Second Edition - August 1984

This manual describes the KD32-AA and KD32-AB central processing units used in the MicroVAX I.

MicroVAX I
CPU Technical Description

Document Order Number: **EK-KD32A-TD-002**

digital equipment corporation
maynard, massachusetts

First Edition, January 1984
Second Edition, August 1984

Printed in U.S.A.

Contents

Chapter 4: Functional Overview

Chapter 5: Data Path Microcode

Chapter 9: Q22 Bus Controller

List of Figures

List of Tables

Preface

Manual Scope

This manual is a technical description of the central processing unit (CPU) used in the MicroVAX I system. The MicroVAX I can have either the KD32-AA CPU (F_ and G_floating point) or the KD32-AB CPU (F_ and D_floating point). Both processors consist of two quad-height modules:

- The data path module (M7135 for the KD32-AA CPU, M7135-YA for the KD32-AB CPU), and
- The memory controller module (M7136).

This technical description is intended as a field reference for DIGITAL Field Service personnel and a resource for training programs conducted by Educational Services and Manufacturing. A knowledge of VAX architecture is assumed.

Chapter 1 is a general description of the MicroVAX I system.

The next two chapters comprise a user's guide for the KD32-AA and KD32-AB processors. Chapter 2, "Programming Interface," contains information a MicroVAX I programmer needs to know such as the system physical address space, the macrolevel registers, the boot EPROM, and machine checks. Chapter 3, "Module Configuration," describes the factory configuration of the processor and how to change it, plus power and cooling specifications.

Beginning with Chapter 4, the remaining chapters provide a "theory of operation" description of the

processor modules. Chapter 4 is a functional overview of the CPU, Chapters 5 and 6 describe the data path module microcode and hardware, Chapters 7 and 8 describe the memory controller module microcode and hardware, and Chapter 9 describes the Q22 bus controller.

The *MicroVAX I Field Service Print Set*, MP-01896-01, contains schematic diagrams for the processor modules. Signal names in the *MicroVAX I CPU Technical Description* are prefaced by four-letter codes which reference pages in the *Print Set*. You may find it helpful to refer to the *Print Set* as you read the *Technical Description*.

Related Documentation

The *MicroVAX I CPU Technical Description* is part of the hardware documentation set for the MicroVAX I system. Related manuals that may be of interest are:

- *MicroVAX I Owner's Manual*, EK-KD32A-OM. This book contains installation, operation, diagnostics, troubleshooting, removal and replacement procedures, and system configuration information for the MicroVAX I system.
- *MicroVAX Handbook*, EB-25156-47. This book contains descriptions of the MicroVAX I system and related products: peripherals, interfaces, operating systems, and communications software.
- *VAX Architecture Handbook*, EB-19580-20. The MicroVAX I system design is based on the VAX architecture described in this handbook.
- *Microcomputer Interfaces Handbook*, EB-20175-20. This handbook is a reference guide for the interface and peripheral hardware options that can be

installed on the Extended LSI-11 Bus used in the MicroVAX I system.

- *Microcomputers and Memories*, EB-20912-20. This manual contains a detailed description of the Extended LSI-11 Bus.

Chapter 1
Introduction

This chapter introduces the MicroVAX I system. It contains information about the system necessary for understanding the MicroVAX I central processing unit (CPU).

System Overview

The MicroVAX I system is a 32-bit, high-performance, microprogrammed computer. The processor executes the VAX-11 instruction set and contains an interface to the extended LSI-11 bus (Q22 bus). PDP-11 compatibility mode is not supported.

The major components of the MicroVAX I system, shown in Figure 1-1, are:

- The processor, which consists of two modules:
  - data path module (DAP)
  - memory controller module (MCT)
- Q22 bus
- RQDX1 controller
- RX50 diskette drives
- RD51 or RD52 fixed disks
- Q22 memory, with block mode capability
- Console terminal
- Front control panel
- Rear patch panel assembly

The Q22 bus-compatible system box also contains a backplane and power supply.

Processor

The processor consists of two quad-height modules and contains:

- An interface to the Q22 bus which supports block mode transfers and up to four megabytes of physical memory
- An 8 KB direct-mapped cache
- A 512 entry (longword) translation buffer
- A 10 ms nonprogrammable interval timer
- An interface to a console serial line unit
- An 8 KB or 16 KB boot EPROM
- Interfaces to the front control panel and rear patch panel assembly.

Two processors are available for the MicroVAX I system. The KD32-AA processor contains microcode to handle F_ and G_floating point instructions. The KD32-AB processor contains microcode to handle F_ and D_floating point instructions.

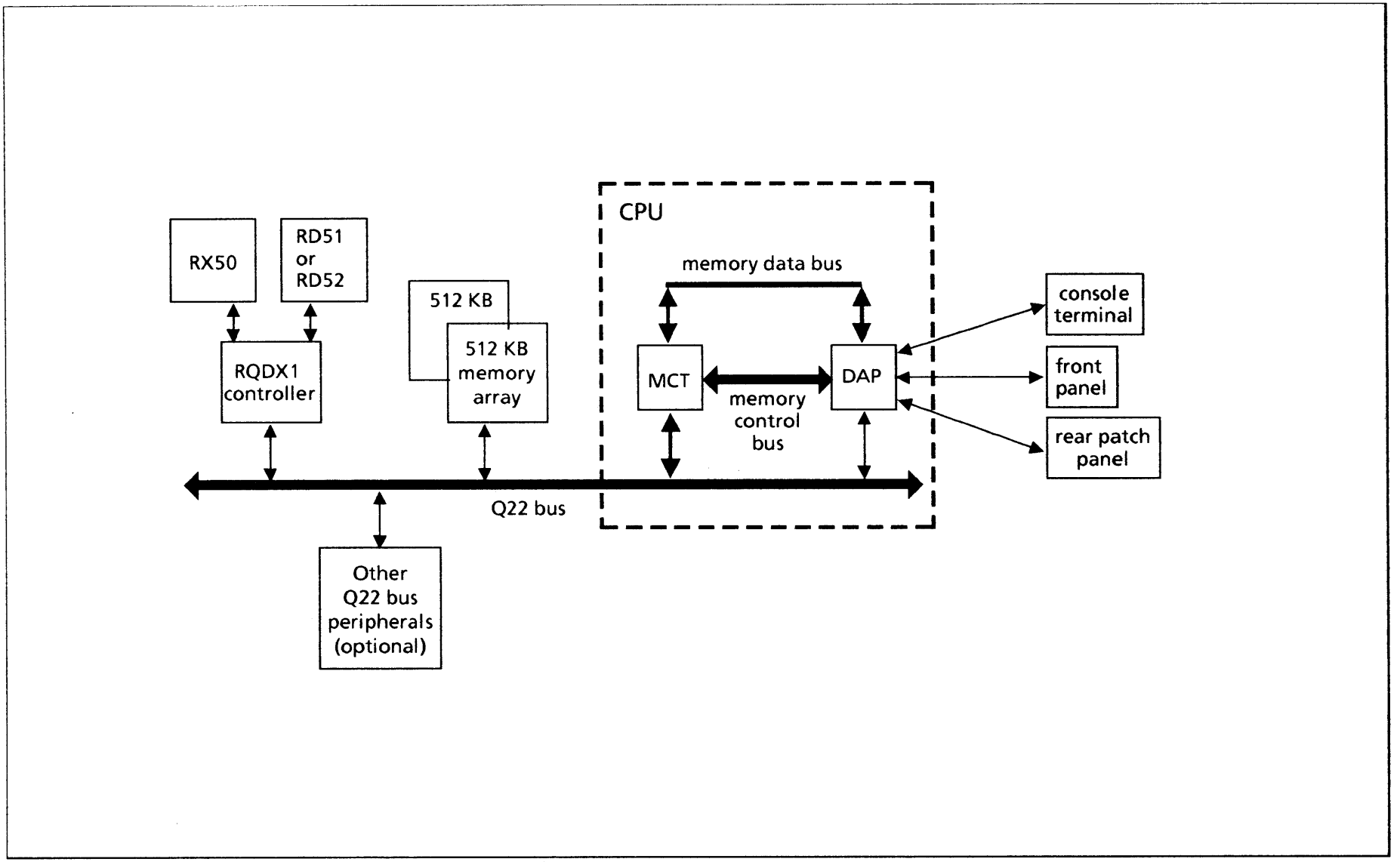


Figure 1-1. MicroVAX I System

Q22 Bus

The MicroVAX I system backplane uses the extended LSI-11 bus (also called the Q22 bus), which has 22-bit addressing. The Q22 bus consists of 42 bidirectional and 2 unidirectional signal lines. These are the lines along which the processor, memory, and I/O devices communicate with each other. MicroVAX I performs the following Q22 bus data transfer functions:

DATI	read word
DATO	write word
DATOB	write byte
DATIO	read, modify, write word
DATIOB	read, modify, write byte
DATBI	read block
DATBO	write block

RQDX1 Controller

The RQDX1 controller (M8639) is a quad-height module that occupies the last-used slot in the backplane. It is the interface between the Q22 bus and the RX and RD disk drives. The controller is a direct memory access (DMA) interface and uses mass storage control protocol (MSCP). It also provides support for gather-read and scatter-write operations; that is, transfers do not have to be physically contiguous.

RX50 Diskette Drive

The RX50 is a random access storage device with two diskette drives. It uses single-sided 5.25 inch (13.34 cm) diskettes. The total drive capacity is 800K bytes of formatted data. Each drive has an access door and slot for inserting and removing diskettes. A head load LED

for each diskette slot informs the user when that unit is busy.

The RX50 is a field replaceable unit (FRU) that mounts in the MicroVAX I system box. Cables connect the RX50 to the RQDX1 controller and the power supply. See the *MicroVAX I Owner's Manual* for removal and replacement procedures.

RD51 and RD52 Fixed Disk Drives

The RD51 is a random access storage device which uses two nonremovable 5.25 inch (13.34 cm) disks as storage media. One movable head per disk surface services 153 data tracks. The total formatted capacity of the four heads and surfaces is 10 megabytes.

The RD52 is a random access storage device which also uses nonremovable 5.25 inch (13.34 cm) disks as storage media. The total formatted capacity of the RD52 is 31 megabytes.

The RD51 and RD52 are field replaceable units (FRUs) that mount in the MicroVAX I system box. A control cable and one data cable connect the RD51 or RD52 drive to the RQDX1 controller. Another cable connects the RD51 or RD52 drive to the power supply. See the *MicroVAX I Owner's Manual* for removal and replacement procedures.

The RD51 is also available as the RD51-D (desk top) or RD51-R (rack mount) disk subsystem. Similarly, the RD52 is available as the RD52-D (desk top) or RD52-R (rack mount) disk subsystem. The RD51-D, RD51-R, RD52-D, and RD52-R are freestanding, outboard fixed disk subsystems that contain their own power, cooling, console, and I/O cable. The RQDX1 controller plus the RQDX1-E bus extender card are the interface between the Q22 bus and the disk subsystem.

Memory

MicroVAX I relies on block mode Q22 bus data transfer functions to realize its performance goals. Therefore, MicroVAX I systems are configured with MSV11-P memory modules, which have block mode capability.

The MSV11-P family of memory modules are quad-height modules that implement an 18-bit wide random access memory array (16 data bits and 2 parity bits), parity generation and detection, and on-board refresh circuitry. There are two variations:

MSV11-PL	512 KB of storage using 64K MOS RAMs
MSV11-PK	256 KB of storage using 64K MOS RAMs

Console Terminal

The console terminal may be any member of the VT100 or VT200 family of terminals. A cable connects the terminal to an EIA connector on the CPU patch panel, located at the back of the MicroVAX I system box. Appendix C lists the pinout for the EIA connector.

An internal cable attaches the EIA connector on the CPU patch panel to a 10-pin connector on the data path module. A terminal interface UART and an RS232/423 driver and receiver pair are located on the data path module.

Front Control Panel

The front panel provides control and status of the various components of the system. The switches and indicators are shown in Figure 1-2. The switches and their functions are listed in Table 1-1. The indicators and their meanings are listed in Table 1-2.

The front control panel is a field replaceable unit (FRU). See the *MicroVAX I Owner's Manual* for removal and replacement procedures.

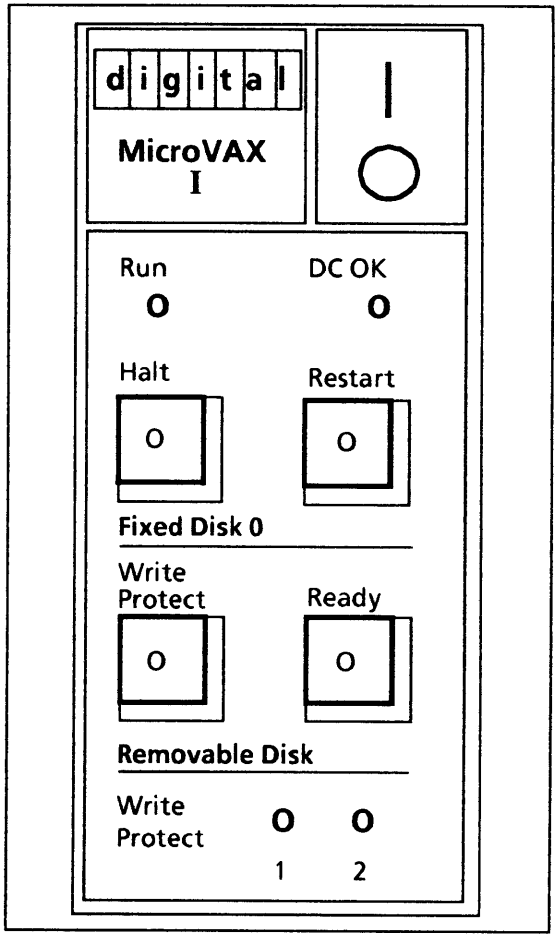
Patch Panel Assembly

External option cables and serial lines connect to the MicroVAX I through the rear patch panel assembly. The patch panel assembly provides shielding for EMI and accommodates a variety of connectors by providing six areas for patch panel inserts; the connectors are an integral part of the patch panel inserts. The inserts mount in cutouts in the sheet-metal frame of the patch panel assembly. Four screws hold each insert in place. When an area is not occupied by an insert, a metal plate covers the cutout for the insert.

Four of the patch panel areas are 2×3 inches; the other two areas are each 1×4 inches. An alternate configuration can be created by removing the divider post between the third and fourth patch panel areas and installing three 1×4 inch inserts.

The 2×3 inch patch panel areas can each accommodate an insert with four 25-pin EIA connectors. The 1×4 inch areas can each accommodate an insert with one 40-pin or one 50-pin EIA connector.

The insert for the KD32-AA or KD32-AB CPU is installed in the first 2×3 inch patch panel area. This CPU patch panel contains one 25-pin EIA connector, one rotary baud rate select switch, and a two-digit LED display. The rotary switch sets the system baud rate; the choices are 300, 1200, 9600, and 19,200 baud.



Front Panel
Floor Mount Version

Table 1-1. Front Panel Switches

Switch	Position	Function
1, 0	1	Turns on the system power.
	0	Turns off the system power.
Halt	In (LED lit)	The processor halts and responds to console commands.
	Out (LED off)	Enables the processor to run.
Restart	In (momentary switch)	When the halt switch is out (LED off), the processor carries out a power-up sequence. When the halt switch is in (LED lit), this button has no effect.
Write Protect	In (LED lit)	Write protects fixed disk 0.
	Out (LED off)	Enables writing to fixed disk 0.
Ready	In (LED off)	Places fixed disk 0 off-line.
	Out (LED lit)	Places fixed disk 0 on-line.

Table 1-2. Front Panel Indicators

LED	Function
Run	The Run LED is on when the processor is operating; the LED goes off when the processor is not executing instructions.
DC OK	This LED is on when the power supply is generating correct DC power output voltages.
Removable Disk Write Protect	
1	When lit, the diskette in drive 1 is write protected.
2	When lit, the diskette in drive 2 is write protected.

Figure 1-2. MicroVAX I Front Panel

Backplane

The backplane (H9278-A) is a four-row by eight slot backplane capable of accepting either quad- or double-height modules. The backplane uses the Q22 bus structure in the A and B connectors of slots 1 through 8, and in the C and D connectors of slots 4 through 8. A slot-to-slot interconnection scheme (referred to as the CD interconnect) is wired in the C and D connectors of slots 1 through 3. The CD interconnect connects selected side two pins in rows C and D of a given slot to side one pins of the slot immediately following. There are 32 such connections per slot.

The backplane receives and distributes two voltages and ground. Maximum ratings are +5 volts at 36 amps, and +12 volts at 6 amps.

The backplane includes four connectors, J1 through J4, which are mounted on side two of the backplane. J1 (eighteen pins), J3 (four pins), and J4 (four pins), connect power supply outputs to the backplane. J2 (ten pins) connects the backplane to the front panel.

The backplane also includes provision for the insertion of four resistor packs (p/n 1318110-00) into positions XZ1, XZ2, XZ3, and XZ4. In a MicroVAX I system (single backplane), these resistor packs are inserted to terminate the Q22 bus lines. (Characteristic impedance is 220 ohms).

Figure 1-3 shows the backplane organization. The numbers in the parentheses following the Q22 designations show the path of interrupt and direct memory access grant continuity for options installed in the backplane; increasing value denotes lower priority. Each slot requires the insertion of a module or a bus grant continuity card to pass these grant signals on as

no jumpers are provided on the backplane for this purpose.

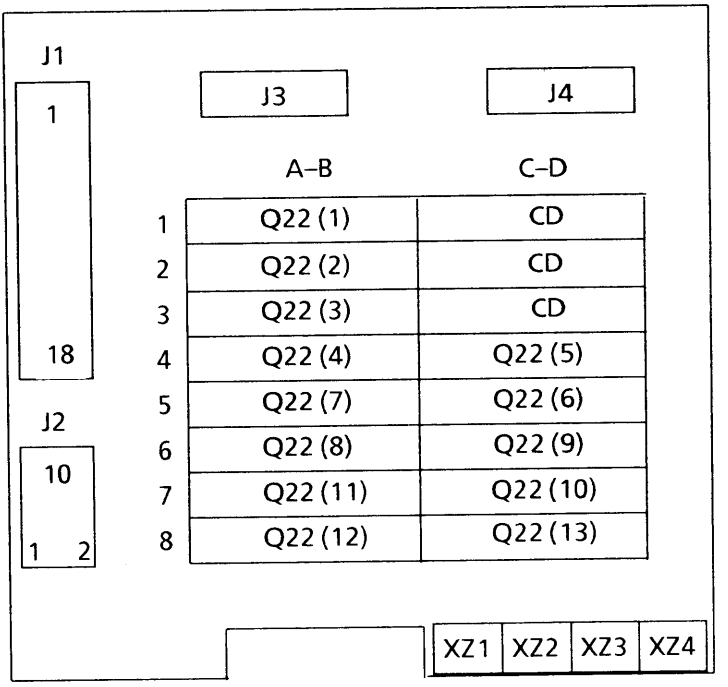


Figure 1-3. MicroVAX I Backplane

Power Supply

The power supply (H7864) is a modular, 230-watt power supply that supplies from 4.5 amps minimum to 36 amps maximum at +5 volts, and 0 to 7 amps at +12 volts. There are also two outputs designed to accommodate DC brushless fans, not included in the 230-watt power specification. These outputs supply 0.45 amps at +12 volts and +9 volts.

Other power supply features include thermal shut down, overvoltage and overcurrent protection, AC input transient suppression, and three Q22 bus signals (BPOK, BDCOK, BEVNT; see Appendix A for signal definitions). In addition, the power supply has a full cycle ride-through feature; that is, the power supply maintains the output voltages at operating level for a minimum of one 60 Hz cycle if the AC input voltage drops.

The power supply includes connectors that provide the necessary power and signal interfaces to the logic backplane, mass storage units, front panel, and fans.

The power supply is a field replaceable unit (FRU). See the *MicroVAX I Owner's Manual* for removal and replacement procedures.

System Architecture

The VAX architecture is an architecture designed by DIGITAL for its family of 32-bit, virtual memory minicomputers. DIGITAL has also defined a subset of the VAX architecture called the MicroVAX architecture.

MicroVAX Architecture

The MicroVAX architecture is tailored to facilitate low-end implementations of the VAX family of computers. The features of the MicroVAX architecture are:

- A four gigabyte virtual address space
- 32-bit word size
- Sixteen 32-bit general purpose registers
- 32 interrupt levels
- Vectored hardware and software interrupts

- 21 addressing modes
- Variable instruction size
- Full memory management
  - virtual to physical address translation
  - page protection mechanism
- Stack processing
- Full VAX instruction set (except PDP-11 compatibility mode)

The MicroVAX architecture specifies a subset of instructions that **must** be implemented in hardware. The remaining instructions may optionally be implemented in hardware, or emulated in software. The instructions that do not have to be implemented in hardware are:

- decimal string instructions
- character string instructions except MOVC3 and MOVC5
- EDITPC or CRC instructions
- D_floating, F_floating, G_floating, and H_floating instructions.

Thus, any machine implementing the MicroVAX architecture can execute the full VAX instruction set (minus PDP-11 compatibility mode).

For those instructions that do not have to be implemented in hardware, the MicroVAX architecture specifies two kinds of emulation support: instructions emulated strictly in software, and instructions emulated in software with a hardware assist.

The MicroVAX architecture specifies that all floating point instructions (D, F, G, and H) are emulated strictly in software (if they are not implemented in hardware).

The MicroVAX architecture specifies that the following instructions are emulated in software with a hardware assist (if they are not implemented in hardware):

- decimal string: MOVP, CMPP3, CMPP4, ADDP4, ADDP6, SUBP4, SUBP6, MULP, DIVP, ASHP, CVTPL, CVTLP, CVTPS, CVTSP, CVTTP, CVTPT
- character string: MOVTC, MOVTUC, SKPC, LOCC, SCANC, SPANC, MATCHC, CMPC3, CMPC5
- cyclic redundancy check: CRC
- edit: EDITPC

The MicroVAX architecture supports a subset of VAX processor registers. The following internal processor registers (IPRs) are either not required by the MicroVAX architecture, or are specified differently by MicroVAX architecture:

ICCS	interval clock control/status register
NICR	next interval count register
ICR	interval count register
TODR	time of year register
RXCS	console receive control status
RXDB	console receive data buffer
TXCS	console transmit control status
TXDB	console transmit data buffer
TBIS	translation buffer invalidate single
PMR	performance monitor enable
TBCHK	translation buffer check

MicroVAX I Implementation

The MicroVAX I system implements a superset of the MicroVAX architecture in that it implements more than the specified subset of instructions in hardware, and implements some IPRs not required by the MicroVAX architecture. The differences between the MicroVAX architecture, and the MicroVAX I implementation of it, are as follows.

- The MicroVAX architecture specifies emulation support for the following character string and floating point instructions, whereas MicroVAX I implements them in hardware:
  - CMPC3
  - LOCC
  - SCANC
  - SKPC
  - SPANC
  - F_floating point instructions
  - G_floating point instructions (KD32-AA CPU)
  - D_floating point instructions (KD32-AB CPU)
- The MicroVAX architecture does not specify the implementation of the following six IPRs. MicroVAX I implements them as defined by the VAX architecture:

RXCS	console receive control status
RXDB	console receive data buffer
TXCS	console transmit control status
TXDB	console transmit data buffer
TBIS	translation buffer invalidate single

TBCHK translation buffer check

- MicroVAX I implements these IPRs uniquely:

  ICCS interval clock control/status register

  CADR cache disable

  MCESR machine check error summary

  IORESET initialize bus

- The MicroVAX architecture specifies that physical addresses can be up to 30 bits long. A physical address on MicroVAX I is 23 bits long, allowing a physical address space of eight megabytes. (The MicroVAX I physical address space is covered in more detail in Chapter 2 of this manual.)

The differences between the VAX and MicroVAX architectures have been outlined here. The differences between the MicroVAX architecture, and the MicroVAX I implementation of that architecture have also been discussed. For more information about VAX architecture, see the *VAX Architecture Handbook*, EB-19580-20.

For more information about the MicroVAX I implementation of the MicroVAX architecture, see Chapter 2 of this manual, "Programming Interface." The MicroVAX instruction set is listed in Appendix E.

System Timing

The MicroVAX I system clocks are generated on the CPU memory controller module (MCT). A basic clock with a 64 MHz frequency is generated by a crystal oscillator. All the other clocks in the data path (DAP) and memory controller (MCT) modules are derived from this basic clock.

MicroVAX I is a pipelined, microprogrammed machine. The basic microcycle is 250 ns, and the pipeline is one deep. A new microinstruction on the DAP module is accessed every 250 ns, and requires two 250 ns microcycles to complete. The first 250 ns is DECODE, and the second 250 ns is EXECUTE. The EXECUTE microcycle of the first microinstruction is overlapped with the DECODE microcycle of the next microinstruction. Thus, one microinstruction is retired every 250 ns.

The main clock on the data path module (CPU CLOCK) has a symmetrical 250 nanosecond period. The start of a microcycle is defined as occurring on the leading edge of this clock and is referred to as T0. The trailing edge of the clock occurs 125 ns later.

This timing is illustrated in Figure 1-4.

The memory controller module implements a separate micromachine which cycles at 125 ns to accomplish memory-related activities.

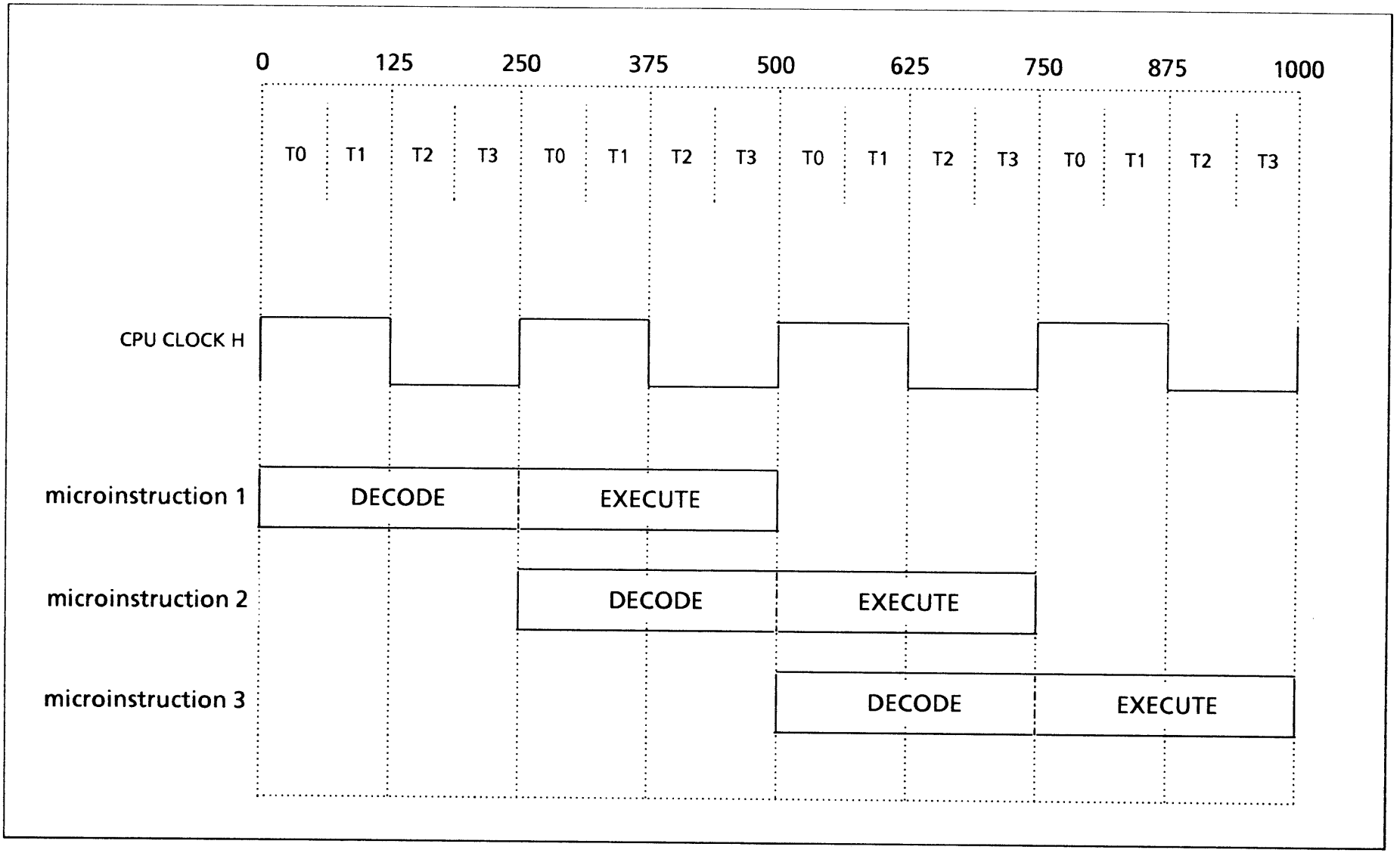


Figure 1-4. Microinstruction Timing

System Bus Summary

The system buses which interconnect the modules in the MicroVAX I system are the memory data bus (MDB), the memory control bus (MCB), and the extended LSI-11 bus (Q22 bus). (Those buses that are completely contained within a module are not discussed in this section.)

The memory data bus and the memory control bus connect the two CPU modules (DAP and MCT). The memory data bus is implemented using an over-the-top 50-pin cable. It is a 32-bit bidirectional data bus. The 8-bit memory control bus is implemented using the CD interconnect on the backplane. The remaining lines on the CD interconnect are used for clock distribution, status, and miscellaneous control logic. Slots 1 and 2 on the backplane are reserved for the two CPU modules as both must be placed in Q22/CD slots (see Figure 1-3).

The Q22 bus connects the CPU to the system's memory and peripheral I/O devices. Four basic kinds of transactions take place on the bus:

- Power up/down signal sequencing
- Transfer of bus mastership from the CPU to a direct memory access (DMA) device
- Transfer of data between a bus master and a slave
- Interrupts to the CPU

Most of the bus interface logic is located on the memory controller module. The data path module contains logic to handle power up and down signal sequencing and interrupts. Chapter 9 of this manual, "Q22 Bus Controller," describes the bus transactions in more detail.

For more information about extended LSI-11 bus signals and protocols in general, see the handbook *Microcomputers and Memories*, EB-20912-20.

This chapter is a brief overview of the MicroVAX I system components. For more detail about the MicroVAX I system, see the *MicroVAX I Owner's Manual*, EK-KD32A-OM, or the *MicroVAX I Field Service Print Set*, MP-01896-01.

The rest of this manual describes the MicroVAX I processor.

Chapter 2
Programming Interface

This chapter contains programming information for the KD32-AA and KD32-AB CPUs. It describes the physical address space, the internal processor registers available to the software, how the bootstrap works, what Microverify and the console microcode do, and what happens during exceptions and interrupts.

Physical Address Space

The MicroVAX architecture specifies that physical addresses may be up to 30 bits long. A physical address on MicroVAX I is 23 bits long, providing a physical address space of eight megabytes; four megabytes are in memory space, and four megabytes are in I/O space. Address bit <29> is used to select memory or I/O space, and bits <21:0> select an address within memory or I/O space. Address bit <28> is not part of the 23-bit physical address, but it is used as an internal state flag, called the no-cache flag, to indicate that the address is located in shared memory and should not be cached. Address bits <27:22> are ignored. Physical memory starts at address 00000000 (hex) and is contiguous to the end of installed memory (up to 4 megabytes, address 00400000 hex).

The I/O space is largely empty, containing only Q22 bus I/O space, which is the first 8K bytes (20000000 to 20001FFF). Bit <29> is set in physical address references to I/O space, and bits <21:13> are ignored.

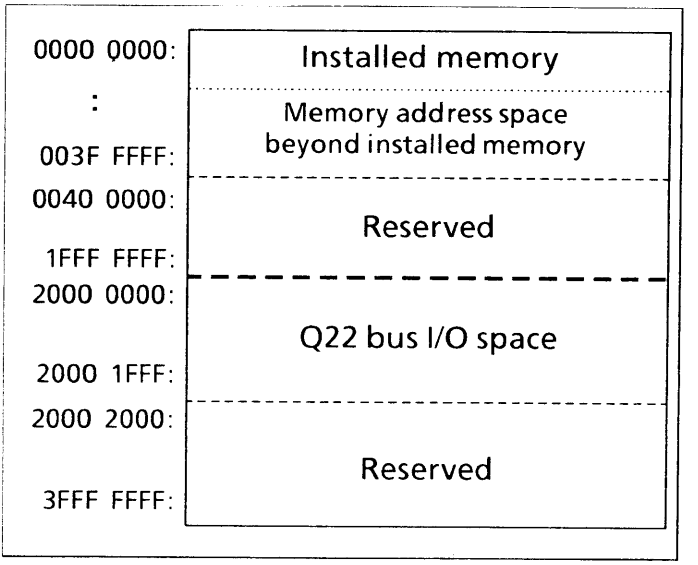


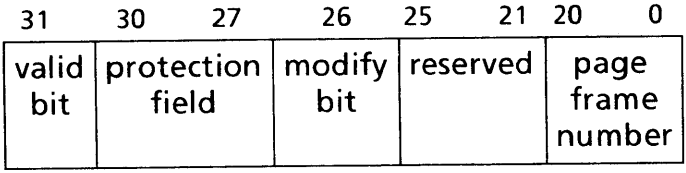
Figure 2-1. MicroVAX I Physical Memory

Address Translation

MicroVAX I implements full VAX memory management, so virtual addresses are translated to physical addresses just as they are for the VAX minicomputers.

MicroVAX I virtual addresses are 32-bits long, allowing a virtual address space of 4 gigabytes. Virtual address space is divided into process space (low-addressed half) and system space (high-addressed half). Process space is again divided into the P0 region and the P1 region.

When memory management is enabled, system and process space virtual addresses are translated into physical addresses by means of page table entries (PTEs). Virtual memory is partitioned into 512-byte pages; there is one PTE for each page of virtual memory. Each PTE has this format:



A physical address is formed for a given virtual address as follows. Bits <31:30> of a virtual address select one of three page tables, each of which contains PTEs for that region of memory:

<31:30> = 2 selects the system page table (SPT)
<31:30> = 1 selects the P1 page table (P1PT)
<31:30> = 0 selects the P0 page table (P0PT)

Bits <29:9> of the virtual address, called the virtual page number (VPN), are used as a longword index into the selected page table to select the corresponding PTE. Each page table has a length associated with it for determining virtual addresses that are out of bounds.

Once the PTE for the page that contains the given virtual address is found, bits <20:0> of the PTE (the page frame number, or PFN) form bits <29:9> of the physical address. Bits <8:0> of the given virtual address form bits <8:0> of the physical address. Thus, bit <29> of the physical address comes from bit <20> of the PTE, and bit <28> of the physical address comes from bit <19> of the PTE.

Figure 2-2 shows the translation from a system virtual address to a physical address, Figure 2-3 shows the translation from a P0 virtual address to a physical address, and Figure 2-4 shows the translation from a P1 virtual address to a physical address.

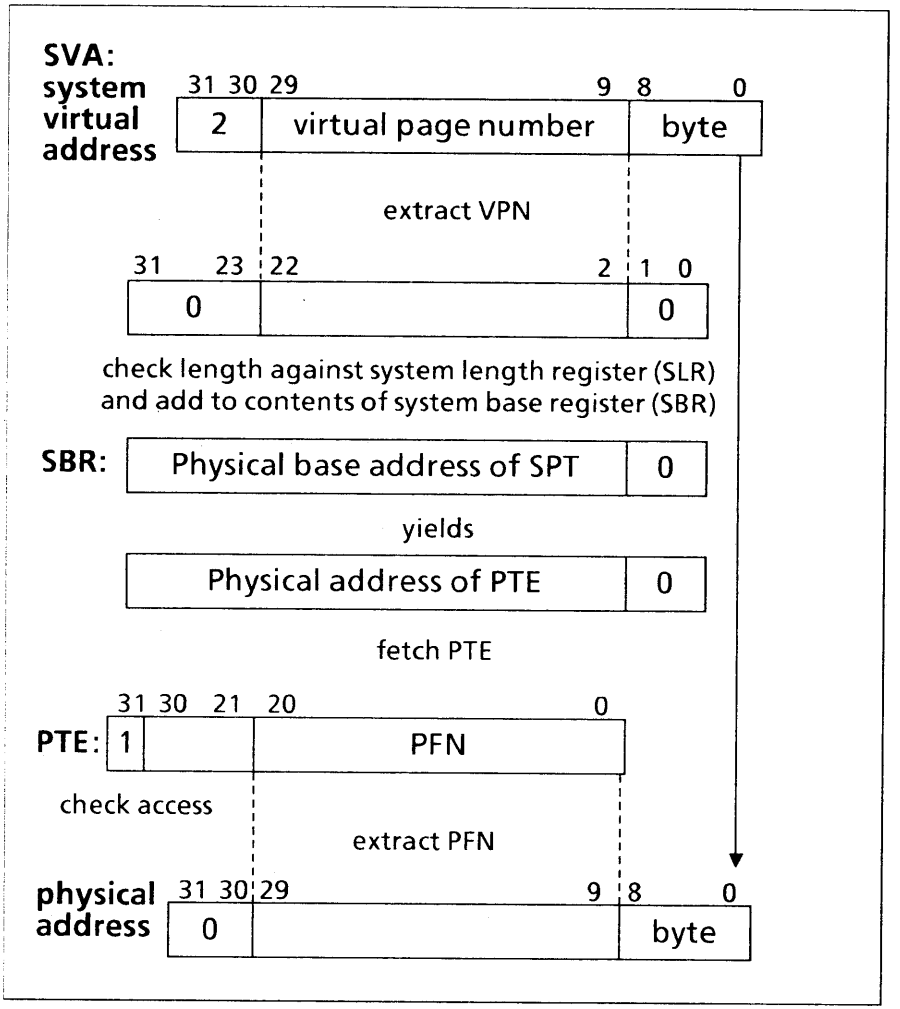


Figure 2-2. System Virtual to Physical Translation

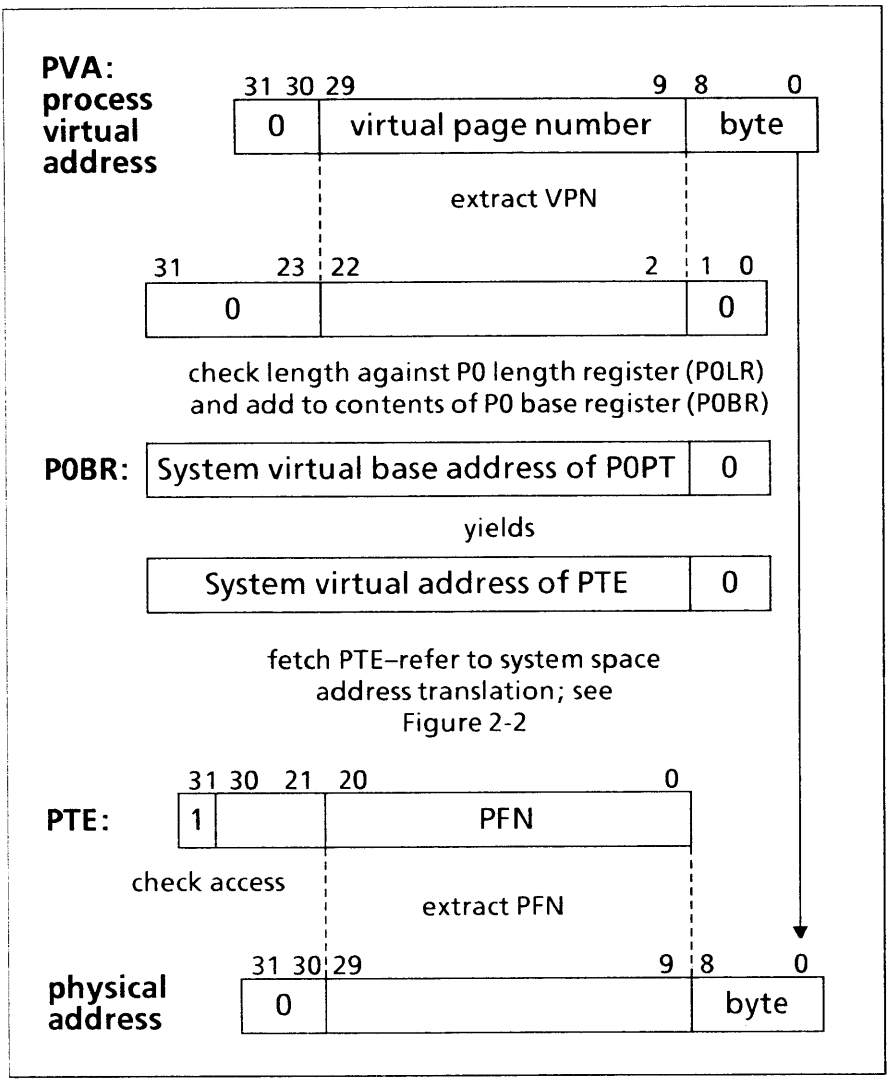


Figure 2-3. P0 Virtual to Physical Translation

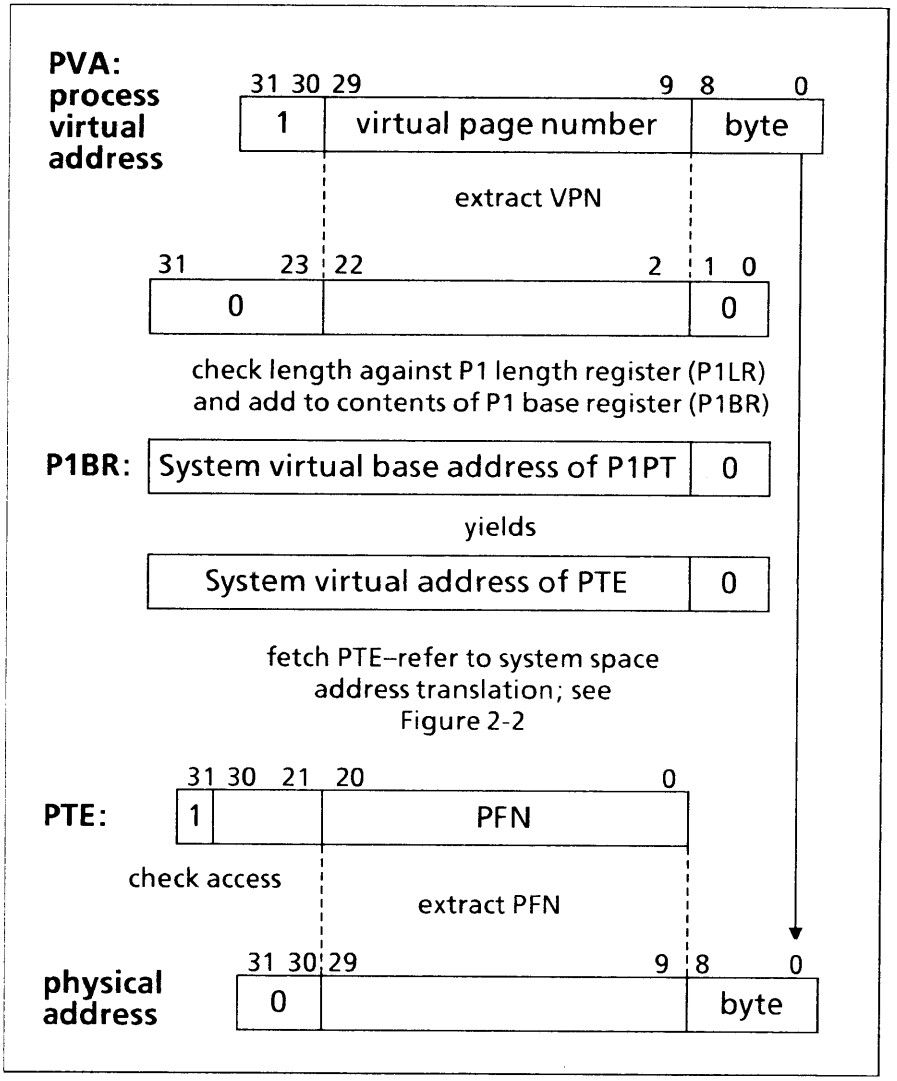


Figure 2-4. P1 Virtual to Physical Translation

I/O Space Programming Constraints

If either bit <29> or bit <28> of a MicroVAX I physical address is set, certain programming constraints apply. Again, note that bit <28> is not used as part of the physical address, but as the no-cache flag. If this flag is set, the physical address is subject to the same constraints that apply to an I/O space address. The following considerations apply when bit <29> or bit <28> is set in a physical address:

- The data at the specified physical address are not cached.
- Only byte and word-aligned accesses are allowed. Thus, the physical address must be on a byte boundary for instructions using a length attribute of byte, and on an even-byte boundary for instructions using a length attribute of word. Instructions using a length attribute other than byte or word are not allowed.
- In certain devices, byte accesses to word-length registers produce unpredictable results.
- String, quad, floating point, and field references result in undefined behavior.

