovered to the operating system using IPL 17(hex). Nonrecoverable errors are separated into two categories: nonfatal (vectored interrupts) and fatal.

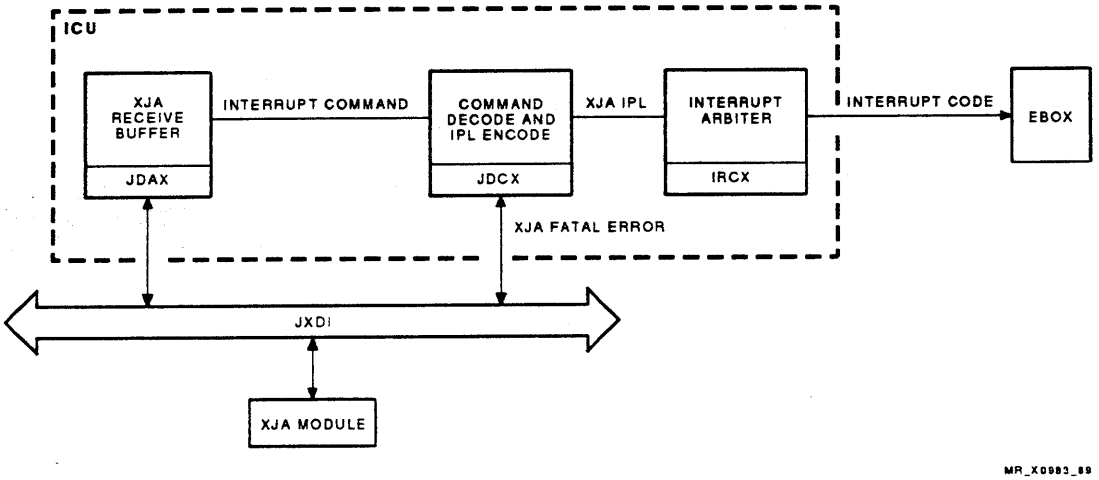


Figure 6–80 XJA Interrupts

The following steps summarize how IRCX sends the interrupt code to the EBox:

1. The XJA, console (SPU), or CPU sends an interrupt request to ICU.
2. ICU decodes the request and sends the interrupt, IPL (XJA), interrupt, and CPU ID (SPU) to IRCX. The IRCX determines which IPL has the highest priority and sends the winning IPL to the EBox.
3. The EBox completes the current instruction or reaches a well-defined point where an instruction can be interrupted (if the interrupt has a higher priority than the current IPL).
4. The EBox microcode initiates the interrupt sequence. Hardware interrupts are serviced from the interrupt stack in the EBox. The microcode examines the vector in the system control block (SCB). The SCB contains vectors that the EBox uses to dispatch all interrupts. Each device controller has a separate set of interrupt vector locations in the SCB.
5. The EBox uses the SCB vector, creates a new PC, uses the IPL associated with the interrupt (from the interrupt code sent by the IRCX MCA), and creates a new IPL.
6. The EBox executes the interrupt service routine identified by the SCB vector (new PC) and completes the routine with a return from exception or interrupt instruction.
7. The EBox restores the PC and PSL and continues from where it left off when the IRCX MCA sent the interrupt.

6.14.4.1 XJA-Vectored Interrupts

XJA delivers vectored interrupts to the ICU using the XJA interrupt packet format over JXDI. The interrupt packet contains the IPL in [05:04]. The JDAX MCA encodes [05:04] on JDA_IRC_INTR_H[01:00]. IRCX decodes the JDA_IRC_INTR_H[01:00] and sends the interrupt to an EBox. IRCX uses the CPU configuration register, not XJA, to determine which EBox(es) to interrupt.

Upon receiving the interrupt request from the interrupt arbiter, as soon as its IPL is lower than the IPL of the requested interrupt, the EBox issues a read request to one of four XJA SCB offset registers corresponding to the four IPLs. 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.

6.14.4.2 Fatal XJA Interrupts

The XJA asserts the JXDI signal XJA_FATAL_ERROR_H when any of the following errors occur:

- Fatal XMI errors:

  Node reset
  Node halt
  XMI fault
  Write error interrupt
  XMI powerfail
  XMI arbitration timeout

- Fatal XJA errors:

  XCE MCA internal state machine error
  ICU buffer count error
  CPU request overrun
  JXDI receive buffer overrun
  JXDI receive command error
  XJA CBI (CMOS to bipolar interconnect)

The JDCX MCA passes this signal to the IRCX MCA, which delivers the requested interrupt IPL 1D(hex) to the EBox(es). The EBox calculates the SCB offset and points to an XJA fatal error routine. XJA_FATALERR does not necessarily indicate that XJA is incapable of responding to further CPU requests.

6.14.5 Console Interrupts

The console subsystem consists of a BI MicroVAX system, one RD54 disk drive, one TK50 tape console block storage devices, a console terminal, and a remote diagnosis port.

IRCX distinguishes between terminal operations and block storage device operations by decoding the command field in the interrupt packet that the SPU sends to the ICU. (SPU sends an interrupt packet to ICU through the CTLD MCA.) When the JDCX MCA decodes the SPU command and finds that the command is an interrupt, it generates an IPL and sends the command and IPL to the IRCX MCA for interrupt arbitration.

Transfers to console devices are made through internal processor registers and device drivers in I/O space. No direct memory transfer is made between a VAX CPU and console device; instead the SPU sends an interrupt to the CPU, which executes an interrupt service routine and reads the appropriate registers in the SPU internal register area.

The console subsystem contains the following four internal SPU processor registers for communication with a console terminal:

RXCS — Console receive control and status register
RXDB — Console receive data buffer register
TXCS — Console transmit control and status register
TXDB — Console transmit data buffer register

The SPU supports three console terminal devices: local terminal (CTY), remote terminal (RTY), and the logical console port. The SPU uses the RXCS and RXDB registers to enable the terminals and to hold the data entered at the terminal. The SPU loads the RXCS and RXDB registers and sets the interrupt enable bit in RXCS. The SPU builds the interrupt packet with the interrupt command (console receive — IPL 14[hex]) and CPU ID, sending the packet to the IRCX MCA.

The SPU uses the TXCS and TXDB registers to determine which of the three terminals (CTY, RTY, or logical) is enabled and to hold the data to be transferred to the terminal. The SPU loads the TXCS and TXDB registers and sets the interrupt enable bit in TXCS. The SPU builds the interrupt packet with the interrupt command (console transmit — IPL 14[hex]) and CPU ID, sending the packet to the IRCX MCA.