Internal Processor Registers

There are 64 internal processor registers (IPRs) in the MicroVAX I processor register space. These registers can only be accessed by a Move to Processor Register (MTPR) instruction, or a Move from Processor Register (MFPR) instruction. MTPR and MFPR instructions must be executed in kernel processor mode.

The MicroVAX I internal processor registers can be grouped into four categories, as follows.

1. Registers that are implemented as defined by the VAX architecture.
2. Registers that are implemented uniquely by the MicroVAX I system.
3. Registers to which access is allowed, but which have no effect on the MicroVAX I system; these registers are read as zero, and no operation is performed for writes.
4. Registers to which access is not allowed; an access results in a reserved operand fault.

Table 2-1 lists each internal processor register and its category.

The following paragraphs describe all of the registers that are uniquely implemented in MicroVAX I (category 2). The System Identification (SID) register and the four console terminal registers are also described. Although these are in category 1 (implemented as defined by the VAX architecture), some of the bits are uniquely defined for MicroVAX I where allowed by the architecture.

Table 2-1. Internal Processor Registers

Reg. No.	Register Name	Mnemonic	Type	Category
0	Kernel Stack Pointer	KSP	R/W	1
1	Executive Stack Pointer	ESP	R/W	1
2	Supervisor Stack Pointer	SSP	R/W	1
3	User Stack Pointer	USP	R/W	1
4	Interrupt Stack Pointer	ISP	R/W	1
5	Reserved			4
6	Reserved			4
7	Reserved			4
8	P0 Base Register	P0BR	R/W	1
9	P0 Length Register	P0LR	R/W	1
A	P1 Base Register	P1BR	R/W	1
B	P1 Length Register	P1LR	R/W	1
C	System Base Register	SBR	R/W	1
D	System Length Register	SLR	R/W	1
E	Reserved			4
F	Reserved			4
10	Process Control Block Base	PCBB	R/W	1
11	System Control Block Base	SCBB	R/W	1
12	Interrupt Priority Level	IPL	R/W	1
13	AST Level	ASTLVL	R/W	1
14	Software Interrupt Request	SIRR	W	1

Category Key:

1 = implemented as defined by VAX architecture
2 = implemented by MicroVAX I uniquely
3 = read as zero, no operation on writes
4 = access not allowed; results in reserved operand fault

Table 2-1. Continued

Reg. No.	Register Name	Mnemonic	Type	Category
15	Software Interrupt Summary	SISR	R/W	1
16	Reserved			4
17	CMI Error Register	CMIERR		4
18	Interval Clock Control	ICCS	R/W	2
19	Next Interval Count	NICR	W	3
1A	Interval Count	ICR	R	3
1B	Time of Year	TODR	R/W	3
1C	Console Storage Receiver Status	CSRS	R/W	4
1D	Console Storage Receiver Data	CSRD	R	4
1E	Console Storage Transmit Status	CSTS	R/W	4
1F	Console Storage Transmit Data	CSTD	W	4
20	Console Receive C/S	RXCS	R/W	1
21	Console Receive D/B	RXDB	R	1
22	Console Transmit C/S	TXCS	R/W	1
23	Console Transmit D/B	TXDB	W	1
24	Translation Buffer Disable	TBDR	R/W	3

Category Key:

1 = implemented as defined by VAX architecture
2 = implemented by MicroVAX I uniquely
3 = read as zero, no operation on writes
4 = access not allowed; results in reserved operand fault

Table 2-1. Continued

Reg. No.	Register Name	Mnemonic	Type	Category
25	Cache Disable	CADR	R/W	2
26	Machine Check Error Summary	MCESR	R/W	2
27	Cache Error	CAER	R/W	3
28	Accelerator Control/Status	ACCS		4
29	Accelerator Maintenance	ACCR		4
2A	Reserved			4
2B	Reserved			4
2C	Writable Control Store Address	WCSA		4
2D	Writable Control Store Data	WCSB		4
2E	Reserved			4
2F	Reserved			4
30	SBI Fault/Status	SBIFS	R/W	3
31	SBI Silo	SBIS	R	3
32	SBI Silo Comparator	SBISC	R/W	3
33	SBI Maintenance	SBIMT	R/W	3
34	SBI Error Register	SBIER	R/W	3
35	SBI Timeout Address	SBITA	R	3
36	SBI Quadword Clear	SBIQC	W	3

Category Key:

1 = implemented as defined by VAX architecture
2 = implemented by MicroVAX I uniquely
3 = read as zero, no operation on writes
4 = access not allowed; results in reserved operand fault

Table 2-1. Continued

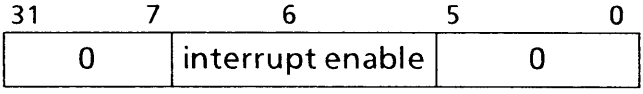
Reg. No.	Register Name	Mnemonic	Type	Category
37	Initialize Bus	IORESET	W	2
38	Memory Management Enable	MAPEN	R/W	1
39	Translation Buffer Invalidate All	TBIA	W	1
3A	Translation Buffer Invalidate Single	TBIS	W	1
3B	Translation Buffer Data	TBDATA	R/W	3
3C	Microprogram Breakpoint	MBRK	R/W	3
3D	Performance Monitor Enable	PMR	R/W	3
3E	System Identification	SID	R	1
3F	Translation Buffer Check	TBCHK	W	1

Category Key:

1 = implemented as defined by VAX architecture
2 = implemented by MicroVAX I uniquely
3 = read as zero, no operation on writes
4 = access not allowed; results in reserved operand fault

Interval Clock Control/Status Register

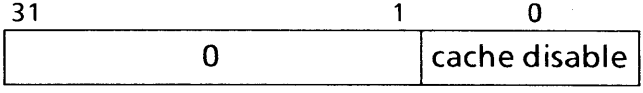
The MicroVAX I processor includes a 10 millisecond interval timer. The Interval Clock Control/Status Register (ICCS) contains the interrupt enable bit for the timer. The interrupt enable bit is bit <6>. It is cleared when the system is booted.



When the interrupt enable bit is set, an interrupt request is generated at interrupt priority level 16 (hex) when the 10 ms timer overflows. When the interrupt enable bit is clear, the timer overflow is ignored; no interrupt is generated.

Cache Disable Register

The memory controller module of the processor contains an 8 KB cache. The Cache Disable Register (CADR) contains the disable bit for the cache. The disable bit is bit <0>. It is cleared during machine initialization so that the cache is enabled.

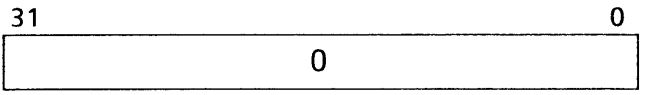


When the cache disable bit is set, the cache is disabled. When the cache disable bit is clear, the cache is enabled.

Machine Check Error Summary Register

The Machine Check Error Summary Register (MCESR) is always read as zero. A write to this register clears the machine-check-in-progress flag. This flag is an internal state flag that is set when a machine check occurs.

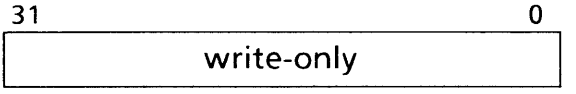
This flag should be cleared by software when a machine check has been handled. If a second machine check occurs while this flag is set, the CPU halts with a halt code of 5, indicating that a double machine check has occurred. (See Table 2-6 in this chapter for a description of the console halt codes.)



Initialize Bus Register

The Initialize Bus Register (IORESET) is write-only; an attempt to read this register causes a reserved operand fault.

Any write to this register causes a system reset which initializes the processor and causes a Q22 bus initialization sequence.

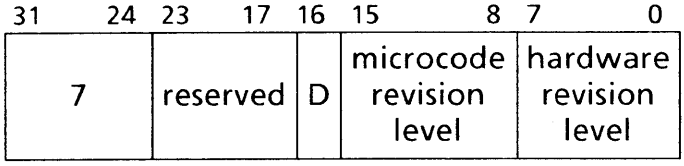


System Identification Register

The System Identification Register (SID) is a read-only constant register that specifies the processor type. The information in the SID register is used for error logs and to check engineering change order (ECO) levels.

The SID register is four bytes wide. The three high-order bytes are built by the processor microcode and the low-order byte is set by an eight-switch DIP on the data path module. The high-order byte contains a number that uniquely identifies the processor; the MicroVAX I processor is identified by the number 7.

If the processor is the KD32-AA CPU with F_ and G_floating point, bit 16 of the SID register is a 0. If the processor is the KD32-AB CPU with F_ and D_floating point, bit 16 of the SID register is a 1.



Console Terminal Registers

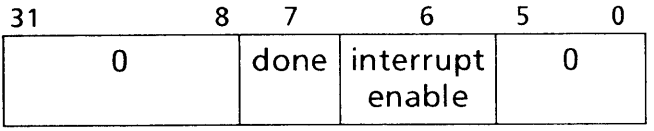
The console terminal is accessed through four internal registers. Two registers are associated with receiving from the terminal, and two with writing to the terminal. In each direction there is a control/status register and a data buffer register. The bit assignments for each register are described in the following paragraphs.

Console Receive Control/Status Register (RXCS)

Bits <31:8> and <5:0> of the RXCS register are read as zero and ignored on writes.

Bit 7 is the done bit. It is a read-only bit that is set by the console whenever a character is received. The done bit is initialized to 0 at bootstrap time and is cleared when the RXDB (console receive data buffer) register is read.

Bit 6 is the interrupt enable bit. When this bit is set, an interrupt is generated at interrupt priority level 14 (hex) when the done bit is set. An interrupt is also generated if the done bit is already set when the software sets the interrupt enable bit. The interrupt enable bit is set to 0 at bootstrap time, and can be read or written by software.



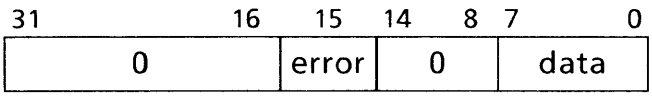
Console Receive Data Buffer Register (RXDB)

This register is a read-only register.

Bits <31:16> and <14:8> of the RXDB register are read as zero.

Bit 15 is the error bit. If the received character contains an error, this bit is set.

Bits <7:0> are the data field. This field contains the actual character received by the console.

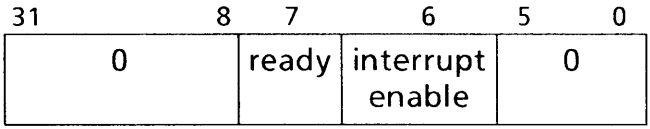


Console Transmit Control/Status Register (TXCS)

Bits <31:8> and <5:0> of the TXCS register are read as zero and ignored on writes.

Bit 7 is the ready bit. It is a read-only bit that is set at bootstrap time, and whenever the console transmitter is not busy. The ready bit is cleared when the TXDB (console transmit data buffer) register is written.

Bit 6 is the interrupt enable bit. If this bit is set by software, an interrupt is generated at interrupt priority level 14 (hex) when the ready bit becomes set. If the ready bit is already set and software sets the interrupt enable bit, an interrupt is also generated.



Console Transmit Data Buffer Register (TXDB)

This register is a write-only register.

Bits <31:12> of the TXDB register are ignored.

Bits <11:8> form the ID field. The encoding in this field determines whether characters are written to the console terminal, or whether internal functions are executed. If <11:8> = 0, the character contained in bits <7:0> of the TXDB register is written to the console terminal. If <11:8> = F, the internal function

specified by bits <7:0> of the TXDB register is performed. If <11:8>=1–E, a reserved operand fault occurs.

Bits <7:0> are the data field. This field contains the actual character transmitted to the console if the ID field is 0. If the ID field contains the value F, the data field specifies the internal function to be performed. The internal functions are encoded as shown in Table 2-2.

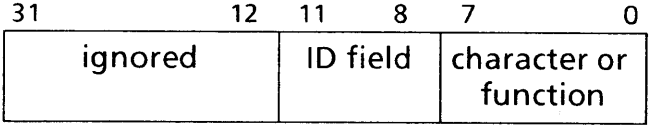


Table 2-2. TXDB Register Encoding

TXDB <11:0>	Internal Function
F00	no operation
F01	no operation
F02	boot the system
F03	clear the restart-in-progress flag
F04	clear the boot-in-progress flag
F05	causes the system to enter console mode (halt code 2; see Table 2-6)
F06	illegal; causes a machine check
F07	illegal; causes a machine check
F08	write 000 to the diagnostic LEDs (all LEDs on)
F09	write 001 to the LEDs (on, on, off)
F0A	write 010 to the LEDs (on, off, on)
F0B	write 011 to the LEDs (on, off, off)
F0C	write 100 to the LEDs (off, on, on)
F0D	write 101 to the LEDs (off, on, off)
F0E	write 110 to the LEDs (off, off, on)
F0F	write 111 to the LEDs (all LEDs off)

MicroVAX I System Bootstrap

The data path module of the processor contains an EPROM (erasable programmable read-only memory). The EPROM contains the primary bootstrap. The purpose of the primary bootstrap is to load a secondary bootstrap. In some cases, the secondary bootstrap is the entire system image. In other cases, the secondary bootstrap loads the system image. The primary bootstrap, then, is a set of instructions that cause

additional instructions to be loaded until the complete computer program is stored in memory.

The primary bootstrap checks each device that could contain the secondary bootstrap or a system image. It checks these devices in a predetermined order. When the primary bootstrap finds the first device (usually a disk drive) that contains a secondary bootstrap or an appropriate system image, the primary bootstrap causes the secondary bootstrap or system image supplied by the device to be copied into memory and executed.

In short, the primary bootstrap provided with the MicroVAX I system does the following:

- Initializes the machine to a known state
- Locates, determines the size of, and tests the memory
- Locates and reads the secondary bootstrap or system image into memory
- Begins execution of the secondary bootstrap or system image

Note: The last 16-bit word of the primary bootstrap must be the twos complement of the sum of all previous 16-bit words. See the section titled "Microverify" in this chapter for more information about this checksum.

Bootstrapping Methods

There are five ways to bootstrap a MicroVAX I system.

- The data path module contains an eight-switch DIP called the option switches. Option switches 3 and 4 determine the series of activities that the processor attempts during power on. If either "boot, halt" or "warm start, boot, halt" is the action

set in the switches, the system boots when system power is turned on.

- If "boot, halt" is the action set in the switches, the system boots when a Halt instruction is executed in kernel mode. If "warm start, boot, halt" is the action set in the switches and warm start fails, the system boots when a Halt instruction is executed in kernel mode.
- The system boots when the boot command is entered from console mode. (Pressing the Halt button on the front panel places the system in console mode. The Halt button must be pressed again to unlatch it before the boot command is entered.) The boot command is described below.
- When the Restart button on the front panel is pressed, the system examines option switches 3 and 4 for the recovery action. If "boot, halt" is the action set in the switches, the system boots when the Restart button is pressed. If "warm start, boot, halt" is the action set in the switches and warm start fails when the Restart button is pressed, the system boots. (The Halt button must be latched out for the Restart button to work.)

  Note: With all software currently sold by DIGITAL for the MicroVAX I, pressing the Restart button always causes the system to reboot.
- The system boots when the value F02 (hexadecimal) is written to the console transmit data buffer register (TXDB).

When the MicroVAX I system is booted by one of these methods, the console microcode searches for 64K bytes of good memory. If 64K bytes of good memory cannot be found, the boot fails and a message is displayed on the console terminal.

If sufficient memory is found, the primary bootstrap stored in the EPROM is read into successive locations in memory, starting at the base address of the found memory plus 0200 (hexadecimal) and continuing for 8K or 16K bytes (depending on whether an 8K or 16K EPROM is installed). When the transfer is complete, a jump to the beginning of the primary bootstrap is executed (that is, a jump to the base address plus 0200). The primary bootstrap is then executed using the information in the following general registers:

R0	zero, or the four character device name specified in the boot command
R1	settings of the option switches
R5	zero, or the flag value specified in the boot command
R10	program counter at the time boot was requested
R11	processor status at the time boot was requested
AP	halt code (see Table 2-6)
SP	base address of the found memory plus 0200 (hexadecimal)

When the primary bootstrap is executed, it locates the appropriate secondary bootstrap or system image and reads it into memory.

Boot Command

The format of the console boot comand is:

B[[/n]<sp><xxxx>]<cr>

The two acceptable forms of the command, therefore, are:

B<cr>

B[/n]<sp><xxxx><cr>

where /n is an optional hexadecimal flag value that is passed to the primary and secondary bootstrap in general register R5, and xxxx is an optional device name that is passed to the primary bootstrap in general register R0. If no flag value or device name is specified, the value zero is passed to R5 and R0.

The device name must be exactly four characters. Table 2-3 lists the allowable device names.

Table 2-3. Device Names

Name	Device
DUA0	disk or diskette unit 0
DUA1	disk or diskette unit 1
DUA2	disk or diskette unit 2
DUA3	disk or diskette unit 3
PRA0	MRV11 PROM
XQA0	DEQNA

Table 2-4 lists the hexadecimal flags for the boot command parameter /n that are meaningful to the primary bootstrap. Leading zeros need not be supplied.

Note: Other flags not listed in Table 2-4 may be meaningful to various operating systems available for the MicroVAX I. For example, the boot command **B/1** invokes a conversational boot when the operating system software is MicroVMS.

Table 2-4. Boot Command Flags

Bit No.	Hex Flag Value	Flag Name and Meaning
3	00000008	BOOTBLOCK – Secondary boot from bootblock. When this bit is set, the primary bootstrap skips the normal operation, which is to search the volume as a Files-11 volume. Instead, the primary bootstrap reads logical block number 0 of the volume and tests it for conformance with the bootblock format.
4	00000010	DIAGNOSTIC – Diagnostic boot. When this bit is set, the secondary bootstrap is the image called [SYS0.SYSMAINT]DIAGBOOT.EXE.
6	00000040	HEADER – Image header. If this bit is not set, the primary bootstrap transfers control to the first byte of the secondary bootstrap file. If this bit is set, the primary bootstrap transfers control to the address of the secondary bootstrap obtained from that file's image header.
7	00000080	NOTEST – Memory test inhibit. This flag disables parity checking during boot.
8	00000100	SOLICT – Solicit file name. When this bit is set, the primary bootstrap prompts for the name of a secondary bootstrap file.

Table 2-4. Continued

Bit No.	Hex Flag Value	Flag Name and Meaning
9	00000200	HALT – Halt before transfer. When this bit is set, it causes a Halt instruction to be executed before transferring control to the secondary bootstrap.
31:28	X0000000	TOPSYS. X can be any value from 0 through F (hex). The TOPSYS flag changes the top level directory name for system disks with multiple operating systems. For instance, if X = 1, the top level directory name is [SYS1. ...].

Bootstrap Operation

The primary bootstrap stored in the EPROM contains code that:

- initializes the system control block (SCB),
- initializes the restart parameter block (RPB),
- initializes a PFN (page frame number) bit map and the relevant RPB fields,
- selects a boot device, and
- performs a Files-11 ODS2, bootblock, PROM, or down-line load boot.

The primary bootstrap finds the device which contains the secondary bootstrap in one of two ways.

1. If a device name is specified in the boot console command, that device is searched for the secondary bootstrap.

2. If no device name is specified, and option switch number 1 is off (which is the default position), the following devices are searched in the order listed here until the secondary bootstrap is found, or the list is exhausted.

   All diskette drives in ascending unit order

   All fixed disk drives in ascending unit order

   MRV11 PROM

   DEQNA for down-line loading

   If option switch number 1 is on, the primary bootstrap bypasses the diskettes and disks and searches only the MRV11 PROM and the DEQNA for the secondary bootstrap.

Booting from Disk

For the system to boot successfully from diskette or disk, the RQDX1 controller must be configured at Q22 bus address 772150 (octal). The module is shipped this way from the factory. (For more information about configuring the RQDX1 module, see the "System Configuration" section of the *MicroVAX I Owner's Manual*, EK-KD32A-OM.)

The primary bootstrap begins by searching the diskette drives for the secondary bootstrap. (This is assuming that if the boot console command is used to bootstrap the system, none of the optional parameters are specified.) The primary bootstrap checks the first diskette drive to determine if a diskette is installed. If no diskette is present, the next diskette drive is checked.

If a diskette is present, the primary bootstrap determines if it is a Files-11 volume. If it is, the primary bootstrap searches for the file

[SYS0.SYSEXE]SYSBOOT.EXE

which contains the secondary bootstrap. If this file is found, the system boots. If this file is not found, the primary bootstrap checks the next diskette drive.

If a diskette is present on the diskette drive being checked but it is not a Files-11 volume, the primary bootstrap checks for the bootblock format in logical block number (LBN) 0 of the diskette. Figure 2-5 shows the bootblock format.

If LBN 0 of the diskette contains the bootblock format, the system boots. If the bootblock format is not present, the primary bootstrap checks the next diskette drive.

If the primary bootstrap cannot find the secondary bootstrap on any diskette, it checks the fixed disk drives in the same manner. First, it checks whether the fixed disk drives are on-line. Then, for every drive on-line, it checks if the disk is a Files-11 volume. If it is, and the file [SYS0.SYSEXE]SYSBOOT.EXE is present, the system boots.

If the disk is not a Files-11 volume, the primary bootstrap checks for the bootblock format in LBN 0. If the bootblock format is present, the system boots.

The sequence just described occurs unless the boot console command is used with any of these hexadecimal flags specified in the /n parameter:

bit <3> = BOOTBLOCK
bit <4> = DIAGNOSTIC
bits <31:28> = TOPSYS

(Table 2-4 summarizes the functions of the /n parameter bits.)

If the BOOTBLOCK bit (bit <3>) is set in the /n parameter of the boot console command, the primary bootstrap does not test the volume for the Files-11 format but just reads LBN 0. If LBN 0 contains the bootblock format shown in Figure 2-5, the secondary bootstrap is located at the LBN specified in the bootblock format, and loaded into memory at the specified offset from the default load address. (The default load address is the base address of the restart parameter block plus 5000 hex.) Execution of the secondary bootstrap begins with the instruction that is located at the offset specified in the bootblock format.

If the DIAGNOSTIC bit (bit <4>) is set in the /n parameter of the boot console command and the volume is a Files-11 volume, the primary bootstrap searches for the secondary bootstrap in the file named

[SYS0.SYSMAINT]DIAGBOOT.EXE.

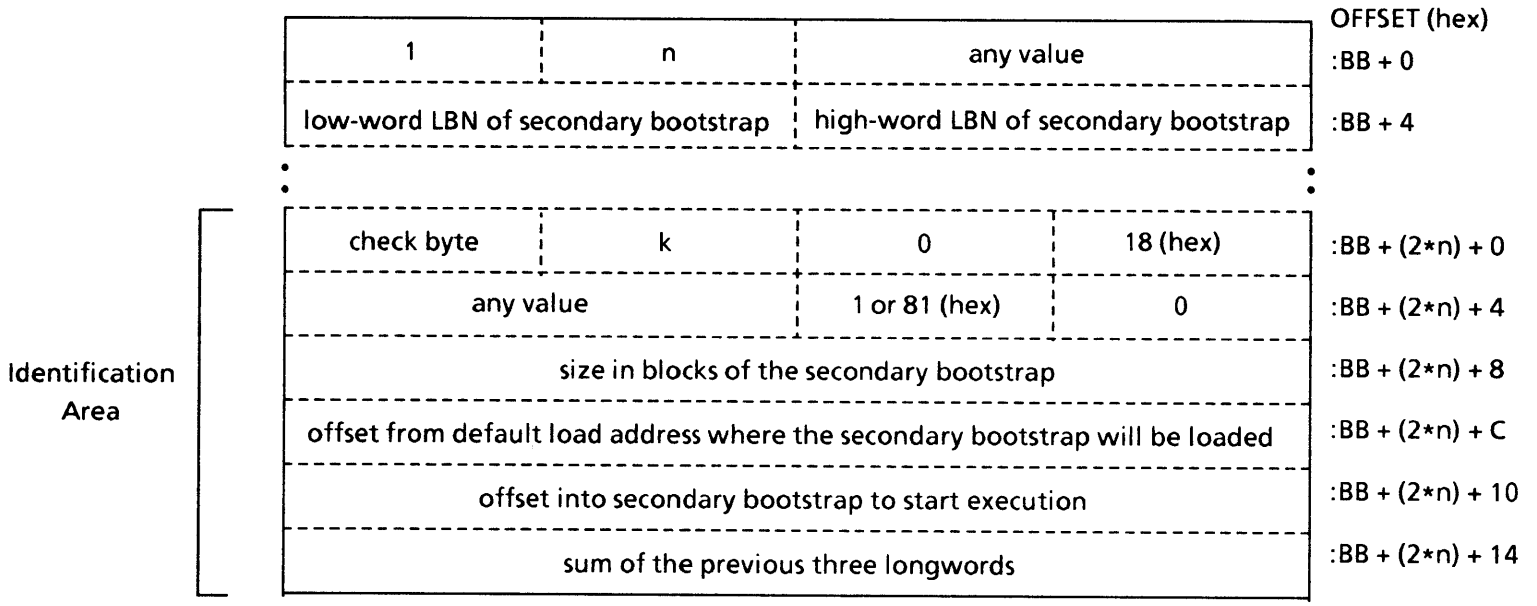
If the TOPSYS bits (bits <31:28>) contain a nonzero value in the /n parameter of the boot console command and the volume is a Files-11 volume, the primary bootstrap searches for the secondary bootstrap in the file named

[SYS*n*.SYSEXE]SYSBOOT.EXE

(or [SYS*n*.SYSMAINT]DIAGBOOT.EXE if the DIAGNOSTIC bit is set), where *n* is the hexadecimal value of bits <31:28>.

For instance, if the TOPSYS value is 4, then the primary bootstrap searches for the secondary bootstrap in the file named [SYS4.SYSEXE]SYSBOOT.EXE (or [SYS4.SYSMAINT]DIAGBOOT.EXE).

Figure 2-6 is a flowchart diagramming the sequence used by the primary bootstrap to find the secondary bootstrap when the secondary bootstrap is located on diskette or disk.



BB + 0: **any value.** These two bytes can be any value.
BB + 2: **n.** The value n is the word offset from the start of the block to the identification area described below.
BB + 3: **1.** This byte must be one.
BB + 4: This entry is a longword containing the logical block number (LBN) where the secondary bootstrap is located. Note that the low- and high-words of the LBN are interchanged.
BB + (2*n) + 0: **18 (hex).** This byte defines the expected instruction set; 18 means VAX.
BB + (2*n) + 1: **0.** This byte defines the expected controller type; 0 means unknown.
BB + (2*n) + 2: **k.** This byte identifies the file structure on the volume. K can be any value.
BB + (2*n) + 3: **check byte.** This byte is the ones complement of the sum of the previous three bytes.
BB + (2*n) + 4: **0.** This byte must be zero.
BB + (2*n) + 5: **1 or 81 (hex).** This byte defines the version number of the format standard and the type of disk. The version is 1. The high bit is 0 for single-sided and 1 for double-sided.
BB + (2*n) + 6: **any value.** These two bytes can be any value.
BB + (2*n) + 8: This entry is a longword containing the size in blocks of the secondary bootstrap.
BB + (2*n) + C: This entry is a longword containing the offset from the default load address where the secondary bootstrap is to be loaded.
BB + (2*n) + 10: This entry is a longword containing the byte offset into the secondary bootstrap where execution is to begin.
BB + (2*n) + 14: This entry is a longword containing the sum of the previous three longwords.

Figure 2-5. Bootblock Format

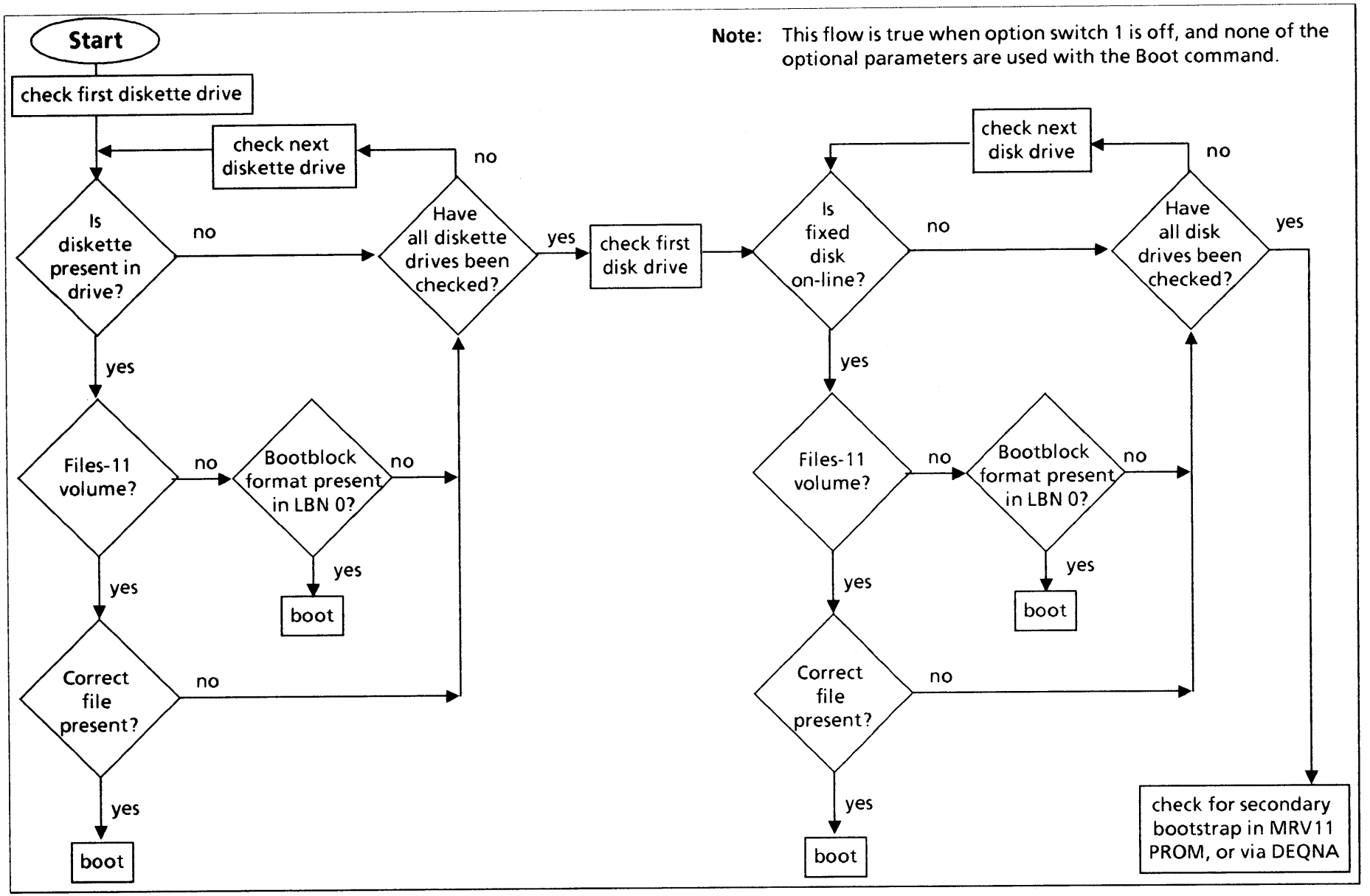
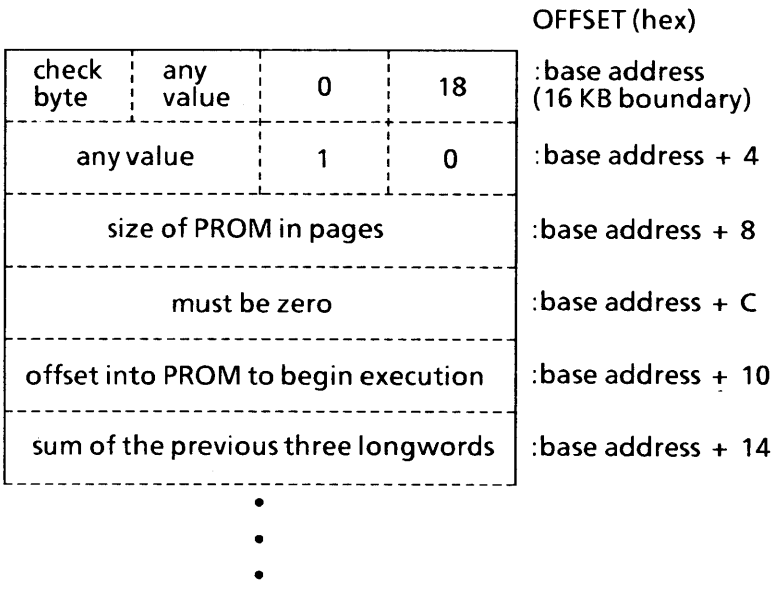


Figure 2-6. Bootstrap Flowchart

Booting from an MRV11-D PROM Module

An MRV11-D PROM module (M8578) can contain the secondary bootstrap or an operating system. To bootstrap from an MRV11-D module, the base address of the module must be on a 16 KB boundary, and the first six longwords of the PROM must contain the following hex values:



The check byte is the ones complement of the sum of the previous three bytes.

If the specified boot device is PRA0, or if none of the diskettes or disks contain the secondary bootstrap, the primary bootstrap searches each 16 KB boundary in all 4 MB of memory for this footprint. If this footprint is found, the primary bootstrap passes control to the

PROM code at the specified offset from the base address where execution is to begin.

The PROM is marked as "not valid" in the PFN bit map built by the primary bootstrap.

The documentation shipped with the MRV11-D module explains how to configure the jumpers and switches on the module. However, the following jumpers and switches must be set as noted for the MRV11-D to work correctly as a boot device within the MicroVAX I system:

- The system size jumper, W3, must be set to select a 22-bit system.
- The bootstrap jumper, W4, must be set to disable bootstrap.
- The page/direct mode DIP switch (switch 1 in the 5-switch DIP) must be set to the on position to select direct mode.

Note: 16K by 8 bit and 32K by 8 bit PROMs are not supported by the array address decoder PROM that is shipped with the MRV11-D module. However, the MRV11-D documentation describes how a decoder PROM can be made to support these larger PROMs.

Booting from DEQNA

If the specified boot device is the DEQNA (Ethernet to QBUS adapter), the secondary bootstrap is supplied by a host on the Ethernet. The primary bootstrap contains a bootstrap loader to downline load the secondary bootstrap via the Ethernet, using the DEQNA. The selection of the secondary bootstrap is the responsibility of the host.

The bootstrap loader implements the DECnet Low-level Maintenance Protocol (MOP) Version 3.0.

Complete details of the protocol are available from DIGITAL in the *DNA Maintenance Operations Functional Specification*, AA-X436A-TK.

The DEQNA module must be configured at Q22 bus address 774440 (octal). The module is shipped this way from the factory. (For more information about configuring DEQNA modules, see the "System Configuration" section of the *MicroVAX I Owner's Manual*, EK-KD32A-OM.)

The downline load process consists of the following steps.

1. The bootstrap loader in the primary bootstrap performs loopback testing of the DEQNA and transceiver to ensure that the Ethernet hardware is functioning properly. After the loopback tests complete, the three LED indicator lights on the DEQNA module are turned off. If one or more of the LEDs remain on, the loopback test has not completed successfully, and the downline load is not attempted.

   All LEDs on: DEQNA could not be initialized
   Two LEDs on: internal loopback failed; DEQNA not functioning completely
   One LED on: external loopback failed; transceiver or cable is bad

2. The bootstrap loader transmits a Program Request MOP message on the Ethernet. The message destination address is the Load Assistant Multicast address (AB-00-00-01-00-00). The message source address is the DEQNA's station address (supplied by a PROM on the DEQNA module). The program type requested is Operating System. (In MOP terminology, the bootstrap loader in the MicroVAX I primary bootstrap is the primary, secondary, and tertiary loader, and therefore it can perform the complete downline load process.)

3. The bootstrap loader waits for approximately 30 seconds to receive an Assistance Volunteer response message from a load server at the host. If it does not receive a message, the bootstrap loader retransmits the Program Request every 30 seconds. If a response is not received in two minutes, the load fails.

4. Once the bootstrap loader receives the Assistance Volunteer response message, it retransmits the Program Request message directly to the load server that sent the response message. It then waits for a Memory Load message.

   When the bootstrap loader receives a valid Memory Load message, it copies the load data into the memory location specified by the message (relative to the first good memory found by the primary bootstrap memory test). The bootstrap loader then transmits a Request Memory Load message, which acknowledges the previous Memory Load message and requests the next Memory Load message.

   The bootstrap loader waits for 30 seconds for the next Memory Load message. If the message is not received, the bootstrap loader restarts the downline load process by transmitting a Program Request message. The load server at the host normally waits for the Request Memory Load acknowledgement message for approximately 4 seconds. If the load server does not receive the message, it retransmits the last Memory Load message. Since Memory Load messages have unique sequence numbers, the bootstrap loader can ignore out-of-order or duplicate messages and can acknowledge the last message it correctly received.

5. The bootstrap loader continues to receive Memory Load messages, copy the received data into memory, and transmit Request Memory Load messages, until

it receives the final message, which is a Parameters with Transfer Address message. The parameters are stored in the restart parameter block (RPB) and control is transferred to the loaded program at the received transfer address.

The downline load parameters stored in the RPB are as follows:

Offset	Value
28 (hex)	Node address (48 bit Ethernet address)
68 (hex)	Node name string (word count followed by string)

These parameters can then be used by the loaded operating system for system communication via the DEQNA.

Interface Between Primary and Secondary Bootstrap

Once the primary bootstrap locates the secondary bootstrap and loads it into memory, the machine is in the following state.

The machine is running in kernel mode on the interrupt stack at IPL 31.

The register contents are:

- R11: base address of the RPB (restart parameter block)
- AP: base address of the argument list prepared by the primary bootstrap for the secondary bootstrap
- SP: current stack pointer and lowest address of secondary bootstrap.

  Note: The lowest address of the secondary bootstrap is either the default load address, which is the base address of the RPB + 5000 hex, or if a bootblock boot is performed, it is the default load

address plus the offset specified in LBN 0. If the secondary bootstrap is contained in an MRV11-D PROM, the lowest address of the secondary bootstrap is the base address of the PROM plus the offset into the PROM where execution is to begin.

- SCBB processor register: SCB address

Figure 2-7 shows the argument list prepared by the primary bootstrap for the secondary bootstrap.

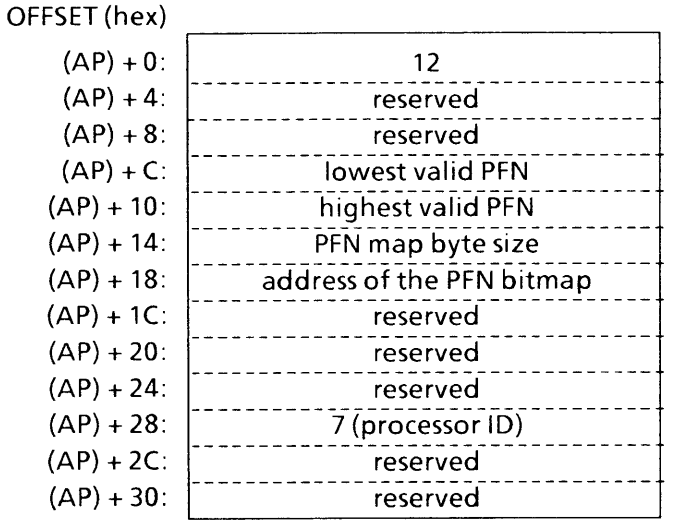
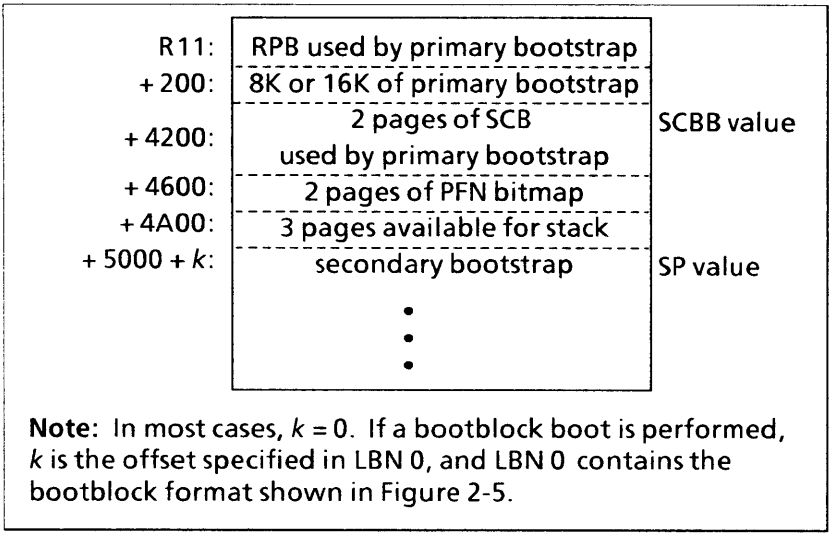


Figure 2-7. Secondary Bootstrap Argument List

Figure 2-8 shows what memory looks like when the secondary bootstrap gains control.