The console subsystem contains four internal SPU processor registers for communication with a console storage device:

RXFCT — Receive function request register
RXPRM — Receive parameters register (data or address)
TXFCT — Transmit function request register
TXPRM — Transmit parameters register (data or address)

The SPU uses the RXFCT and RXPRM registers to operate system and diagnostic functions. The SPU loads the RXFCT and RXPRM registers with a function and parameter, setting the interrupt bit in RXFCT. The SPU builds the interrupt packet with an interrupt command (console storage receive — IPL 17[hex]) and CPU ID. RXFCT functions include receive block of data, halt CPU, send error log entry, get error log entry, and set keep-alive state.

The SPU uses the TXFCT and TXFCT registers to operate system and diagnostic functions. The SPU loads the TXFCT and TXPRM registers with a function and parameter, setting the interrupt bit in TXFCT. The SPU builds the interrupt packet with an interrupt command (console storage transmit — IPL 17[hex]) and CPU ID. Functions include transmit block of data, keep alive, reboot system, boot secondary system, reset the XJA, halt CPU, set interrupt mode, set margin clock, set margin power, and get IPAMM and PAMM configurations.

The console subsystem also sends interrupts for powerfail, CPU halt, and EBox keep alive.

6.14.6 Interprocessor Interrupts

Using the interprocessor interrupt register, a CPU can interrupt one or more CPUs, including itself, to support applications involving more than one processor. Using interprocessor interrupts, processors can share data or send information from one processor to another. The CPU initiates an I/O register write to the interprocessor interrupt register, and sets the corresponding CPU's bits (Section 6.14.3.2). The IRCX encodes the CPU interrupt, IPL 16, and sends the interrupt to the CPUs with bits set.

7
Error Detection and Reporting

The SCU error logic provides maximum error detection coverage on address, data, and control interfaces. The basic intent of the error detection logic is to detect all single intermittent failures. This is accomplished with the use of parity, ECC, and handshaking protocol. The types of errors reported include fatal, read data, and write data errors.

7.1 Detection

The error detection policy in the JBox provides error detectors wherever a bundle of signals for data, address, or control enters a new physical entity (an MCA or MCU). In most cases, a bundle of signals passing between an MCA and any other MCA is accompanied by a parity bit covering those signals. This permits the receiving MCA to do a parity check across these signals, ensuring detection of broken paths between the two MCAs. Figure 7–1 shows the parity checking on data lines in SCU.

7.2 Fault Isolation

The quantity and positioning of the error checkers in SCU implicate the minimum number of components whenever an error is detected. In general, each bundle of signals from an MCA to any other MCA is accompanied by a parity bit covering those signals. To minimize the number of parity signals required to do this, all signals from one MCA to another are grouped together, even if they are logically dissimilar, and are covered by one parity bit. The logic is designed so that all parity bits are true, and therefore checked, on all cycles. Consequently, grouping the signals is straightforward.

Wherever the number of MCAs implicated is unusually high, extra error status latches, such as select logic, help provide better isolation given several instances of the same intermittent failure. Usually, any error detected, other than single-bit errors (SBEs), causes SCU to assert the attention line to SPU. Every location that has duplicate error checkers has a scan latch, which can optionally inhibit the parity error signal from asserting the attention line. This ensures that if the only problem is that the duplicate error checker is broken, the system still runs normally.

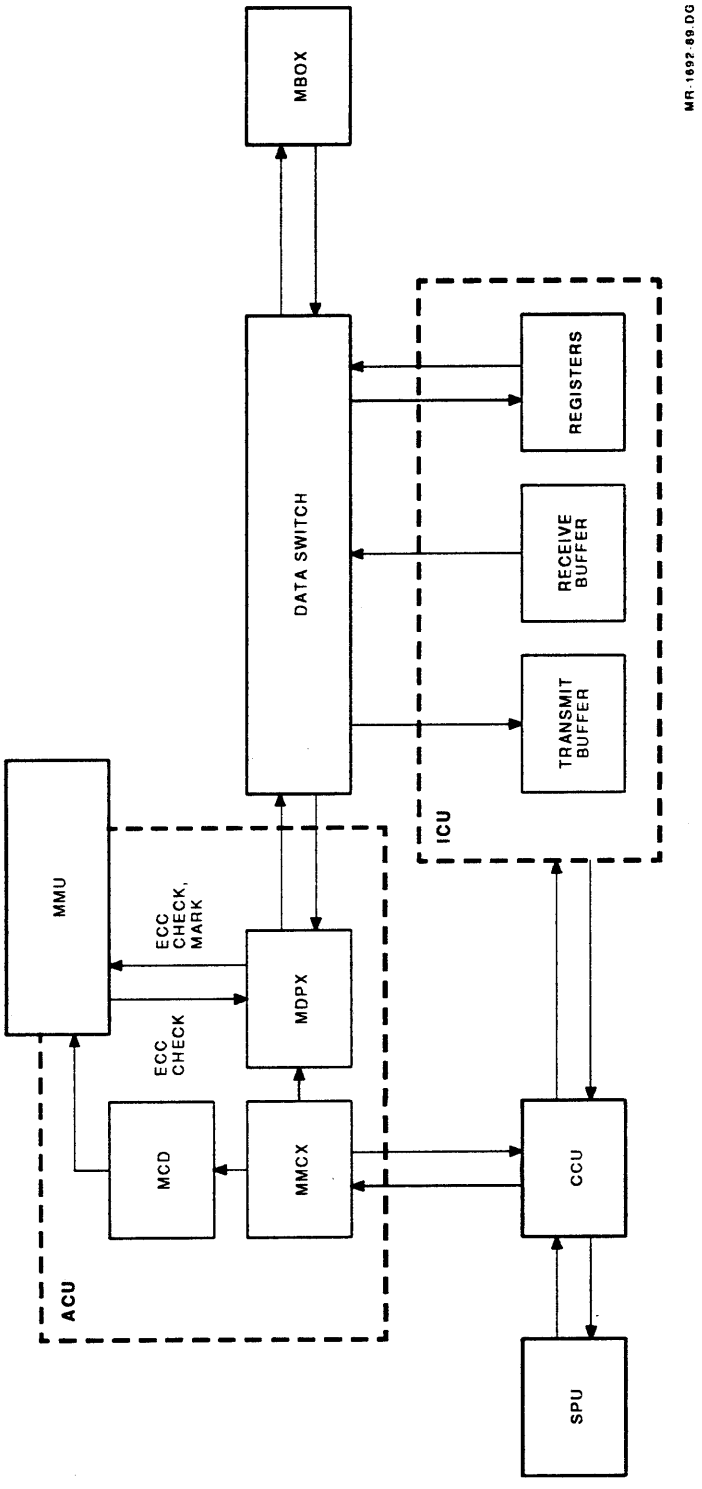


Figure 7–1 Data Parity

7.3 Error Correction Code

A modified Hamming code protects data written to memory. This code operates on 32 data bits and produces 7 check bits. Another bit, the mark bit, is used with the ECC code to enhance error detection. The mark bit is set when bad data is purposely written to memory. This can occur during read-modify-write operations when a multiple-bit error is detected, or during a write operation when parity detection indicates that bad data has been received from the JBox.

7.3.1 Error Syndromes

In all cases except marked bad data, the syndromes listed in Table 7–1 are calculated for single-bit errors (SBEs) and double-bit errors (DBEs). For more than two errors, syndromes calculated could be interpreted as a multiple-bit error (MBE) or single-bit error.

Table 7–1 Syndromes

Mark	Data	Corrected Bit	Syndrome	Error
0	0	0	All 0s	No error
0	1	0	Three 1s	SBE
0	0	1	One 1	SBE
0	2	0	Two, four, or six 1s	MBE
0	0	2	Two 1s	MBE
0	1	1	Two or four 1s	MBE
1	0	0	All 0s	No error
1	1	0	Three 1s	MBE
1	0	1	One 1	SBE

7.3.2 Error Syndromes on Marked Bad Data

During a read-modify-write operation, ECC-detected errors are not reported. If there is a multiple-bit error, marked bad data is loaded into the memory location.

In a read-modify-write operation, the following actions occur:

- SBE
  1. Correct data.
  2. Merge with new data (as defined by mask bits).
  3. Generate ECC check bits on good data.
  4. Set mark bit to zero.
  5. Write good data, mark bit, and check bits.
- DBE
  1. Generate new ECC check bits on old data.
  2. Invert new ECC check bits.
  3. Set mark bit to one.
  4. Write old data, mark bit, and check bits.

In a read operation, the following actions occur:

- SBE
  1. Correct data.
  2. Generate ECC check bits on good data.
  3. Set mark bit to zero.
  4. Write good data, mark bit, and check bits.
- DBE
  1. Generate new ECC check bits on received bad data.
  2. Invert new ECC check bits.
  3. Set mark bit to one.
  4. Write bad data, mark bit, and check bits.

Because the check bits written to memory are inverted on DBEs, a different set of syndromes (for example, XOR of read check bits and new check bits) is generated. Syndromes are dependent on both the previous double-bit error and a single-bit error if it occurs.

7.3.3 Correction

ECC is maintained across 4-byte quantities in memory. The most frequent ECC incident is an SBE, which is dynamically corrected. The syndrome and address are sent to SPU, through the ICU. The data requester (MBox) is unaware of this occurrence.

If a DBE occurs, the syndrome and address are sent to SPU in the same way, but in this case, the consumer of the data (MBox) is informed when the JBox sends a return read error status command to the MBox.

7.3.4 ECC Reporting

The following sequence describes how SCU deals with SBEs and reports them to SPU (which behaves like an I/O controller connected to ICU in this context):

1. MDPX MCA detects an SBE.
2. MDPX MCA locks its error register.
3. MDPX MCA informs MMCX MCA of the SBE.
4. MMCX sends a return memory data command and index field to CCU.
5. MMCX sends a return syndrome command and index field to CCU.
6. CCU checks for a free buffer in the CPU waiting for the memory data.
7. CCU sends an index field to the address crossbar.
8. CCU sends a send data bit to MMCX.
9. MMCX tells MDPX to send out the memory data.
10. CCU sends a command to DSCT MCA to move the memory data.
11. CCU checks for a free buffer in the ICU to which the console is connected.
12. CCU sends an index field to the address crossbar.
13. CCU sends a send data bit to MMCX.
14. MMCX tells MDPX to send out the ECC syndrome.
15. CCU sends a command to DSCT to move the ECC syndrome.
16. MDPX unlocks its error register.
17. JDBX MCA sends a write to console register command.

Each MMU is supported by two MDPX chips: one for bytes 7 through 4 and the other for bytes 3 through 0. Each MDPX includes its own check and correction network.

A 4-bit register in IRCX masks SBE reporting from each MDPX. This register is dynamically writeable, through the write to JBox register command, without stopping the clocks.

The MDPX error lock is set when MDPX detects an error, and it is cleared when MDPX has dispatched the ECC syndrome to ICU. If MDPX detects another SBE while its error lock is set, it sets a bit to imply this, but the ECC syndrome is lost. This bit is included in the report the next time an error is flagged.

7.3.5 SPU Assistance for ECC Handling

SCU does not interrupt the VMS operating system directly in the event of SBEs, although SCU does send syndrome and failing address information to SPU. The SPU correlates successive SBE events, determining whether the events have any systematic relationship, (for example, many of the SBE incidents may be in the same page, or group of pages, in physical memory). SPU notifies the VMS operating system that certain pages should no longer be used. Therefore, SPU acts as the first level of filtering for SBE information.

7.4 MCU Error Detection

Figure 7–2 shows the SCU error reporting logic. Table 7–2 lists events and the corresponding MCU set attention signals.