Note: In most cases, $k = 0$. If a bootblock boot is performed, k is the offset specified in LBN 0, and LBN 0 contains the bootblock format shown in Figure 2-5.

Figure 2-8. Memory Layout for Secondary Bootstrap

The primary bootstrap creates a PFN bit map which contains a bit set for each page in physical memory that is both present and contains correct parity at the time of the bootstrap.

The primary bootstrap also creates an RPB (restart parameter block). Figure 2-9 shows the contents of the RPB when the primary bootstrap transfers control to the secondary bootstrap.

OFFSET	
00:	address of the RPB
04:	0
08:	0
0C:	0
10:	PC at restart/halt
14:	PSL at restart/halt
18:	restart reason from microcode
1C:	saved boot parameter R0
20:	saved boot parameter R1
24:	saved boot parameter R2
28:	saved boot parameter R3
2C:	saved boot parameter R4
30:	saved boot parameter R5
34:	two longwords reserved
3C:	disk block address of secondary image
40:	size of secondary bootstrap file
44:	descriptor for PFN bitmap (two longwords)
4C:	count of good physical pages
50:	reserved
54:	physical CSR address of boot device
58:	four longwords reserved
68:	boot file name in ASCII, up to 40 characters (ten longwords)
90:	eight longwords reserved
B0:	system control block base address

Figure 2-9. Restart Parameter Block

Microverify

Microverify is a microcoded internal test that runs automatically when the system is powered on. In addition, the TEST console command runs Microverify. The purpose of Microverify is to isolate module level errors and return error status to the main data path microcode. It indicates status by lighting LEDs on the data path module and by displaying messages on the console terminal.

In the MicroVAX I system, the LEDs on the data path module are connected to an LED display on the rear patch panel assembly. Thus, if an error occurs during Microverify, the system user can glance at the back of the system to obtain the error code.

Microverify runs under control of the data path microcode and tests the data path module, the memory controller module, and the interface between them. The successful completion of Microverify indicates that the CPU works and can run macrocode. Therefore, the primary bootstrap in the EPROM can be loaded and executed.

Microverify also tests for the checksum in the primary bootstrap by adding all of the 16-bit words in the primary bootstrap together and checking for a sum of zero. (The last 16-bit word in the primary bootstrap must be the twos complement of the sum of all previous 16-bit words.) If the sum is not zero, Microverify fails and displays the error code 6 in the data path module LEDs and in the LED display on the rear patch panel.

Microverify does not test the Q22 bus, external memory, or any devices.

Console Terminal Messages

When Microverify begins running and determines that the console terminal is usable, it displays the message "MICROVERIFY STARTED" on the console terminal.

Within five seconds after the "MICROVERIFY STARTED" message is displayed, Microverify completes either successfully or unsuccessfully. If it completes successfully, the message "MICROVERIFY PASSED" is displayed on the console terminal. If it does not complete successfully, the message "MICROVERIFY FAILED" is displayed.

Two other situations are also possible. The "MICROVERIFY STARTED" message may never appear because the "MICROVERIFY FAILED" message is displayed immediately. The second situation is that "MICROVERIFY STARTED" is displayed, but no other message appears. You should not wait any longer than ten seconds for a second message. If either of these situations occurs, Microverify has failed and the data path module is probably at fault.

LEDs

When Microverify begins, all three LEDs on the data path module are lit. In the system, the LED display on the rear patch panel assembly shows the number 7.

If Microverify fails, it lights the LEDs in certain combinations to indicate where the error occurred, and it sends an error code to the rear patch panel assembly LED display. Table 2-5 lists the light combinations, the corresponding error code displays, and their meanings.

If Microverify completes successfully, it sets the LEDs to 3 and passes control to the console microcode.

Normally, Microverify is invoked because the system is powered on, and "warm start, boot, halt" is the default action set in the option switches. Assuming the normal case, the console microcode searches for 64K bytes of contiguous good memory after it receives control. If the console microcode finds 64K bytes of good memory, it loads the primary bootstrap from the boot EPROM into the found memory, and transfers control to the primary bootstrap.

If the console microcode does not find 64K bytes of contiguous good memory, the LEDs remain set to 3 (off, on, on).

If control passes to the primary bootstrap, the primary bootstrap also lights the LEDs on the DAP module, and the LED display on the rear patch panel assembly, to indicate its progress. If the primary bootstrap fails, an error code of 3, 2, or 1 is displayed in the LEDs to indicate the problem. If the primary bootstrap detects a parity error when it tests memory, it leaves the LEDs set at 3. Table 2-5 also lists the light combinations, the corresponding error code displays, and their meanings for the error codes controlled by the primary bootstrap.

When the primary bootstrap completes successfully and passes control to the secondary bootstrap, all three LEDs on the DAP module are turned off, and the LED display is blank except for a lighted dot in the lower right-hand corner of the display.

Table 2-5. LEDs and Patch Panel Display

DAP LEDs	LED Display	Meaning
on, on, on	7	Microverify failed before completing the data path microsequencer test. (The error is most likely on the DAP module.)
on, on, off	6	Error found on DAP module.
on, off, on	5	Error found on MCT module.
on, off, off	4	Undetermined error in DAP/MCT interface.
off, on, on	3	Microverify worked as expected. If bootstrapping was attempted, bad memory was found. (An error code of 3 in the LEDs has several meanings. Please read the section titled "LEDs" on the previous two pages for a more exact description.)
off, on, off	2	No boot device was found.
off, off, on	1	Unable to boot from selected device.
off, off, off	•	Control has passed to the secondary bootstrap.

Note: A number displayed in the LEDs is meaningful only if the system is in console halt mode, as indicated by the system prompt >>>.

Modes of Operation

Microverify can run in one of two modes: single pass mode, or infinite loop mode. A jumper on the data path module selects the mode.

The DAP module is shipped with the Microverify jumper in the correct position (in) for single pass mode. This is the normal operating mode. The messages, lights, and error codes appear in single pass mode.

Infinite loop mode can be used to isolate intermittent failures. However, Microverify never returns control to the main microcode in this mode.

Figure 2-10 shows the location and proper placement of the Microverify jumper for single pass mode, and the correct orientation for reading the LEDs. The LEDs may be seen by looking at the data path module edge-on from the edge of the board that the connectors are mounted on.

For more information about Microverify, see Appendix D.

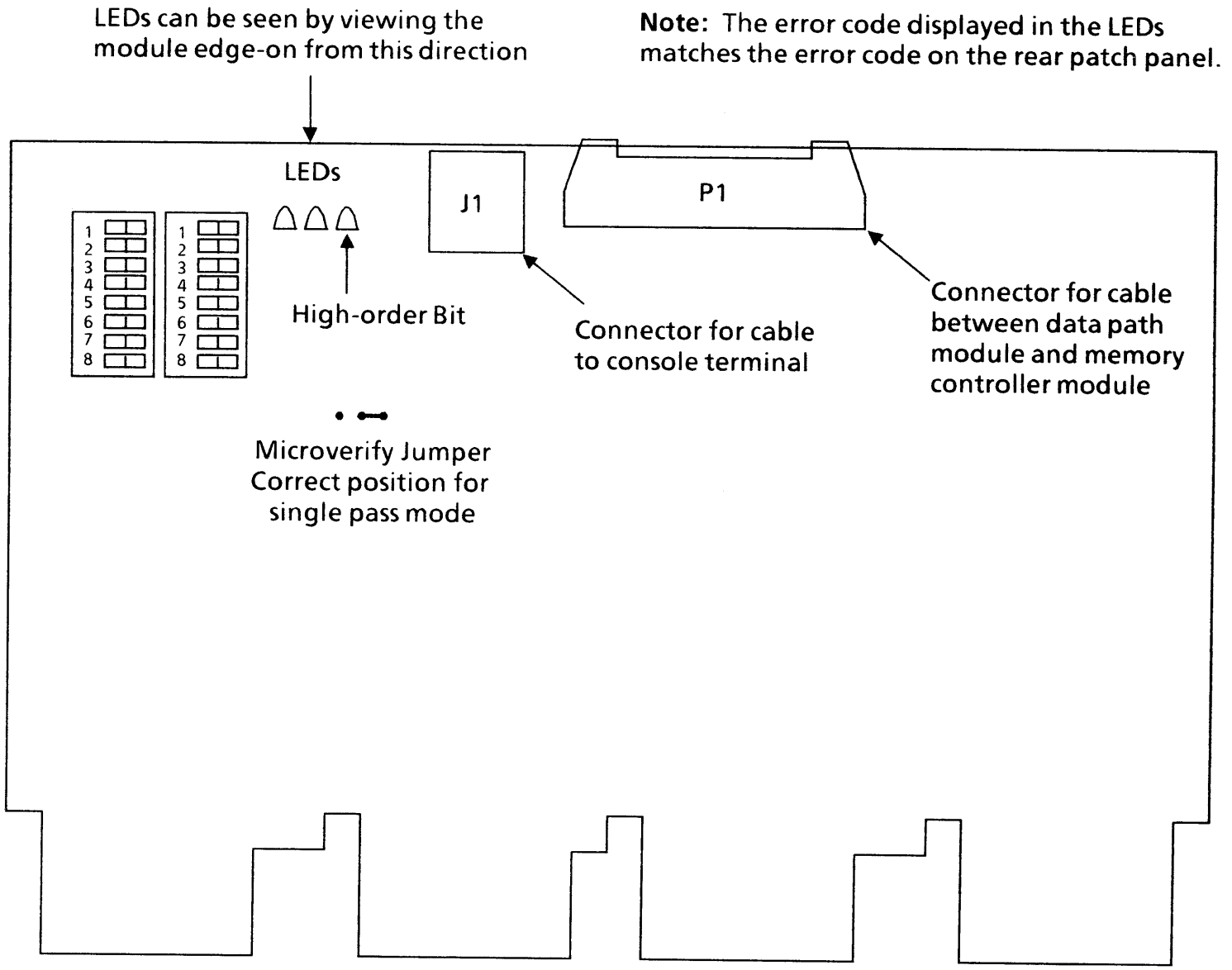


Figure 2-10. Location of LEDs and Microverify Jumper on Data Path Module

Console Microcode

The MicroVAX I console microcode provides the following functions.

- It controls bootstrapping, and initializes the state of the machine and the processor registers.
- It controls restart and halt procedures, and responds to internal halts.
- It provides a means to view the internal state of the machine for diagnostics and debugging.

Console Terminal Modes

The MicroVAX I console operates in one of two modes: program I/O mode, or console I/O mode.

In program I/O mode, normal user and system programs are running, and any characters typed at the console terminal are transmitted directly to the processor console registers.

In console I/O mode, characters typed at the console terminal are handled directly by the console microcode.

The following events all place the MicroVAX I system in console I/O mode:

- The Halt button on the MicroVAX I system front panel is pressed.
- A Halt instruction is executed in kernel mode with the "halt" action selected in the data path module DIP switches.
- The system is powered on or the Restart button is pressed with any action selected in the data path module DIP switches that results in a Halt. For example, if "warm start, boot, halt" is the action set in the switches (the default), and both warm

start and bootstrap fail when the system is powered on or when the Restart button is pressed, the system halts in console I/O mode.

- The BREAK key is pressed on the console terminal keyboard when break detect is enabled in the data path module option DIP switches.
- The hex value F05 is written to the TXDB register.
- One of the errors listed in Table 2-6 below occurs.

A user can return the MicroVAX I system to program I/O mode from console mode by issuing the boot, start, or continue console commands with the Halt pushbutton in the "out" (released) position. For more information about these console commands, see Chapter 6 of the *MicroVAX I Owner's Manual*, EK-KD32A-OM.

Console Halt Codes

The console microcode displays a halt code and the contents of the program counter on the console terminal when a halt occurs. Table 2-6 shows the available console halt codes and their meanings.

Table 2-6. Console Halt Codes

Code	Meaning
00	Reserved
01	Microverify succeeded.
02	Console halt/break
03	Power up
04	The interrupt stack was not valid when the CPU tried to push the PC/PSL during an exception or an interrupt.
05	A second machine check occurred while the CPU was processing an existing machine check.
06	A Halt instruction was executed while the processor was in kernel mode.
07	Reserved
08	Reserved
09	Reserved
0A	A CHMx instruction was executed when the CPU was executing on the interrupt stack.
0B	Reserved
0C	A hard memory error occurred while the CPU was trying to read a system control block vector.
0D–10	Reserved
FF	Microverify failed.

Bit-mapped Video Interface

The MicroVAX I processor is also used in other DIGITAL products; for example, as the processor for a bit-mapped video workstation. The workstation includes a video controller, monitor and keyboard. For this monitor to work with the MicroVAX I processor as

the console terminal, the following requirements must be met.

- The MicroVAX I data path module must have the 16 KB boot EPROM (rather than the 8 KB EPROM). The additional 8K holds the font table, the keyboard translation tables, and the video controller parameter tables.
- Option switch number 2 in the data path module option switch pack must be set on.
- The base address for the video controller's control and status registers (CSRs) must be 777200 octal (2000 1E80 hex), located in Q22 bus I/O space.
- The starting address of the video controller's bitmap RAM must be 3840K decimal (003C 0000 hex).

When option switch 2 on the data path module is set on, the console microcode assumes that a 16 KB boot EPROM is present on the data path module, and translates keyboard input and output using the tables supplied in the second 8K bytes of the boot EPROM.

When the MicroVAX I system is powered on, the console microcode initializes the processor, then tests option switch number 2 on the data path module. If option switch 2 is on, communication between the processor and the user is through the monitor and keyboard. The console microcode then does the following.

- It loads the display screen parameters from the boot EPROM into the video controller.
- It initializes the scan line map RAM.
- It clears the bitmap RAM.
- It initializes the keyboard UART.
- It enables the video logic.

The console microcode assumes the starting address of the video controller's scan line map RAM is 003F F800 (hex), and reserves hexadecimal addresses 003F F7E0 and 003F F7E2 to store the current row and column of the cursor position.

After the video controller is initialized, Microverify runs. Microverify does not check the bit-mapped video interface. The Microverify success or failure messages are displayed on the monitor (and indicated in the DAP module and patch panel LEDs in case the monitor or video controller is not working). Once Microverify completes successfully, the console microcode regains control.

Assuming that the default action "warm start, boot, halt" is set in the option switches, the console microcode searches for 64K bytes of contiguous good memory and upon finding it, loads the complete boot EPROM (the entire 16K) into memory. The console microcode also does an I/O reset, and then reinitializes the video controller and the keyboard.

Control then transfers to the primary bootstrap, which is the first 8K bytes copied into memory from the boot EPROM. The primary bootstrap also examines option switch 2 and if it is on, uses the monitor for console I/O. Therefore, all error messages are displayed on the monitor as well as any interactive input and output. The primary bootstrap does not reinitialize the video controller; it assumes that the console microcode has already done this.

The primary bootstrap locates the secondary bootstrap, loads it into memory, and transfers control. When the system finishes bootstrapping, it is in program I/O mode, and the video controller is no longer under control of the console microcode.

Note: The MicroVAX I console terminal registers always communicate with the data path UART.

If the system is placed in console I/O mode, the monitor is again used by the console microcode as the console terminal. The console microcode does not reinitialize the video controller; it assumes it is still initialized.

Interrupts and Exceptions

Interrupts and exceptions divert execution from the normal flow of control. An interrupt is caused by some activity outside the CPU, while an exception is caused by the execution of the current instruction.

Interrupts

The MicroVAX architecture specifies 31 interrupt priority levels (IPLs), which occur in MicroVAX I as shown in Table 2-7.

Table 2-7. Interrupt Priority Levels

IPL	Interrupt
1F	unused
1E	power fail
1D	memory write error
1C–18	unused
17	Q22 bus IRQ7
16	interval timer
16	Q22 bus IRQ6
15	Q22 bus IRQ5
14	console receive
14	console transmit
14	Q22 bus IRQ4
13–10	unused
01–0F	software interrupts

Interrupts from Q22 bus devices are arbitrated by comparing the level of the interrupting device to the current processor IPL. However, when an interrupt from a bus device is serviced, the interrupt priority level is raised to 17 (hex). Software may choose to subsequently lower the IPL to the level of the interrupting device. Subsequent interrupts lower than the current IPL of the processor are blocked, including other interrupts from the bus device being serviced.

The interrupt subsystem is controlled by the Interrupt Priority Level (IPL) processor register, the Software Interrupt Request Register (SIRR), and the Software Interrupt Summary Register (SISR). These registers are implemented as defined by the MicroVAX architecture, which is identical to the VAX architectural specification for these registers.

Exceptions

The MicroVAX architecture recognizes seven classes of exceptions:

1. Arithmetic Traps/Faults
2. Memory Management Exceptions
3. Operand Reference Exceptions
4. Instruction Execution Exceptions
5. Trace Faults
6. Instruction Emulation Exceptions
7. System Failure Exceptions

The MicroVAX architecture specifications for classes 1 through 5 are the same as the VAX architecture specifications.

Arithmetic Traps/Faults

Arithmetic traps/faults occur as the result of an arithmetic or conversion operation. They are mutually exclusive and are assigned the same vector in the system control block (SCB). Each indicates that an exception occurred during the last instruction and that the instruction has been completed (trap) or backed up such that the instruction can be restarted (fault). An appropriate distinguishing code is pushed on the stack as the first of three longwords:

type code	:SP
PC of next instruction to execute*	
processor status longword	

*same as the instruction causing
the exception in the case of a fault

The arithmetic traps/faults and the corresponding type codes pushed on the stack are shown in Table 2-8.

Table 2-8. Arithmetic Traps/Faults

Type	Code (hex)	Exception Type
	1	integer overflow
TRAPS	2	integer divide by zero
	7	subscript range
	8	floating overflow
FAULTS	9	floating divide by zero
	A	floating underflow

Memory Management Exceptions

Two types of faults are associated with memory mapping and protection: access control violation and translation not valid.

An access control violation fault is an exception indicating that the process attempted a reference not allowed at the access mode at which the process is operating. An access control violation fault is also taken if a length violation is detected; that is, if the virtual address is beyond the end of the applicable page table.

A translation not valid fault is an exception indicating that the process attempted a reference to a page for which the valid bit in the page table entry was not set.

These faults have distinct vectors in the system control block, and two longword parameters are pushed on the stack in each case, in addition to the PC and PSL. The first parameter contains zero in bits <31:3>.

Bit <1> of the first parameter is set if the fault occurred during the reference to the process page table (as opposed to the system page table reference, which could also fault during a process space reference).

Bit <2> when set indicates that the intended access was a write or modify; when bit <2> is clear, the intended access was a read.

Bit <0> when set indicates that an access control violation was the result of a length violation rather than a protection violation; this bit is always clear for a translation not valid fault.

The second parameter pushed is the virtual address which caused the fault.

If a process attempts to reference a page for which the page table entry specifies both not valid and access violation, an access control violation fault occurs.

Operand Reference Exceptions

There are two types of exceptions that can occur during operand reference: reserved addressing mode fault and reserved operand exception (fault or abort).

A reserved addressing mode fault is an exception indicating that an operand specifier attempted to use an addressing mode that is not allowed in the situation in which it occurred. No parameters are pushed on the stack.

A reserved operand exception indicates that an operand accessed has a format reserved for future use by DIGITAL. No parameters are pushed on the stack. This exception always backs up the PC to point to the opcode.

The exception service routine may determine the type of operand by examining the opcode using the stored PC. Only the changes made by instruction fetch and because of operand specifier evaluation are guaranteed to be restored. Therefore, some instructions are not restartable. These exceptions result in aborts rather than faults. The PC is always restored properly unless the instruction attempted to modify it in a manner that results in unpredictable results.

Instruction Execution Exceptions

There are three types of instruction execution exceptions: reserved/privileged instruction fault, extended function fault, and breakpoint fault.

A reserved or privileged instruction fault occurs when the CPU encounters an opcode that is not specifically defined, or that requires higher privileges than the current mode. No parameters are pushed on the stack.

An extended function fault occurs when an opcode reserved to the customer or to DIGITAL's Computer Special Systems (CSS) group is executed. The operation is identical to the reserved/privileged instruction fault except that the event is caused by a different set of opcodes, and faults through a different vector. All opcodes reserved to customers and CSS start with FC (hex), which is the XFC instruction.

A breakpoint fault is an exception that occurs when the breakpoint instruction (BPT) is executed. No parameters are pushed on the stack.

Trace Faults

A trace is an exception that occurs between instructions when trace is enabled. Tracing is used for tracing programs, for performance evaluation, or for debugging purposes. It is designed so that one and only one trace fault occurs before the execution of each traced instruction. The saved PC on a trace is the address of the next instruction that would normally be executed. The trace fault for an instruction takes precedence over all other exceptions.

To ensure that exactly one trace occurs per instruction despite other traps and faults, the PSL contains two bits: trace enable (T), and trace pending (TP). The trap is implemented by copying PSL <T> to PSL <TP>. PSL <TP> is the bit that actually generates the exception; the fault is generated before any other processing at the start of the next instruction.

Instruction Emulation Exceptions

The MicroVAX architecture specifies emulation support for instructions that are not implemented in hardware.

The MicroVAX architecture specifies two kinds of emulation support: software emulation, and software emulation assisted by hardware.

When an instruction is executed that is emulated strictly in software, a reserved instruction fault occurs.

When an instruction is executed that is emulated in software with a hardware assist, an instruction emulation exception occurs.

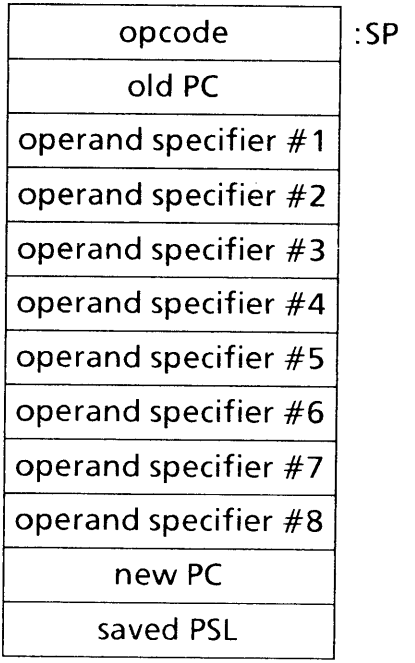
Of those instructions emulated by software with a hardware assist, there are five character string instructions that the MicroVAX I implements in

hardware; emulation is not required for these instructions. Thus, on the MicroVAX I, the following instructions are emulated in software with a hardware assist:

- decimal string:
  ADDP4, ADDP6, ASHP, CMPP3, CMPP4,
  CVTLP, CVTPL, CVTPS, CVTPT, CVTSP,
  CVTTP, DIVP, MOVP, MULP, SUBP4, SUBP6
- character string:
  CMPC5, MATCHC, MOVTC, MOVTUC
- integer:
  CLRO, MOVO
- address manipulation:
  MOVAO, PUSHAO
- EDITPC and CRC

If one of these instructions is executed, the emulation process consists of the following steps, assuming that the TP (trace pending) and FPD (first part done) bits in the PSL are clear:

1. The operand specifiers are evaluated as they occur in the instruction stream. The address is saved for address and modify access types; the operand is saved for read access types.
2. Ten longword parameters are pushed on the stack (in addition to the PC and PSL to be restored on returning from this exception):



The opcode parameter contains the faulting opcode; that is, the opcode of the instruction to be emulated. The old PC points to the location of the faulting instruction.

The specifier parameters contain the address of the operand or the operand itself: for an .rx specifier, the parameter is the operand value; for .wx and .ax specifiers, the parameter is the operand address. A register is indicated by a reserved system space address corresponding to the ones complement of the register number. The parameter corresponding to a specifier that does not exist is unpredictable. The new PC points to the instruction following the faulting instruction.

3. An exception is initiated in the current mode through the emulated instruction vector using C8 (hex) as the system control block offset. The FPD (first part done) bit in the saved PSL is clear. The TP bit in the saved PSL is set if T was set. The TP bit and bits <7:0> in the new PSL are cleared. All other bits in the new PSL are unchanged from their previous state.

If the FPD bit was set, a suspended emulated instruction fault is taken in the current mode through a vector at system control block offset CC (hex). No parameters are pushed on the stack other than the PC and PSL. The TP bit in the saved PSL is set if T was set; all other bits are unchanged. The TP bit, the FPD bit, and bits <7:0> in the new PSL are cleared. All other bits in the new PSL are unchanged from their previous state.

System Failure Exceptions

There are three types of system failure exceptions: kernel stack not valid abort, interrupt stack not valid halt, and machine check.

A kernel stack not valid abort indicates that the kernel stack was not valid while the processor was pushing information onto it during the initiation of an exception or interrupt. Usually, this is an indication of a stack overflow or other executive software error. The attempted exception is transformed into an abort that uses the interrupt stack. No extra information is pushed on the interrupt stack in addition to the PSL and PC. The IPL is raised to 1F (hex). If the kernel stack is not valid during the normal execution of an instruction (including CHMK or REI), the normal memory management fault is initiated.

An interrupt stack not valid halt indicates that the interrupt stack was not valid or that a memory error

occurred while the processor was pushing information onto it during the initiation of an exception or interrupt. No further interrupt requests are acknowledged, and the processor halts.

A machine check exception indicates that the CPU detected an internal error. All machine checks have a common format and always push twelve bytes of data. The pushed information consists of a machine check code and two parameters. The location addressed by the stack pointer always contains the hexadecimal value 0000000C to indicate that twelve bytes of data are on the stack. Figure 2-11 shows the general format of the machine check stack.

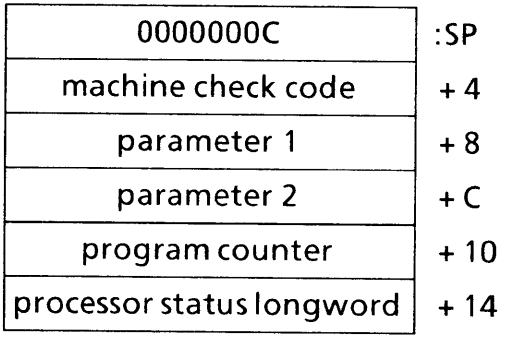


Figure 2-11. Machine Check Stack

The machine check microroutine tests the internal machine-check-in-progress flag. If the state of the flag is true, then another machine check has occurred while a previous machine check was being processed. A double error halt (console halt code 05) is initiated.

If the state of the flag is false, then it is set true. It is the responsibility of the software to clear the flag by writing to the machine check error summary register (MCESR) when processing of the machine check is

complete. Table 2-9 describes the machine checks that can occur, and the parameters associated with them.

These parameters are preserved on a "best effort" basis since some processor errors cannot guarantee the state of the machine.

Table 2-9. Machine Checks

Error Code	Name	Description
0	memory controller bug check	An invalid state was reached in the memory controller and it was unable to successfully complete the last function. parameter 1: The contents of the memory controller register "virtual." This register usually holds the physical address of the last function (see Note 1 below). parameter 2: The address presented to the memory controller at the start of the function.
1	unrecoverable read error	An unrecoverable read error occurred on the last memory controller function. The error may have been a parity error or an ECC error, depending on the type of memory present. parameter 1: The physical address of the page containing the error (see Notes 1 and 2 below). parameter 2: The address presented to the memory controller at the start of the function.
2	nonexistent memory	A bus timeout occurred during the last memory controller read function. parameter 1: The physical address of the nonexistent memory (see Notes 1 and 2 below). parameter 2: The address presented to the memory controller at the start of the function.
3	illegal operation	An attempt was made to access an unaligned word or a longword in I/O space. parameter 1: The physical address of the illegal I/O reference (see Notes 1 and 2 below). parameter 2: The address presented to the memory controller at the start of the function.

Table 2-9. Continued

Error Code	Name	Description
4	unrecoverable page table read error	An unrecoverable error occurred while attempting to read a page table entry. The error may have been a parity, ECC, or timeout error. parameter 1: The physical address of the page table entry (see Note 1 below). parameter 2: The virtual address associated with the page table entry; that is, the address that caused the page table entry to be read.
5	unrecoverable page table write error	An unrecoverable error occurred while attempting to write the modify bit in a page table entry. This error reflects hardware that is in an unrunnable state and should be treated as a write timeout error. parameter 1: The physical address of the page table entry (see Note 1 below). parameter 2: The virtual address associated with the page table entry; that is, the address that caused the page table entry to be read.
6	control store parity error	A control store parity error has occurred. This is a fatal data path module error. parameter 1: zero parameter 2: zero
7	micromachine bug check	An invalid state has been reached in the micromachine. This is a fatal hardware or microcode error. parameter 1: zero parameter 2: zero

Table 2-9. Continued

Error Code	Name	Description
8	Q22 bus vector read error	An error was encountered while attempting to read an interrupt vector address from the Q22 bus. parameter 1: zero parameter 2: zero
9	write parameter error	An error was encountered during an exception while attempting to write the user, supervisor, or executive stack after having verified that the write would succeed (that is, CHMx and emulation). parameter 1: The virtual address that was being written (see Note 2 below). parameter 2: zero

Note 1: Only the I/O space flag (bit <29>) and the low 22 bits of the physical address in parameter 1 are meaningful.

Note 2: Physical and virtual addresses returned on memory controller errors may not be the actual address of the error if a page crossing occurs. If the page offset (that is, bits <8:0>) is:

000000001 binary and the data length was word, or

000000001 binary and the data length was long, or

000000010 binary and the data length was long, or

000000011 binary and the data length was long

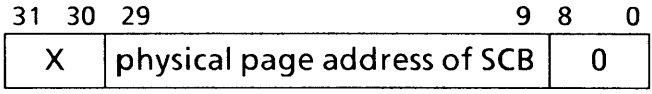
then the page in which the error occurred could be the one addressed or the one logically preceding the one specified.

System Control Block

The system control block (SCB) is a page containing the vectors by which exceptions and interrupts are dispatched to the appropriate service routines.

System Control Block Base Register

The system control block base (SCBB) is a processor register containing the physical address of the system control block, which must be page-aligned.



At bootstrap time, the contents of the SCBB are unpredictable. On writes, bits <8:0> are ignored. These bits are always read as zero.

System Control Block Vectors

A system control block vector is a longword in the SCB that is examined by the CPU when an exception or interrupt occurs to determine how to service the event.

Bit <1> of each vector is ignored. Bit <0> of each vector contains a code interpreted by the hardware as follows:

0 Service this event on the kernel stack unless already running on the interrupt stack, in which case, service on the interrupt stack.

1 Service this event on the interrupt stack. If this event is an exception, raise the IPL to 1F (hex).

Bits <31:2> of each vector contain the virtual address of the service routine, which must begin on a longword boundary. Table 2-10 shows the organization of the system control block including the offset into the SCB, a description of the type of event, and the number of longword parameters pushed on the stack in each case in addition to the PC and PSL.

This concludes the description of the programming interface for the KD32-AA and KD32-AB CPUs. The next chapter describes the CPU configuration.

Table 2-10. System Control Block Organization

Offset (hex)	Implementation	Type	Parameters Pushed	Notes
00	unused	–	–	reserved
04	machine check	abort/fault/trap	3	The parameters are dependent on the type of error (see Table 2-9, "Machine Checks").
08	kernel stack not valid	abort	0	This event is serviced on the interrupt stack. The IPL is raised to 1F (hex).
0C	power fail	interrupt	0	The IPL is raised to 1E (hex).
10	reserved/privileged instruction	fault	0	
14	customer reserved instruction	fault	0	This is the XFC instruction.
18	reserved operand	fault/abort	0	
1C	reserved addressing mode	fault	0	
20	access control violation	fault	2	The parameters are the virtual address causing the fault and a status code.
24	translation not valid	fault	2	The parameters are the virtual address causing the fault and a status code.
28	trace pending (TP)	fault	0	
2C	breakpoint instruction	fault	0	
30	unused			
34	arithmetic	trap/fault	1	The parameter is the type code.
38–3C	unused			reserved
40	CHMK	trap	1	The operand word is sign-extended and pushed on the stack.
44	CHME	trap	1	The operand word is sign-extended and pushed on the stack.

Table 2-10. Continued

Offset (hex)	Implementation	Type	Parameters Pushed	Notes
48	CHMS	trap	1	The operand word is sign-extended and pushed on the stack.
4C	CHMU	trap	1	The operand word is sign-extended and pushed on the stack.
50–5C	reserved			
60	write bus timeout	interrupt	0	The IPL is raised to 1D.
64–80	unused			reserved
84–BC	software levels 1–F	interrupt	0	
C0	interval timer	interrupt	0	The IPL is raised to 16 (hex).
C4	reserved			
C8	emulated instruction	fault	10	This vector is used when the FPD (first part done) bit in the PSL is clear.
CC	emulated instruction	fault	10	This vector is used when the FPD bit in the PSL is set.
D0–DC	reserved			
E0–EC	reserved			These vectors are reserved for CSS and customers.
F0–FC	reserved			
100–1FC	reserved			
200–3FC	device vectors	interrupt	0	These vectors are used by MicroVAX I to directly vector device interrupts from the Q22 bus. The vector is determined by adding 200 to the vector supplied by the device. Only device vectors in the range from 0 to 1FC (hex) are allowed. Interrupt priority levels 14 to 17 (hex) correspond to Q22 bus levels IRQ4 to IRQ7.

Chapter 3
Processor Configuration

The KD32-AA and KD32-AB processors are configured at the factory for normal operation before they are shipped. This chapter describes the normal configuration and how to change it if necessary. Also described are the power and cooling specifications for the board set.

Processor Configuration Overview

The KD32-AA or KD32-AB processor consists of two printed circuit boards that plug into the system backplane. The two boards are:

- the data path module, M7135 (KD32-AA) or M7135-YA (KD32-AB), and
- the memory controller module, M7136.

Two sets of dual-in-line package (DIP) switches are mounted on the data path module. Each package contains eight switches. One package is the low-order byte of the system identification (SID) register. These switches identify the hardware revision level and should not be changed.

The other package is the option switches. These switches determine the baud rate, break detect enable, the system recovery action, the console terminal type, and the bootstrap search order.

The option switches are set at the factory for normal operation. There is no need to change the setting of the option switches unless you have a special situation that calls for a different setting in the switches. The follow-

ing sections in this chapter explain how to reset the switches.

Option Switches

Figure 3-1 shows the data path module and the location of the option switch pack.

The option switches have these functions:

8:7 baud rate selection
These switches specify the rate of data transmission between the processor and the console terminal.

6 reserved

5 break detect enable
This switch determines whether or not a break condition on the serial line causes a halt.

4:3 recovery action
These switches determine the series of activities that the processor attempts during power on.

2 console terminal type
This switch identifies the type of console terminal connected to the system.

1 bootstrap search order
This switch determines which devices are searched when the system bootstraps.

Table 3-1 shows what the switch settings mean. The settings listed in **bold** are the switch positions set at the factory. **For normal operating conditions, all switches should be set off.**

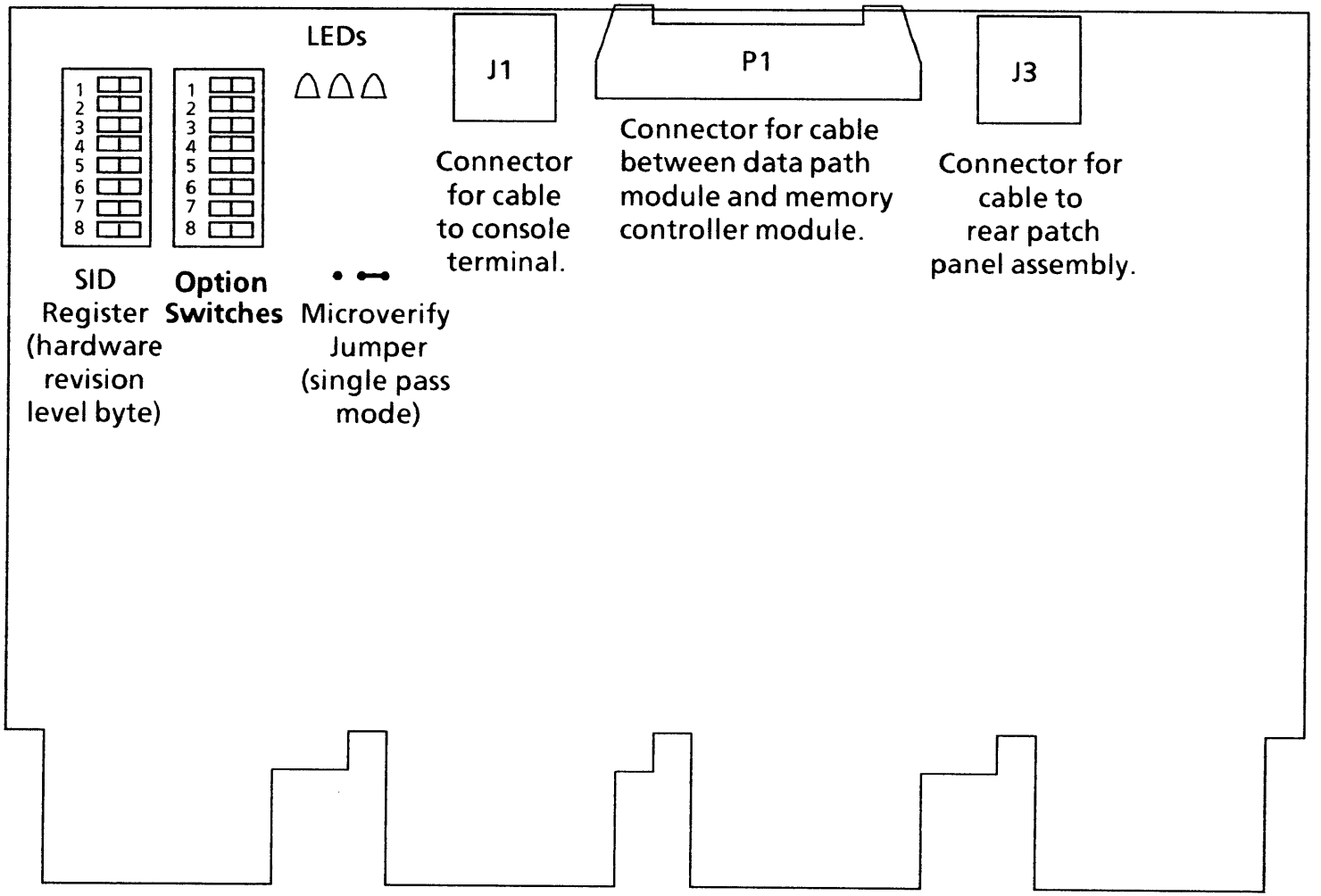


Figure 3-1. Location of Switch Packs on Data Path Module

Table 3-1. Option Switch Settings

Switch	On/Off	Meaning
8:7 baud rate select	**both off**	**9600**
	8 off, 7 on	19,200
	8 on, 7 off	300
	both on	1200
6	**off**	no effect
5 break detect	**off**	**break key disabled**
	on	break key enabled
4:3 recovery action	**both off**	**warm start, boot, halt**
	4 off, 3 on	boot, halt
	4 on, 3 off	warm start, halt
	both on	halt
2 console terminal	**off**	**VT100 compatible**
	on	bit-mapped graphics terminal
1 bootstrap search order	**off**	**all devices searched**
	on	disk/diskette drives bypassed

Each switch setting is explained in more detail in the following paragraphs.

Baud Rate Select

When the system is powered on, the processor examines option switches 7 and 8 to set the data transmission speed between itself and the console terminal.

If option switches 7 and 8 are both set to the *off* position, data are transmitted between the system and the

terminal at a rate of 9600 bits per second. The switches are set for this rate at the factory.

If option switch 8 is *off* and switch 7 is *on*, data are transmitted at 19,200 bits per second.

If option switch 8 is *on* and switch 7 is *off*, data are transmitted at 300 bits per second.

If option switches 7 and 8 are both set to the *on* position, data are transmitted between the system and the terminal at a rate of 1200 bits per second.

Option switches 7 and 8 are logically ORed with the rotary baud rate select switch on the MicroVAX I system patch panel assembly. The encoding for the rotary switch is as follows:

Rotary Switch Setting	Baud Rate Selected
3	9600
2	19,200
1	300
0	1200

The rotary switch settings apply as long as DIP switches 7 and 8 are both set off. Thus, for the rotary baud rate select switch to have the desired effect, DIP switches 7 and 8 must be left in their default setting: both off.

Break Detect

The MicroVAX I system is designed to be connected to any of the console terminals in DIGITAL's VT100 or VT200 family of terminals. The keyboard that comes with these terminals has a key marked BREAK. Switch number 5 in the option switch pack determines whether the BREAK key on the console terminal keyboard has any effect on the MicroVAX I system.

If switch number 5 is on, the BREAK key has the same effect as the HALT button on the system front panel; that is, pressing BREAK halts the processor.

The KD32-AA and KD32-AB processors are shipped with switch number 5 set to the *off* position so that the BREAK key is disabled.

Recovery Action

The processor examines option switches 3 and 4 to determine what actions to take next when one of the following situations occurs.

- The system is powered on.
- The RESTART button on the front panel is pressed. (The RESTART button is a momentary switch and releases automatically.)
- A Halt macroinstruction is executed in kernel mode.

Warm Start, Boot, Halt

If option switches 3 and 4 are both set to the *off* position, and one of the situations listed above occurs, the MicroVAX I system attempts a warm start. If the warm start fails, the system tries to bootstrap. If bootstrap fails, the system enters console mode and waits for a command to be entered from the console terminal.

The difference between a warm start and a bootstrap is that when the processor is able to perform a warm start, the program it was running is still in memory and the processor can resume executing the program. When the processor bootstraps, it begins by loading the secondary bootstrap or system image into memory.

The processor is shipped with switches 3 and 4 off; warm start, boot, halt is the default recovery action.

Note: With all software currently sold by DIGITAL for the MicroVAX I, a warm start will not succeed when the Restart button on the front panel is pressed. Pressing the Restart button will always bootstrap the system.

Boot, Halt

If switch 4 is *off*, switch 3 is *on*, and one of the situations listed above occurs, the MicroVAX I system tries to bootstrap by searching the available devices for the secondary bootstrap or a system image. If bootstrap fails, the system enters console mode and waits for a command to be entered from the console terminal.

Warm Start, Halt

If switch 4 is *on*, switch 3 is *off*, and one of the situations listed above occurs, the MicroVAX I system attempts a warm start. If the warm start fails, the system enters console mode and waits for a command to be entered from the console terminal.

Halt

If option switches 3 and 4 are both set to the *on* position, and one of the situations listed above occurs, the MicroVAX I system enters console mode and waits for a command to be entered from the console terminal. The console commands are described in Chapter 6, "The Console Interface" of the *MicroVAX I Owner's Manual*.

Console Terminal Type

When the processor bootstraps, the bootstrap examines option switch 2 to determine what type of console terminal it is attached to.

If option switch 2 is set to the *off* position, a console terminal that is compatible with DIGITAL's VT100/VT200 family is connected, and the processor uses the CPU patch panel for console command input and output. The processor is shipped with switch 2 set *off*.

The MicroVAX I processor is also used in other DIGITAL products; for example, as the processor for a bit-mapped video workstation. When option switch 2 is set to the *on* position, a bit-mapped graphics terminal is connected, and the processor uses this bit-mapped graphics terminal for console command input and output.

This switch is set correctly at the factory for the system that the processor is installed in, so there is normally no requirement for this switch setting to be changed.

Bootstrap Search Order

Option switch 1 controls which devices are searched when the system bootstraps.

If option switch 1 is set to the *off* position, the following devices (if present) are searched in the order listed here until the secondary bootstrap or a system image is found: diskette drives, disk drives, MRV11 PROM, and DEQNA. The processor is shipped with switch 1 set *off*.

If option switch 1 is set to the *on* position, the primary bootstrap bypasses the diskette and disk drives, and searches only the MRV11 PROM and the DEQNA (if these devices are present) for the secondary bootstrap.

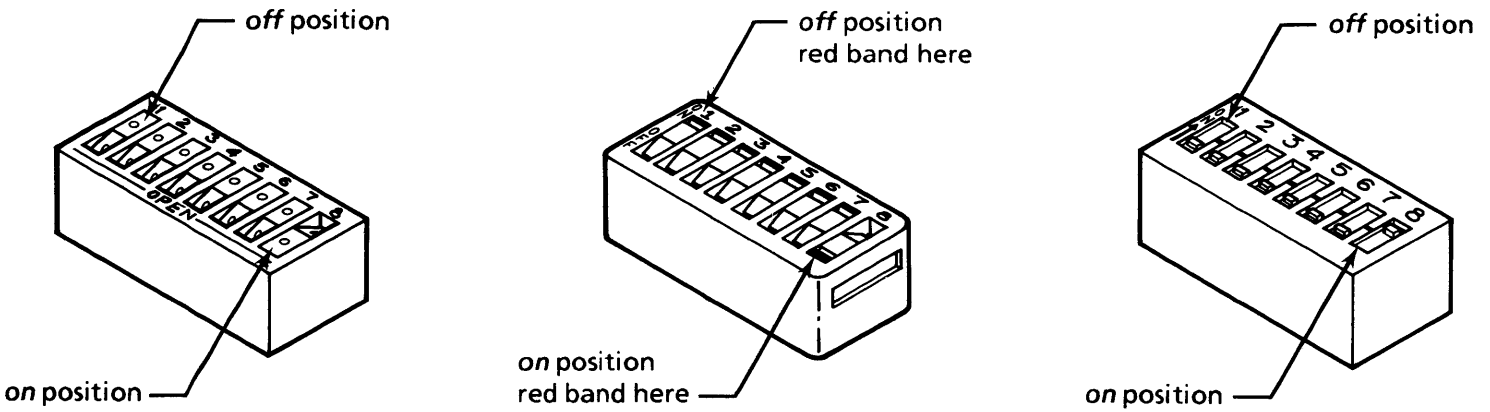
(This switch setting would be useful, for example, if you always wanted the secondary bootstrap to be provided by a host computer over the Ethernet.)

Resetting the Option Switches

If you need to enable the BREAK key on the console terminal keyboard, change the bootstrap search order, or select a different baud rate or recovery action, you can change the option switch settings. The following paragraphs describe how to do this.

Locate the option switch pack on the data path module, as shown in Figure 3-1.

The data path module can have one of three types of DIP switches: rocker, modified rocker, or slider. These three types are illustrated in Figure 3-2. Use Figure 3-2 to determine the type of switch pack mounted on the data path module. Then, use Table 3-1 to determine the switch settings you need, and set the switches according to the instructions in the following paragraphs.



Note: In each picture, switches 1 through 7 are shown in the *off* position, and switch 8 is shown in the *on* position.

Figure 3-2. Three Types of DIP Switches

Rocker Switches

A rocker switch pivots from the center and has two sides: an *on* side and an *off* side. Each side has a small depression in it, just large enough for the tip of a ballpoint pen. When one side of the switch is pressed down into the plastic switch casing, the opposite side is up, flush with the top of the casing. The word "open" is printed on the bottom edge of the casing. The switch numbers: "1," "2," and so on, are printed in white across the top edge of the casing. Think of the "open" side as the *off* side, and the numbered side as the *on* side.

A rocker switch is *off* when the side nearest the word "open" is depressed; that is, pushed down into the casing so that the depression on the opposite side is up. A rocker switch is *on* when the side nearest the switch number is pushed down into the casing so that the depression on the "open" side is up.

For example, to turn rocker switch number 8 *off* (assuming it is *on* to begin with), place the point of a pen (or any small pointed object except a pencil) in the depression of switch number 8 that is closest to the word "open" and press firmly. The side you are pressing on goes down into the switch casing, and the side nearest the number "8" comes up flush with the casing. Similarly, to turn switch number 8 *on* again, place the pen in the depression nearest the number "8" and press firmly so that this side of the switch goes down into the casing.

Modified Rocker Switches

A modified rocker switch also pivots from the center and has two sides: an *on* side and an *off* side. Each side has a red mark on it. When one side of the switch is pressed down into the plastic switch casing, the

opposite side is up, flush with the top of the casing, and the red mark on that side shows. The switch numbers: "1," "2," and so on, are printed in white across the top edge of the casing. The word "on" is printed on the upper left edge of the casing. The word "off" is printed on the lower left edge of the casing.

A modified rocker switch is *off* when the side nearest the word "off" is depressed; that is, pushed down into the casing so that the red mark on the "on" side is up. A rocker switch is *on* when the side nearest the word "on" is pushed down into the casing so that the red mark on the "off" side is up.

For example, to turn modified rocker switch number 8 *off* (assuming it is *on* to begin with), place the point of a pen (or any small pointed object except a pencil) on the side of switch number 8 that is closest to the word "off" and press firmly. The side you are pressing on goes down into the switch casing, and the side nearest the number "8" comes up flush with the casing. Similarly, to turn switch number 8 *on* again, place the pen on the switch side nearest the number "8" and press firmly so that the "on" side of the switch goes down into the casing.

Slider Switches

A slider switch slides in the plastic casing. It has a bump in the middle that is flush with the top of the casing. The switch numbers: "1," "2," and so on, are printed across the top edge of the casing. The word "on" is printed near the upper left edge of the casing, and an arrow indicates the *on* direction.

A slider switch is *on* when the bump is pushed toward the side of the casing labeled "on," in the direction of the arrow. A slider switch is *off* when the bump is pushed in the direction opposite to the arrow.

Place the pointed end of a pen behind the bump and slide the switch in the direction the arrow is pointing to set the switch to the *on* position. To turn the switch off, slide the switch in the opposite direction.

Power and Cooling

The data path module (M7135 or M7135-YA)) of the MicroVAX I processor draws a maximum of 7.0 amps at +5 volts (±5%) and 0.5 amps at +12 volts (±5%).

The memory controller module (M7136) draws a maximum of 7.0 amps at +5 volts (±5%).

The processor (both modules) presents 1.0 DC load and 4.0 AC loads on the Q22 bus.

The operating and storage temperature specifications for the KD32-AA or KD32-AB processor are shown in Table 3-2. **Note:** These specifications apply only to the processor; they do not apply to the MicroVAX I system.

Table 3-2. Processor Temperature Specifications

Mode	Temperature and Humidity
Operating	5° C to 60° C (41° F to 140° F) 10% to 90% relative humidity, non-condensing
Storage	−40° C to 66° C (−40° F to 151° F) up to 95% relative humidity, non-condensing

This concludes the description of the MicroVAX I processor configuration and specifications. The next chapter is a functional description of the processor.

Chapter 4
Functional Overview

This chapter is a functional overview of the major processor components. The general flow of data is discussed using several instructions as examples.

Figure 4-1 is a high-level block diagram of the MicroVAX I processor.

Data Path

The data path module (M7135 or M7135-YA) contains the main data path, register file, instruction decode, microsequencer and miscellaneous logic needed to implement the MicroVAX instruction set. It is contained on a single quad-height printed circuit board and has connectors to interface to the memory controller, the console terminal, and the rear patch panel.

The major components implemented on the data path module are:

- a 32-bit-wide ALU data path and register file implemented as a custom VLSI chip
- an 8K-deep by 40-bit-wide control store
- a 13-bit-wide microsequencer
- instruction decode logic
- a byte-wide internal data path which provides visibility to various processor states
- an 8K or 16K by 8-bit-wide boot EPROM
- a console interface

Each of these components is discussed briefly in the following paragraphs.

Data Path Chip

The execution of each microinstruction takes place in the data path chip. This 68-pin custom VLSI chip contains the main 32-bit data path. The chip is controlled by the microprogram. Twenty-one bits of the 40-bit microinstruction control the chip operations. The chip consists of:

- A 21-bit control store register
- A 32-bit bidirectional I/O port
- Two 32-bit internal buses
- A 32-bit ALU
- A 64-bit barrel shifter (32-bit output)
- Forty-eight 32-bit registers
- Thirty-two 32-bit constants
- A 10 ms nonprogrammable interval timer
- Two register file pointer registers
- Hardware to accomplish parallel program counter and register maintenance
- Hardware support for multiply

The chip is pipelined; each microinstruction requires 500 ns to execute, but microinstructions are retired every 250 ns (see Figure 1-4 in Chapter 1).

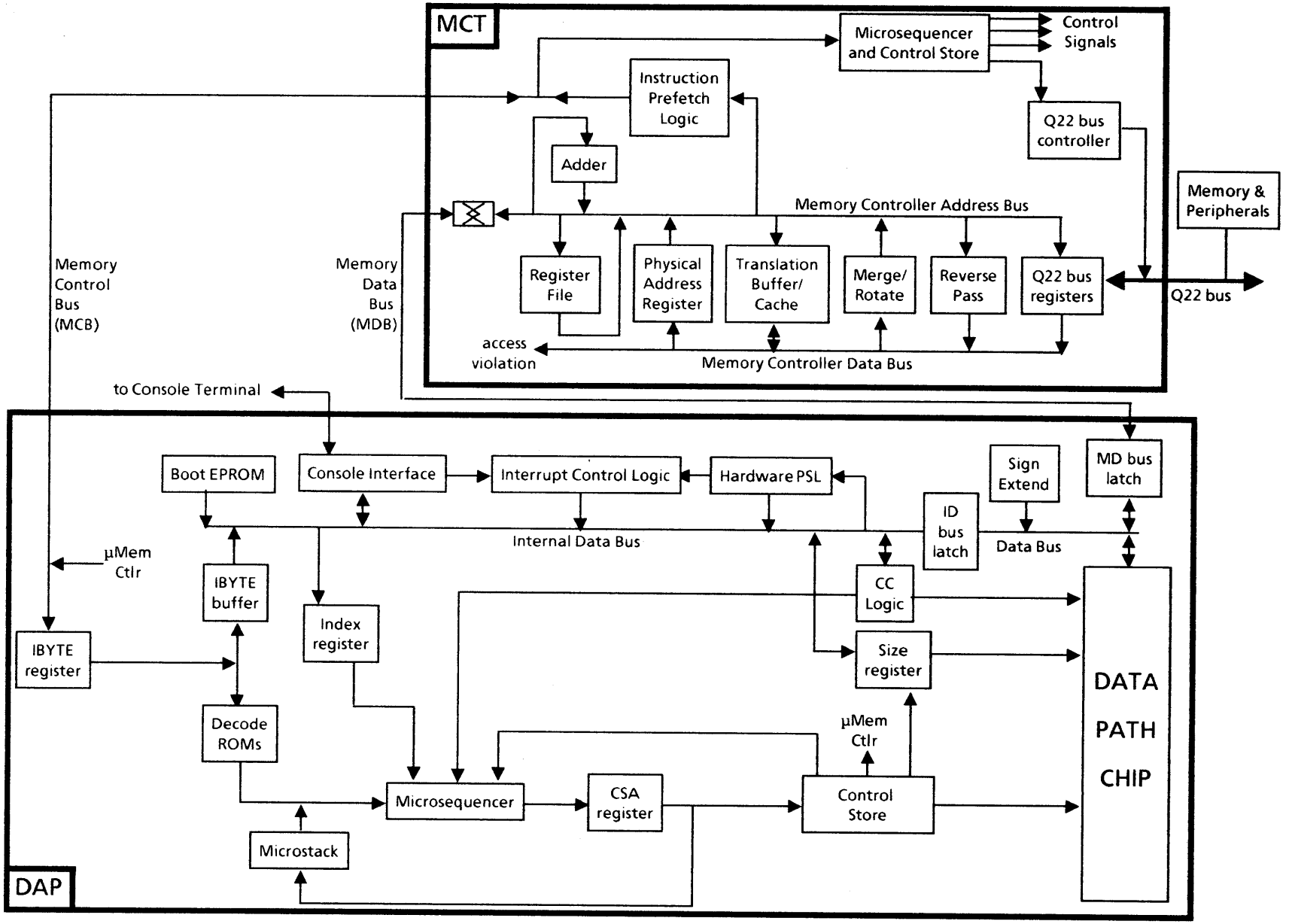


Figure 4-1. CPU Block Diagram

Control Store

The data path microinstruction is 40 bits wide, implemented in five 8K by 8 PROMs. This provides 8K microinstructions, which constitute the microprogram. In addition to implementing the instructions defined in Chapter 1, the microprogram includes the console microcode, and a microdiagnostic called Microverify.

Microsequencer

The microsequencer controls the execution flow of the microcode in the CPU. It decodes a portion of the microinstruction and performs condition testing and branching to generate the microaddress of the next microinstruction to be executed. Thus, it generates a 13-bit microaddress every 250 ns. The functions provided by the microsequencer are described further in the next chapter.

Instruction Decode Logic

The instruction decode logic decodes macroinstruction stream bytes. The outputs of this logic provide portions of the next microaddress at the start of macroinstruction execution.

Internal Data Bus

The internal data bus is an 8-bit-wide bus completely contained within the data path module. This bus is the interface between the main data path elements in the data path chip, and control and status information available in the remainder of the machine. The internal data bus is also used during instruction decode to pass operand specifier information to the data path chip.

Boot EPROM

The boot EPROM is 8K by 8-bits-wide, and stores the VAX macrocode necessary to bootstrap the system. The boot EPROM is accessible only to the microcode.

To allow for future expansion, the data path module is designed to accept a 16K by 8-bit-wide EPROM as well.

Console Interface

The data path module contains the hardware and the microcode to provide the interface to a single console terminal, implementing a console protocol which is a subset of the VAX protocol. The hardware is a standard EIA RS232/423 line interface. The external connection to this interface is a 10-pin cable mounted on the data path board.

A UART is connected through a buffer to the internal data bus on the data path module and can be read or written directly by the microcode. The console terminal registers always communicate with the data path UART.

The baud rate is selectable from a switch pack on the DAP module, or from the rotary switch on the CPU patch panel, and can be set for 300, 1200, 9600, or 19,200 baud. Both transmitter and receiver always operate at the same speed. The microcode reads the switch pack and the rotary switch on power up and programs the UART for the selected baud rate.

Memory Controller

The memory controller module (M7136) is the interface between the main data path and micromachine, and the Q22 I/O and memory subsystem.

Each microcycle on the memory controller module is 125 ns while each microcycle on the data path module is 250 ns.

The memory controller is an asynchronous subsystem that provides the following services to the data path micromachine.

- It controls and maintains a translation buffer and a data and instruction cache to reduce the number of memory accesses and increase the effective speed of those accesses.
- The memory controller implements functions that allow Q22 bus memory to be accessed as byte, word, or longword without regard to data alignment; I/O devices can also be accessed for byte and aligned-word data transfers.
- The memory controller module generates all system clocks.
- It maintains a 16-byte instruction prefetch buffer to allow data path opcode and operand specifier decodes to occur rapidly and at the same time as memory accesses.

The memory controller is contained on a single quad-height printed circuit board and has connectors to interface to the data path module. The main interface to the Q22 bus is also implemented on the memory controller module. The major memory controller components are:

- An 8 KB direct-mapped cache
- A 512-entry translation buffer
- A micromachine which consists of a micro-sequencer and a 1K by 64 control store
- Q22 bus interface logic

Each of these components is discussed briefly in the following paragraphs.

Cache

The data and instruction cache consists of a 2K by 32-bit-wide data store, and a 2K by 16-bit-wide tag store. The cache is the main element of the mechanism that transparently translates 16-bit data from the Q22 bus into 32-bit data that the data path micromachine needs. The cache also provides increased system throughput. The cache is a direct-mapped, write-through cache.

Translation Buffer

The translation buffer contains the corresponding physical addresses for recently used virtual addresses. It has a total of 512 entries: 256 entries for mapping system space addresses and 256 entries for mapping process space addresses.

Memory Controller Micromachine

The memory controller micromachine consists of a 1K by 64 control store and a simple microsequencer which, in most instances, generates microaddresses directly from the previous microinstruction. The memory controller microsequencer accepts memory request commands issued by the data path micromachine and sequences the memory controller data path to carry out the command. A wide, parallel microinstruction allows virtually all of the memory controller elements to be used every microcycle.

Q22 Bus Interface Logic

The Q22 bus interface logic allows the MicroVAX I processor to communicate with the Q22 bus. Although most of the interface logic is physically located on the

memory controller module, it is discussed as a separate controller in the next section.

Q22 Bus Interface

The Q22 bus interface consists of a state sequencer, a write register and a read register. The sequencer handles the bus sequencing and arbitration, freeing the memory controller from this task.

Interrupts from Q22 bus devices go directly to the data path module. The data path module arbitrates device interrupts according to their interrupt priority levels (IPLs), and raises the machine IPL to 17 (hex) upon honoring an interrupt from a bus device. Software may subsequently lower the IPL to the level of the interrupting device.

MicroVAX I allows only byte and aligned-word accesses to Q22 I/O space. All other attempted accesses result in a machine check. Additionally, aligned-longword writes to memory are atomic; that is, no other bus operations are allowed between the two 16-bit-writes executed on the Q22 bus to accomplish an aligned longword write.

Memory parity errors are reported to the CPU via the Q22 bus.

Data Flow Overview

This section takes the prefetch operation and two macroinstructions as examples, and describes an overview of the data transfers executed by the CPU at the microprogram level to accomplish these operations. This should illustrate how the major functional components, described above, interact.

Prefetch Operation

The memory controller contains a prefetch buffer and associated logic to prefetch bytes from the instruction stream. The prefetch logic maintains its own program counter, which always contains the physical address of the last instruction stream byte that was loaded into the prefetch buffer.

The prefetch buffer can hold up to 16 bytes of instruction stream data; the memory controller always tries to keep the buffer full so that it can rapidly supply the data path with the next instruction stream byte for decoding. The memory controller accomplishes this by incrementing the prefetch program counter to sequentially access memory; this counter can only be incremented within a 512-byte page.

If a program flow change or a page crossing occurs, the data path module sends the memory controller a new virtual address, so that the memory controller can update its program counter and refill the prefetch buffer starting with the new address. The following steps describe the data transfers that take place between the data path and the memory controller in order to do this. Figure 4-2 illustrates the data path elements that correspond to the steps.

1. After a program flow change or a page crossing, the program counter (PC) located on the data path chip contains the virtual address of the next byte in the instruction stream.
2. The virtual address is transferred to the memory controller along the memory data bus (MDB).
3. The translation buffer on the memory controller translates the virtual address to a physical address. (Assume a translation buffer hit; that is,

the translation buffer contains the PTE for the given virtual address.)

4. The physical address is sent to the cache. (It is also copied into a Q22 bus register in case the address is not in the cache and Q22 memory must be accessed to obtain the data.) Assume a cache hit; that is, the cache contains the data for that physical address. The cache contains the byte of data plus the adjacent bytes in the instruction stream because each cache entry is 32 bits wide. Assume the physical address is longword-aligned so that the accessed cache entry contains the desired instruction stream byte plus the next three bytes in the instruction stream.

5. The cache data (in this case, the four instruction stream bytes) are sent through the rotate/merge logic to the prefetch logic a byte at a time (see Figure 4-1). From the prefetch logic, the first instruction stream byte is sent out the memory control bus to the IBYTE register. As the byte is clocked into the IBYTE register, the prefetch logic drives the next instruction byte onto the memory control bus.

6. From the IBYTE register, the instruction byte is sent to the decode logic and the data path microsequencer for decoding. The proper microinstructions are invoked to interpret it. The hardware on the data path chip increments the program counter (PC) by one. This cycle ends with the PC containing the virtual address of the next byte in the instruction stream.

   While the data path was busy with steps 5 and 6, the memory controller continued to fill the prefetch buffer with successive instruction stream bytes. The prefetch PC now contains the physical

address of the last instruction stream byte that the memory controller loaded into the prefetch buffer.

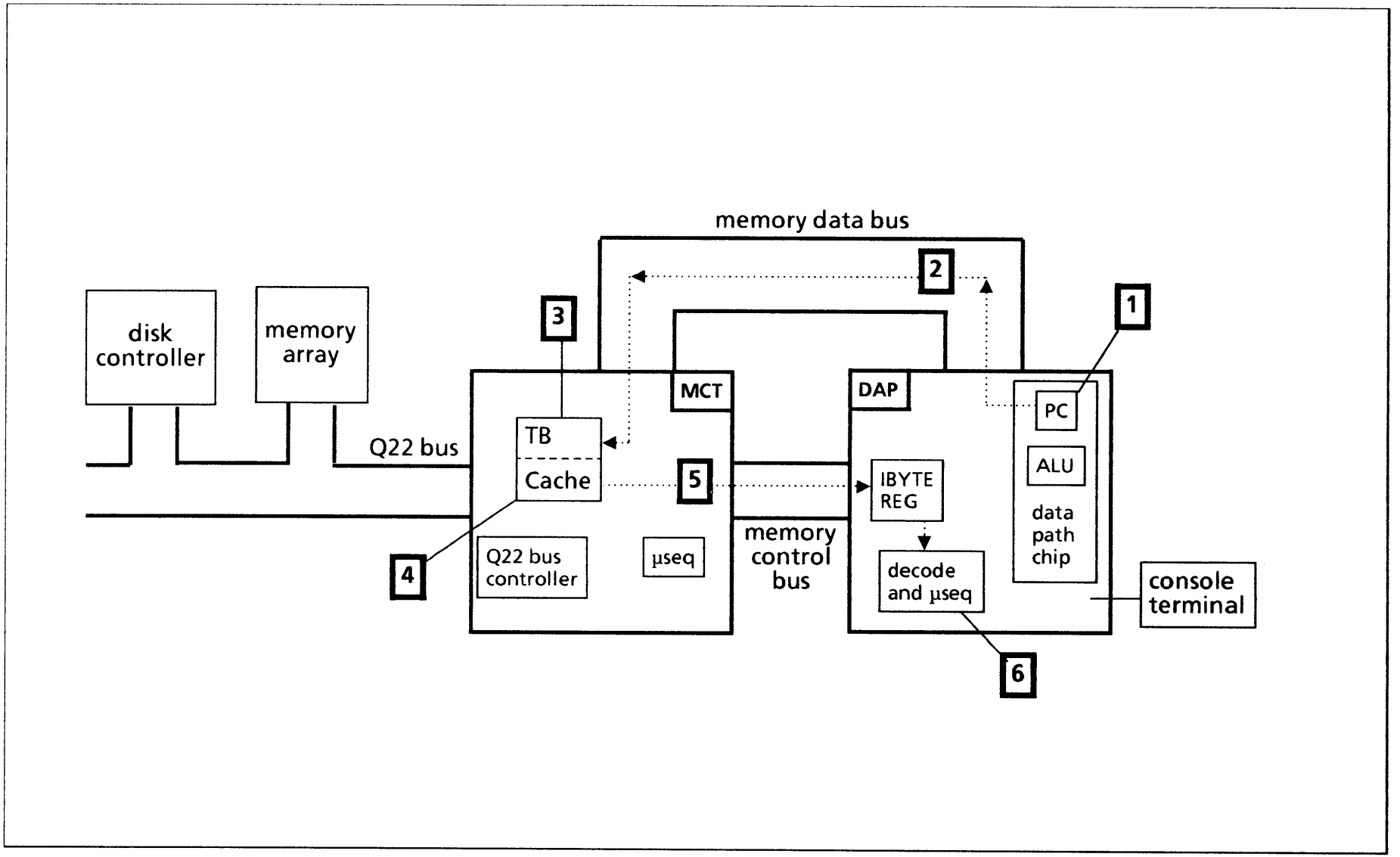
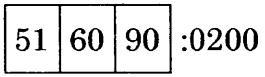


Figure 4-2. Prefetch Operation Data Flow

Move Byte

A move byte (MOVB) macroinstruction copies the byte at the address specified by the first operand into the location specified by the second operand. A sample move byte instruction is: MOVB (R0), R1. This instruction means: locate the byte of data at the address contained in R0 (general processor register 0), and move it to R1 (general processor register 1). At an assigned virtual address in memory, say 0200, the instruction looks like this:



where 90 is the opcode for move byte, 60 is the operand specifier for register deferred mode specifying R0, and 51 is the operand specifier for register mode specifying R1. The following steps describe the data transfers that take place as this instruction is fetched and executed.

Assume that the first byte of the MOVB macroinstruction (the opcode) is already present in the IBYTE register, the next instruction byte (60) is stable on the memory control bus, and the remainder of the macroinstruction is located in the memory controller prefetch buffer. The program counter (PC), located on the data path chip, contains the virtual address of the MOVB opcode (0200). Figure 4-3 illustrates the data path elements used during the corresponding steps.

1. From the IBYTE register, the opcode (90) is sent through the decode logic to the data path microsequencer, causing the first microinstruction in the MOVB microprogram to be executed during the next microcycle.
2. The hardware on the data path chip increments the program counter (PC) by one. The PC now

contains the virtual address of the next byte in the instruction stream; in this case, the virtual address (0201) of the first operand specifier (60).

3. At the start of the next microcycle, the instruction byte sitting on the memory control bus (in this case, 60), is loaded into the IBYTE register and the prefetch logic drives the next instruction byte (51) onto the memory control bus.
4. From the IBYTE register, the operand specifier (60) is sent through the decode logic to the data path microsequencer to define the microprogram flow necessary to process the operand specifier.
5. The hardware on the data path chip increments the PC so that it now contains the virtual address (0202) of the second operand specifier, 51.
6. The contents of R0 are examined. R0 is located on the data path chip and contains some virtual address, say 0100. The 0100 is sent over the memory data bus to the memory controller for address translation.
7. The translation buffer on the memory controller translates the virtual address to a physical address (assuming a translation buffer hit).
8. The physical address is sent to the cache. (It is also copied into a Q22 bus register in case the address is not in the cache and memory must be accessed to obtain the data.) This time, assume a cache miss; that is, the cache does not contain the data at that physical address.
9. The memory controller microsequencer detects the cache miss condition and informs the Q22 bus controller that a data transfer operation is needed.
10. Since the physical address is already conveniently stored in a Q22 bus register, the Q22 bus controller

takes over and initiates two read word data transfers, sending the physical address over the Q22 bus to Q22 bus memory. (Two read word data transfers are necessary to retrieve the 32 bits needed to fill the cache.)

11. The first word of data at the physical address is located in a memory array and sent over the Q22 bus to the Q22 bus read register on the memory controller module.

12. From the Q22 bus read register, the word is sent to the rotate/merge logic (see Figure 4-1) where it is rotated and latched in the two low-order bytes of the merge register. (The rotate/merge logic includes a rotator and a 32-bit-wide merge register). The purpose of the rotation is to position the requested byte of data (the first operand) in the low-order byte of the merge register. From the merge register, the word is sent over the memory data bus to the data path chip on the DAP module. Because this is a move byte instruction, only the low-order byte on the memory data bus (the first operand) is saved in the data path chip. Steps 6 through 12 have all happened as a result of the microinstructions invoked from decoding and interpreting the first operand specifier, 60.

13. After the word containing the desired byte is sent to the DAP module, the second word is sent over the Q22 bus and latched in the Q22 bus read register. As with the first word, it is rotated and latched in the merge register, but in the upper two bytes. The first word is still saved in the lower two bytes. Once both words are latched in the merge register in the proper order, all 32 bits are written into the cache.

14. At this point, the byte to be moved into R1 has been obtained from memory and stored in a temporary register on the data path chip. As a result of the decode executed in step 4, the next instruction byte, 51, which was sitting on the memory control bus, was clocked into the IBYTE register.

15. From the IBYTE register, the second operand specifier (51) is sent through the decode logic to the data path microsequencer to define the microprogram flow necessary to process this operand specifier.

16. Decoding and interpreting the operand specifier 51 causes the first operand, which was stored in a temporary register, to be moved into R1. The move byte macroinstruction is now complete.

17. The hardware on the data path chip increments the PC to point at the next byte in the instruction stream (in this case, the opcode of the next instruction).

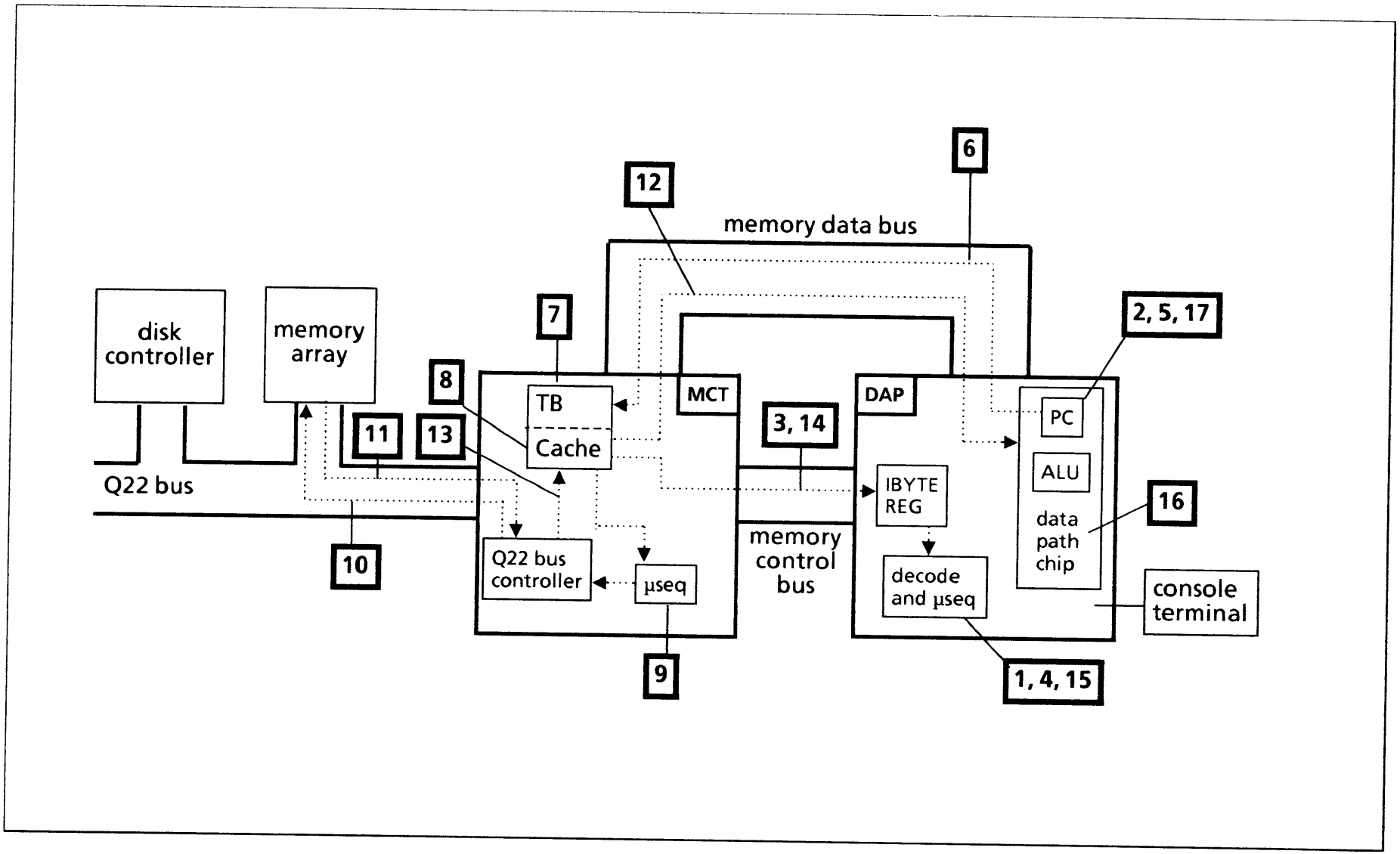


Figure 4-3. MOVB Macroinstruction Data Flow

Subtract One and Branch

The subtract one and branch on greater (SOBGTR) macroinstruction maintains a loop count and a branch address, causing the macroprogram to loop on a set of instructions a desired number of times. The loop count is decremented by 1 each time the instruction is executed and a branch is taken to the starting address of the loop until the loop count is less than or equal to 0. As long as the loop count is greater than 0, the sign-extended branch displacement is added to the PC and the PC is replaced by the result to cause the branch to the first instruction in the loop.

At an assigned virtual address in memory, say 0203, a SOBGTR instruction might look like this:

E0	52	F5	:0203

where F5 is the opcode for SOBGTR, 52 is the operand specifier for register mode specifying R2, and E0 is the displacement value −32. R2 contains the loop count that is decremented each time the loop is executed; assume it contains the number 10. The −32 is sign-extended and added to the PC to compute the branch destination which is the start of the loop.

Assume that the SOBGTR instruction follows right behind the MOVB instruction in the instruction stream, and that the memory controller had successfully prefetched the entire SOBGTR instruction.

Then, as a result of the execution of the MOVB as described in the previous section, the PC contains 0203, the SOBGTR opcode is in the IBYTE register, the next byte of the instruction (52) is stable on the memory control bus, and the last byte (E0) is available in the prefetch buffer.

The following steps describe the data transfers that now take place as the SOBGTR instruction is executed. Figure 4-4 illustrates the data path elements that correspond to the steps.

1. From the IBYTE register, the opcode (F5) is sent through the decode logic to the data path microsequencer causing the first microinstruction in the SOBGTR microprogram to be executed during the next microcycle.
2. The hardware on the data path chip increments the program counter (PC) by one. The PC now contains the virtual address of the next byte in the instruction stream; in this case, the virtual address (0204) of the first operand specifier (52).
3. At the start of the next microcycle, the instruction byte sitting on the memory control bus (in this case, 52), is driven into the IBYTE register and the prefetch logic drives the next instruction byte (E0) onto the memory control bus.
4. From the IBYTE register, the first operand specifier (52) is sent through the decode logic to the data path microsequencer to define the microprogram flow necessary to process the operand specifier.
5. The hardware on the data path chip increments the PC to 0205 (the virtual address of E0, the next instruction byte), and E0 is loaded into the IBYTE register from the memory control bus.
6. The data path chip hardware causes the contents of R2 to be decremented by 1. R2 now contains the loop count 9. Since 9 is greater than 0, the condition codes are cleared; that is, set to zeros. (Z is set to 1 when the loop count equals 0; N is set to 1 when the loop count is less than 0.)

7. From the IBYTE register, E0 is driven over the internal data bus (see Figure 4-1), and sign-extended on the data bus to FFFFFFE0. From the data bus, FFFFFFE0 is sent to the data path chip.

8. The microinstructions invoked from the opcode decode compute the virtual address for the start of the loop as: PC+1+FFFFFFE0=01E6 (the PC contains 0205). This branch destination address is stored in a temporary register on the data path chip.

9. Next, the condition codes are tested. Since condition codes Z and N are clear, the loop count contained in R2 is still greater than 0 (in fact, it is 9). Therefore, the virtual address 01E6 stored in a temporary register is moved into the PC to cause the program to branch back to the beginning of the loop.

10. The virtual address 01E6 (the new virtual program counter) is sent to the memory controller over the memory data bus, and the memory controller prefetch buffer is purged.

11. The translation buffer on the memory controller translates the virtual address to a physical address (assuming a translation buffer hit).

12. The physical address is sent to the cache. (It is also copied into a Q22 bus register in case the address is not in the cache and memory must be accessed to obtain the data.) Assume a cache hit; that is, the cache contains the instruction bytes at the physical address for the start of the loop.

13. The cache data is sent through the rotate/merge logic to the prefetch logic a byte at a time. The first byte is then sent out onto the memory control bus to the IBYTE register.

The memory controller continues to fetch additional instruction stream data to refill the prefetch buffer, while instruction execution proceeds on the DAP module.

14. From the IBYTE register on DAP, the first byte is sent to the decode logic and the data path microsequencer for decoding. The proper microinstructions are invoked to process this instruction stream byte.

15. This flow continues: moving the next byte from the instruction stream into the IBYTE register, decoding and executing it, until the opcode for the SOBGTR instruction, F5, is once again loaded into the IBYTE register. Steps 1 through 14 are repeated nine more times until the loop count, when decremented at step 6, is zero. When this occurs, condition code Z is set.

16. When the condition codes are tested at step 9, Z is found to be set indicating that the loop should be exited. The branch destination address, 01E6, is left in the temporary register and *not* moved into the PC. Instead, the hardware on the data path chip increments the PC to 0206. This is the virtual address of the next byte in the instruction stream; in this case, the opcode of the macroinstruction that follows SOBGTR.

17. If the memory controller successfully prefetched the opcode of the macroinstruction that follows SOBGTR, that opcode is available in the IBYTE register, and the process of decoding and executing continues.

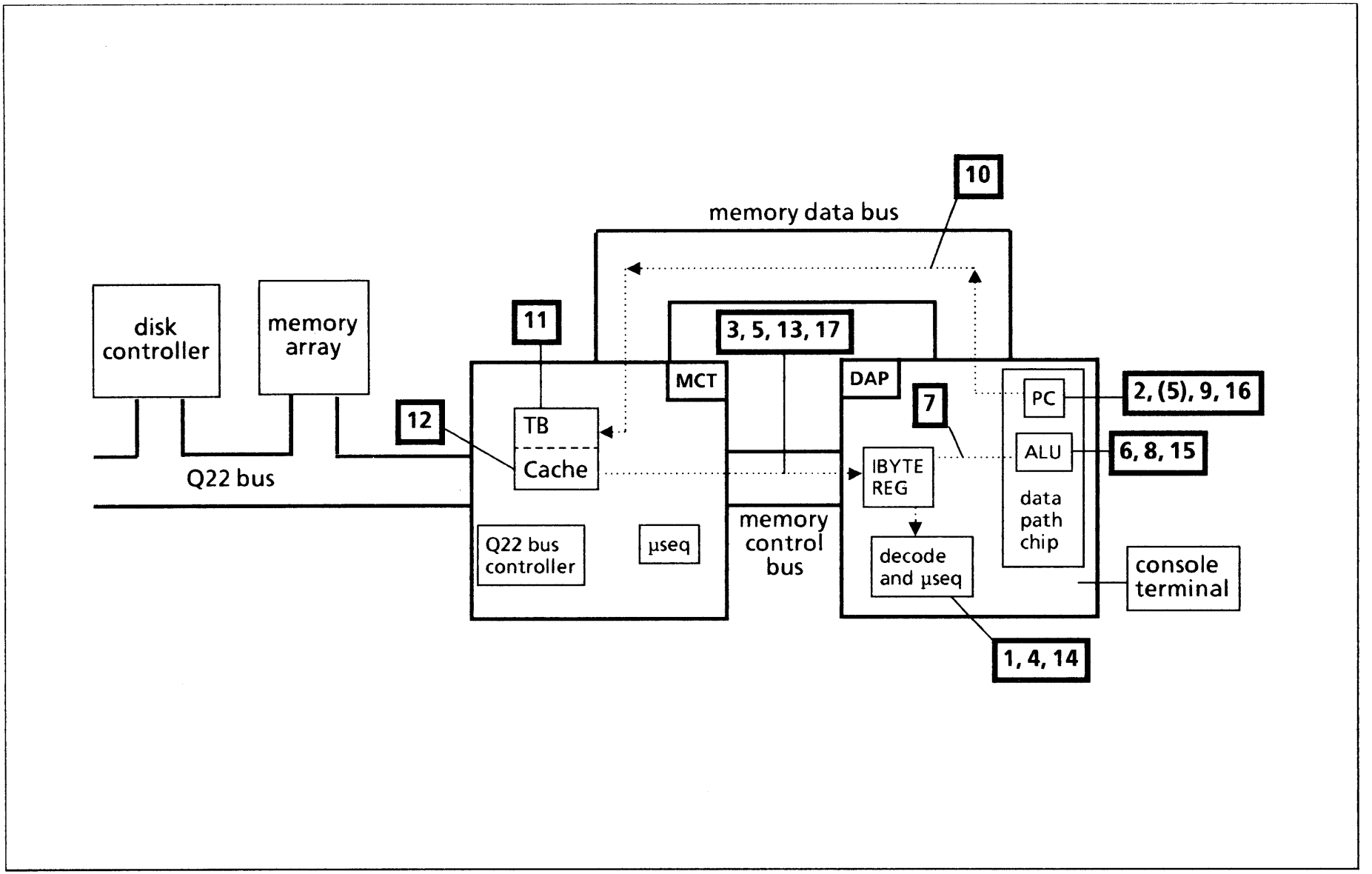


Figure 4-4. SOBGTR Macroinstruction Data Flow

Microcode

The microcode controls all the functions on both modules. Each macroinstruction in the MicroVAX instruction set is implemented by an associated routine of microinstructions. All of the macroinstruction data transfers described above, for example, happen as a result of their associated microinstructions.