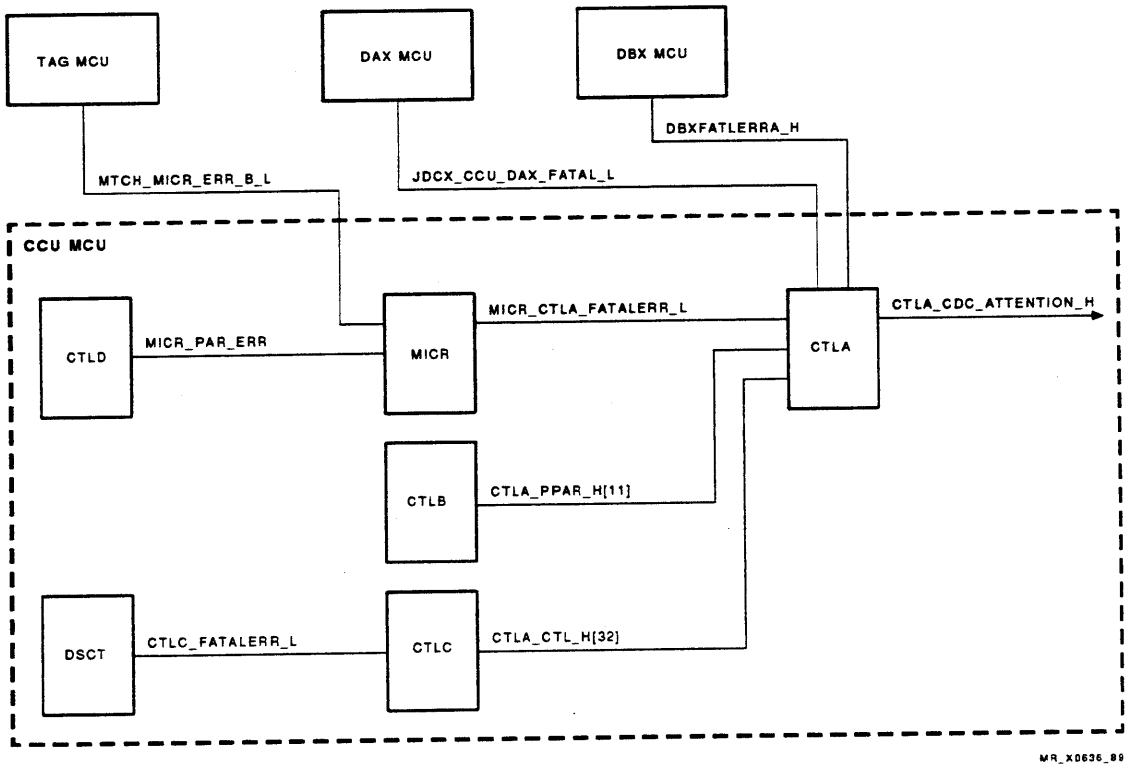


Figure 7–2 SCU Error Reporting

Table 7–2 MCU Attentions

Event	CCU	DA0	DA1	DB0	DB1	Tag
Error detected in CCU	Y	N	N	N	N	N
Error detected in DA0	Y	Y	N	N	N	N
Error detected in DA1	Y	N	Y	N	N	N
Error detected in DB0	Y	N	N	Y	N	N
Error detected in DB1	Y	N	N	N	Y	N
Error detected in Tag	Y	N	N	N	N	Y
XJA0 or XJA1 retry	N	Y	N	N	N	N
XJA2 or XJA3 retry	N	N	Y	N	N	N
End of BIST MMU0	N	N	N	Y	N	N
End of BIST MMU1	N	N	N	N	Y	N

7.4.1 DAX MCU Errors

Figure 7–3 shows the DAX errors.

On the DAX MCU, six MCAs report their errors to IRCX, which sends FATAL_ERROR_L to JDCX. JDCX sends JDCX_CCUX_DAX_FATAL_L to CTLA, resulting in attention being asserted by CDC on CCU. JDCX also asserts ERR_ATTEN_H to CDC on DAX, resulting in attention being asserted by that CDC.

Each DSXX MCA sends a fatal error signal to either the IRCX MCA (DAX MCU) or the MMCX MCA (DBX MCU). The signal is asserted only when a DSXX detects a parity error on control lines from the data switch controller DSCT.

The data switch MCAs also check parity on incoming data but do not assert fatal error parity errors. When DSXX detects a data parity error from MDPX, MBox, or JDAX, it generates and latches an error signal. For MDPX, the signal is MDPLATPARERR. The latched error signal is held in its latch while the error is allowed to propagate to the next MCA, where it is detected again. Error data is sent to the following:

- **ICU** — An error is flagged in JDBX and results in attention to the SPU.
- **IRCX** — An error is flagged and results in set attention.
- **Memory** — An error is flagged in MDPX but not reported, and the bad data is written into memory.

Data received from the other data switch is handled in the same manner.

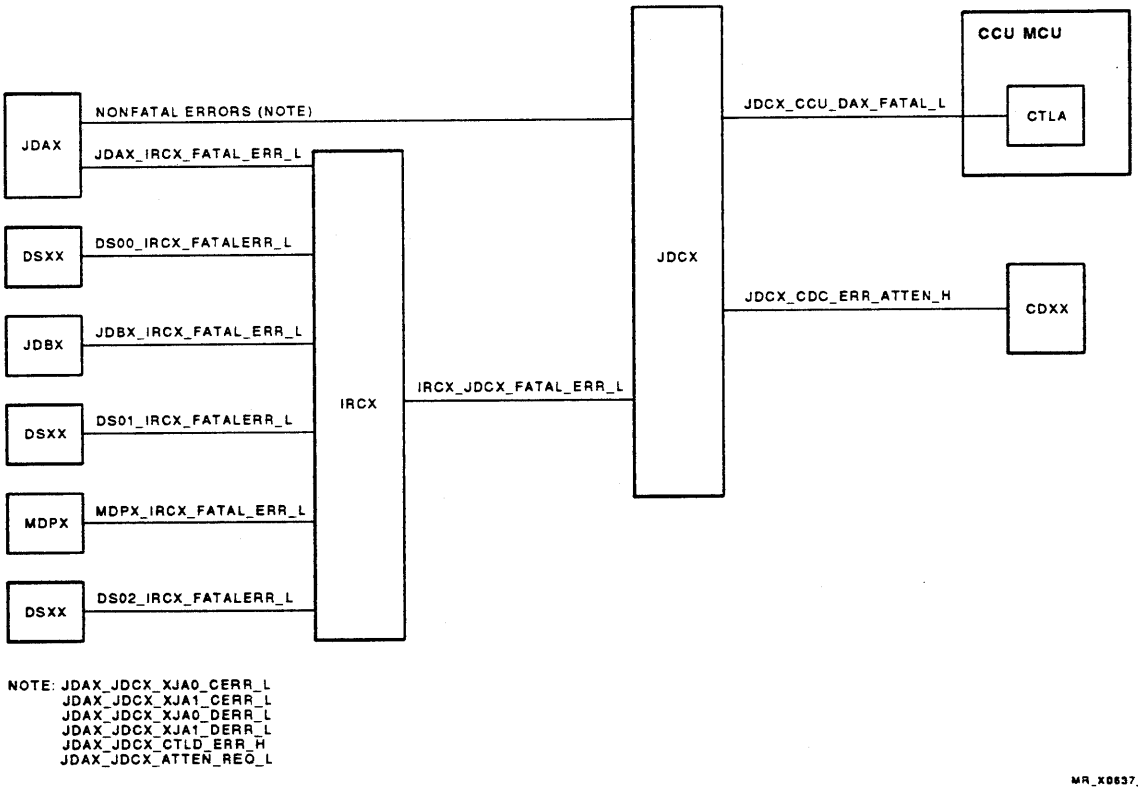


Figure 7–3 DAX Errors

7.4.2 DBX MCU

Figure 7–4 shows the fatal errors sent to the MMCX MCA on the DBX MCU.