The microinstruction routines are stored in two places: the control store on the DAP module, and the control store on the MCT module. The flow from one microinstruction to the next is controlled by two microsequencers, one for each control store: the data path microsequencer, and the memory controller microsequencer. Understanding these two microsequencers and the microinstructions they execute is the key to understanding the MicroVAX I processor.

The data path microsequencer and control store are the master source of control. The microinstructions in the data path control store are invoked to execute the decoded macroinstruction. The data path microsequencer controls the microinstruction flow.

The memory controller microsequencer acts as a slave receiving commands from the data path microsequencer, performing the desired function, and delivering data or status back to the data path micromachine. The control store for the MCT microsequencer contains the microinstructions that enable the MCT microsequencer to perform the desired memory control function.

The Q22 bus controller (located mostly on the MCT module) contains sequencing logic that enables it to accept commands for data from the MCT microsequencer, and handle the bus sequencing and arbitration to

get the requested data. The Q22 bus controller returns data and status back to the MCT microsequencer.

The remainder of this manual describes these three micromachines, and the hardware that implements and surrounds them, in detail. Chapter 5 describes the data path microcode and Chapter 6 describes the data path hardware. Similarly, Chapter 7 describes the memory controller microcode and Chapter 8 describes the memory controller hardware. Finally, Chapter 9 describes the Q22 bus controller.

Chapter 5
Data Path Microcode

All of the functions that occur in the KD32-AA or KD32-AB processor happen as the result of microinstructions. This chapter describes the microinstructions that control the data path module.

Microinstruction Format

The data path microinstruction is forty bits wide. The bits are divided into four fields that accomplish different functions. These fields are parity, condition code/data type, data path control, and next address control. Memory controller functions are encoded within the data path control field.

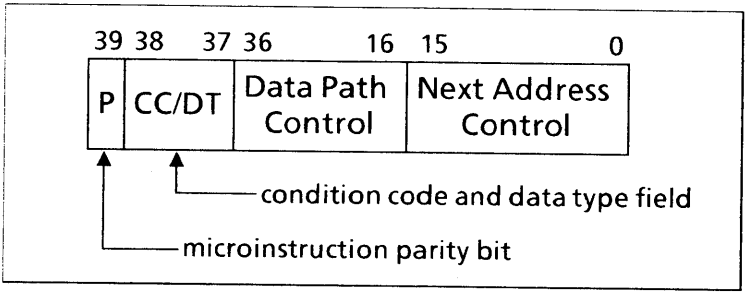


Figure 5-1. DAP Microinstruction Format

The following sections describe each of these fields in more detail.

Parity Field

The highest order bit (bit 39) of the microinstruction contains the parity bit. It is used to detect single bit errors across the entire microinstruction. Odd parity is used; that is, the parity bit is a one when the sum of the one bits in the remainder of the microinstruction is even.

Condition Code/Data Type Field

This field has two functions. It controls the setting of the condition codes, and it determines the data type to be used for the current operation. (Data type is also referred to as size.)

For Memory Request and I-stream Request microinstructions, bits <38:37> are interpreted as data type. Decode microinstructions are a special case. For all other microinstructions, bits <38:37> control the setting of the condition codes, and implicitly specify the data type. Table 5-4 in this chapter summarizes the microinstruction types and which way the CC/DT field is interpreted for each.

Table 5-1 below shows the encoding for bits <38:37> when they are interpreted as the condition code field, which is the case for most microinstructions.

Table 5-1. Condition Code Field Encoding

<38:37>	CC Function
0	condition codes are unaffected; data type is long
1	set ALU condition codes; data type is long
2	set ALU and PSL condition codes; data type is long
3	set ALU and PSL condition codes; data type is determined by the size register

For Memory Request and I-stream Request microinstructions, bits <38:37> are interpreted as the data type field and encoded as shown in Table 5-2.

Table 5-2. Data Type Field Encoding

<38:37>	Data Type
0	byte
1	word
2	determined by size register
3	longword

The Decode microinstruction decodes macroinstruction opcodes and macroinstruction operand specifiers. When a Decode microinstruction decodes an opcode, bits <38:37> of the Decode microinstruction are ignored. When a Decode microinstruction decodes an operand specifier, bits <38:37> control the loading of the size register and are encoded as shown in Table 5-3.

Table 5-3. Operand Specifier Decodes: CC/DT Field Encoding

<38:37>	Value Loaded in Size Register	Data Type
0	0	byte
1	1	word
2	size register unaffected	existing data type
3	3	longword

Data Path Control Field

The data path control field is the 21 bits that are sent to the data path chip to control its functions. The data path control field for all microinstructions (except Memory Requests and I-stream Requests) is divided into six function fields, as shown in Figure 5-2. (The encoding of the data path control field for Memory Request and I-stream Request microinstructions is shown in Figure 5-4 later in this chapter.)

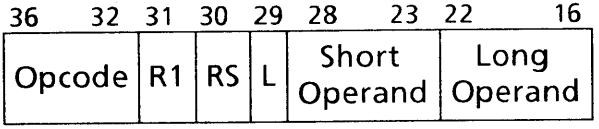


Figure 5-2. Data Path Control Field

The opcode field, bits <36:32>, defines the microinstruction type. Table 5-4 shows the available opcodes and their functions.

The result register bit, bit <31>, selects the destination result register for the current ALU operation. The

result of any ALU operation is stored in one of two result registers on the data path chip: result register 0 or result register 1. If bit <31> is clear, the result of the current ALU operation is stored in result register 0. If bit <31> is set, the result of the current ALU operation is stored in result register 1.

The register save bit, bit <30>, determines whether or not a register save operation occurs. (This is true unless the microinstruction is a NOP, Decode, Restore, Clear Save Stack, Multiply Step, I-stream Request, or Memory Request; bit <30> is ignored in these microinstructions.)

The data path chip contains a register save stack, which is a pushdown stack capable of holding seven 36-bit items. When bit <30> is set, the contents of the register specified by the short operand, plus the low four bits of the register address, are pushed onto the register save stack. When bit <30> is clear, no register save operation occurs. (For a register save operation to occur, the short operand cannot be a literal and must specify a register address from 0 through F.)

The literal bit, bit <29>, determines the interpretation of the short operand field. If bit <29> is clear, the short operand field specifies a register. If bit <29> is set, the short operand field is literal data. If the short operand is literal data, the data path chip zero-extends the data to 32 bits for use inside the chip.

The short operand field, bits <28:23>, is the first operand of the data path control field. The short operand field can specify register addresses 0 to 3F and may designate a register directly or indirectly. If the literal bit is set, the short operand field is a 6-bit literal value. The short operand field is encoded uniquely for the Decode microinstruction; the encoding is described later in this chapter.

The long operand field, bits <22:16>, is the second operand of the data path control field. It can specify any register address that the short operand can, and in addition, specify addresses 40 to 7F. Thus, the long operand can designate any internal or external register, or any constant (the constants are implemented as ROM on the data path chip).

The encodings for long and short operands are described further in the section titled "Operand Field Encoding" in this chapter.

Table 5-4. Opcode Assignments

Opcode	CC/DT	Function	Interpretation
0	CC	NOP	no operation
1	CC	AND	dest ← short operand AND long operand
2	CC	OR	dest ← short operand OR long operand
3	CC	XOR	dest ← short operand XOR long operand
4	CC	Mask	dest ← (NOT short operand) AND long operand
5	CC	Reverse Mask	dest ← short operand AND (NOT long operand)
6	CC	NOT	dest ← NOT short operand
7	CC	Reverse NOT	dest ← NOT long operand
8	CC	Add	dest ← short operand + long operand
9	CC	Add + 1	dest ← short operand + long operand + 1
A	CC	Addwc	dest ← short operand + long operand + ALU carry
B	CC	Sub	dest ← short operand – long operand
C	CC	Sub – 1	dest ← short operand – long operand – 1
D	CC	Reverse Sub	dest ← long operand – short operand
E	CC	Reverse Sub – 1	dest ← short operand – long operand – 1
F	CC	Compare	CCs ← short operand – long operand (The result registers are unaffected.)
10	CC	Shift Left	dest ← long operand shift left logical by shift count register
11	CC	Shift Right	dest ← long operand shift right logical by shift count register
12	CC	Shift Right Arithmetic	dest ← long operand shift right arithmetic by shift count register
13	CC	Double Shift	dest ← 32 bits from 64-bit quantity formed from short operand and long operand, shift right by shift count register

Table 5-4. Continued

Opcode	CC/DT	Function	Interpretation
14	CC	Shift Left Literal	dest ← long operand shift left logical by literal
15	CC	Shift Right Literal	dest ← long operand shift right logical by literal
16	CC	Shift Right Arith. Lit.	dest ← long operand shift right arithmetic by literal
17		reserved	undefined
18	See Note	Decode	Decode generates a new microaddress for the current macroinstruction opcode or operand specifier.
19	CC	Restore	The top entry in the register save stack is moved to the register whose address is stored in the entry.
1A	CC	Clear Save Stack	All entries in the register save stack are marked as being empty.
1B	CC	Multiply Step	Multiply Step controls the "shift and add" algorithm for multiplication.
1C	DT	I-stream Request	A memory request in which the long operand specifies IB.BYTE, IB.WORD, IB.LONG, or IB.SIZE.
1D	CC	Move	Move from long operand to short operand.
1E	DT	Memory Request	A memory request in which the long operand is the memory address of the desired data.
1F	CC	Moveout	Move from short operand to an external destination specified by the long operand.

Note: The CC/DT field is ignored for opcode Decodes, and used to control the loading of the size register for operand specifier Decodes. See Table 5-3 for the CC/DT field encoding.

Next Address Control Field

The next address control field determines the next microinstruction address. As each microinstruction is retrieved from control store, the microsequencer decodes this field and generates the next microaddress. The next microaddress is used to access control store to retrieve the next microinstruction.

The control store address space is divided into 32 pages; each page is 256 words. The next address control field of some microinstructions specifies an address within the current page. Other next address control fields specify a full 13-bit address. The next address control field always has one of the nine formats shown in Figure 5-3.

A next address control field must be specified for every microinstruction. In the microcode listing, there are microinstructions with no explicit next address control field given. For these instances, an unconditional jump to the current microaddress plus 1 is supplied by default.

Six of the nine formats shown in Figure 5-3 have jump control fields, either JC<3:0> or JC<1:0>, corresponding to next address control field bits <11:8> and <9:8>, respectively. The return format has a split jump control field, consisting of JC<2> (bit 12), and JC<1:0> (bits 9:8).

The jump control field is used to specify conditions which are being tested by the microcode. If the condition is not met, the next microaddress is the current microaddress plus 1. If the condition is met, the next address is within the current page at the offset specified by the jump address field, JA<7:0>. This is true unless the next address control field format is trap or branch to subroutine. If the condition specified by the

jump control field is met for a trap or a branch to subroutine, the next address is within page zero at the offset specified by JA<7:0>.

The jump control field encoding is shown in Table 5-5. A jump control field value of 0 means there are no jump conditions to be tested and the next microaddress is conditioned only by the output of the OR MUX.

Five of the nine formats shown in Figure 5-3 use the OR field, either OR<2:0> or OR<1:0>, corresponding to next address control field bits <12:10> and <11:10>, respectively. The OR bits control the OR MUX, one of the hardware components of the data path microsequencer. The OR MUX is discussed in Chapter 6, but some information about it is called for here.

Conceptually, there are eight sets of inputs to the OR MUX, and each set contains four signals which are used to conditionally affect the low-order four bits of the microaddress. Table 5-6 shows the eight sets of inputs with four signals in each set. The value of the OR field selects one set of four signals, thereby determining the output of the OR MUX.

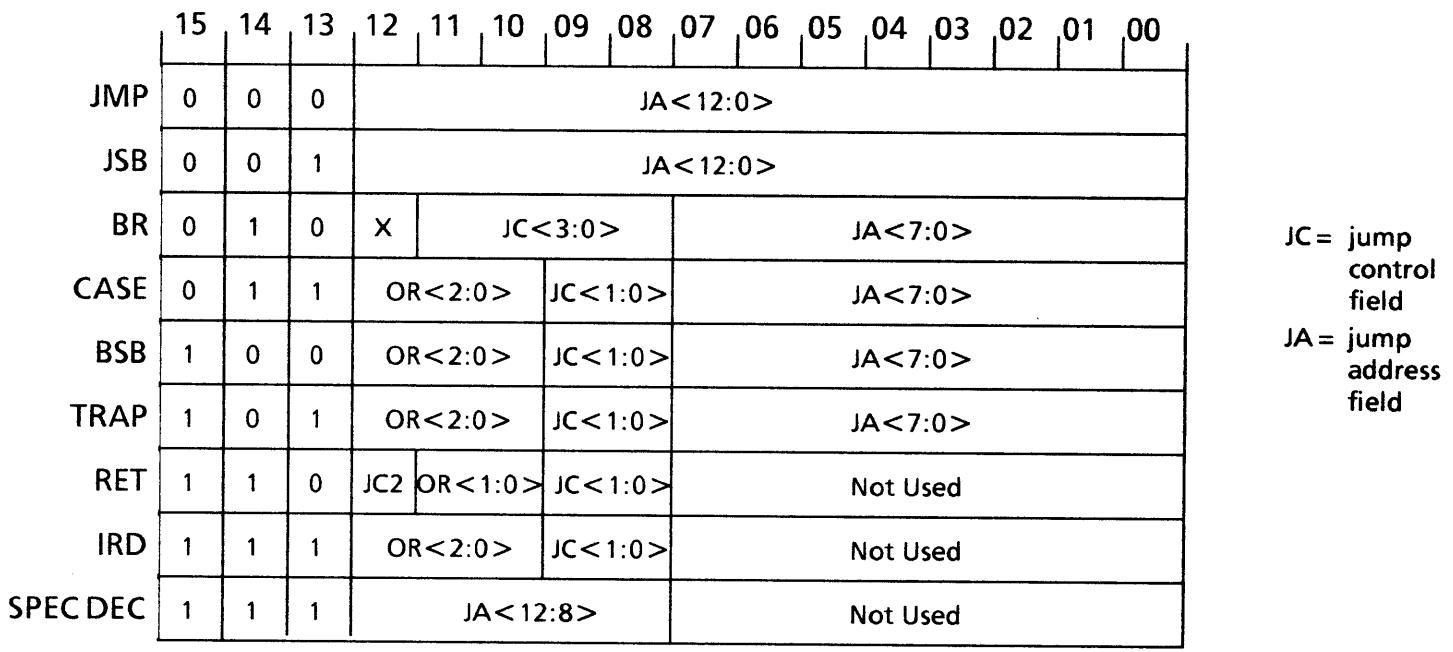


Figure 5-3. Next Address Control Field Formats

Table 5-5. Jump Control Field

JC <3:0>	Condition Tested	JC <3:0>	Condition Tested
0	Use OR MUX	8	Console Halt
1	OR MUX = 0	9	Interrupt
2	OR MUX ≠ 0	A	Stack Register
3	IB OK	B	Register Dest. (not PC)
4	ALU N clear	C	ALU V clear
5	ALU Z clear	D	ALU C clear
6	ALU N set	E	ALU V set
7	ALU Z set	F	ALU C set

Table 5-6. OR<2:0>

2:0	ORMUX3	ORMUX2	ORMUX1	ORMUX0
0	0	0	0	0
1	0	0	0	IB invalid
2	0	0	1	0
3	MEM ERR	Page Crossing	TB Miss	Modify Refuse
4	0	0	BR False	IB invalid
5	Overflow & Chk or trap request	Interrupt Request	T Bit or Console halt	IB invalid
6	INDEX<3>	INDEX<2>	INDEX<1>	INDEX<0>
7	0	0	SIZE<1>	SIZE<0>

For example, one of the inputs to the OR MUX has these four signals on it: MEM ERR, page crossing, TB miss, and modify refuse (see the fourth row in Table 5-6). Suppose that just after a memory request completes, the MEM ERR signal is set (a one), and the other three are clear (zeros). This input to the OR MUX, then, is 1000 binary. Now, if the value of the OR field in the microinstruction at this same point in time is 3, this input is selected. Therefore, the output from the OR MUX is 1000 binary and this output is ORed with the low four bits of the next microaddress.

Given this general information about the next address control field, each of the nine formats is described in more detail in the following paragraphs.

Jump and Jump to Subroutine

The jump (JMP) format causes an unconditional jump to any address in control store. The 13-bit control store address is supplied by next address control field bits, labeled JA <12:0>.

The jump to subroutine (JSB) format causes an unconditional jump to any address in control store. The 13-bit control store address is supplied by next address control field bits, JA <12:0>. A JSB also saves the current microaddress plus 1 on the microstack.

Branch

The branch (BR) format causes a jump to a destination within the current page, if the jump condition specified is met. The jump condition is specified in next address control bits <11:8>. These bits are labeled JC<3:0> in Figure 5-3.

There is no OR field in the branch format, so when the next address control field format is branch, the OR

MUX input is forced to zero. As a result, if the jump control field of a branch format contains the value 0 indicating "use OR MUX" (see Table 5-5), the output of the OR MUX (0000 binary) is ORed with the low four bits of JA<7:0>.

If the jump control field of a branch format contains the value 1, the condition OR MUX=0 is tested. Because the OR MUX input is forced to zero for the branch format, this condition is true, and again, the output of the OR MUX (0000 binary) is ORed with the low four bits of JA<7:0>.

If the jump control field of a branch format contains the value 2, the condition OR MUX≠0 is tested. Because the OR MUX input is forced to zero, this condition is false, and the branch is not taken.

Thus, if the jump condition is met, the next micro-address is constructed from the current page, and next address control bits <7:0>. Bits <12:8> of the address come from the current page, and bits <7:0> specify the address within that page.

If the jump condition is not met, the next microaddress is the current microaddress plus 1.

Case

The case format causes a branch to a destination within the current page if the condition specified in JC <1:0> is met. The value of the field OR<2:0> determines the output of the OR MUX.

If the jump condition is met, the next microaddress is JA<7:4> and the logical sum of the OR MUX output and JA<3:0>. The current page is specified in bits <12:8>.

If the jump condition is not met, the next microaddress is the current microaddress plus 1.

Branch to Subroutine

The branch to subroutine format (BSB) causes a branch to a microaddress within page zero if the jump condition specified in JC<1:0> is met. The value of OR<2:0> determines the output of the OR MUX. A BSB also saves the current microaddress plus 1 on the microstack.

If the jump condition is met, the next microaddress is <12:8> = 0, JA<7:4>, and the logical sum of the OR MUX output and JA<3:0>.

If the jump condition is not met, the next microaddress is the current microaddress plus 1.

Trap

The trap format causes a trap to a destination within page zero if the jump condition specified in JC<1:0> is met. The value of OR<2:0> determines the output of the OR MUX. A trap also saves the current microaddress on the microstack.

If the jump condition is met, the next microaddress is <12:8> = 0, JA<7:4>, and the logical sum of the OR MUX output and JA<3:0>.

If the jump condition is not met, the next microaddress is the current microaddress plus 1.

Return

The return format causes a return to the microaddress at the top of the microstack if the jump condition specified in JC<2:0> is met. The JC field is split for the return format: JC<1:0> correspond to next address control field bits <9:8>, and JC<2> corresponds to bit <12>. The value of OR<1:0> deter-

mines the output of the OR MUX (OR<2> is defined as zero for the return format).

If the jump condition is met, the next microaddress is the logical sum of the top entry in the microstack and the OR MUX output.

If the jump condition is not met, the next microaddress is the current microaddress plus 1.

Instruction Read and Decode (IRD)

The Decode microinstruction (opcode 18 in Table 5-4) is used to decode macroinstruction opcodes and macroinstruction operand specifiers. A macroinstruction opcode decode is also called IRD: instruction read and decode. When the Decode microinstruction is used for IRD, its next address control field has the IRD format shown in Figure 5-3. A jump condition is specified in JC<1:0> and the value of OR<2:0> determines the output of the OR MUX.

If the jump condition is met, the next microaddress is <12:4>=0 and the OR MUX output as bits <3:0>. The current microaddress is saved on the microstack.

If the jump condition is not met, the next microaddress is <12>=0 and decode ROM <11:0>. The address of the current microinstruction plus 1 is pushed on the microstack.

The decode ROMs (shown in Chapter 4, Figure 4-1) are used to select the proper microcode routine to process the macroinstruction specified by the IBYTE register. There are two decode ROMs; one for single byte opcodes, and one for two-byte opcodes. Bits <24:23> of the Decode microinstruction select one of these two ROMs, and the content of the IBYTE register is used as a direct address into the selected ROM. The output from the decode ROM is twelve bits of microaddress.

Chapter 6 contains more information about the decode ROMS.

Operand Specifier Decode

When the Decode microinstruction is used to decode a macroinstruction operand specifier, its next address control field has the SPEC DEC (specifier decode) format shown in Figure 5-3. Operand specifier decode microinstructions have three possible sources for the next microaddress.

If the content of the IBYTE register is valid and the operand is not contained in a general register, or the register is PC, the next microaddress is JA<12:8> and decode ROM <7:0>. When the IBYTE register contains a macroinstruction operand specifier, the decode ROMs supply eight bits of microaddress and four bits of control information. Again, the contents of the IBYTE register and bits <24:23> from the Decode microinstruction are used as a direct address into the decode ROMs. The address of the current microinstruction plus 1 is pushed on the microstack.

If the content of the IBYTE register is valid and the operand *is* contained in a general register other than the PC, the decode ROM generates a control signal that causes the next microaddress to be the address of the current microinstruction plus 1.

If the content of the IBYTE register is not valid, the next microaddress is JA<12:4>=0 and OR MUX <3:0>. The OR MUX encoding is defined as 1 for this condition, so OR MUX <3:0> is 0, 0, 0, IB invalid, and IB invalid=1 (see Table 5-6). Thus, a trap to microaddress 0001 (hex) occurs. The address of the current microinstruction is saved on the microstack.

Data Path Microinstructions

The microinstructions listed in Table 5-4 are grouped by function and discussed in more detail in the following paragraphs.

ALU Microinstructions

Data path microinstructions involving the ALU are those with opcodes 0 through F. The ALU is located on the data path chip. The result of an ALU operation is written into one of two registers on the data path chip: RESULT0 or RESULT1. Each destination ("dest") listed in Table 5-4 is one of these two result registers. The ALU microinstructions also set the N, Z, V, and C condition codes.

When a NOP microinstruction is executed (opcode 0), no operations occur in the data path chip; bits <31:16> of the microinstruction are ignored.

Shift Microinstructions

Shift microinstructions are those with opcodes 10 through 16. For these microinstructions, the shift count can come from either the shift count register which is located on the data path chip, or from a literal in the short operand field. The range of the shift count is limited to 0 through 31. Different opcodes are used to select the type of shift and the source of the shift count. The result of the shift is always placed in the RESULT2 register, also located on the data path chip. The shift microinstructions set the Z condition code when bit <0> of the RESULT2 register is a 1.

A Double Shift microinstruction (opcode 13) concatenates the short operand and the long operand, and selects 32 bits from this 64-bit quantity. The long

operand specifies the lower-order longword. The shift count comes from the shift count register. A rotate operation is obtained by making the long and short operands the same.

Move Microinstructions

The two move microinstructions are Move, opcode 1D, and Moveout, opcode 1F.

Move transfers the contents of the location specified by the long operand to the location specified by the short operand. The short operand cannot be a literal. The data transfer takes place within the data path chip.

Moveout transfers the contents of the location specified by the short operand to the external data pins of the data path chip. The external destination is specified by the long operand. The range of the destination address must be 60 to 7F.

Other Microinstructions

The following microinstructions don't fit in any of the categories listed above.

17	Reserved
18	Decode
19	Restore
1A	Clear Save Stack
1B	Multiply Step
1C	I-stream Request
1E	Memory Request

Reserved is simply an unassigned opcode. Clear Save Stack causes all of the entries in the register save stack to be marked as empty. The register save bit (bit <30> in the data path microinstruction) is ignored.

The remaining microinstructions are described in the following paragraphs.

Decode

The Decode microinstruction selects the routine of microinstructions to be executed to process the macroinstruction opcode or operand specifier in the IBYTE register. For the Decode microinstruction, the low five bits of the short operand are redefined as shown in Table 5-7.

Table 5-7. Decode Microinstruction Short Operand

Bit	Function	Explanation
27	Enable V bit and check, or trap request	This bit enables the OR MUX input signal "overflow and check, or trap request."
26	Pointer Register	This bit selects which of the two pointer registers is loaded from the data bus.
25	Register Save Stack Initialize	When set, this bit resets the register save stack to empty and pushes the old PC onto the stack.
24	IFUNC 1	This bit is used by the decode ROMs to distinguish an opcode decode from an operand specifier decode.
23	IFUNC 0	This bit is used with bit 24 to define the type of decode selected.

Redefining these bits in this manner enables the following functions to be performed during a Decode:

- If the Decode is for an operand specifier, and the operand specifier is a short literal, bits <5:0> of

the byte in the IBYTE register are extracted from the data bus (see Figure 4-1) and written into one of two 6-bit pointer registers located on the data path chip: pointer 1 or pointer 2.

If the Decode is for an operand specifier, and the operand specifier is not a short literal, bits <3:0> from the IBYTE register are written into one of the two pointer registers.

The pointer registers are used to hold register numbers and literals. The bits from the IBYTE register are written into pointer 1 if bit <26> of the microinstruction is a zero, and into pointer 2 if bit <26> is a one.

- If bit <25> of the Decode microinstruction is a one, the register save stack is cleared, and the unincremented content of the PC is pushed on the register save stack.
- Bits <24:23> of the Decode microinstruction form a two-bit control field which is part of the input to the decode ROMs. (The rest of the input is the macroinstruction byte from the IBYTE register.) Bits <24:23> are encoded as follows:

24	23	Selected Decode
0	0	operand specifier decode type 1
0	1	operand specifier decode type 2
1	0	IRD for single byte opcodes
1	1	IRD on second byte of two byte opcode

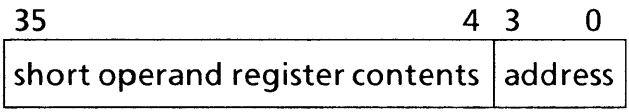
Operand specifier decode type 1 and type 2 refer to different ways the short operands are handled. Basically, type 1 indicates that an integer macroinstruction operand specifier is to be decoded. Type 2 indicates that a floating-point macroinstruction operand specifier is to be decoded.

- The macroprogram counter (PC) on the data path chip is incremented by one.
- If the Decode is an IRD, microinstruction bits <31:28> are ignored, and bits <22:16> always specify R0 so that nothing is driven onto the internal data bus. (For operand specifier decodes, bits <22:16> specify IB.BYTE so that the contents of the IBYTE register are driven onto the internal data bus.)

The next address control field of the Decode microinstruction then generates the next microaddress, which is the address of the appropriate microinstruction routine for executing the current macroinstruction byte.

Restore

The register save stack, located on the data path chip, is a pushdown stack capable of holding seven 36-bit items. When bit <30> of a microinstruction is set, both the contents of the register specified by the short operand and the low four bits of the register address are pushed on the stack in this format:



The following microinstructions are exceptions to this in that bit <30> (the RS bit) is ignored: NOP, Decode, Restore, Clear Save Stack, Multiply Step, I-stream Request, and Memory Request.

The Restore microinstruction pops the top entry off the register save stack; that is, the top entry in the register save stack is moved to the register whose address is stored in bits <3:0> of the stack. If the register save stack is empty, a Restore microinstruction is effectively a NOP.

Multiply Step

The Multiply Step microinstruction performs multiplication using a "shift and add" algorithm. The two ALU result registers, RESULT0 and RESULT1, are combined to form a 64-bit shift register with RESULT1 the lower order longword. The required setup conditions are:

- The multiplier is placed in RESULT1. RESULT0 is cleared, or may contain an initial addend.
- The multiplicand is specified by the long operand (range = 0:95).
- The data type is longword.

Multiply Step is actually called a total of thirty-two times to complete one multiplication of two longwords. The following functions are performed for each Multiply Step microinstruction.

1. If RESULT1<0> = 1, add the multiplicand to RESULT0.
2. Right shift RESULT0 and RESULT1 by one bit, such that RESULT0<0> becomes RESULT1 <31>.
3. If the add in step 1 was executed, set RESULT0 <31> equal to the sign bit (bit <31>) of the multiplicand. If the add was not executed, RESULT0 <31> is unchanged.
4. Set the data path chip condition code bits according to the result of the addition of the multiplicand and RESULT0 in step 1. (If RESULT1 <0> is not equal to one in step 1, the addition does not actually happen and the Multiply Step consists of steps 2 through 4.)

Once these four parts of the Multiply Step microinstruction are executed thirty-two times, the longword multiplication is complete.

Memory Request

The Memory Request microinstruction is an explicit request from the data path to the memory controller to read and write instruction data. The microinstruction supplies a 9-bit function code and 32 bits of data. The 32 bits of data are a virtual address, a physical address, or the actual data to be written.

The 9-bit function code is encoded in bits <31:23> of the Memory Request microinstruction. These nine bits describe the memory function to be performed. The function code is described in more detail in the section titled "Memory Function Encoding" later in this chapter.

The 32 bits of data are contained in a register on the data path chip. The register address is specified by the long operand of the Memory Request microinstruction. The 32 bits of data are driven from the register onto the data bus, and sent to the memory controller over the memory data bus (see Figure 4-1). The 32 bits of data are also saved in a temporary register [TEMP(0)] on the data path chip. If the 32 bits of data are a physical or virtual address, the data located at this address are the data to be read or written.

A Memory Request microinstruction is followed by one intervening cycle, and then the proper microinstruction (either a Move or a Moveout) is executed to move the requested data to or from the data path. The long operand of the appropriate move instruction specifies MEMORY.DATA to indicate that the data to be moved are the 32 bits currently on the memory data bus. If the requested data are not available at this time, the

microsequencer stalls the execution of the microprogram by continuously repeating the Move or Moveout microinstruction until the requested data are available.

The status of the memory function is available at the same time the requested data are available, and remains valid until the next memory function. The memory function status consists of the four signals on the fourth OR MUX input: MEM ERR, Page Crossing, TB Miss and Modify Refuse (see Table 5-6).

I-stream Request

The I-stream Request microinstruction acts as a command from the data path to the memory controller to read bytes, sign-extended words, and longwords from the instruction stream. It can be thought of as a special case of the Memory Request microinstruction where the 9-bit function code can only be an instruction stream read (IB.READ) and the 32 bits of data are the unincremented contents of the PC on the data path chip. The data located at this 32-bit address in the instruction stream are the data to be read.

The long operand of the I-stream Request microinstruction specifies IB.BYTE, IB.WORD, IB.LONG, or IB.SIZE to indicate the amount of data to be read from the instruction stream. The unincremented contents of the PC on the data path chip are driven onto the data bus, and sent to the memory controller over the memory data bus. The PC on the data path chip is then incremented by 1, 2, or 4, depending on the amount of data read from the instruction stream.

The prefetch logic on the memory controller keeps the prefetch buffer filled with instruction stream bytes. An I-stream Request first clears the prefetch buffer, then reads a byte, word, or longword from the cache or memory, sends it to the data path module via the

memory data bus, and refills the prefetch buffer beginning with the next byte in the instruction stream following this byte, word, or longword.

After the I-stream Request microinstruction, one intervening microinstruction is executed. Then a microinstruction, such as Move, is executed to return the byte, word, or longword from the cache to the data path over the memory data bus. If a word is read from the cache or memory, it is returned over the memory data bus and sign-extended on the data bus. If a byte is read from the cache or memory, it is returned over the memory data bus and **not** sign-extended on the data bus. (The data bus is shown in Figure 4-1.)

Operand Field Encoding

The short and long operands of the microinstructions can specify addresses 0 through 3F. In addition, the long operand can specify addresses 40 through 7F. Table 5-8 shows the address space organization. Chapter 6 describes the registers in more detail.

Table 5-8. Register Address Organization

ADDR	+0	+1	+2	+3
00	R0	R1	R2	R3
04	R4	R5	R6	R7
08	R8	R9	R10	R11
0C	R12	R13	R14	PC
10	TEMP(0)	TEMP(1)	TEMP(2)	TEMP(3)
14	TEMP(4)	TEMP(5)	TEMP(6)	TEMP(7)
18	TEMP(8)	TEMP(9)	TEMP(10)	TEMP(11)
1C	TEMP(12)	TEMP(13)	TEMP(14)	TEMP(15)
20	TEMP(16)	TEMP(17)	TEMP(18)	TEMP(19)
24	TEMP(20)	TEMP(21)	TEMP(22)	TEMP(23)
28	TEMP(24)	TEMP(25)	TEMP(26)	TEMP(27)
2C	TEMP(28)	TEMP(29)	TEMP(30)	TEMP(31)
30	RESULT0	RESULT1	RESULT2	Shift Count
34	PTR 1	PTR 2	*Pointer 1	*Pointer 2
38	Timer Control/Status (TMRCSR)	Reserved	Reserved	Reserved
3C	Reserved	Reserved	Reserved	Reserved
40	ROM Constant (hex): 1	ROM Constant (hex): 4	ROM Constant (hex): 4000	ROM Constant (hex): 14
44	ROM Constant (hex): 16	ROM Constant (hex): 0FF	ROM Constant (hex): 0FF00	ROM Constant (hex): 3000
48	ROM Constant (hex): 0FFF	ROM Constant (hex): 0E0	ROM Constant (hex): 0FFFF	ROM Constant (hex): 0FFFFFF
4C	ROM Constant (hex): 8000	ROM Constant (hex): 7FFF	ROM Constant (hex): 2000	ROM Constant (hex): 1FF
50	ROM Constant (hex): 1F0000	ROM Constant (hex): 1E	ROM Constant (hex): 7FF	ROM Constant (hex): 0B020FF00
54	ROM Constant (hex): 80	ROM Constant (hex): 7FFFFF	ROM Constant (hex): 800000	ROM Constant (hex): 7F80
58	ROM Constant (hex): 10	ROM Constant (hex): 0	ROM Constant (hex): 0FFFFF	ROM Constant (hex): 100000
5C	ROM Constant (hex): 7FF0	ROM Constant (hex): 400	ROM Constant (hex): 80000000	ROM Constant (hex): 0FFFFFFFF
60	CON.DATA (UART data)	CON.STATUS (UART status)	CON.MODE (UART mode)	CON.CMD (UART command)
64	Reserved	Reserved	Reserved	Reserved
68	Size register	Index register	PSL.MODE	MISC register, bits <3:0>
6C	PSL.EN (write), REQ.ST (read)	PSL.IPL (write), INT.SRC (read)	PSL.CC	ALU.CC
70	System ID	Option switches	MISC register, bits <7:4>	Boot ROM
74	Reserved	Reserved	Reserved	Reserved
78	IB.BYTE	IB.WORD	IB.SIZE	IB.LONG
7C	MEMORY.DATA	MEMORY.DATA	MEMORY.DATA	MEMORY.DATA

Memory Controller Interface Microcode

The data path has three ways in which it requests data from the memory controller.

1. The memory controller prefetch logic keeps the prefetch buffer filled with instruction stream bytes so there is a valid byte on the memory control bus as often as possible. The data path signals the memory controller when it needs the next instruction stream byte, so the data path implicitly fills the IBYTE register from the memory control bus as a side effect of instruction and operand specifier decodes.

   If the long operand of a microinstruction (other than an I-stream Request) specifies IB.BYTE, which is the unique address of the IBYTE register, the byte is read from the IBYTE register and sign-extended on the data bus. In this manner, the data path also implicitly fills the IBYTE register from the memory control bus.

2. The data path executes Memory Request microinstructions to explicitly read and write instruction data.

3. The data path executes I-stream Request microinstructions to explicitly read bytes, sign-extended words, and longwords from the instruction stream.

The first item is actually a function of the data path hardware and is discussed further in Chapter 6. The Memory Request and I-stream Request microinstructions are described earlier in this chapter. The memory functions that can be encoded within these microinstructions are described in more detail in the following paragraphs.

Memory Function Encoding

The microinstruction format is different for Memory Request and I-stream Request microinstructions. Bits <31:23> of these microinstructions form a 9-bit function code field which is used by the DAP microsequencer to specify a function for the memory controller to perform.

The format of memory request microinstructions is shown in Figure 5-4. The data size is specified separately in the data type field (bits<38:37>) of the microinstruction.

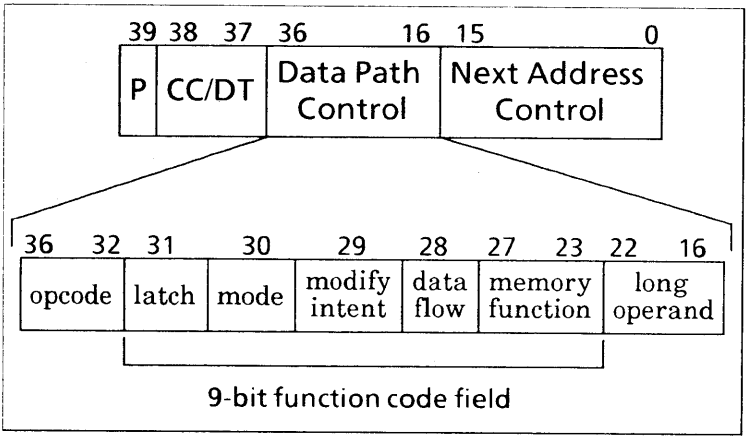


Figure 5-4. Memory Request Format

Bits <31:23> are formatted as follows:

<31> latch function parameters:
When this bit is set, the microsequencer latches the state of the other eight bits of the function code field in a register called the previous function latch. These eight bits, plus one bit provided by the data

path hardware, are referred to as the function parameters.

<30> access mode:
When this bit is set, the access mode is kernel. A zero bit specifies current mode. The current access mode is obtained from the PSL.MODE register on the internal data bus. The access mode is used to check that the protection allows the specified operation to be performed.

<29> modify intent:
When this bit is set, the intended access is write. A zero bit specifies read intent. Modify intent does not signify whether data will be read or written, but rather which access intent is to be checked.

<28> data flow:
When this bit is set, data flows from the data path chip to the memory controller (write). A zero bit specifies that data flows from the memory controller to the data path chip (read).

<27:23> memory function:
This is a 5-bit encoded value that specifies which memory function to execute.

The data path hardware expands the 9-bit function code field from the microinstruction into a 10-bit function code. This 10-bit function code, plus the two bits specifying the data size, are what is actually delivered to the memory controller. The 10 bits have the format:

<9:8> access mode: these two bits indicate the mode for which all accesses are to be checked. When bit <30> of the memory request microinstruction is a 1, these bits

are 00 to indicate kernel mode. When bit <30> of the memory request microinstruction is a 0, these bits are encoded for the current mode, as specified by the PSL.MODE register.

<7> modify intent: this bit indicates the intended access type as specified by the modify intent bit, bit <29>, of the memory request microinstruction.

<6> data flow: this bit indicates the direction in which data will flow on the data bus as specified by the data flow bit, bit <28>, of the memory request microinstruction.

<5:1> memory function code: the 5-bit encoded value from memory request microinstruction bits <27:23> that specifies the memory function to be performed.

<0> second part: This is the function parameter bit supplied by the data path hardware. When clear, this bit specifies that the first part of a memory function is to be executed. When set, it specifies that the second part of a memory function is to be executed. The second part bit is set when a Memory Request with the REPEAT.SECOND function code is executed. It is cleared when a Memory Request microinstruction is executed that has the latch bit (bit <31>) set. The second part bit is used only during memory management error recovery for unaligned reads and writes across page boundaries.

Memory Functions

The following describes each memory function as specified by microinstruction bits <27:23>. The descriptions give the hex values for bits <29:23> (see Figure 5-4) because the state of the data flow and modify intent bits is the only difference between some memory functions.

READ.VECTOR

Memory Request microinstruction bits <29:23> have the hex value 00. This memory function grants bus mastership to the highest level interrupting device and reads its vector from the Q22 bus. The memory controller completes the bus grant cycle and transmits the vector address to the data bus on the data path module.

The contents of the register specified by the long operand are ignored as a physical or virtual address is meaningless for this memory function. The microinstruction data type field must specify byte even though a word of data is returned.

VREAD.RCHECK

This is a virtual read with read check Memory Request microinstruction; bits <29:23> have the hex value 01. This memory function requests that a virtual read operation, with a check for read access, be performed.

The data type field of the microinstruction specifies the amount of data to be read; byte, word, longword, or use size register may be specified.

The register specified by the long operand contains a 32-bit virtual address. If mapping is not enabled, then

no access check is performed and the virtual address is interpreted as a physical address.

VREAD.WCHECK

This is a virtual read with write check Memory Request microinstruction. Bits <29:23> have the hex value 41; that is, the 5-bit memory function code is the same as for VREAD.RCHECK (a value of 01) but the modify intent bit is set. This memory function requests that a virtual read operation, with a check for write access, be performed.

The data type field of the microinstruction specifies the amount of data to be read; byte, word, longword, or use size register may be specified.

The register specified by the long operand contains a 32-bit virtual address. If mapping is not enabled, then no access check is performed and the virtual address is interpreted as a physical address.

If the resultant physical address has bit <29> set, it is a reference to I/O space. Only word and byte references to I/O space are legal, and all word references must be word aligned. If bit <29> is set, the memory controller converts the read to an interlocked read on the Q22 bus (a DATIO—read, modify, write word, or DATIOB—read, modify, write byte).

VWRITE.WCHECK

This is a virtual write with write check Memory Request microinstruction. Bits <29:23> have the hex value 61; that is, the 5-bit memory function code has the value 01 but the modify intent bit and the data flow bit are set. This memory function requests that a virtual write operation, with a check for write access, be performed.

The data type field of the microinstruction specifies the amount of data to be written; byte, word, longword, or use size register may be specified.

The register specified by the long operand contains a 32-bit virtual address. If mapping is not enabled, then no access check is performed and the virtual address is interpreted as a physical address.

If the previous operation was an interlocked read, this function is effectively a write unlock.

VREAD.LOCK

This is a virtual read with write check interlocked Memory Request microinstruction. Bits <29:23> have the value 42; that is, the 5-bit memory function code has the value of 02 but the modify intent bit is set. This memory function requests that a virtual read operation, with a check for read access, be performed.

The data type field specifies byte or word. The register specified by the long operand contains a 32-bit virtual address. If mapping is not enabled, then no access check is performed and the virtual address is interpreted as a physical address.

This microinstruction causes a byte or word to be read from memory with a locked Q22 bus cycle; that is, either a DATIOB or a DATIO data transfer takes place. The memory controller is bus master and will not release the bus until a write is performed. The VREAD.LOCK memory function must be followed by a VWRITE.WCHECK function or by a PWRITE function.

IB.REFILL

This is an instruction stream refill Memory Request microinstruction. Bits <29:23> have the hex value 03. This memory function requests that a new origin in

the instruction stream be selected. The data type field must specify byte.

The register specified by the long operand contains a 32-bit virtual address which is the address of the new origin in the instruction stream. No data are delivered on the move in from memory.

If mapping is enabled, a read access check is performed for the specified mode. As subsequent sequential bytes are read from the instruction stream, no access check is necessary until a page boundary crossing. If mapping is not enabled, then no access check is performed.

PREAD

This is a physical read Memory Request microinstruction. Bits <29:23> have the hex value 04. This memory function requests a physical read operation. The data type field of the microinstruction specifies the amount of data to be read; byte, word, longword, or use size register may be specified. The register specified by the long operand contains a 32-bit physical address that is the address of the data to be read.

PWRITE

This is a physical write Memory Request microinstruction. Bits <29:23> have the hex value 64; that is, the 5-bit memory function code has the value 04 but the modify intent bit and the data flow bit are set. This memory function requests a physical write operation. The data type field of the microinstruction specifies the amount of data to be written; byte, word, longword, or use size register may be specified. The register specified by the long operand contains a 32-bit physical address that is the address of the data to be written.

If the previous function was an interlocked read, this function is effectively a write unlock.

XLATE.RCHECK

This is a translate virtual address with read check Memory Request microinstruction. Bits <29:23> have the hex value 05. This memory function translates virtual addresses to physical addresses to check if certain operations such as pushing onto the stack can be performed without a fault. This memory function insures that the page is accessible and that the appropriate entry is in the translation buffer.

The data type is specified by the data type field of the microinstruction. The register specified by the long operand contains the 32-bit virtual address to be translated.

The data returned to the data path is the physical address of the first byte corresponding to the translated virtual address. Both the address and the address plus the data size − 1 are checked for read access. If mapping is not enabled, then no access check is performed and the virtual address is treated as a physical address.

XLATE.WCHECK

This is a translate virtual address with write check Memory Request microinstruction. Bits <29:23> have the hex value 45; that is, the 5-bit memory function code has the value 05, the modify intent bit is set and the data flow bit is clear. (A zero data flow bit specifies that the data flows from the memory controller to the data path chip—a read.) This memory function translates virtual addresses to physical addresses to check if certain operations such as pushing onto the stack can be performed without a fault. This

memory function insures that the page is accessible and that the appropriate entry is in the translation buffer.

The data type is specified by the data type field of the microinstruction. The register specified by the long operand contains the 32-bit virtual address to be translated.

The data returned to the data path is the physical address of the first byte corresponding to the translated virtual address. Both the address and the address plus the data size − 1 are checked for write access. If mapping is not enabled, then no access check is performed and the virtual address is treated as a physical address.

IB.READ

This is the only memory function allowed for the I-stream Request microinstruction. Bits <29:23> have the hex value 0D. This memory function reads a byte, sign-extended word, or longword from the instruction stream. The instruction stream PC is implicitly incremented so that the next instruction stream read addresses the correct data. A new instruction stream origin is established as with the IB.REFILL function.

The amount of data to be read from the instruction stream is specified as IB.BYTE, IB.WORD, IB.SIZE, or IB.LONG in the long operand. The data type field also specifies byte, word, size, or longword and must match the data type specified in the long operand. The data read from the instruction stream is returned to the data path over the memory data bus. (Sign-extended bytes can also be read via the data path by specifying IB.BYTE as the long operand specifier.)

REPEAT.FIRST

This is a Memory Request microinstruction that repeats a previous memory function. REPEAT.FIRST is only meaningful when preceded by an error condition. It is intended for use in memory management error recovery microcode that fills the translation buffer and handles memory modify refuse. Bits <29:23> have the hex value 06.

When the latch bit, bit <31>, of a Memory Request microinstruction is set, the data path microsequencer latches the current function parameters (except for the second part bit) in a register. REPEAT.FIRST references this register, enabling the previous Memory Request microinstruction to be repeated. The data type field in the REPEAT.FIRST microinstruction is ignored; the data type field from the previous Memory Request microinstruction is used.

When any Memory Request microinstruction is executed, the 32 bits of data it supplies are saved in a register on the data path chip. The long operand of the REPEAT.FIRST microinstruction specifies the address of this register. Thus, the 32 bits of data supplied by the REPEAT.FIRST microinstruction are the same 32 bits supplied by the previous Memory Request microinstruction.

If mapping is not enabled, then no access check is performed and the virtual address is treated as a physical address. Note that the REPEAT.FIRST memory function is only meaningful for virtual functions since the failure of a physical function is never retried.

REPEAT.SECOND

This Memory Request microinstruction acts just like REPEAT.FIRST in that it repeats the previous memory function. In addition, it sets the second part flag. REPEAT.SECOND is only meaningful when preceded by a page crossing error condition. It is intended for use in memory management error recovery microcode for unaligned reads or writes across page boundaries. Bits <29:23> have the hex value 07.

REPEAT.SECOND references the previous function latch, enabling the previous Memory Request microinstruction to be repeated. (The previous function latch is the register used to save the function parameters when the latch bit of a Memory Request microinstruction is set.) The data type field in the REPEAT.SECOND microinstruction is ignored. The REPEAT.SECOND memory function is used specifically for memory operations that read or write across a page boundary.

The desired virtual address in the next page is stored in a result register on the data path chip. It is computed by adding 4 to the 32-bit virtual address supplied by the previous Memory Request microinstruction; that is, 4 is added to the address in the last page.

The long operand of the REPEAT.SECOND microinstruction specifies the address of the result register where the computed virtual address is stored. Therefore, the 32-bit address supplied by the REPEAT.SECOND microinstruction is the desired virtual address in the next page.

If mapping is not enabled, then no access check is performed and the virtual address is treated as a physical address. Note that the REPEAT.SECOND memory function is only meaningful for virtual

functions since the failure of a physical function is never retried.

READ.CACHE

This is a Memory Request microinstruction that reads a cache entry. Bits <29:23> have the hex value 0B. This memory function requests a read operation. The data type field must specify byte, even though a longword of data is returned.

The register specified by the long operand contains a 32-bit physical address that is the address of the data to be read from the cache. The cache RAM contents at that address are returned whether or not a cache hit occurs.

WRITE.CACHE

This is a Memory Request microinstruction that writes a cache entry and its associated tag, and marks the entry valid. Bits <29:23> have the hex value 2C; that is, the 5-bit memory function code has the value 0C but the data flow bit is set. This memory function requests a write operation.

The data type field must specify byte, even though a longword of data is written. The register specified by the long operand contains a 32-bit physical address that is the address of the data to be written to the cache.

WRITEP

This is a Memory Request microinstruction that is used in the memory management microcode to write a page table entry (PTE) with the modify bit set. Bits <29:23> have the hex value 60; that is, the 5-bit memory function code has the value 00 but the modify intent bit and the data flow bit are set. This memory function requests a write operation and writes an

aligned longword to the translated address that was used for the previous memory function.

The data type field must specify longword, and a longword of data is written. The long operand is ignored.

Read MCT Registers

This function represents a group of Memory Request microinstructions that read the memory controller internal registers. The internal register number is supplied as part of the function code; bits <29:23> have the hex values 10 through 17. The data type field must specify byte, even though a longword of data is returned. The contents of the register specified by the long operand are ignored. Table 5-9 shows the values for bits <29:23> and the corresponding functions performed.

Table 5-9. Read MCT Function Codes

<29:23>	Function
10	READ.MAP.ENABLE
11	READ.CACHE.ENABLE
12	READ.ERROR.FLAG
13	READ.IB.ERROR
14	READ.VIRTUAL
15	READ.PHYSICAL
16	READ.ISTREAM.PC
17	READ.ERROR.CODE

WRITE MCT Registers

This function represents a group of Memory Request microinstructions that write data to the memory controller internal registers. The internal register number is supplied as part of the function code; bits

<29:23> have the values 18 through 1F. The data type field must specify byte, even though a longword of data is written. The register specified by the long operand contains the actual data to be written. Table 5-10 shows the values for bits <29:23> and the corresponding functions performed.

Table 5-10. Write MCT Function Codes

<29:23>	Function
18	WRITE.MAP.ENABLE
19	WRITE.CACHE.ENABLE
1A	WRITE.ERROR.FLAG
1B	WRITE.IB.ERROR
1C	WRITE.VIRTUAL
1D	WRITE.PHYSICAL
1E	WRITE.ISTREAM.PC
1F	WRITE.ERROR.CODE

READ.TB

This Memory Request microinstruction reads a translation buffer entry. Bits <29:23> have the hex value 08. The data type field must specify byte, even though a longword of data is returned. The register specified by the long operand contains a 32-bit virtual address. The translation buffer entry specified by the virtual address is read regardless of whether a tag match occurs or whether mapping is enabled or not.

WRITE.TB

This Memory Request microinstruction writes a translation buffer entry and its associated tag, and marks the entry valid. Bits <29:23> have the hex value 29; that is, the 5-bit memory function code has the value 09 but the data flow bit is set. The data type

field must specify byte, even though a longword of data is written. The register specified by the long operand contains a 32-bit virtual address. The translation buffer entry specified by the virtual address is written regardless of whether mapping is enabled or not.

INVALID.SINGLE

This is an invalidate single Memory Request microinstruction. Bits <29:23> have the hex value 0E. This memory function invalidates a single translation buffer entry. The data type field must specify byte.

The register specified by the long operand contains a 32-bit virtual address. If the specified virtual address is in the translation buffer, then that entry is set invalid. Otherwise, no operation is performed. No useful data are returned by the memory controller on the move in from memory.

INVALID.MULTIPLE

This is an invalidate multiple Memory Request microinstruction. Bits <29:23> have the hex value 0F. This memory function invalidates all of the translation buffer entries. The data type field must specify byte.

The register specified by the long operand contains a 32-bit virtual address. Translation buffer entries are unconditionally invalidated. Bit <10> of the virtual address selects whether the process or system translation buffer is invalidated. Translation buffer entries are invalidated starting with the specified address and continuing until a page crossing occurs. No useful data are returned by the memory controller on the move in from memory.

RCHECK

This is a read check Memory Request microinstruction. Bits <29:23> have the hex value 0A. This memory function performs a read check to determine the accessibility of the first byte of a virtual address. The data type field must specify byte.

The register specified by the long operand contains the 32-bit virtual address of the byte to be checked. If this virtual address is not in the translation buffer, a translation buffer miss is reported in the error summary register as the TB-Check code. If mapping is not enabled, then no access check is performed. No useful data are returned by the memory controller on the move in from memory. The purpose of RCHECK and of WCHECK is to determine the accessibility of data without forcing the translation buffer to be filled.

WCHECK

This is a write check Memory Request microinstruction. Bits <29:23> have the hex value 4A; that is, the 5-bit memory function code has the value 0A but the modify intent bit is set. This memory function performs a write check to determine the accessibility of the first byte of a virtual address. The data type field specifies byte.

The register specified by the long operand contains the 32-bit virtual address of the byte to be checked. If this virtual address is not in the translation buffer, a translation buffer miss is reported in the error summary register as the TB-Check code. If mapping is not enabled, then no access check is performed. No useful data are returned by the memory controller on the move in from memory.

Memory Controller Status

After a Memory Request microinstruction and an intervening microinstruction have been executed, the next microinstruction executed can test the results of a memory function. The status of a memory function is available at the same time that the data requested by the memory function are available on a read, or at the same time that the data are presented on a write. This status remains available until another memory function is executed.

Memory controller status is returned to the data path via four bits of status, available as microsequencer OR MUX inputs:

- TB Miss. The memory controller cannot complete the current virtual function because the appropriate page table entry is not in the translation buffer.
- Memory Modify Refuse. The memory controller cannot complete the current virtual function because the modify bit is not set in the translation buffer copy of the page table entry.
- Page Crossing. The memory controller cannot complete the current virtual read/write function because a page crossing is necessary.
- Error Summary. An error code has been written into the memory controller's error code register, indicating one of the following errors:
  - Access Violation. The memory controller cannot complete the current virtual function because the desired access is not allowed.
  - Parity Error. A memory read error that is not correctable has been detected.

- Nonexistent Memory. An attempt has been made to read a nonexistent memory location.
- Illegal Operation. An attempt has been made to access I/O space as a longword or as an unaligned word, or an attempt has been made to execute an interlocked read/write to a longword or an unaligned word.
- Translation Buffer Check. A read or write check function encountered a translation buffer miss.

The DAP microcode determines which of these errors occurred by reading the error code register. This is accomplished by issuing the READ.ERROR.CODE Memory Request microinstruction.

Chapter 6
Data Path Module

This chapter is a detailed description of the components on the data path module and how they interact. First, the major logic elements and their hardware components are described. Then, the basic transfers of data between the logic elements are described on a microprogram level.

Overview of DAP Functions

The data path module contains hardware to perform the following eight functions:

- control microinstruction flow
- decode macroinstructions
- execute microinstructions
- transfer data within the data path module
- process interrupts
- communicate with the console terminal
- power on
- communicate with the memory controller

The next eight sections describe these functions, and the hardware components that implement them, in detail. The hardware components are illustrated in the DAP block diagram, Figure 6-1.

Controlling the Microinstruction Flow

Controlling the microinstruction flow is the main function of the data path module, and much of the hardware is dedicated to it. The hardware components are the CPU clocks, the control store, the control store address register, the parity checker, the index register, the microsequencer, and the microstack and microstack pointer. These components, plus some control signals, determine which microinstruction is executed next. The following paragraphs describe each of these components in turn.

Clock Signals

The clocks for the system are generated on the MCT module. A basic clock with a 64 MHz frequency is generated by a crystal oscillator and is used to derive all the other clocks in the system.

The CPU clock (DAPL CPU CLOCK) consists of a symmetrical 250 ns period clock. The start of a microcycle is defined as occurring on the leading edge of this clock and is referred to as T0. All the internal data bus registers are written on this edge. The trailing edge of the clock occurs 125 ns later.

The signal DAPL CPU PHASE is a clock with the same timing as CPU CLOCK, but it is not affected by stall conditions.

The delayed CPU clock (DAPL DLYD CPU CLK) is asserted from T62.5 to T187.5. This clock is used to clock PALs which first decode the microinstruction and generate discrete control signals.

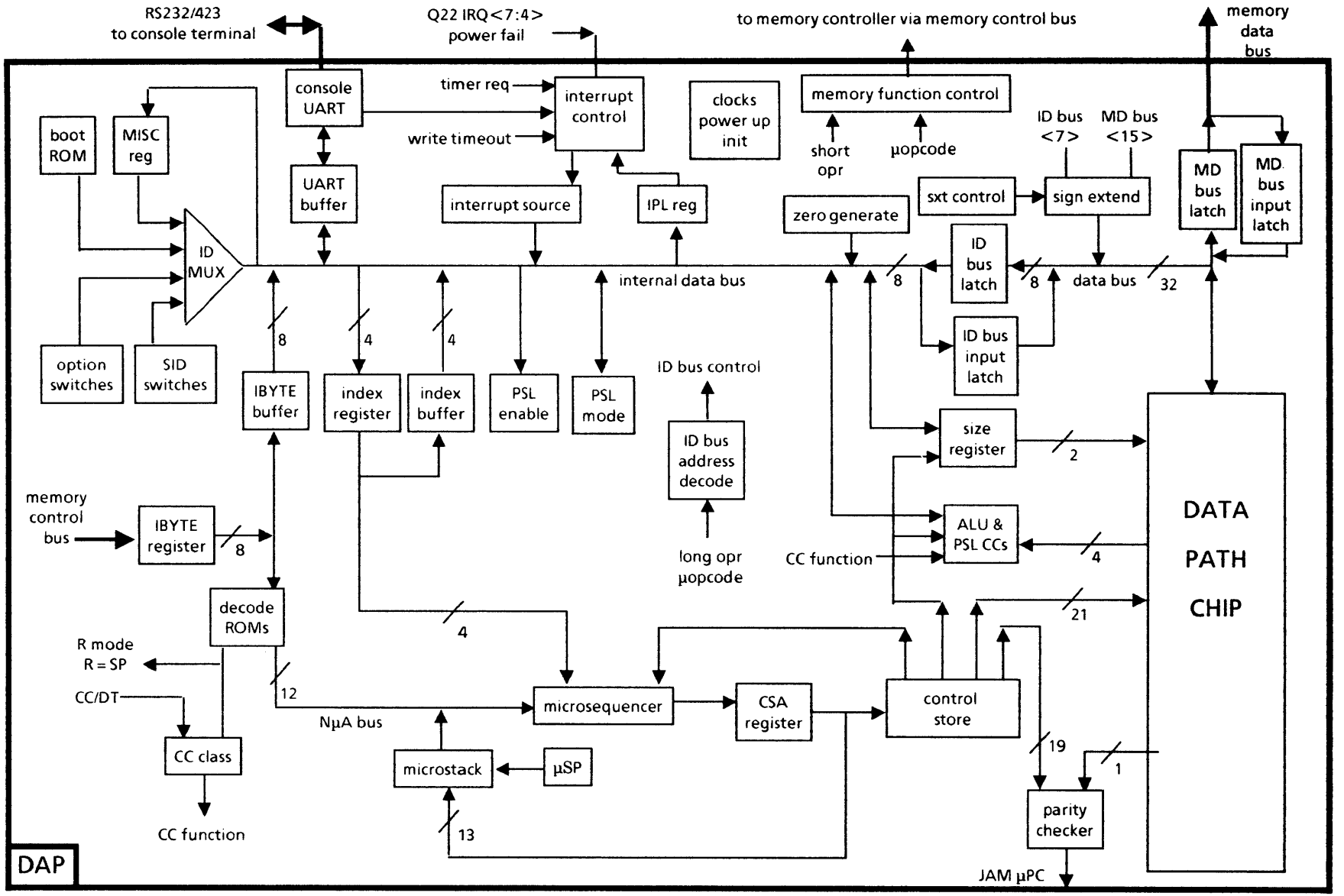


Figure 6-1. Data Path Block Diagram

The control store address clock (DAPL CPU PHASE) is asserted from T125 to T250. This signal clocks the control store address register.

Control Store

The control store consists of five 8K by 8-bit PROMs. The address space within the control store is organized as 32 pages, each page containing 256 words. Each word is one data path microinstruction and is 40 bits wide. All the data path microinstructions are stored in this control store.

The input to the control store is a 13-bit microaddress supplied by the control store address register (DAPB CSA <12:00>). The high-order five bits specify the page, and the low-order eight bits select the word within the page.

The control store output is a 40-bit microinstruction (DAPA CS <39:00>). The various microinstruction bits are sent different places.

- Nineteen bits are sent from the control store to the parity checker: CS <39>, CS <38:37>, and CS <15:00>.
- CS <36:16> are sent to the data path chip (the data path control field).
- The thirteen low-order bits of the next address control field, CS <12:00>, are sent to the jump register in the microsequencer.
- The eight high-order bits of the next address control field, CS <15:08>, and CS <24>, are sent to the jump MUX control logic and the OR MUX control logic in the microsequencer. The logic decodes CS <15:08> and generates various

control signals and selects to govern the microsequencer elements.

- Microinstruction bits CS <38:37>, the CC/DT field, are sent to a flip-flop and then to the PAL that contains the size register; they are also sent to the condition code control logic.
- Microinstruction bits CS <24:23> are sent to the decode ROMs to indicate the type of opcode or operand specifier decode.
- The microinstruction opcode CS <36:32>, CS <24>, and the long operand CS <22:16>, are sent to the block of logic labeled ID bus address decode. This block of logic controls the driving of the appropriate data on the internal data bus when a Move or Moveout microinstruction is executed, and the long operand specifies an address external to the data path chip.
- The microinstruction memory function bits CS <31:23> are sent to the block of logic labeled memory function control in case the microinstruction is a memory request. Information about the microinstruction opcode is sent to the memory function control logic from the ID bus address decode logic.
- Microinstruction bits CS <28:24> are sent to the IBYTE control logic. Information about the microinstruction opcode is sent to the IBYTE control logic from the ID bus address decode logic. Information about the long operand is sent to the IBYTE control logic from the memory function control logic.

Control Store Address Register

The control store address (CSA) register holds the microaddress used to access the control store. While the inputs to the control store are held stable in this register, the outputs can be used to control the operation of the data path.

The input to the CSA register is the thirteen microaddress bits from the next microaddress MUX in the microsequencer. The output from the CSA register is the input to the control store: CSA <12:00>. The control store address register is clocked at T2.

Parity Checker

The parity checker consists of three chips that check the parity of the 40-bit microinstruction. If a parity error is found, the next microaddress is forced to zero, and a flag is set. The microinstruction causing the parity error is executed but produces undefined results. The microinstruction executed at location zero reads the flag by reading bit 5 at the same address as the index register. (The index register itself is four bits wide.)

The input to the parity checker is bits <39:37> and <15:0> from the control store, and one parity bit from the data path chip. When the data path chip receives control store bits <36:16> (the data path control field), it generates a parity bit for these bits; this data path chip parity bit is sent to the parity checker.

If no parity error is found, there is no output from the parity checker. If a parity error is detected, the output from the parity checker generates the signal JAM μPC, which forces the next microaddress to zero.

Index Register

The index register is a four bit register used to store and test the microcode state. The input to the index register is the low four bits from the internal data bus (BUS ID <03:00>). The four bits in the index register are sent as one of the four-signal inputs to the OR MUX (INDEX <3:0>, see Table 5-6), or they can be driven back onto the ID bus through the index buffer (BUS ID <03:00>).

Microsequencer

The microsequencer generates a 13-bit microaddress every 250 ns. It accomplishes this by decoding certain bits in the previous microinstruction while monitoring certain control and status lines.

The microsequencer consists of these components: the page register, the microprogram counter (μPC), the conditional decrementer, the microstack, the microstack pointer, the jump register, the OR MUX, the jump MUX, and the next microaddress MUX (NμA MUX). These components are described briefly in the following paragraphs. Figure 6-2 is a block diagram of the data path microsequencer.

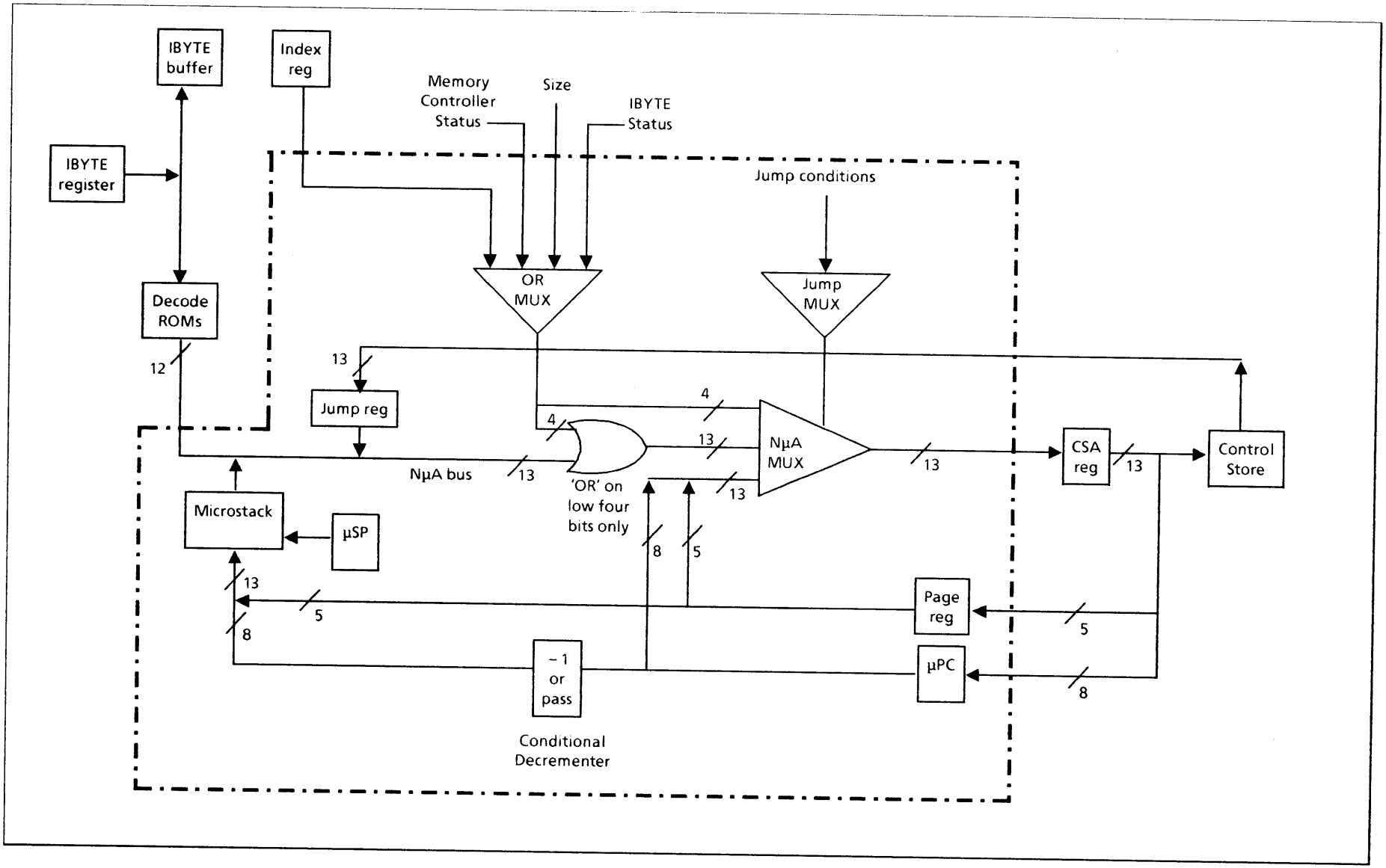


Figure 6-2. Microsequencer Block Diagram

Page Register and Microprogram Counter

These two components together hold the next 13-bit microaddress. The page register contains bits <12:8>; these form the page address. The μPC contains bits <7:0>; these form the address of the word within the page. The μPC is loaded with the address of the current microinstruction plus one, and cannot count beyond the end of the current page. The μPC is clocked at T0 of each microcycle.

Conditional Decrementer

The conditional decrementer is an adder located in the microsequencer logic between the μPC and the microstack. All eight bits from the μPC run through the decrementer. When a microtrap is taken, the decrementer adds a negative one to the μPC bits. Otherwise, the μPC bits pass through unaffected.

Microstack

The microstack is a 16 deep, LIFO (last-in-first-out) stack used to save return microaddresses when subroutine calls or microtraps are executed. The address of the current microinstruction plus 1 is also saved on the microstack when a valid operand specifier decode is executed and the operand is not contained in a general register.

If the current microinstruction is a subroutine call or an operand specifier decode (not register mode), the conditional decrementer adds zeros to the address in the microprogram counter, causing the microaddress of the current instruction plus 1 to be saved on the stack. If the current microinstruction is a trap, or a Decode and the OR MUX is not equal to zero, the conditional decrementer subtracts one from the address in the

microprogram counter, causing the microaddress of the current instruction to be saved on the stack.

The input to the microstack is supplied by the conditional decrementer. The decrementer always supplies a microaddress to the microstack, but the microaddress does not get written into the microstack unless the current operation is a subroutine call, a trap, an operand specifier decode (not register mode), or an IRD and the OR MUX is not equal to zero. The microstack is written at T250 of the microcycle (T0 of the next cycle).

The signal DAPC STACK PUSH is asserted by the microsequencer control logic when it decodes microinstruction bits CS <15:08> and <24> and determines that the next address control field format is a subroutine call, a trap, or an operand specifier decode. STACK PUSH enables writes to the microstack.

The output from the microstack is the top entry in the stack which is driven onto the next microaddress bus when the operation is a return.

Microstack Pointer

The microstack pointer (μSP) always points to the top entry in the microstack; that is, the microstack pointer contains the address of the microstack location that contains the most recently stored microaddress.

When the operation is a "push" (a subroutine call or a microtrap), the address of the next location in the microstack is calculated and used to address the microstack so that the microaddress from the conditional decrementer is written into the microstack at that location. If the branch is taken, the calculated microstack address is stored in the microstack pointer.

When the operation is a "pop" (a return), the current microstack location address in the microstack pointer is used to address the microstack so that the microaddress at that location is written onto the next microaddress bus. Then the microstack pointer is updated at the next T0 clock edge to contain the previous microstack location address.

The inputs to the microstack pointer are signals to indicate when the current operation is a push or a return. Another input to the microstack pointer is the TAKE BRANCH signal which is asserted during conditional subroutine calls and returns. The outputs from the microstack pointer are four microstack address lines.

Jump Register

This register is used to allow the outputs from control store to be driven onto the next microaddress bus (NμA bus). The jump register is open from T0 to T125.

The input to the jump register is the microinstruction next address control field, CS <12:0>, from the control store. When enabled, the jump register drives these same thirteen bits onto the next microaddress bus (NμA bus).

OR MUX

The OR multiplexer allows the microcode to "case" (that is, perform a multiway branch) on certain signals in the data path, based on the value of the OR field in the current microinstruction. Conceptually, there are eight inputs to the OR MUX, each with four signals. Some signal values are fixed, others reflect the microcode state (see Table 5-6 in Chapter 5).

The OR MUX is enabled when the microinstruction next address control field format is CASE, BSB, TRAP, RET, IRD, or SPEC DEC. (Although there is no OR field in the SPEC DEC next address control field format, the OR MUX is enabled to test for IB invalid.) The OR MUX logic decodes the format and OR fields, and enables the appropriate input. The value of the four signals on that input then becomes the value of the OR MUX output. (For an example, see the section titled "Next Address Control Field" in Chapter 5.) The OR MUX output is then logically ORed with the low four bits of the microaddress on the next microaddress bus (NμA bus). The result is the low four bits of the address of the next microinstruction.

The input to the OR MUX logic is bits <24> and <15:8> of the microinstruction; the input to the OR MUX itself is the various microcode state signals listed in Table 5-6. The OR MUX output is the value of the signals on the selected input line. This value is sent to the NμA bus.

Jump MUX

The jump MUX is part of the jump control logic that controls the next microaddress multiplexer (NμA MUX). The jump MUX selects input signals according to the JC (jump control) field of the current microinstruction (see Figure 5-3). The JC field specifies conditions to be tested (see Table 5-5).

If the specified condition is met, the jump control logic enables the NμA MUX to select the address specified in JA<7:0> of the current microinstruction as the next microaddress. If the specified condition is not met, the jump control logic enables the NμA MUX to select the current microaddress plus one for the next microaddress.

The input to the jump control logic is next address control field bits <24> and <15:8> of the current microinstruction, and the current values for the conditions that could be tested. The output of the jump control logic is the select lines to the next microaddress MUX.

Next Microaddress MUX

This multiplexer provides the inputs to the CSA register. It is used to select either the contents of the page register and the microprogram counter (μPC), or the contents of the next microaddress bus (NμA bus).

When performing conditional jumps, the desired jump-to address is driven onto the NμA bus early in the microcycle. Later in the cycle, the NμA MUX select lines are changed by the jump control logic depending on whether the jump is to be taken.

The NμA MUX actually consists of two 2-to-1 MUXs and three 4-to-1 MUXs. Two of the 4-to-1 MUXs make up the low 4-bit slice of the NμA MUX. One of following three inputs to the NμA MUX is selected by the jump control logic as the NμA MUX output:

- the current microaddress contained in the page register and the microprogram counter, which is the current microaddress plus one, or
- the microaddress currently on the NμA bus, but with the value of the low four bits determined by the OR MUX output ORed with NμA bus <3:0>, or
- microaddress bits <12:4> forced to zero and the value of the low four bits determined directly by the output of the OR MUX.

These inputs to the NμA MUX are stable at T0+112. The third case described above allows traps which may

be taken during decode instructions to use the output of the OR MUX directly. For all other instances, the low four bits of the NμA MUX input are determined by the OR MUX output ORed with the address supplied from the NμA bus (the second case described above), or by μPC + 1 (the first case described above).

In short, the inputs to the NμA MUX are: NμA bus <12:0>, OR MUX <3:0>, and μPC <12:0>. The output of the NμA MUX is referred to as NμA <12:0>; these bits have the same value as the bits of the selected input.

When a microinstruction has a BR or CASE next address control field format (see Figure 5-3), the destination microaddress must be within the current page. The NμA MUX has separate selects for bits <12:08> and bits <07:00> so that for BR and CASE, the select for bits <12:08> is not changed even if the branch is taken.

During certain microinstructions, it is necessary to force zeros to be output from the NμA MUX. Table 6-1 lists these conditions.

Table 6-1. Forced Zeros on NμA MUX Output

Bits	Conditions
12	IRD BSB trap control store parity error power up
11:08	BSB trap decode and trap control store parity error power up
07:04	Decode microinstruction when a trap is being taken control store parity error power up
03:00	control store parity error power up

Figure 6-3 lists the possible next address control field formats, and shows which hardware components of the microsequencer provide the bits for the next microaddress.

Next Address Control Field Formats	Next Microaddress Bits													Comments
	12	11	10	9	8	7	6	5	4	3	2	1	0	
JMP	<12:0> of current microinstruction, through jump register, onto NμA bus													
JSB	<12:0> of current microinstruction, through jump register, onto NμA bus													current microaddress + 1 saved on microstack
BR: JC true	<12:8> from page register					<7:0> of microinstruction, through jump register to NμA bus								
CASE: JC true	<12:8> from page register					<7:0> of microinstruction, through jump register to NμA bus; <3:0> modified by OR MUX output								
BSB: JC true	<12:8> = 0					<7:0> of microinstruction, through jump register to NμA bus; <3:0> modified by OR MUX output								current microaddress + 1 saved on microstack
TRAP: JC true	<12:8> = 0					<7:0> of microinstruction, through jump register to NμA bus; <3:0> modified by OR MUX output								current microaddress saved on microstack
RET: JC true	<12:0> from top entry in microstack onto NμA bus; <3:0> modified by OR MUX output													
JC not true for BR, CASE, BSB, TRAP, RET	<12:0> from page register and microprogram counter													
IRD: JC true	<12:4> = 0									<3:0> = OR MUX output				current microaddress saved on microstack
IRD: JC not true	0	<11:0> from decode ROMs onto NμA bus												current microaddress + 1 saved on microstack
SPEC DEC: IBYTE valid, operand not in GPR	<12:8> of microinstruction, through jump register to NμA bus					<7:0> from decode ROMs onto NμA bus								current microaddress + 1 saved on microstack
SPEC DEC: IBYTE valid, operand in GPR	<12:0> from page register and microprogram counter													
SPEC DEC: IBYTE not valid	<12:4> = 0									<3:0> = OR MUX output = 0001 (binary)				current microaddress saved on microstack
control store parity error or power up	<12:0> = 0													

Figure 6-3. Next Microaddress Sources

Decoding Macroinstructions

Decoding macroinstructions is the second of the eight functions that the data path module performs. The hardware components are the IBYTE register, IBYTE control, the decode ROMS, condition code control, condition code class register, condition code PALs, macrolevel branch control, PSL enable, and the size register. These components decode macroinstruction opcodes and operand specifiers. The following paragraphs describe each of these components in turn, and the ALU and PSL condition codes.

IBYTE Register

The instruction byte register is an eight-bit register that holds the next byte of instruction stream data to be evaluated at the inputs to the decode ROMs; that is, it contains the macrolevel instruction byte currently being processed.

The IBYTE register is read on the internal data (ID) bus when the long operand of the current microinstruction specifies IB.BYTE, which is the IBYTE register's unique address. The contents of the IBYTE register are also driven on the ID bus during operand specifier decodes and stored in one of the two pointer registers in the data path chip. If the operand specifier mode is not short literal, bits <5:4> of the IBYTE register are forced to zero to extract the register number. The high two bits need not be set to zero because the pointer registers are only six bits wide.

The IBYTE register is loaded at T0 from the memory control bus whenever the signal DAPR LOAD I BYTE is asserted. LOAD I BYTE asserted means that the next byte from the instruction stream is needed at the

end of the current microcycle. There are two reasons why the next I-stream byte is needed. The first reason is that the byte currently in the IBYTE register is valid, but the current microinstruction uses that byte, so at the end of this microcycle, the byte in the IBYTE register will no longer be needed. The second reason is that the byte currently in the IBYTE register is not valid. The signal DAPR IB INVALID is asserted to indicate when this is the case.

The input to the IBYTE register is the byte from the memory control bus, BUS MEM CTL <7:0>. The output from the IBYTE register is eight bits labeled DAPF I BYTE <7:0>. These bits go two places; they are the input to the decode ROMs, and they are latched in the IBYTE buffer (see Figure 6-1).

IBYTE Control

The IBYTE register is controlled by the IBYTE control PAL. The IBYTE control logic informs the memory controller when the next instruction stream byte is needed by asserting DAPR IB TAKEN. The next instruction stream byte is needed either because the byte currently in the IBYTE register is valid and is used by the current microinstruction, or because the byte in the IBYTE register is not valid.

When DAPR LOAD I BYTE is asserted, and the clock signal DAPL CPU PHASE is asserted, the signal DAPR CLOCK I BYTE is generated. DAPR CLOCK I BYTE clocks the bits BUS MEM CTL <7:0> off the memory control bus into the IBYTE register at T0.

When the IBYTE control logic asserts DAPR LOAD I BYTE and the clock signal DAPL ALLOW STALL is asserted, the signal DAPR IB TAKEN is generated. DAPR IB TAKEN informs the memory controller that the instruction stream byte that was on the memory

control bus has been loaded into the IBYTE register, and another instruction stream byte needs to be sent from the prefetch logic to the memory control bus.

Thus, DAPR IB TAKEN causes the memory controller prefetch logic to drive an instruction stream byte onto the memory control bus. DAPR LOAD I BYTE, DAPR CLOCK I BYTE, and DAPR IB TAKEN are the signals asserted when the next instruction stream byte is needed because the byte currently in the IBYTE register is valid but is no longer needed because it was just used by the current microinstruction.

If the byte in the IBYTE register is not valid, the IBYTE control logic assserts the signal DAPR IB INVALID. The memory controller continues to send instruction stream bytes to the memory control bus as long as IB INVALID is asserted; that is, LOAD I BYTE is always true when DAPR IB INVALID is asserted. When IB INVALID is deasserted, this means the byte in the IBYTE register is valid, and the memory controller stops sending instruction stream bytes from the prefetch logic.

The memory controller, meanwhile, generates the signal MCTT NXT VALID REG, which when asserted means that the byte on the memory control bus is valid data. The memory controller deasserts MCTT NXT VALID REG when the byte on the memory control bus becomes invalid for any reason; for example, the prefetch buffer becomes empty, an I-stream Request microinstruction (which flushes the prefetch buffer) is executed, a Memory Request microinstruction with the IB.REFILL function is executed, or the microinstruction long operand specifies IB.BYTE.

IB INVALID is deasserted by the signal NXT VALID REG from the memory controller. As long as the memory controller can supply valid instruction stream

bytes from the prefetch logic to the memory control bus, NXT VALID REG remains asserted.

IB INVALID is asserted by any of the following microinstructions if the NXT VALID REG signal from the memory controller is deasserted: Decode, I-stream Request, Memory Request specifying IB.REFILL, a microinstruction in which the long operand specifies IB.BYTE.

Figure 6-4 illustrates the timing relationship between these signals for both cases:

Case 1: The IBYTE register needs to be refilled because the current byte is valid but a Decode microinstruction was just executed.

Case 2: The IBYTE register needs to be refilled because the current byte is not valid.