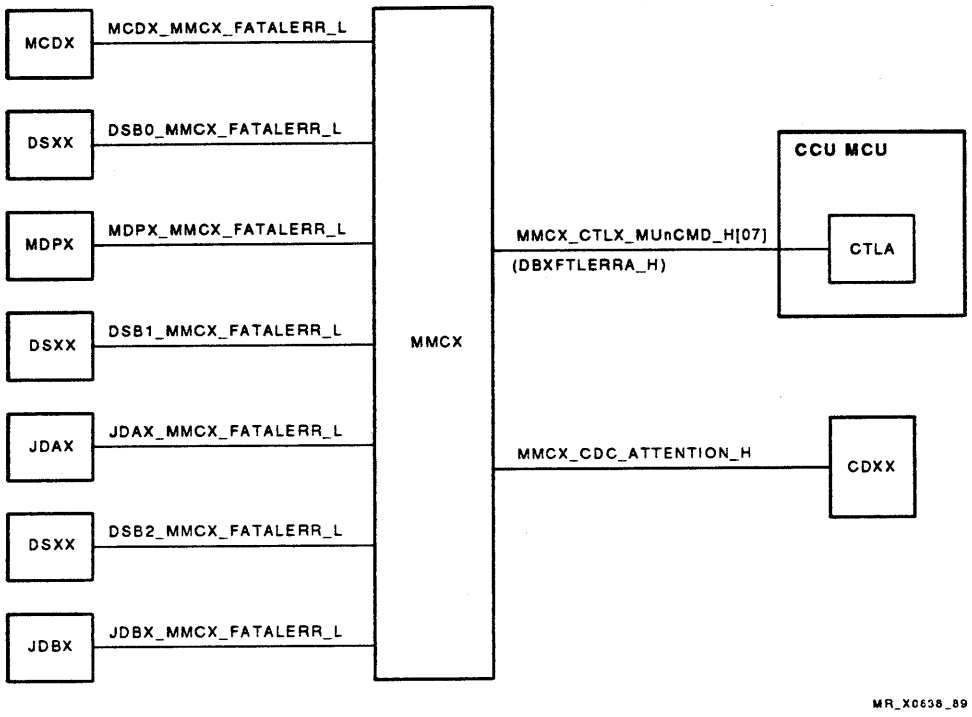


Figure 7–4 DBX Errors

7.4.3 Tag MCU Errors

Figure 7–5 shows the tag MCU errors.

On the tag MCU, the MTCH chip collects error signals from the four ADRX MCAs and reports them to CTLA using MTCH_MICR_ERR_B_L. The STRAMs on the tag MCU do not report errors to MTCH. Instead, parity bits are written into the global tag STRAMs with the data and are checked on MTCH when the data is read. The MTCH MCA also has the ability to assert MTCH_CDC_EXCEPTION_B_H to the CDC chip on the tag MCU.

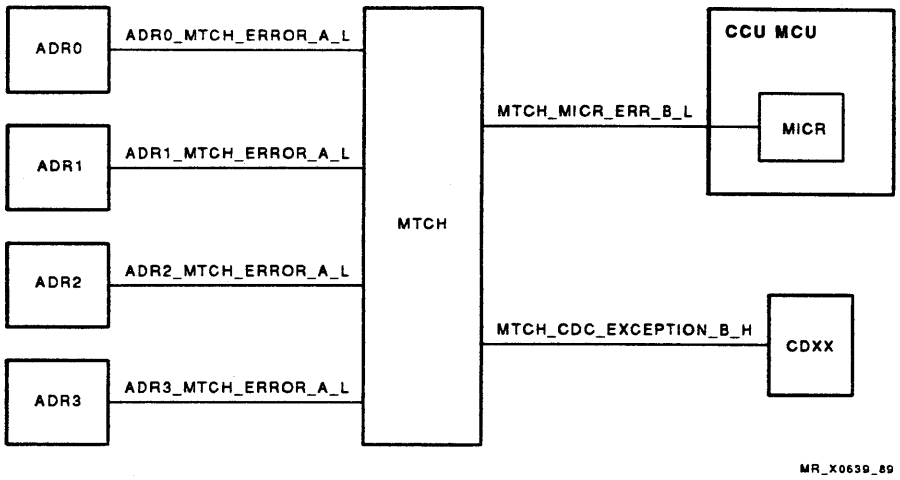


Figure 7–5 Tag Errors

7.4.3.1 Stop on Address Match

The MTCH MCA contains stop on address match logic, which compares PA [32:06] from the ADRX MCA and the tenth and eleventh bits from CTLD, with an address supplied by scan. Figure 7–6 shows the stop on address match logic. If a match occurs, STOP_MATCH_B_H can be sent to the CSSE connector, if enabled by scan. Scan determines whether a match on PA [32:06], the tenth and eleventh bits, or both sets STOP_MATCH_B_H.

STOP_MATCH_B_H can be used to send an attention to SPU as an error condition. The SPU takes the appropriate action for the address match condition.

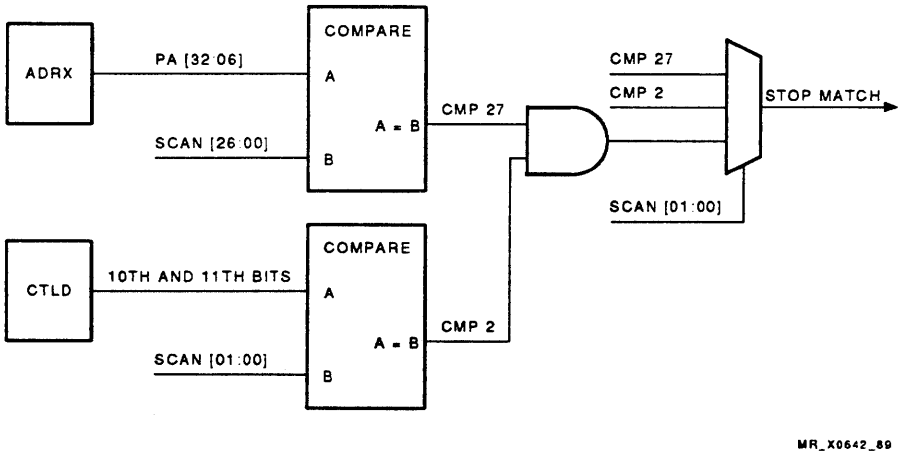


Figure 7–6 Stop on Address Match

7.4.3.2 Force Attention Logic

MTCH receives CSSE_TAG_FORCE_ATTN_H from the CSSE connector on the SCU planar module. Figure 7–7 shows the force attention logic. This signal is asserted by a logic analyzer or other test equipment in response to a predefined combination of signals output to the connector. This signal forces the error conditions, MTCH_CDC_EXCEPTION and MTCH_MICR_ERROR_B_H, when selected to do so by scan, which tests to see if MICR detects that it is set.