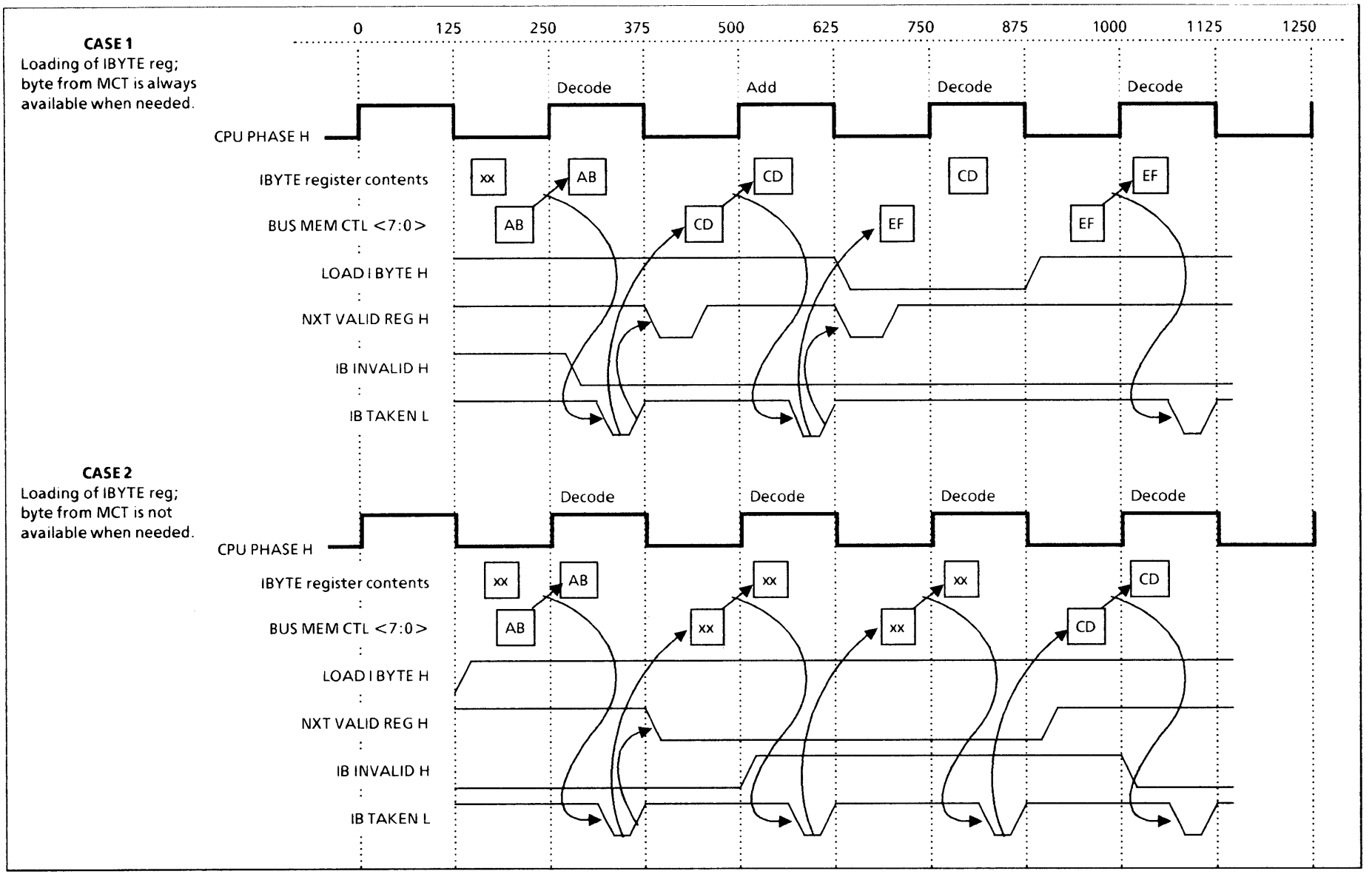


Figure 6-4. IBYTE Register Loading

Decode ROMs

The decode ROMs are logically 1K by 16 bits. They are used to select the microcode routine to be executed depending on the current contents of the IBYTE register. When the current microinstruction is a Decode, the output from the decode ROMs is driven onto the NμA bus.

The inputs to the ROMs are bits <7:0> from the IBYTE register (DAPF I BYTE <7:0>), and the two bit control field from the current Decode microinstruction, bits <24:23> (DAPA CS <24:23>). Bits <24:23> are encoded as follows:

24	23	Selected Decode
0	0	operand specifier decode type 1
0	1	operand specifier decode type 2
1	0	IRD for single byte opcodes
1	1	IRD on second byte of two byte opcode

If the decode operation is an IRD, the outputs from the ROMs are:

- two bits of condition code class (DAPF CC CLASS <1:0>). For all instructions except conditional branches, these two bits define how the PSL condition codes are set. The encoding is shown in Table 6-2.
- two bits of data type (DAPF DT1/RMODE and DAPF DT0/SP). For all instructions except conditional branches, these two bits are encoded as follows:

  00 byte 10 not used
  01 word 11 longword

- twelve bits of microaddress (BUS NUA <11:00>)

- if the instruction is a macrolevel conditional branch, the low-order bit of the condition code class (CC CLASS <0>) and the data type field are combined to form a code that indicates which condition codes need to be tested for that specific branch. The encoding is listed in the section titled "Macrolevel Branch Control" in this chapter.

If the decode operation is an operand specifier decode, the outputs from the ROMs are:

- eight bits of microaddress (BUS NUA <07:00>)
- one bit to specify register mode and not PC (DAPF DT1/RMODE)
- one bit to indicate that the stack pointer (R14) is specified (DAPF DT0/SP)
- one bit to indicate that the operand specifier being decoded is not a short literal (DAPF CC CLASS 0).

ALU and PSL Condition Codes

There are two separate sets of condition codes stored in the data path. The first set is the ALU condition codes which are used at the microprogram level. These condition codes result from the last ALU operation in the data path chip. They are available as jump conditions to the microcode and are also used to load the PSL condition codes.

The other set of condition codes is the PSL condition codes. These are part of the PSL and are available to the macrolevel code. They are used to determine if a macrobranch should be taken.

Both sets of condition codes (ALU and PSL) can be read or written on the internal data bus as bits <3:0>.

Condition Code Control

The setting of the condition codes is controlled by the CC/DT field of the microinstruction, the microinstruction opcode, and the condition code class register.

For microinstruction opcodes Move, Moveout, Memory Request, I-stream Request, Multiply Step, Restore, Clear Save Stack, and Decode, the condition codes are never set and the CC/DT field is used only for data type.

For all other microinstruction opcodes, the condition codes are set as follows for the given values of the CC/DT field:

0 data type is long, CCs not affected
1 data type is long, set ALU CCs
2 data type is long, set ALU and PSL CCs
3 data type is SIZE, set ALU and PSL CCs

Condition Code Class Register

The condition code class register is part of the logic that sets the condition codes. It is loaded from the decode ROMs at the end of every macroinstruction opcode decode (also referred to as an instruction read and decode, or IRD). The bits DAPF CC CLASS 1, DAPF CC CLASS 0, and DAPF DT0/SP are loaded into this register from the decode ROMs. The first two, CC CLASS <1:0>, contain an encoded value; the encoding is shown in Table 6-2. These register encodings are essentially setup conditions; when the value of the two bits is as given, the PSL condition codes will be set as defined in the Function column.

Table 6-2. Condition Code Class Register Encoding

<1:0>	CC Class	Function
0	Logical	ALU N to PSL N ALU Z to PSL Z ALU V to PSL V PSL C to PSL C
1	Arithmetic	ALU N to PSL N ALU Z to PSL Z ALU V to PSL V ALU C to PSL C
2	Compare	ALU N to PSL N ALU Z to PSL Z Clear PSL V ALU C to PSL C
3	Floating Point	ALU N to PSL N ALU Z to PSL Z ALU V to PSL V Clear PSL C

The output of the condition code class register is the same two CC CLASS bits, labeled DAPE CC CLASS <1:0>.

Condition Code PALs

The ALU and PSL condition codes are stored in two PALs. One PAL stores the ALU and PSL N and V bits, and the other stores the ALU and PSL Z and C bits. PSL <3:0> are contained in these two PALs; that is, these two PALs contain the low four bits of the PSL register. The PALs are controlled by a four-bit condition code function field, DAPE CC <F3:F0>. This CC function field is the output of another PAL, called the CC Function, or CC Pipeline PAL. The CC

function field is generated from the following five bits: DAPE CC CLASS <1:0>, DAPC CS REG <38:37>, and DAPC NO CC OP (1). This last bit indicates whether the current microinstruction is one that affects the condition codes. When NO CC OP is low, the CC function field is 0000 binary. The encoding of the CC function field is shown in Table 6-3.

Table 6-3. CC Function Field Encoding

DAPE CC <F3:F0>	Function
0000	no operation, CCs unaffected
0100	load ALU CCs logical
0101	load ALU CCs arithmetic
0110	load ALU CCs compare
0111	load ALU CCs floating
1100	load ALU/PSL CCs logical
1101	load ALU/PSL CCs arithmetic
1110	load ALU/PSL CCs compare
1111	load ALU/PSL CCs floating

Because the data path chip is pipelined (that is, the microcycles overlap; see Figure 1-4 in Chapter 1), the condition codes are affected by the previous microinstruction and not the current one. The first microinstruction is decoded and the control information (the CC function field) stored until the following T0. The stored information is then used to directly control the PALs that store the condition codes. Figure 6-5 shows when the ALU condition codes are available and when they are loaded into the PALs from the data path chip for a Compare microinstruction.

Condition codes are set as follows. The signals DAPF CC CLASS <1:0> are the output from the decode ROMS during IRDs. These signals are the input to the

condition code class register, and also the output from the condition code class register as DAPE CC CLASS<1:0>. These two bits, plus DAPC CS REG <38:37> and DAPC NO CC OP (1) generate the CC function field, labeled DAPE SET CC <F3:F0>. The CC function field bits are sent through a flip-flop to delay them one microcycle. As the output from the flip-flop, they are labeled DAPE CC <F3:F0>. From there, the CC function field bits become part of the input to the two condition code PALs that store the ALU and PSL condition codes. The other inputs to these two PALs are the N, Z, V and C condition codes themselves from the last data path chip operation (DAPH DPC <N,Z,V,C>). The PSL condition codes that are stored in these PALs (PSL <3:0>) are set according to the encoding of the CC function field and the values of DAPH DPC <N,Z,V,C> from the data path chip.

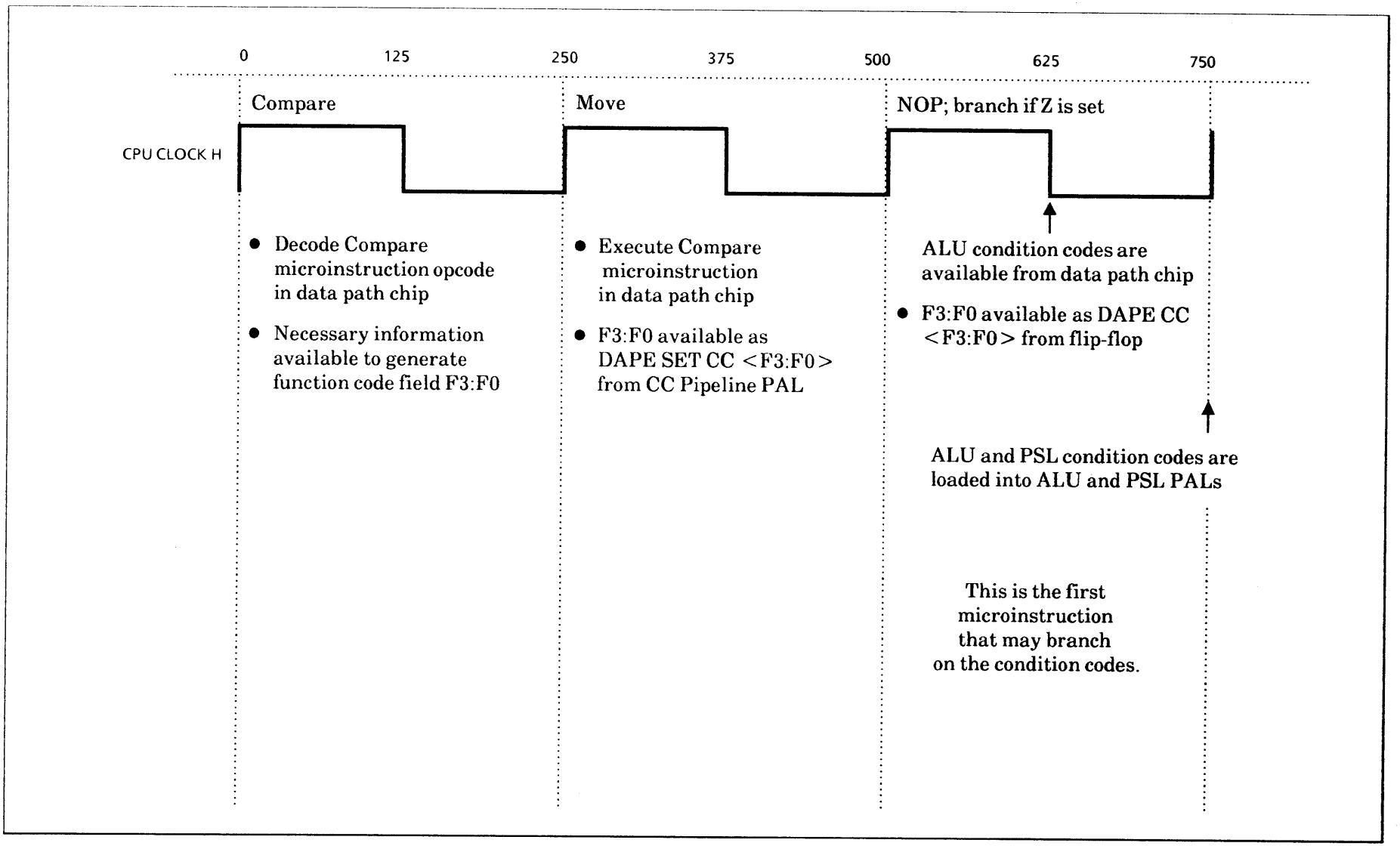


Figure 6-5. Condition Code Setting Timing Diagram

The ALU condition codes available as the output of these PALs (DAPE ALU <N,Z,V,C>) are the stored ALU condition codes, and are available as jump conditions to the microcode, along with DAPH DPC <N,Z,V,C>. These eight signals are the inputs to a multiplexer (ALU BR MUX) that allows microbranches to be taken on either the result of the current data path chip operation (DAPH DPC <N,Z,V,C>) or the stored ALU condition codes (DAPE ALU <N,Z,V,C>). Both the true and the inverted output of this MUX (DAPE ALU BR H and DAPE ALU BR L) go to the jump MUX as part of the microbranch control logic.

Macrolevel Branch Control

The condition code test for the macrobranch instructions is performed in the CC Class & Branch PAL. (This is the same PAL that contains the condition code class register.) The inputs to this PAL are:

- the same three bits from the decode ROMs used for the condition class register: DAPF CC CLASS 1, DAPF CC CLASS 0, and DAPF DT0/SP,
- bit zero from the IBYTE register, and
- the PSL condition code bits, DAPE PSL <N,Z,V,C> from the output of the condition code PALs.

The three bits in the first category of inputs listed above are output bits 15, 14, and 12 from the decode ROMs. These three bits form a hex code to indicate what PSL condition code bits need to be checked:

Decode ROMs <15:12>	Hex Code to PAL	CCs Checked	Opcodes (hex)
0	0	N	BGEQ, BLSS (18, 19)
1	1	Z	BNEQ/BNEQU (12), BEQL/BEQLU (13)
4	2	V	BVC, BVS (1C, 1D)
5	3	C	BGEQU, BCC (1E, 1F)
8	4	N OR Z	BGTR, BLEQ (14, 15)
9	5	C OR Z	BGTRU, BLEQU (1A, 1B)

Bit zero from the IBYTE register is the low-order bit of the macroinstruction opcode and indicates whether or not the branch should be taken if the tested condition is met.

The PSL condition code inputs are the current values of the conditions being checked.

The output from this CC Class & Branch PAL is the signal DAPE BR FALSE. This signal is one of the inputs to the OR MUX, indicating that the branch is not to be taken. At IRD, this signal is always true.

PSL Enable

The PSL enable logic is contained in a PAL. This PAL stores PSL bits 5 and 4: the integer overflow enable bit (IV) and the trace trap bit (T).

These two bits are shown on the DAP block diagram, Figure 6-1, as PSL enable. The bits are written at T0 with internal data bus <5:4>.

Size Register

The size register is used to control the data type of operations being performed in the data path chip, or the

size of a datum to be transferred during a memory request operation.

Size Register Value	Data Type
0	byte (8 bits)
1	word (16 bits)
2	not used
3	longword (32 bits)

The size register is loaded at T0 of the next cycle from:

- internal data bus <1:0> when the size register is explicitly specified in the long operand of a Moveout microinstruction.
- the decode ROMs during macroinstruction opcode decodes (IRDs).
- microinstruction bits <38:37> (the data type field) during macroinstruction operand specifier decodes if the data type field specifies byte, word, or long. If an encoding of 2 is specified, then the size register is unaffected.

The outputs of the size register are:

- DAPE DPC DT1 and DAPE DPC DT0. These bits are sent to the data path chip.
- DAPE SIZE 1 and DAPE SIZE 0. These two signals are two of the OR MUX inputs (see Table 5-6).
- BUS ID 01 and BUS ID 00. These signals are driven through the PSL.ENABLE PAL onto the internal data bus.

The size register is controlled by read and write signals from the ID bus address decode logic.

Executing Microinstructions

Executing microinstructions is the third of the eight functions that the data path module performs. The execution phase of almost all microinstructions takes place in the data path chip (DPC). The data path chip consists of a 32-bit data path, register file, and ALU, and is implemented in 3 micron NMOS technology. Figure 6-6 is a block diagram of the data path chip. The chip components are described in the following paragraphs.

Clock Signal

Internally, the chip runs on a two-phase clock system consisting of Phase 1 (PH1) and Phase 2 (PH2). The clock phases are derived by dividing the clock input signal, DAPL DPC CLK, by two internally on the chip. The clock circuitry external to the chip synchronizes the internal clock phases with the signal DAPL DPC RESET. The low going edge of DPC CLK that occurs immediately after DPC RESET is deasserted forces the internal clock phases to PH1.

DPC RESET is an active low signal which has the following effects on the data path chip:

- it disables the data bus tri-state drivers,
- it presets the timer, and clears bits 0 and 1 in the timer control/status register (TMRCSR), and
- it clears the control store register, so the chip will execute NOPs.

The DPC RESET signal is typically used on power up or testing. Parity and condition code signals are undefined during this time. The DPC RESET signal must be active for at least eight clock periods (four

microcycles). It can be asserted asynchronously to DPC CLK, but is deasserted synchronously to DPC CLK. Figure 6-7 shows the timing relationship between the chip clock signals and phases.

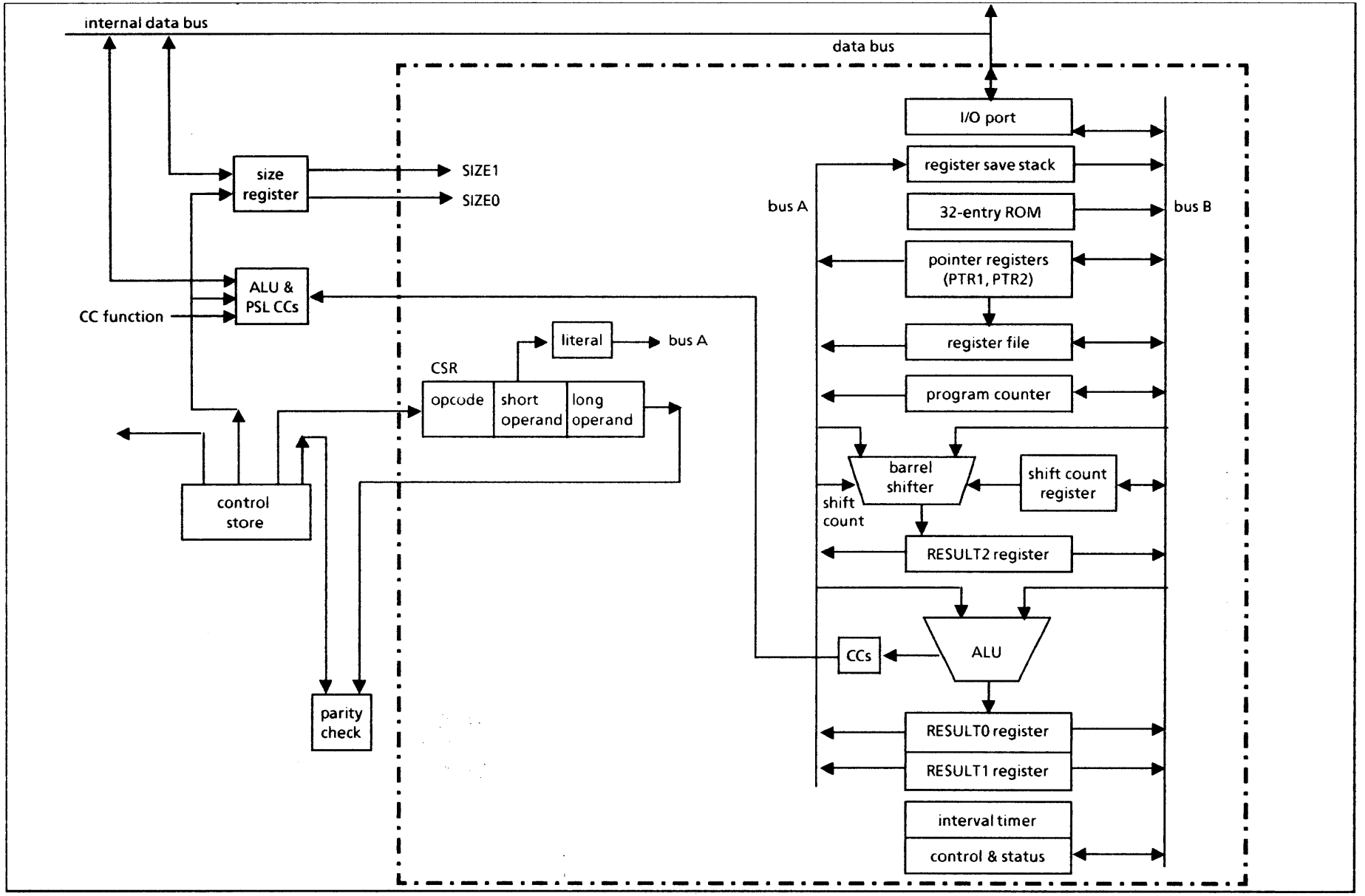


Figure 6-6. Data Path Chip Block Diagram

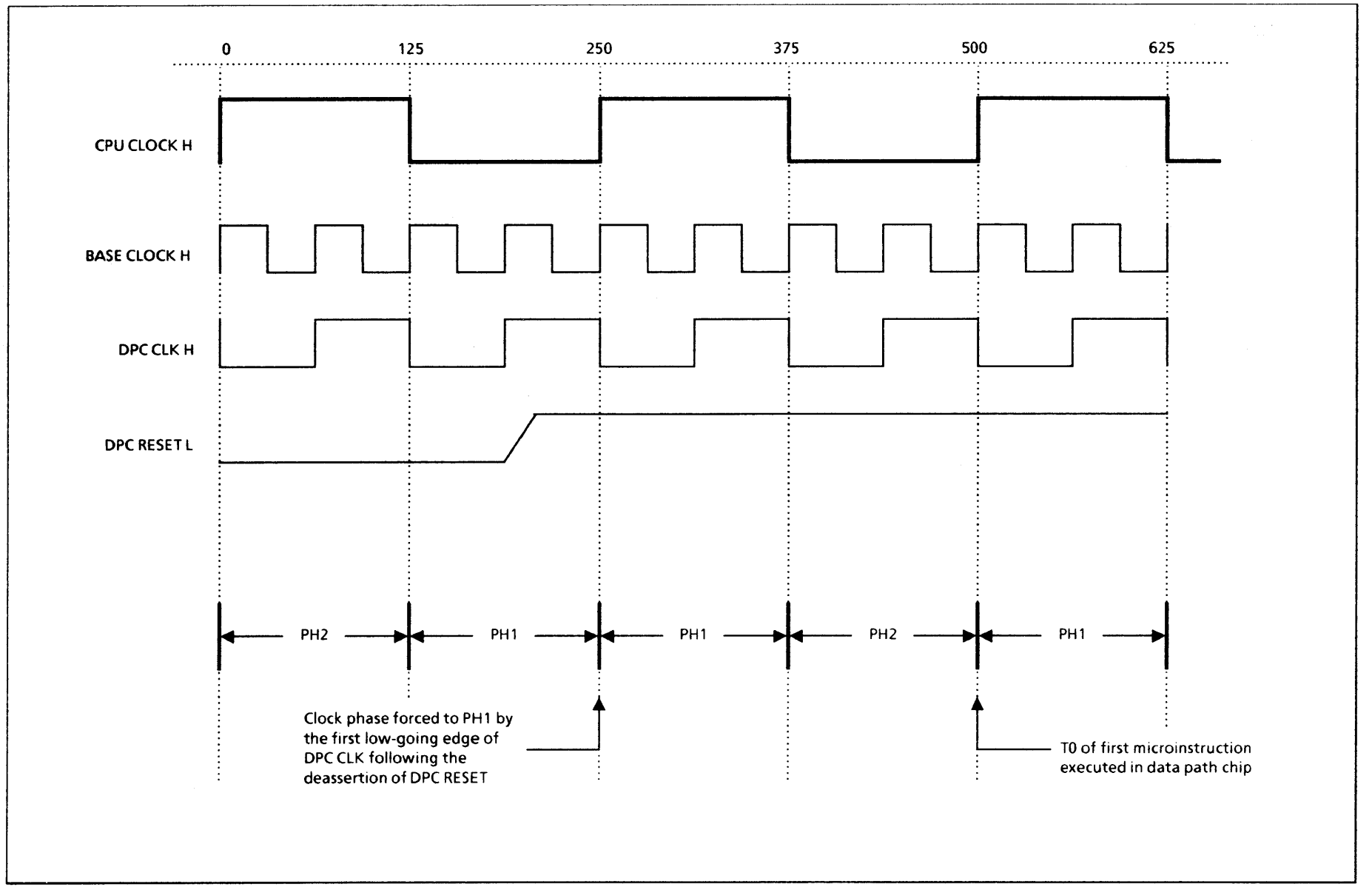


Figure 6-7. Data Path Chip Timing Diagram

Control Store Register

The control store register (CSR) is the 21-bit register that holds the data path control field (bits <36:16>) of the current microinstruction. The data path control field (DAPA CS <36:16>) is loaded into the CSR at the leading edge of every PH1.

Parity Generator

The parity generator on the data path chip computes parity on the 21-bit data path control field contained in the control store register. The result is driven on the parity output pin to the parity checker external to the chip. Odd parity is generated; the parity bit is set to one if the sum of the one bits in the data path control field is even.

Size Control

The chip supports three data types: byte, word, and longword. The size of the operation performed in the data path chip is controlled by the CC/DT field of the current microinstruction and the size register.

For all microinstructions except three, the CC/DT field determines the size information that is sent to the data path chip, and is encoded as follows:

0, 1, or 2	data type is longword
3	data type is determined by the size register

The three microinstructions that are the exceptions are Memory Request, I-stream Request, and Decode. For Memory Request and I-stream Request microinstructions, the CC/DT field determines the size information that is sent to the data path chip, but the field encoding is interpreted this way:

0 byte
1 word
2 determined by size register
3 longword

For Memory Requests and I-stream Requests, the size information in the CC/DT field is also sent to the memory controller as the data type of the memory request.

For opcode Decode microinstructions, the CC/DT field (bits <38:37>) is ignored, and the size register is loaded from the decode ROM signals DAPF DT1/RMODE and DAPF DT0/SP.

For operand specifier Decode microinstructions, the size register may be loaded from the CC/DT field, signal names DAPC CS REG <38:37>. In this case, the CC/DT field controls the loading of the size register and is encoded this way:

0 load 0 into size register, data type byte
1 load 1 into size register, data type word
2 size register unaffected
3 load 3 into size register, data type longword

At the leading edge of every PH1, the data type for the current microinstruction is sent to the data path chip via the size control pins, SIZE1 and SIZE0. The signals DAPE DPC DT1 and DAPE DPC DT0 carry the encoded data type to the pins. The encoded data type on the size control pins is longword for all microinstructions (except the special group of three) if the CC/DT field does not contain the value 3.

If the current microinstruction CC/DT field does contain the value 3, or the current microinstruction is one of the special three and the CC/DT field contains the value 2, the data type on the size control pins is the

same as the data type currently stored in the size register and is encoded as follows:

Size Register	SIZE1	SIZE0	Data Type
0	0	0	byte
1	0	1	word
2	1	0	not used
3	1	1	longword

The data type specified by the size control pins affects the writing of the general purpose registers and the setting of the ALU condition codes; the shift microinstructions are not affected.

Data Path Chip Buses

The data path is 32 bits wide and contains two 32-bit buses called bus A and bus B. The buses are precharged during PH2 and are selectively discharged during PH1.

Bus A is used for short operand sources, with the following exceptions:

- During the Multiply Step microinstruction, bus A transfers RESULT0 back to the ALU.
- During the Decode microinstruction for a macroinstruction opcode (IRD), bus A transfers the PC to the register save stack if bit <25>, the register save stack initialize bit, is set.

Bus B is used for long operand sources and short operand destinations, with the following exceptions:

- During the Moveout microinstruction, bus B is used for the short operand source.
- During the Decode microinstruction, bus B transfers the data on the external data bus to the pointer registers.

- During the Restore microinstruction, bus B transfers the contents of the register save stack to the specified general purpose register (GPR).
- During the I-stream Request microinstruction, bus B transfers the PC to the external data bus.

Arithmetic and Logic Unit

The ALU reads two input longwords, one from bus A and one from bus B, operates on the longwords, and writes the result into one of the result registers: either RESULT0 or RESULT1. The ALU microinstructions are those with opcodes 0 through F (hex) and are defined in Table 5-4.

Barrel Shifter

The barrel shifter provides four primitive functions: left shift, right shift, arithmetic right shift, and double right shift (extract).

The barrel shifter concatenates two longwords, one from bus A, bits A<31:0>, and one from bus B, bits B<31:0>, to form a quadword. The higher-order longword is B<31:0>. The longword result, R<31:0>, is extracted as 32 consecutive bits from the quadword and is written in register RESULT2. The bit-offset of the 32 consecutive bits extracted from the quadword is determined by the shift count, which can come from either the shift count register, or from a literal in the short operand field. The range for the shift count is 0–31. Table 6-4 summarizes the input configurations and extract counts for the four primitive functions of the barrel shifter. "LOP" means long operand, "SOP" means short operand, and N represents the shift count.

Table 6-4. Barrel Shifter Functions

Function	B<31:0>	A<31:0>	Extract Count
left shift	LOP	zeros	32 – N
right shift	zeros	LOP	N
arith. right shift	sign ext.	LOP	N
double shift	SOP	LOP	N

Register File

The register file is a RAM array containing 47 registers, each 32 bits wide. The registers can be read from bus A and bus B, and can be written from bus B. The register addresses are 00 through 0E and 10 through 2F; register address 0F is the program counter. Registers with addresses 00–0E are general purpose registers (GPRs) and may be written as bytes, words, or longwords. When a GPR is written with a length less than longword, the higher order portion is not affected. Registers with addresses 10–2F are always written as entire longwords.

Table 6-5 briefly describes the registers contained on the data path chip. Registers with addresses 30–5F are described in more detail later in this section.

Table 6-5. DPC Registers

Address	Register Name	B bus	A bus	Description
00–0E	GPR(0)–GPR(14)	R/W	R	Macrolevel general purpose registers; writable as B, W, L
0F	PC	R/W	R	program counter
10–2F	TEMP(0)–TEMP(31)	R/W	R	General purpose temporary registers
30	RESULT0	R	R	Result register 0 from ALU
31	RESULT1	R	R	Result register 1 from ALU
32	RESULT2	R	R	Result register from barrel shifter
33	SC	R/W		shift count register
34	PTR1	R/W	R	Pointer register for first operand specifier; pointer registers are zero-extended when read
35	PTR2	R/W	R	Pointer register for second operand specifier; pointer registers are zero-extended when read
36	*PTR1	indirect	indirect	Select working register specified by PTR1 register
37	*PTR2	indirect	indirect	Select working register specified by PTR2 register
38	TMRCSR	R/W		timer control and status register
39–3F	RSVD			Reserved internal to chip
40–5F	ROM	R		Constants ROM

Program Counter

The macrolevel program counter (PC) is GPR(15); it is readable from both buses and writable from bus B. An entire longword must always be written to the PC.

The PC can be incremented by 1, 2, or 4. It is incremented by hardware on the data path chip for each of the following situations:

- An opcode Decode microinstruction is executed.
- An operand specifier Decode is executed.
- The long operand of the current microinstruction specifies IB.BYTE
- An I-stream Request microinstruction is executed.

Result Registers

The result of any ALU operation (except Compare) is stored in one of two 32-bit ALU result registers, RESULT0 or RESULT1, as specified by the result bit (bit<31>) in the microinstruction (see Figure 5-2). RESULT0 and RESULT1 can be addressed using the short or long operand, and the register contents driven onto either bus A or bus B.

RESULT0 and RESULT1 combine to form a 64-bit wide shift register which is used for Multiply Step microinstructions. RESULT0 is the high-order longword. During a Multiply Step microinstruction, RESULT0 and RESULT1 are shifted right so that the least significant bit (LSB) of RESULT0 becomes the most significant bit (MSB) of RESULT1. (For more information about Multiply Step, see the section titled "Multiply Step" in Chapter 5.)

The result of any barrel shifter operation is stored in the 32-bit wide shift result register: RESULT2.

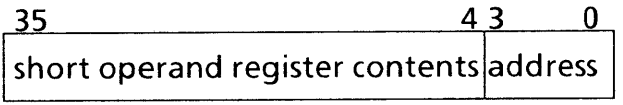
ROM

There are 32 constants stored in ROM. ROM locations are addressed by the long operand and are read onto bus B. (See Table 5-8, addresses 40–5F.)

Register Save Stack

The register save stack is a pushdown stack capable of holding seven 36-bit items.

When bit <30> of the microinstruction is set, both the contents of the register specified by the short operand, and the low four bits of the register address are pushed onto the register save stack in the following format:



The following microinstructions are exceptions to this: Decode, NOP, Restore, Clear Save Stack, Multiply Step, I-stream Request, and Memory Request. During these microinstructions, bit <30> is ignored, and nothing is saved on the register save stack.

The register save stack is popped using the Restore microinstruction, and is initialized by the Clear Save Stack microinstruction or by setting the register save stack initialize bit (<25>) in a Decode microinstruction. (For more information about the register save stack initialize bit, see Table 5-7.)

Pointer Registers

Two 6-bit pointer registers, PTR1 and PTR2, can be used to indirectly address registers 00–1F.

The pointer registers can also be used directly as source operands. When this is the case, their contents are zero-extended.

PTR1 and PTR2 can be written from bus B, and read on either bus A or bus B. One of the pointer registers is always written during a Decode microinstruction with the number of the register specified in the operand specifier, or with literal data if the operand specifier is a short literal; bit <26> of the microinstruction selects which one (see Table 5-7). During a Decode microinstruction, data from the DBUS (data bus, Figure 6-1), are written into PTR1 for bit <26> = 0, and into PTR2 for bit <26> = 1.

Shift Count Register

The shift count register is a 5-bit register that controls the shift amount in a barrel shifter operation. The shift count register is readable and writable via bus B, and it is zero-extended when read.

Interval Timer and TMRCSR

The data path chip contains an interval timer that is available for use by any macrolevel software running on the system. The interval timer is controlled by the timer control/status register, TMRCSR.

The interval timer is a 16-bit counter which is clocked once every microcycle. (One microcycle is 250 ns.) The counter is loaded with the constant 40,000, which causes the counter to overflow once every 10 msec. Every time the counter overflows, TMRCSR <1> is set, and the counter reloads itself with the constant. TMRCSR <1> stays set until it is written with a zero via microcode.

TMRCSR <0> is the interrupt enable bit. The timer interrupt pin of the data path chip is the logical AND of TMRCSR <0> and TMRCSR <1>. When TMRCSR <0> and TMRCSR <1> are both set, the signal DAPH TIMER REQ is sent from the timer interrupt pin to the interrupt control logic on the external data path. DAPH TIMER REQ is the signal represented by the label "timer req" in Figure 6-1.

Writing a zero to TMRCSR <1> clears the interrupt. Writing the timer control/status register has no effect on the contents of the counter. The Reset signal loads the constant into the counter, and clears TMRCSR <0> and TMRCSR <1>.

Condition Codes

The condition codes are the N, Z, V, and C bits. The N, Z, V and C bits are set when an ALU microinstruction (opcodes 1–F) or a Multiply Step microinstruction (opcode 1B) is performed.

During a logical microinstruction (opcodes 1–7 hex), both the V bit and the C bit are cleared. The Z bit is set for a Shift microinstruction (opcodes 10–16 hex).

The N, Z, V, and C bits are set according to the size of the operand as specified by the size control pins. Table 6-6 describes how the condition codes are set.

Table 6-6. Data Path Chip Condition Codes

CC	Affected by Opcodes	Description
N	1–F or 1B	N is set when the MSB of the result = 1; that is, the result is negative. For Compare, N is set if N XOR V is true.
Z	1–F or 1B	Z is set when the result = 0.
Z	10–16	Z is set when RESULT2<0> = 1.
V	1–F or 1B	V is set when an overflow on arithmetic operations (opcodes 8–F and 1B) occurs. Overflow is implemented by taking the XOR of the carry in and carry out of the MSB of the ALU. V is cleared on logical operations (opcodes 1–7).
C	1–F or 1B	For addition and multiplication (opcodes 8–A and 1B), C is the carry out from the MSB of the ALU. For subtraction and Compare (opcodes B–F), C is the complement of the carry out. C is cleared on logic operations (opcodes 1–7).

I/O Port

The external registers (addresses 3C–7F) are located outside the data path chip in the external data path; they can only be referenced in the microinstruction long operand. The I/O port is the interface between the external data path and the data path chip. The I/O port

is connected to bus B. Table 6-7 briefly describes the registers located outside the data path chip.

There are 32 data path chip pins that connect the chip to the external data bus (DBUS). These pins carry the bidirectional tri-state signals labeled BUS DBUS <31:00>. The outputs from these pins are disabled during Reset. The I/O port drives the DBUS pins only for the Moveout, I-stream Request, and Memory Request microinstructions.

Table 6-7. External Registers

Address	Register Name	Read/Write	Description
60	CON.DATA	R/W	UART data register
61	CON.STATUS	R/W	UART status register
62	CON.MODE	R/W	UART mode register
63	CON.CMD	R/W	UART command register
68	size register	R/W	Bits <1:0> only; zero-extended when read
69	index register	R/W	Bits <3:0> are index register, read/write. On writes only, bits <7:0> are the low eight address bits of the boot EPROM. On reads only: bit 7 transmit done bit 6 receiver ready bit 5 control store parity error bit 4 Microverify jumper
6A	PSL.MODE	R/W	Bits <1:0> only; zero-extended on reads.
6B	MISC.WR	Write	Bits<7:5> diagnostic LEDs Bit 4 break detect enable Bit 3 UART transmit interrupt enable Bit 2 UART receive interrupt enable Bit 1 request arithmetic trap Bit 0 send Q22 bus init. Reading address 6B clears console mode.
6C	PSL.EN	Write only	Writing bits <5:4> to this register sets the PSL IV and T bits, respectively.
6C	FUNCTION	Read only	These bits indicate the status of the saved memory request: Bit 1 When set, the memory request mode is kernel; when clear, the mode is current. Bit 0 When set, access type is DAP to MCT (write); when clear, MCT to DAP (read).
6D	PSL.IPL	Write only	Bits <4:0>. Also, writing the low-order six bits to address 6D selects the high six address bits for the boot EPROM.
6D	INT.SRC	Read only	Interrupt source register; encoding: 0, 1 reserved; 2, 3, 10 timer request; 4, 5 reserved; 6 Q22 bus level 4 7 power failure; 8 write timeout; 9 Q22 bus level 7; 11 Q22 bus level 6 12 Q22 bus level 5; 13 console receive; 14 console transmit; 15 no interrupt
6E	PSL.CC	R/W	Bits <3:0>; zero-extended when read.
6F	ALU.CC	R/W	Bits <3:0>; zero-extended when read. Writing bit <4> to address 6F sets console mode.

Table 6-7. Continued

Address	Register Name	Read/Write	Description
70	HD.SID	Read	System ID register switch pack, bits <7:0> only.
71	SWITCHES	Read	Option switch pack, bits <7:0> only.
72	MISC.RD	Read	See 6B.
73	BOOT.ROM	Read	A single byte from the boot EPROM.
74–77	RSVD		Reserved
78	IB.BYTE	Read	Read a byte from the I-stream; PC incremented by one.
79	IB.WORD	Read	Read a word from the I-stream; PC incremented by two.
7A	IB.SIZE	Read	Read from the I-stream and increment the PC; data type determined by current contents of size register.
7B	IB.LONG	Read	Read a longword from the I-stream; PC incremented by four.
7C–7F	MEMORY.DATA	R/W	External memory.

Transferring Data

Transferring data within the data path module is the fourth of the eight functions that the data path module performs. The hardware components are the internal data bus (ID bus) and the data bus (DBUS), the sign-extension logic, the ID bus latch, the ID MUX, the IBYTE buffer, the miscellaneous register, ID bus control, and zero-generator. These components transfer data within the DAP module. The following paragraphs describe each of these components in turn, and the ID bus timing.

Internal Data Bus

There is an 8-bit data path on the DAP module used to access the registers that must be visible to external hardware, such as the console UART and the switch packs. This data path is also used during instruction decode to pass operand specifier information into the data path chip. The information transfer portion of this data path is a tri-state bus called the internal data bus (ID bus).

All of the tri-state enables on the ID bus are disabled during T1. The control outputs are changed during this time and the bus re-enabled at T2. Data are always clocked into the ID bus destination at T0.

Data may be driven onto the ID bus from one of several sources, and may be written from the ID bus to one of several destinations. The following components can be sources or destinations: size register, ALU and PSL condition code PALs, index register, console UART, MISC register, and the data path chip (via the ID bus latch when it is a source and via the ID bus input latch when it is a destination).

These components can only be sources: the boot EPROM and two switch packs which are inputs to the ID MUX, the interrupt source register, and the IBYTE register. The three separate registers that make up the hardware PSL: the current mode register (PSL.MODE), PSL enable, and the IPL register, can only be destinations.

Data Bus

The DAP module communicates with the MCT module over a 32-bit tri-state bus called the memory data bus, implemented in a 50-pin, over-the-top cable. The extension of this bus on the DAP module is the data bus, or DBUS. The DBUS transfers data between the data path chip and the memory controller, and between the data path chip and the rest of the DAP module. There is buffering between the DBUS and the memory data bus (the MD bus latches in Figure 6-1) to provide the required drive for the signals transmitted over the cable.

Sign-Extension

The DBUS may also be driven by the sign-extenders. The sign-extend logic is used when displacements from the instruction stream are read into the data path chip. Word displacements are read from the memory controller over the memory data bus, while byte displacements are read from the IBYTE register directly. The sign-extension control enables the sign-extenders for a read from the ID bus, or for a word displacement read during an I-stream Request microinstruction.

The input to the sign-extenders is bit 7 from the internal data bus (ID bus), bit 15 from the data bus, and information about the data type. The output is data bus

bits <31:16> for words (BUS DBUS <31:16>) or data bus <31:08> for bytes (BUS DBUS <31:08>).

ID Bus Latch

This latch holds data being driven from the low eight bits of the data path chip. The ID bus latch is needed because of the data hold times required by the UART.

ID MUX

The ID bus multiplexer gates one of the following sets of inputs onto the ID bus:

- miscellaneous register <7:0>
- boot EPROM <7:0>
- option switches <7:0>
- system ID switches <7:0>

The output of the ID MUX is ID bus bits <7:0>, labeled BUS ID <07:00>.