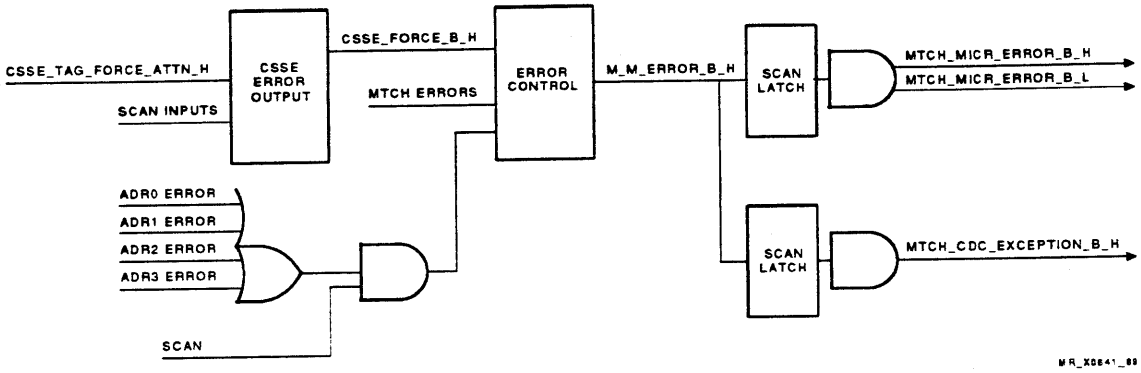


Figure 7–7 Force Attention Logic

7.4.4 CCU MCU Errors

Errors other than SBEs are handled by invoking SPU, in its capacity as a scan controller, to assist in evidence collection and recovery activities.

7.4.4.1 History Buffer

The control store STRAMs include a 1K visibility section that stores the last 1K microaddresses. The history buffer can be stopped by a scan, address match, or CSSE connector.

The history buffer is a free-running FIFO and can be locked by any of the following mechanisms:

- Directly by scan, by the assertion of SCAN_DUMP_H.
- Directly by the CSSE connector, by the assertion of CSSE_STOP_DUMP_H. A logic analyzer at the connector can be set up to assert this signal on any combination of other signals.
- Microaddress match, by the assertion of MATCH_STOP_H. Scan defines a microaddress (SCAN_MATCH_H[09:00]). When each microaddress appears at the output of the microcode STRAM as the next microaddress to be accessed, it is compared with the scan input. If a match occurs between the two, UADR_MATCH_H[09:00] is asserted. As a result, MATCH_H is asserted and causes MATCH_STOP_H to be asserted if enabled by scan.

Figure 7–8 shows the signals that can stop the history buffer.

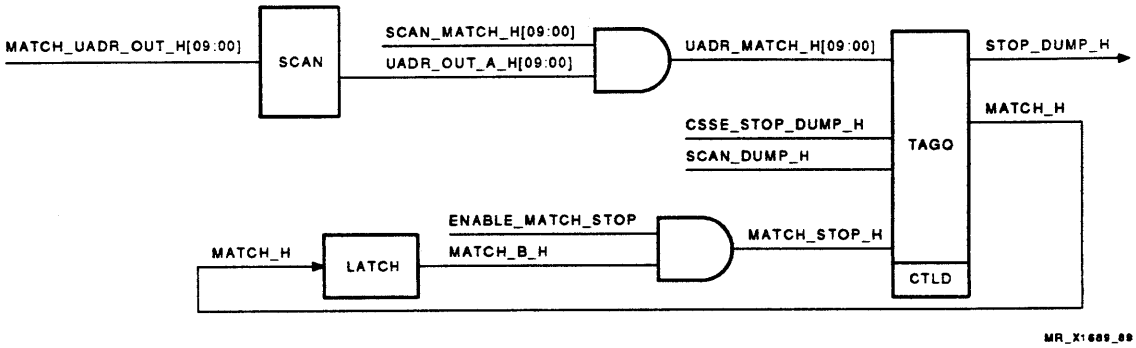


Figure 7–8 History Buffer

7.4.4.2 Error Detectors

All of the MCAs in SCU have some number of error detectors. In general, each MCA ORs its error signals and sends one line to the CCU logic.

The CCU ORs all error signals and sends the resulting signal to the clock chip on the CCU MCU. This clock chip asserts the attention line to the scan controller in SPU. Note that the CTLC MCA contains the logic that interfaces to the scan controller.

CTLA receives composite error signals from other MCAs on the CCU MCU and from other MCUs in SCU. CTLA sends attention to CDCX. CDCX asserts SCAN_BUS_DATA_OUT_H, which alerts the scan control module in SPU.

Following receipt of the attention line, the scan controller does a dynamic scan, with the system clocks running, of the loop containing the attention bits and determines which MCU asserted the attention line. Next SPU stops the system clocks, and uses the scan mechanism to extract error information from various registers within SCU. Once this has been done, SPU restarts the system clocks. Then SPU examines the error evidence to diagnose the problem and the appropriate corrective action. There are then two ways that SPU can affect error recovery as follows:

- Stop the system clocks again and use its scan capability to fix the problem. Then restart the system clocks.
- Use its capability as an I/O controller to fix the problem without stopping the system clocks.

If an interlock bit is stuck set in SCU, perhaps due to a faulty I/O controller, SPU uses its capability as an I/O controller to issue a cancel lock command to SCU, which clears the problem.

7.5 Recovery

This section describes the types of error recovery mechanisms. Key error signals involved with the error detection mechanism are also described.

7.5.1 Dynamic Error Recovery

Dynamic error recovery is provided for single-bit memory errors only. Other types of errors require the assistance of SPU and the scan system to support error recovery.

7.5.2 SPU-Assisted Recovery

When SCU detects an error, it asserts the attention line to SPU, logs error status, and sends error signals to other units (MBox), if appropriate. SPU stops the clocks, scans out the error evidence, and clears the error flags and status register locks. SPU then restarts the system clocks. The SPU examines the error evidence to diagnose the problem and the appropriate corrective action.

7.5.3 Operating System Implications

During the SPU-assisted recovery process, the system clocks have been stopped for a period of time. Depending on how long the clocks have been stopped, some I/O transfers may have timed out.

If this happens frequently when the error recovery strategy is invoked, the operating system must be able to deal with the I/O timeouts and to reinitiate I/O transfers if necessary.

7.5.4 Key Signals

Table 7–3 lists the signal groups and the corresponding error detection mechanisms.

Table 7–3 Error Signals

Signal Group	Detection	Fatal	Description
MMCX	1 parity	Y	JBox command, control, and status to MMCX
	1 parity	Y	JBox mask to MMCX
	1 parity	Y	MCDX command, control, and status to MMCX
	Toggle	Y	JBox address to MMU and toggle to MMCX
	1 parity	Y	MMCX command, control, and status to MCDX
	1 parity	Y	MMCX control to MDPX
	1 parity	Y	MMCX command and status to JBox
	1 parity	Y	MMCX index row and column select to JBox

Table 7–3 (Cont.) Error Signals

Signal Group	Detection	Fatal	Description
MCDX	1 parity	Y	MMCX command, control, and status to MCDX
	2 parity	Y	MCDX DRAM control to MMU, parity back
	1 parity	Y	MCDX command, control, and status to MMCX
MDPX	1 parity	Y	MMCX control to MDPX
	2 parity	N	JBOX data to MDPX
	ECC	N	MMU data to MDPX
	2 parity	Y	MDPX data to JBox
DSXX	1 parity	Y	DSCT control to DSXX
DSCT	1 parity	Y	DSCT control to DSXX
	1 parity	Y	DSCT status to CTLX
	1 parity	Y	DSCT status to MICR

7.6 Types of Error Detection

The types of detection include the following:

DRAM control errors
JBox-to-MMU address errors
Protocol errors
Incidental errors
Forced errors

7.6.1 DRAM Control Error Detection

Each of the two memory segments receives a group of DRAM control signals. DRAM control includes the RAS, CAS, WE, and CAS mask signals. The MCDX MCA supplies the RAS, CAS, and WE signals. The MMCX supplies the CAS mask signals.

The CAS mask control signals are used to set bits in the CAS mask segment register in the DRAM control and address (DCA) gate array on the memory module. During a write operation, DCA generates odd parity across the eight bits in this register, along with the RAS, CAS, and WE signals. This odd parity is transmitted to MCDX.

MCDX receives the odd parity (CTLPAR) bit from each DCA for each segment. MCDX also receives CAS mask parity from MMCX. The MCDX then checks the parity against the expected parity at distinct times. The parity check is done at the third clock cycle after CAS is asserted and when the cycle is inactive. Waiting three cycles allows for the round-trip delay. If MCDX does not receive correct parity, it signals a fatal error.

7.6.2 JBox-to-MMU Address Error Detection

The JBox transmits the row and column address from four ADRX MCAs. Each ADRX transmits four address bits and one parity bit. Therefore, a total of 16 signals are received by each DCA.

DCA performs a parity check on the address lines. If parity error is not detected, the DCA toggles the ADRPARTOG line to MMCX. This technique checks both the address lines and control lines used by MMC to strobe the address. If MMCX does not receive the ADRPARTOG signal when expected, it signals a fatal error.

7.6.3 Protocol Error Detection

The following protocol errors are detected by MMCX:

- MMC receives BOD without first getting a write command from the JBox.
- MMC does not receive a BOD after receiving a write command.
- Timeout occurs before receiving cancel/OK status.

Protocol errors are reported as fatal errors.

7.6.4 Incidental Error Detection

Incidental errors do not affect normal system operation, but these errors do imply that a failure has occurred. The following incidental errors are detected:

- A parity error is detected by MMCX on inactive command lines from the JBox.
- A parity error is detected by MDPX on inactive write data lines.

Incidental errors are picked up during scan operation.

7.7 Types of Errors

The following are types of errors:

Fatal
Write data
Read data
Forced

7.7.1 Fatal Errors

Each MCA on ACU dedicates one signal out to report fatal errors. On each MCU, all of the fatal error signals from the MCAs are connected to one MCA. MMCX serves this purpose on the DB0 and DB1 MCUs. The OR of the fatal error signals then routes to the CCU MCU in the JBox. CCU, in turn, pulls the attention line on the CDCX chip, which signals the service processor to take action.

7.7.2 Write Data Errors

Write data is split between two MDPs. Each MDPX receives data grouped with odd parity, which is assigned as follows:

- 1 parity bit for 16 data bits, 2 mask bits, and 1 BOD bit
- 1 parity bit for 16 data bits and 2 mask bits

MDPX checks the write data parity from the data switch and generates LWPARERR_H, HWPARERR_H, or both if it detects bad parity on the low word, high word, or both. LWPARERR_H or HWPARERR_H sets the mark bit (bad merge data can also set the mark bit) and error data [13]. The error is not reported until the data is read back from memory. Bad parity and bad merge data cause the check bits to be inverted. The combination of inverted check bits and an asserted read mark bit ensures that the decoder detects the data as uncorrectable even if a double-bit error occurs when the address being written is eventually read.

The MDPX calculates odd parity on each group of signals and compares it to the received parity signal. If an error is detected, MDPX latches the error in the MDPX data error register. In addition, MDPX signals MMCX over the MDPX-to-MMCX status lines.

In response to the write data error, MMCX inhibits transmission of WRT_OK to the JBox, and writes the data to memory, marked bad. Marking the data is done by MDPX.

MDPX marks bad data as follows:

1. Bad data is passed through the check bit to generate logic. Seven check bits are produced.
2. The seven check bits are inverted.
3. The mark bit is set.
4. Bad data, inverted check bits, and the mark bit are written to the addressed memory location.

When the bad data is read back, the ECC checking logic detects the double-bit error and reports it.

When MDPX detects an ECC error on memory read data, it reports the condition to MMCX using MDPX_MMCX_STATUS_H[00], when enabled to do so. Error handling software may disable the error reporting function temporarily when correctable error thresholds are exceeded, in order to minimize the impact on system performance.

7.7.3 Read Data Errors

A read data error is defined as a single- or double-bit error occurring in the read data from MMU. Read data from MMU is split between two MDPX MCAs, with each MDPX receiving 32 data bits, 7 check bits, and 1 mark bit. Read data is sent to the generate syndrome logic. This logic generates new check bits on the received data bits and XOR with the received check bits. The result is a syndrome. An all-zero syndrome indicates there is no error. An even number of ones indicates a double- or multiple-bit error. An odd number of ones is used to indicate a single-bit error.

If an error is detected, MDPX does the following:

1. Latches the error in the MDPX data error register.
2. Signals MMCX of the error over the MDPX-to-MMCX status lines.
3. Corrects the SBE.
4. If a DBE, forces bad parity on the output to JBox.

The last three events can be disabled by scan during initialization.

When MMC receives the error report over the status lines, it does the following:

1. Transmits to the JBox that a read error occurred.
2. Receives send data bit from the JBox.
3. Transfers the contents of the MDPX data error register through the JBox to SPU (this is a one-cycle transfer).

7.8 Error Registers

The SCU has three error registers:

- **MDPX data error register** — Latches the occurrence of control and data errors.
- **MMC error register** — Latches the occurrence of data, address, and protocol errors.
- **MCD error register** — Latches the occurrence of DRAM control errors.

7.8.1 MDPX Data Error Register

The MDPX latches the occurrence of control and data errors. All data errors, except for write parity errors, are logged into an error register. If a data error occurs, the contents of this register are transmitted to the service processor.

Parity errors on write data received from the JBox and parity errors on control signals received from MMC are latched and can be read only by a scanning operation.

When an ECC error occurs, the error information is latched in the data error register. The contents of this register are transferred to SPU. The register is shown in Figure 7–9; the contents are listed in Table 7–4.

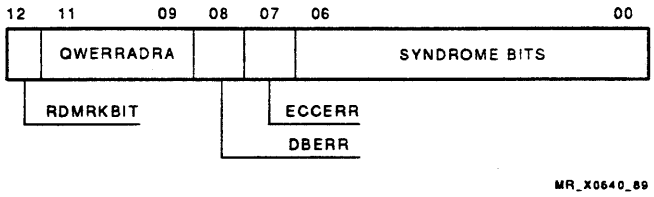


Figure 7–9 MDPX Register

Table 7–4 MDPX Error Register

Bit	Error Type	Description
MDP_ERR[12]	Error flag	When set, this bit indicates that the ECC logic has detected an SBE or DBE.
MDP_ERR[11]	0 = SBE, 1 = DBE	A DBE overwrites an SBE. Otherwise, the first error detected is latched and remains latched until the register contents are read.
MDP_ERR[10:04]	Syndrome	The error syndrome bits indicate which bit is in error for an SBE. SBEs can be a data bit or check bit, or can indicate an address parity error.
MDP_ERR[03]	Bad data flag	This bit indicates that the mark bit was set when read. There are two conditions under which this bit is set. In one, data was purposely marked bad when last written. In this case, the error flag is also set, along with an SBE or DBE indication. The second condition is when the mark bit itself is bad. In this case, the error flag is not set.
MDP_ERR[02:00]	Quadword address	Up to eight quadwords can pass through ACU as part of a read request. When MDP signals an error, MMC loads the associated quadword address.

The mark bit is used to identify bad write data. Write data errors are not reported unless the data is read. The disadvantage of this method is that the time of the error occurrence cannot be known. MDP does check parity on write data coming from the data switch latching LWPARERR_H or HWPARERR_H on detection of an error on the low word and high word, but the error is not reported at that time. These signals are used to generate the mark bit. Also, the ECC check bits are inverted before the data is written into memory.

7.8.2 MMC Error Register

The MMC error register latches the occurrence of data, address, and protocol errors. The register is shown in Figure 7–10. Table 7–5 lists the error bits and their description.

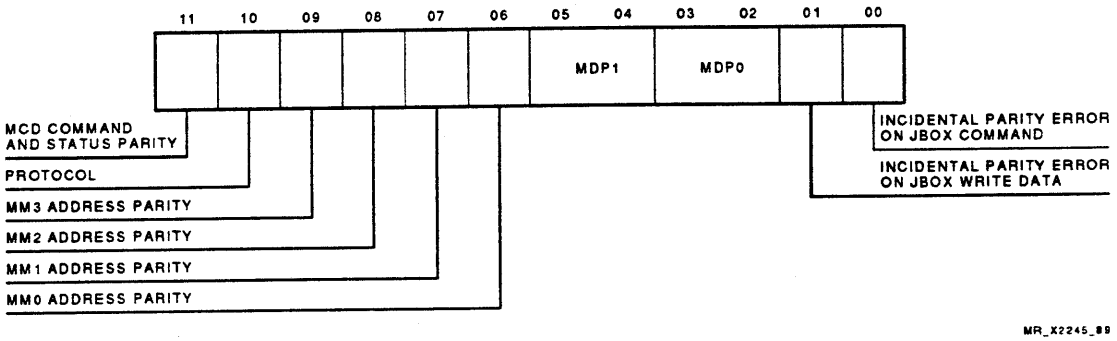


Figure 7–10 MMC Register

Table 7–5 MMC Error Register

Bit	Error Type
MMC_ERR[00]	Incidental parity error on JBox command
MMC_ERR[01]	Incidental parity error on JBox write data
MMC_ERR[03:02]	MDP0[1]
MMC_ERR[05:04]	MDP1[2]
MMC_ERR[06]	MM0 address parity
MMC_ERR[07]	MM1 address parity
MMC_ERR[08]	MM2 address parity
MMC_ERR[09]	MM3 address parity
MMC_ERR[10]	Protocol
MMC_ERR[11]	MCD command and status parity

[1]0x = No error, 10 = SBE/DBE, 11 = Write parity

[2]0x = No error, 10 = SBE/DBE, 11 = Write parity

7.8.3 MCD Error Register

The MCD error register latches the occurrence of DRAM control errors. The register is shown in Figure 7–11. Table 7–6 lists the errors in the MCD error register.

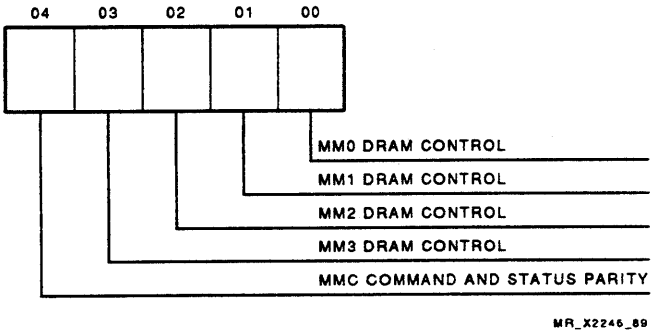


Figure 7–11 MCD Register

Table 7–6 MCD Error Register

Bit	Error Type
MCD_ERR[00]	MM0 DRAM control
MCD_ERR[01]	MM1 DRAM control
MCD_ERR[02]	MM2 DRAM control
MCD_ERR[03]	MM3 DRAM control
MCD_ERR[04]	MMC command and status parity

Index

C

D

E

F

G

I

J

L

M

N

P

R

S

T

U

V

W

X

N

P

R

S

T

U

V

W

X