IBYTE Buffer

The IBYTE buffer is a latch located between the IBYTE register and the ID bus. The contents of the IBYTE register are driven onto the ID bus through the IBYTE buffer.

The contents of the IBYTE register are read on the ID bus when the long operand of the current microinstruction specifies IB.BYTE, which is the IBYTE register's unique address.

The contents of the IBYTE register are also driven on the ID bus during operand specifier decodes, and stored in one of the two pointer registers on the data path chip. When the operand specifier mode is not short literal, bits <5:4> of the IBYTE register are forced to zero to extract the register number. (Except for literal mode,

operand specifier bits <3:0> always specify a register number. The register number is always saved for an operand specifier decode.) The high two bits of the IBYTE register contents (the operand specifier) do not need to be set to zero because the pointer registers are only six bits wide.

The input to the IBYTE buffer is DAPF I BYTE <7:0>. The output is BUS ID <07:00>.

Miscellaneous Register

This is a read/write register that contains various control bits. When a write to this register is performed, the register address specified is 6B; when a read from the MISC register is performed, the register address specified is 72. The register bit definitions are the same regardless of the operation.

The input to this register is the ID bus bits: BUS ID <07:00>. The output is eight lines to the ID MUX; some of these lines are also used for various control functions. The MISC register bit definitions are as follows.

07:05	LED bits Diagnostic LEDs 1, 2, and 3 are lit by writing zeros to these bits.
04	break detect enable When this bit is set, a break condition on the serial line causes a HALT.
03	UART transmit interrupt enable.
02	UART receive interrupt enable.
01	arithmetic trap request When this bit is set, a trap is taken at the next instruction decode.

00 send Q22 bus init
This bit is used to initialize the I/O bus when requested by a MTPR instruction.

ID Bus Address Decode Logic

The operation of the ID bus is controlled by the opcode and long operand field of the microinstruction. The ID bus address decode logic receives bits CS <36:32> from control store (the microinstruction opcode), bits DT1/RMODE and CC CLASS 0 from the decode ROMs, and CS<20:16> from control store (the microinstruction long operand). With these inputs, the ID bus address decode logic generates signals to control read and write operations on the ID bus. The microinstruction opcode specifies the direction of the data transfer, and the long operand is used as an address to determine if an ID bus register is the source or the destination of the data to be transferred.

Because of the pipeline in the data path chip, the timing on the ID bus is different for reads and writes. On a read operation, the data are driven onto the ID bus at T2 of the microinstruction requesting the read. The difference on write operations is that the data are not available from the data path chip until just before T2 of the microinstruction following the one requesting the write. The long operand and some of the control information is stored in a pipeline register which then provides the necessary write enable signals to the destination registers one cycle later. Figure 6-8 shows the timing for reads from ID bus registers. Figure 6-9 shows the timing for writes to ID bus registers.

Logic to decode the microinstruction opcode is part of the ID bus address decode logic; the microinstruction opcode needs to be decoded to allow the data path elements to behave differently depending on the

operation required. For example, the data path needs to detect Memory Request and I-stream Request opcodes. The inputs to the microinstruction opcode decode logic are the microinstruction opcode field CS <36:32>, and CS <24> to differentiate between operand specifier and opcode decodes. The outputs are as follows.

- A signal named NO CC OP informs the condition code logic that the condition codes are not changed for this instruction.
- A signal named DECODE indicates that the current microinstruction is a Decode.
- A signal named MEMORY OP informs the memory controller that the current microinstruction is a memory function.
- A signal named MOVEOUT indicates that the current microinstruction (Moveout, Memory Request, or I-stream Request) causes data to be driven out of the data path chip.
- A signal named NOT LIT indicates that the current operand specifier is not a short literal.
- A signal named CLR IB VALID informs the IBYTE control PAL that the contents of the IBYTE register will not be valid at the end of the current microinstruction.
- A signal named SET OPEN LATCHES controls the data latches driving the memory data bus and the internal data bus.
- A signal named DECODE & RMODE indicates that the current operand specifier is register mode and not PC.

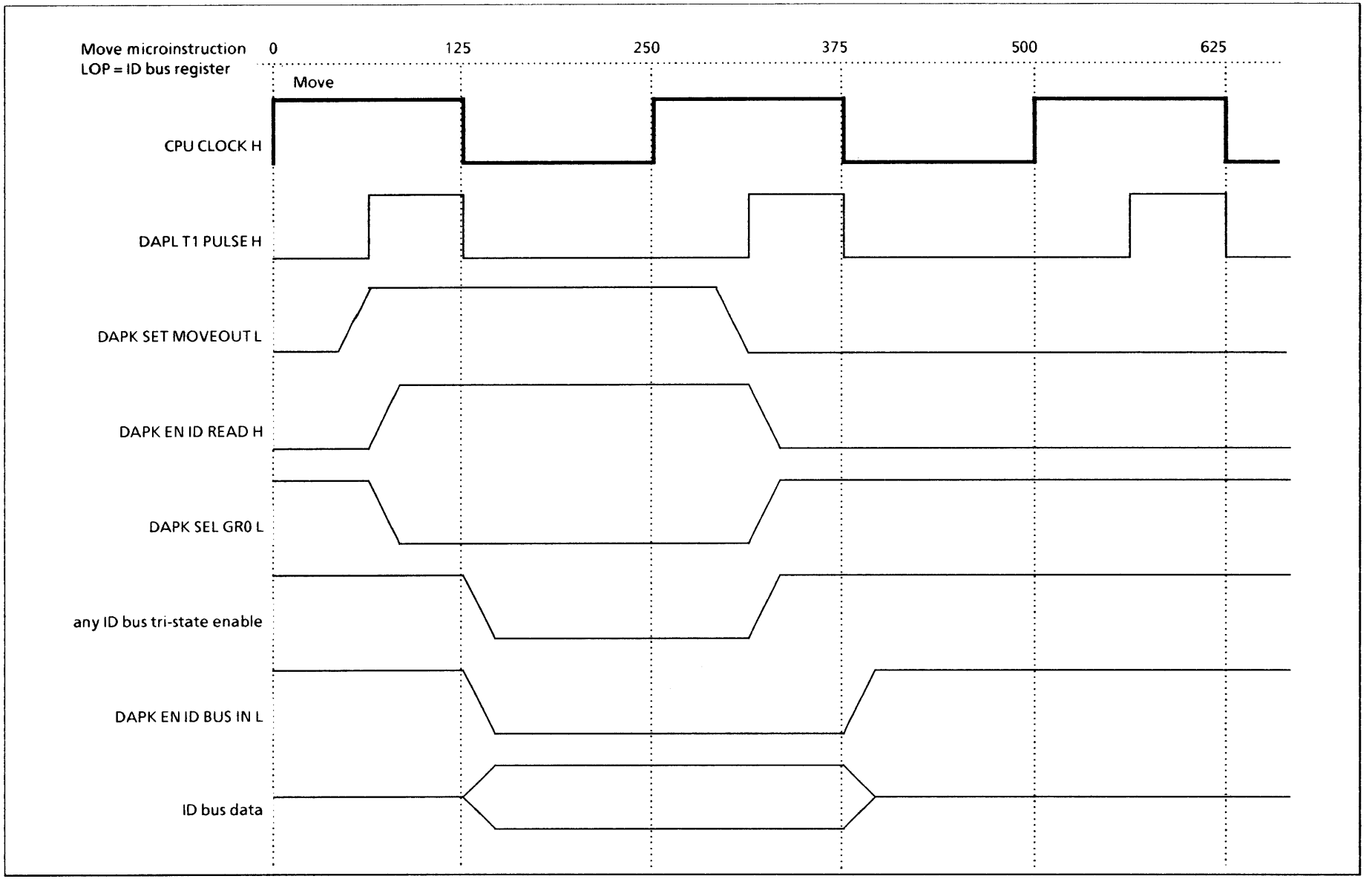


Figure 6-8. Timing of Read from ID Bus Register

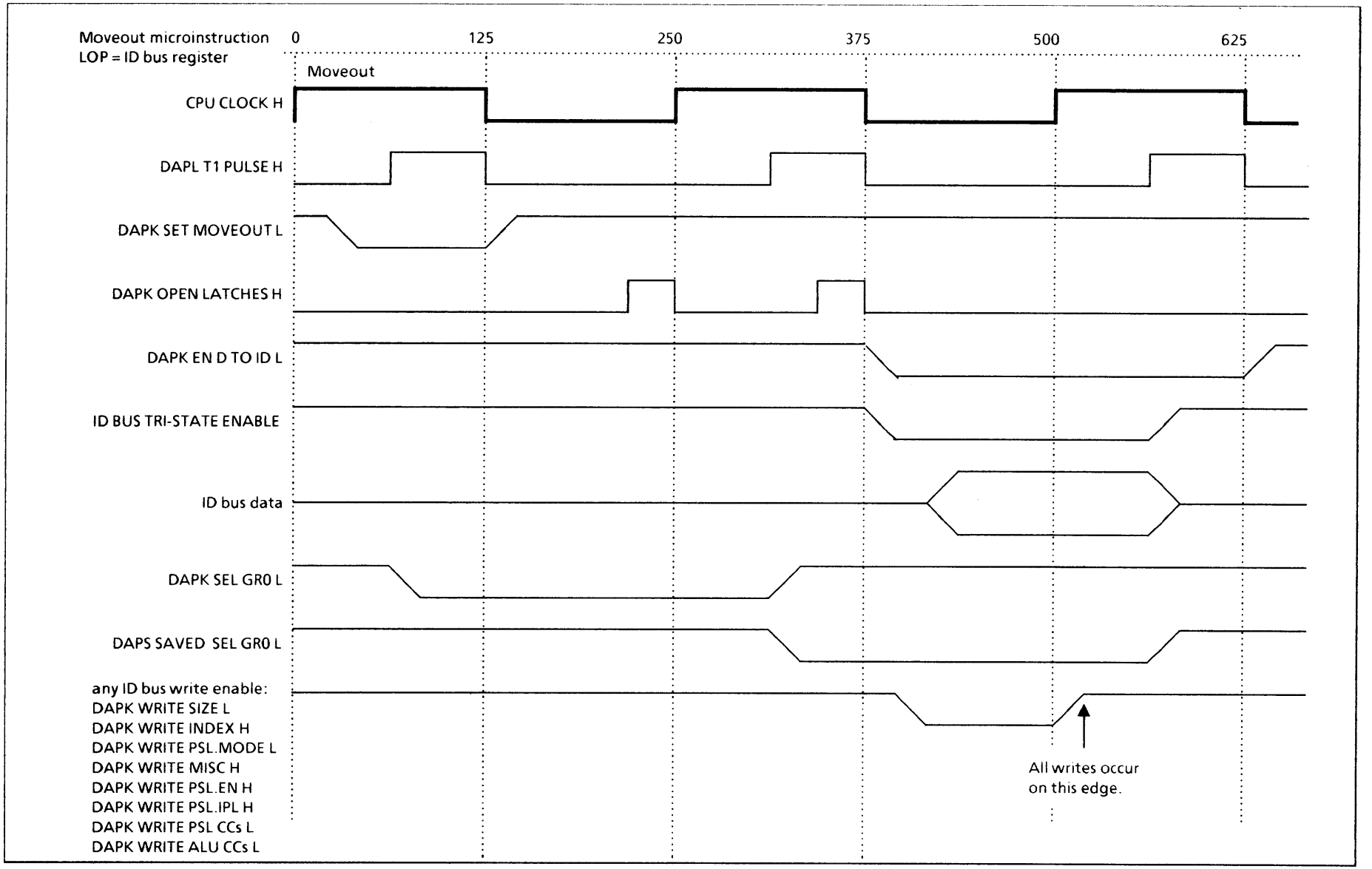


Figure 6-9. Timing of Write to ID Bus Register

Zero-Generator

Several of the readable registers on the ID bus contain less than eight bits. The microcode requires that these registers be zero-extended when read, so a zero-generator is connected to the ID bus. The zero-generator is implemented as a PAL. It is enabled when any register containing less than eight bits is read; the zero-generator drives zeros on the unimplemented bits of that register.

The input to the zero-generator is the low-order three bits of the microinstruction long operand and some control signals. The output is bits BUS ID <07:02>. If the ID bus register being read is INT.SRC (interrupt source), PSL.CC, or ALU.CC, BUS ID <07:04> are driven onto the ID bus as zeros. If the ID bus register being read is the size register, PSL.MODE (current mode register), or FUNCTION (memory function status), BUS ID <07:02> are driven onto the ID bus as zeros.

Processing Interrupts

Processing interrupts is the fifth of the eight functions that the data path module performs. The hardware components are the interrupt priority level (IPL) register, the interrupt control logic, the priority encoder, and the interrupt source register. The following paragraphs describe each of these components in turn.

IPL Register

The interrupt priority level register stores the current processor priority. This priority is used by the

interrupt control logic to determine if an interrupt request is to be granted.

The IPL is changed when an interrupt is taken, when a MTPR to IPL or REI macroinstruction is executed, or during certain exception conditions. These instructions use a temporary register on the data path chip to store the new IPL until it is written into the IPL register. The IPL register is written at T0 from ID bus bits <4:0>.

Interrupt Control Logic

The interrupt control logic on the data path module informs the microcode of pending interrupt requests. These requests can be generated by local hardware (for example, power fail) or can come from the Q22 bus. The priority encoder and the interrupt source register are actually part of the interrupt control logic; this logic is always enabled.

The interrupt request lines from the Q22 bus are received in a bus receiver, synchronized to the CPU clock (DAPL CPU CLOCK) and sent to the priority encoder. Interrupt request signals from internal sources are also sent to the priority encoder.

The hardware compares the IPL of the Q22 bus device requesting the interrupt with the current processor IPL. If the IPL of the Q22 bus device is higher, the interrupt is served at IPL 17 (hex). The microcode that services Q22 bus interrupts then reads the interrupt source register to determine which Q22 bus device actually caused the interrupt.

Priority Encoder

All active interrupt requests are prioritized in the priority encoder. The encoded output value is compared with the interrupt priority level (IPL) from the

hardware PSL. If the priority of the request is greater than the current IPL, the interrupt request flag (DAPN INT REQ) is sent to the OR MUX and jump control logic in the microsequencer.

If an interrupt request is pending during an IRD (macroinstruction opcode decode), it causes a microtrap. INT REQ is one of the OR MUX inputs (see Table 5-6). Since the OR MUX is enabled for an IRD, the OR MUX output is 0100 (binary) if an interrupt request is pending and no other condition is present; the other next microaddress bits are forced to zeros (see Table 6-1). Thus, if an interrupt request is pending and an IRD is executed, a microtrap is taken to control store address 0004 (hex). A microinstruction routine to handle interrupt requests starts at this address.

The comparison between the encoded output value from the priority encoder and the current IPL is done in a PAL; this PAL also contains the interrupt source register (INT.SRC).

Interrupt Source Register

The encoded output value from the priority encoder is the input to the interrupt source register. This value is compared with the processor IPL; the comparison produces a 4-bit code which is loaded into the interrupt source register if the request priority is higher than the current processor IPL. The microcode identifies the source of an interrupt request by reading this 4-bit code in the interrupt source register. The register encoding is shown in Table 6-8.

Table 6-8. Interrupt Source Register Encoding

Interrupting Event	IPL (hex)	INT.SRC Register
none		1111
power failure	1E	0111
write timeout	1D	1000
Q22 bus level 7	17	1001
timer request	16	1010
timer request	16	0010
timer request	16	0011
Q22 bus level 6	16	1011
Q22 bus level 5	15	1100
console receive	14	1101
console transmit	14	1110
Q22 bus level 4	14	0110

When the interrupt source register is read by the microcode, the following interrupt requests are cleared by the hardware if they are the highest priority: write timeout, console receive, and console transmit.

The output from the interrupt source register is bits BUS ID <03:00> and the interrupt request signal to the microsequencer.

Communicating with the Console Terminal

Communicating with the console terminal is the sixth of the eight functions that the data path module performs. The console port consists of an EIA standard RS232/423 line interface and a 2661 UART. The external connection to this interface is through a 10-pin cable header mounted on the DAP board. The hardware components are the console UART and registers, the UART buffer, option switches, the charge

pump, and break and halt detection. The following paragraphs describe each of these components in turn.

Console UART

The console UART and the RS232/423 line interface provide the connection to the console terminal. The UART is connected through the UART buffer to the ID bus, and can be read or written directly by the microcode. The baud rate is selected in the option switches and can be set for 300, 1200, 9600 or 19,200 baud. The transmitter and receiver always operate at the same speed. The microcode reads the option switches during power up and programs the UART for the selected baud rate.

The console UART can request interrupts for either "transmit done" (DAPP XMIT DONE) or "input ready" (DAPP REC RDY).

The UART clock (DAPP UART CLK) comes from a 5.0688 MHz crystal oscillator that is driven directly into the UART.

Console UART Registers

The UART has programmable mode and status registers to select different speed and character length options. There is also a break detect; the signal DAPP BREAK is asserted when the BREAK key on the console terminal is pressed.

The UART mode and status registers are written by the microcode on power up to allow the UART to correctly interface with the console terminal. Four addresses, 60–63 hex, are assigned to the UART in the long operand address space. On power up, the microcode initializes the UART to operate in the mode required. Once the UART is initialized, it is accessed only to read and write characters to the console terminal. Table 6-9

gives the UART register addresses and a brief description.

Table 6-9. UART Registers

Address (hex)	Register	Description
60	CON.DATA	Contains character received or to be transmitted
61	CON.STATUS	Contains UART status
62	CON.MODE	Consists of two mode registers that set operating conditions
63	CON.CMD	UART command register; sets operating mode

The following paragraphs describe these registers in more detail.

UART Data Register

CON.DATA is an eight bit register that contains the ASCII character to be transmitted to the console terminal, or the ASCII character received from the console terminal.

If an ASCII character is to be transmitted to the console terminal, the character written into CON.DATA is ID bus bits <07:00>; BUS ID <07:00> are written into the UART buffer from the ID bus, then transmitted to CON.DATA in the console UART.

Similarly, an ASCII character received from the console terminal and stored in CON.DATA is read onto the ID bus as BUS ID <07:00> through the UART buffer.

UART Status Register

CON.STATUS contains bits that indicate the status of the receiver and transmitter. The bits are defined as follows.

7:6 data set status
MicroVAX I does not use the modem control feature of the 2661 UART, so these bits are ignored by the microcode.

5 framing error
This bit is set when a stop bit is not received following the last data bit of a received character. Bit 5 is cleared by writing a one to the reset error bit in the command register (CON.CMD <4>).

4 overrun error
This bit is set when an incoming character is received before the previous received character has been read by the microcode. This bit is cleared by writing a one to the reset error bit in the command register (CON.CMD <4>).

3 parity error
This bit is not used.

2 data set change
This bit is not used.

1 receiver ready
This bit is set when a character is received from the serial line. It is cleared when the UART data register (CON.DATA) is read.

0 transmit done
This bit is set when the transmitter has completed transmission of a character. It is

cleared when the UART data register (CON.DATA) is written.

External forms of CON.STATUS bits <1> and <0> are available as the signals REC RDY and XMIT DONE. These signals generate the interrupt requests for "input ready" and "transmit done," respectively.

UART Mode Registers

Mode registers 1 and 2 define the general operational characteristics of the UART and are accessed only during power up. The two mode registers are accessed by performing either the read or the write operation at that address twice. The first operation accesses mode register 1, and the second accesses mode register 2.

Mode Register 1. This register is initialized to 4E (hex) in the MicroVAX I system to define the following setup conditions:

bits <7:6> stop bit length
These bits are initialized to 01 to define one stop bit at the end of the eight-bit character being sent or received.

bits <5:4> parity control
These bits are initialized to 00 to define no parity checking.

bits <3:2> character length
These bits are initialized to 11 to define 8-bit characters.

bits <1:0> baud rate multiplier
These bits are initialized to 10 to define an asynchronous, 16X clock rate.

Mode Register 2. This register is used to set operating conditions and the baud rate of the UART. Only four

baud rates are supported. The bit definitions for mode register 2 are:

bits <7:4> clock source, break enable
This bit is initialized to 1111 to define the internal baud rate generator as the clock source, and to enable break detection.

bits <3:0> baud rates:

0101	300 baud
0111	1200 baud
1110	9600 baud
1111	19200 baud

UART Command Register

CON.CMD is used to enable the UART and set the operating mode to either normal or self-test. The bits are defined as follows.

7:6 operating mode
- 00 normal operation
- 01 not used
- 10 local loop back
  In this mode, a character written to the transmitter will be received by the receiver.
- 11 not used

5 request to send (RTS)
This bit is initialized to a one.

4 reset error
Writing a one to this bit clears the receive error flags in the status register (CON.STATUS).

3 force break
This bit is initialized to zero.

2 receiver enable
This bit is initialized to a one.

1 data terminal ready (DTR)
This bit is initialized to a one.

0 transmitter enable
This bit is initialized to a one.

Initializing the UART

The correct sequence must be used to set up the UART initial conditions. The sequence is:

1. write mode register 1
2. write mode register 2
3. write command register

If the baud rate is to be changed, the UART must first be disabled by clearing the receiver and transmitter enable bits in the command register (CON.CMD <2> and <0>). The UART must then be reset following the above sequence.

UART Buffer

The UART buffer is a bus transceiver. The input to the UART buffer is bits BUS ID <07:00> when a write to the UART occurs. The eight bits stored in the UART buffer are then written into the UART.

When a read from the UART occurs, the input to the UART buffer is the eight bits from the UART register. The eight bits stored in the UART buffer are then driven onto the ID bus as BUS ID <07:00>.

Option Switches

The option switches (an eight-switch DIP) select the baud rate, break detect enable, the system recovery

action, the console terminal type, and the bootstrap search order. The output from the switches is eight data lines to the ID bus MUX. The switch definitions are as follows; the default switch settings are in bold.

<8:7> baud rate selection
These switches specify the console terminal baud rate:
0 9600
1 19200
2 300
3 1200

<6> **0**, reserved

<5> break detect enable
This switch determines the state of bit <4> in the MISC register:
0 break disabled (MISC <4> = 0)
1 break enabled (MISC <4> = 1)

<4:3> recovery action
These switches specify the actions to be taken when the machine powers on:
0 warm start, boot, halt
1 boot, halt
2 warm start, halt
3 halt

<2> console terminal
This switch identifies the type of console terminal connected to the system.
0 VT100 compatible
1 bit-mapped graphics terminal

<1> bootstrap search order
This switch determines which devices are searched during bootstrap.

0 search order: diskettes, disks, MRV11 PROM, DEQNA

1 search order: MRV11 PROM, DEQNA

The recovery actions are explained in more detail in the section titled "Powering Up" in this chapter.

– 12 Volt Generator

The RS232/423 drivers require a −12 volt power source; −12 volts is not available from the system power supply. Therefore, the DAP module contains a charge pump circuit to generate this voltage. The circuit operates by alternately charging two capacitors to +12V and using them to charge a third capacitor; the −12 volt output is taken from this third capacitor.

Break and Halt Detection

There are three situations in which break or halt detection needs to occur.

- The HALT button on the MicroVAX I system front panel is pressed.
- The BREAK key on the console terminal is pressed.
- A Halt macroinstruction is executed in kernel mode.

Pressing the HALT button on the front panel asserts the Q22 bus halt line, BHALT. This is received by a bus receiver. The output of the receiver is the signal DAPP RCVD HALT. DAPP RCVD HALT generates the signal DAPS HALT REQ.

Pressing the BREAK key on the console terminal asserts the signal DAPP BREAK. If the break detect enable bit in the MISC register is set (bit <4>), DAPP BREAK generates the signal DAPS HALT REQ. (The setting of option switch 5 determines the state of bit <4> in the MISC register.)

Once DAPS HALT REQ is asserted because either the HALT button or the BREAK key was pressed, it generates two signals: DAPE CONSOLE MODE which is one input to the jump MUX, and DAPE T BIT OR CON which is one input to the OR MUX.

When DAPE T BIT OR CON is asserted, output bit <1> from the OR MUX is set. At the next IRD, a decode trap is taken to address 0002 (hex) in control store (see Table 6-1). From this location, a jump is taken to the address of the "T-bit or Console Halt" microroutine. This microroutine backs up the PC, and checks if CONSOLE MODE is asserted. If CONSOLE MODE is asserted, the microcode branches to the console stop microroutine.

The console stop microroutine displays a halt code on the console terminal, and the system enters console mode. Thus, pressing the BREAK key has the same effect as pressing the HALT button, if MISC register bit <4> is set (that is, if BREAK is enabled). If MISC <4> is not set, pressing the BREAK key has no effect.

When a Halt macroinstruction is decoded, the output from the decode ROMs is the address of a microroutine that checks if the PSL mode is kernel. If the mode is not kernel, a jump to the reserved instruction fault microroutine is taken. If the mode is kernel, a jump is taken to the address of the console stop routine. The console stop microroutine displays a halt code on the console terminal, and examines the option switches to determine the recovery action.

Powering Up

Powering up is the seventh of the eight functions that the data path module performs. This section describes the power up signals, power failure, the initialization state of the CPU, initialization signals on power up, the option switches, and the boot EPROM.

Power Up Signals

A power up sequence is usually the result of turning the MicroVAX I system power switch on; however, a power up sequence is also initiated on the Q22 bus when the RESTART button on the system front panel is pressed, or when power fails then returns.

When DC voltages are first supplied to the Q22 bus from the power supply, the power supply logic negates the signal BDCOK, and then asserts it 3 ms after DC voltages have reached their specified levels.

The signal DAPL INIT is asserted by the DAP module as soon as DC voltages appear, synchronized with the 62.5 ns clock (MCTM BASE CLOCK), and deasserted two clock cycles (125 ns minimum) after BDCOK is asserted. DAPL INIT is generated from BDCOK.

The signal BINIT is asserted by the DAP board as soon as any DC voltages appear; it is deasserted as soon as DAPL INIT is deasserted. BINIT initializes the Q22 bus.

Another power supply logic signal, BPOK, is negated when DC voltages first appear, and is asserted 70 ms after BDCOK is asserted. If power does not remain stable for 70 ms, BDCOK is negated; therefore, Q22 bus devices must suspend critical actions until BPOK is

asserted. BPOK must remain asserted for a minimum of 3 ms.

Power Failure

The DAP module monitors the Q22 bus power status signals: BPOK and BDCOK. A power failure occurs when the AC voltage to the power supply drops below 75% of the nominal voltage for one full line cycle (15–24 ms). When a power failure is detected, BPOK is negated. Once BPOK is negated, the entire power down sequence, as follows, must be completed.

BPOK is synchronized with the CPU clock (CPU CLOCK) to generate the signal DAPK PWR DWN; DAPK PWR DWN is asserted when BPOK is negated. DAPK PWR DWN is an input to the interrupt control logic. A power fail interrupt is initiated if the current IPL is less than 1E (see Table 6-8).

The microcode sets bit <0> of the MISC register; bit <0> is the "send Q22 bus initialization" flag. This occurs no later than 3 ms after the negation of BPOK, and causes BINIT to be asserted for 8 to 20 μs.

Once BPOK is negated, the power supply guarantees a minimum of 4 ms before BDCOK is negated. This 4 ms allows mass storage and similar devices to protect themselves against erasures and erroneous writes during a power failure.

DAPL INIT is a synchronized version of BDCOK; it is asserted two clock cycles (125 ns minimum) after BDCOK is deasserted.

The DAP module asserts BINIT again, no later than 1 μs after the negation of BDCOK.

DC power must remain stable for a minimum of 5 μs after BDCOK is negated.

Figure 6-10 shows the power up and power down sequences, and the signal states for normal power.

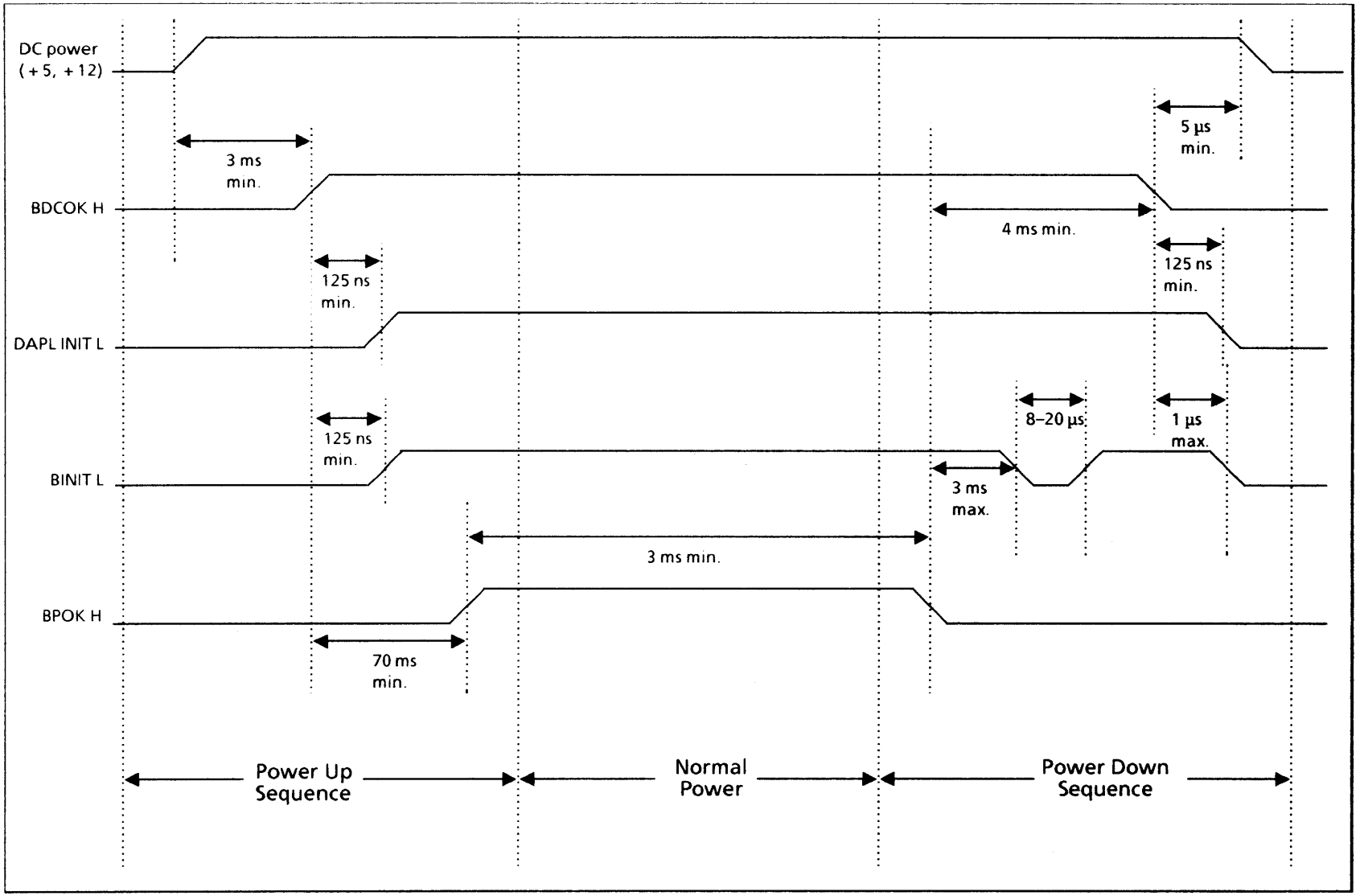


Figure 6-10. Power Up/Power Down Timing

Initialization State

The initial state of the processor is set by the INIT signals: DAPL INIT, DAPL INIT A, and DAPL MCT INIT, which are the buffered outputs of DAPL INIT. (DAPL INIT is generated by the Q22 bus signal BDCOK.) The initial state of the processor is defined as follows.

- The current microaddress is ZERO; that is, the first microinstruction executed following the deassertion of DAPL INIT will be from location 0000 in control store.
- The control store parity error flag is cleared.
- No interrupt requests are pending.
- The index register is cleared.
- The IPL register is cleared.
- The MISC register is cleared, causing the three diagnostic LEDs to be lit.
- The memory request signal (DAPR MEM REQUEST) is in the deasserted state.
- Q22 bus signal BINIT is asserted during the deassertion of BDCOK.

In addition, all of the flip-flops that synchronize the DAP clock signals are set to a known state. This guarantees that the clock signals have the correct relationship to each other.

Initialization Signals on Power Up

When the signal DAPL INIT is asserted during power up, it generates the signal JAM UPC, which causes the microprogram to jump to the microinstruction located at control store address 0000.

The signal BPOK is synchronized with the CPU clock (CPU CLOCK) to generate the signal DAPK PWR DWN. The signal DAPB PUP is generated by DAPK PWR DWN, but is synchronized with the low-going edge of CPU clock.

The assertion of DAPB PUP causes the deassertion of the signal DAPL SET DPC INIT, which in turn causes the deassertion of the signal DAPL DPC RESET on the next leading edge of the 125 ns data path chip clock, DPC CLK.

The deassertion of DAPL SET DPC INIT also causes the signal DAPC DPC INIT to be deasserted on the next low-going edge of DPC CLK.

DAPC DPC INIT generates the signal DAPC DLY STRTUP. The deassertion of DAPC DPC INIT causes the signal DAPC DLY STRTUP to be deasserted on the next leading edge of CPU CLOCK. The deassertion of DAPC DLY STRTUP marks T0 of the first microinstruction to be executed by the data path chip.

On the next leading edge of the CPU clock following the deassertion of DLY STRTUP, the first microinstruction executed by the data path chip is executed in the chip again, and this time, executed by the entire data path module as well; that is, the data path module is synchronized to T0 of the first microinstruction. The data path chip executes the first microinstruction twice to deliver correct parity to the data path module.

Figure 6-11 shows all of these power up and initialization signals, and their relationship to the various clock signals.

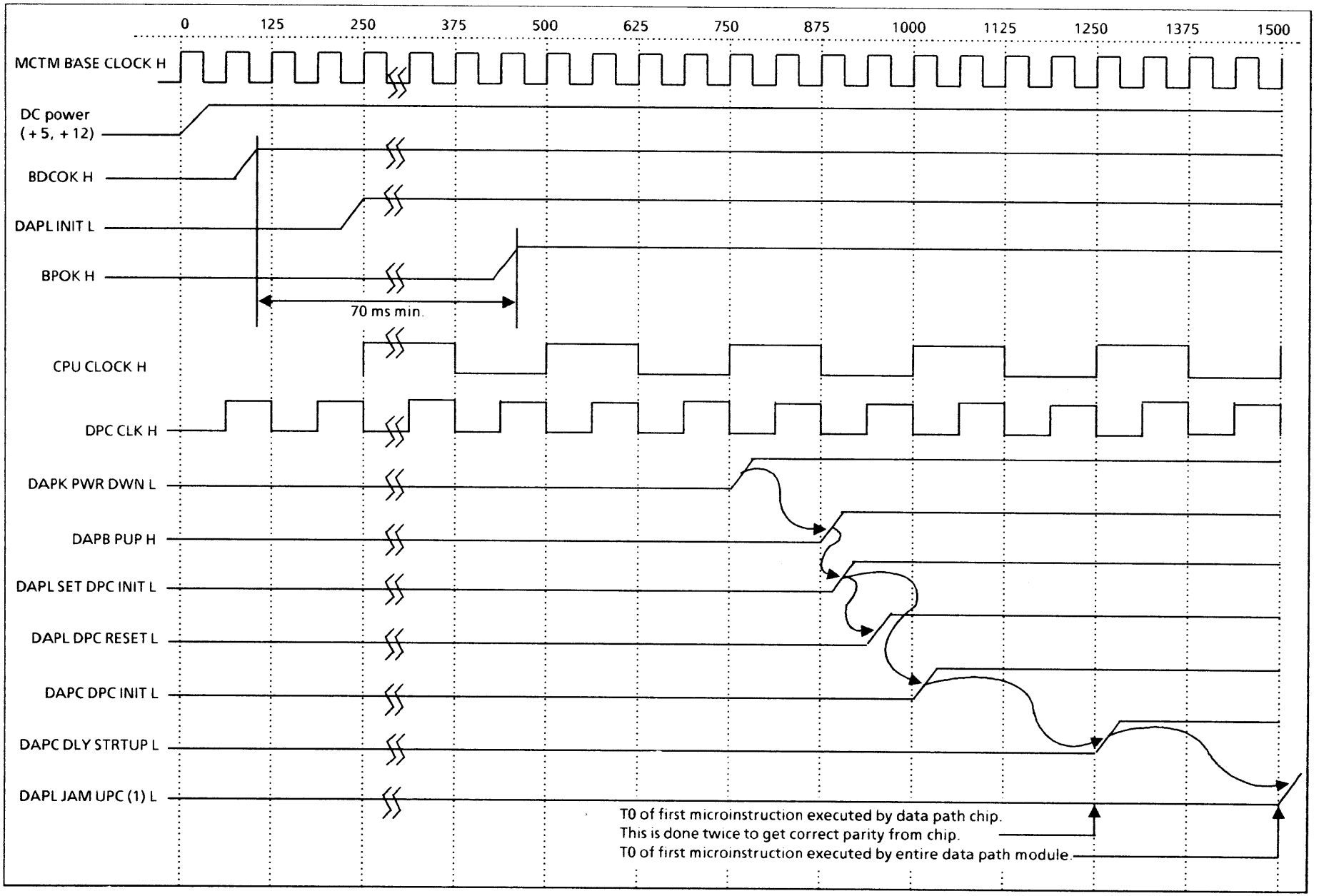


Figure 6-11. DAP Initialization Signals

Option Switches

In addition to specifying the baud rate, break enable, console terminal type, and the bootstrap search order, the option switches select the recovery action.

When a halt condition is encountered, the console stop microroutine prints a halt code on the console terminal. The microcode then examines option switches 3 and 4 to determine the recovery action. These switches are also examined during power up after the successful completion of Microverify to determine the power up action. The switches select one of four possible strategies:

- Warm start, Boot, Halt
  If <4:3> = 0, the system attempts a warm start. If the warm start fails, the system tries to boot. If bootstrap fails, the system enters console mode. This is the default configuration for the switches.
- Boot, Halt
  If <4:3> = 1, the system tries to boot. If bootstrap fails, the system enters console mode.
- Warm start, Halt
  If <4:3> = 2, the system attempts a warm start. If the warm start fails, the system enters console mode.
- Halt
  If <4:3> = 3, the system enters console mode and waits for input from the console terminal.

Boot EPROM

This is an 8192 by 8-bit-wide EPROM used to store the VAX macrocode necessary to boot the MicroVAX I system. (An EPROM that is 16,384 locations long by 8 bits wide may also be used.)

The macrocode stored in the EPROM is the primary bootstrap. The EPROM is addressed by loading the low eight bits into the index register and the high six bits into the PSL.IPL register. The proper address bits are loaded into the index and PSL.IPL registers by the initialization microcode routine. Thus, a byte at a time is addressed in the primary bootstrap.

As each byte is accessed, it is driven onto the ID bus, into the data path chip, and then out the memory data bus to the memory controller. From there, the byte is written into the Q22 bus memory. In this manner, the entire primary bootstrap from the EPROM is copied into main memory. Once it is copied, the main memory address of the first byte is loaded into the program counter, and the bootstrap macrocode executed.

The bootstrap macrocode locates the device that contains the secondary bootstrap and loads the secondary bootstrap into main memory. The primary bootstrap then sets the stack pointer and the program counter to the starting address of the secondary bootstrap, and execution of the secondary bootstrap begins. It is the responsibility of the secondary bootstrap to complete the bootstrap process. (For more information about bootstrapping, see the section titled "MicroVAX I System Bootstrap" in Chapter 2.)

Communicating with the MCT

Communicating with the memory controller is the eighth of the eight functions that the data path module performs. This section describes the data interface, the control interface, interface control signals, stalls, the MD bus latches, memory function latches, memory function control, the sign extenders, the PSL.MODE register, and memory reference timing.

Data Interface

The data interface between the data path and memory controller modules is the memory data bus (MDB) which carries 32 tri-state signals. The memory data bus is part of the 50-pin, over-the-top cable that connects the two boards. The tri-state signals are named BUS MEM DATA <31:00>. The tri-state enables for these data bus signals are controlled by either the DAP module or the MCT module, depending on the direction of data transfer.

The following situation causes BUS MEM DATA <31:00> to be sent from the memory controller to the data path module over the memory data bus: a memory request microinstruction is executed, followed two cycles later by the execution of a microinstruction that is not a Moveout and that has MEMORY.DATA specified as the long operand. (MEMORY.DATA is selected by addresses 7C–7F; these addresses are allocated as a block. MEMORY.DATA can be thought of as the address of the memory data bus. When the long operand of a microinstruction specifies MEMORY.DATA the data to be operated on are the 32 bits currently on the memory data bus.)

There are three microinstructions that cause BUS MEM DATA <31:00> to be sent from the data path module to the memory controller over the memory data bus. They are:

- a Memory Request; BUS MEM DATA <31:00> represent a virtual address.
- an I-stream Request; BUS MEM DATA <31:00> are the unincremented contents of the program counter.

- a Moveout; the long operand specifies MEMORY.DATA.

Control Interface

The control interface between the data path and memory controller modules consists of eight bidirectional signals, seven unidirectional lines from DAP to MCT, and eleven unidirectional lines from MCT to DAP, which return the status of the memory controller to the DAP microsequencer. All of these control signals and the clocks are carried on the CD slots of the backplane (see Figure 1-3 in Chapter 1). The eight bidirectional signals are the memory control bus (MCB) and are named BUS MEM CTL <7:0>.

The memory control bus is a time-multiplexed tri-state bus which may be driven from either the DAP or the MCT module. Control information from the DAP microinstruction is driven in the first half of the microcycle (during T1). Instruction stream bytes are driven from the memory controller to the IBYTE register during the second half of the microcycle (during T3).

Interface Control Signals

When a microinstruction specifying a memory request function is decoded, the data path module drives the contents of the register specified by the long operand (usually a virtual address, but possibly a physical address or the actual data) out the data bus and onto the memory data bus. The encoded memory function is driven onto the memory control bus. The data path module then asserts the memory request line, DAPR MEM REQUEST. This signal informs the memory controller that a new function code is on the control bus.

The memory controller responds by accepting the 32 bits on the memory data bus (a virtual address, physical address, or data), starting the appropriate cache or bus cycle, and asserting the request acknowledge signal, MCTN REQ ACK. When the data path sees the request acknowledge signal, it removes the 32 bits from the data bus. If the memory function is a read, the data path also disables the tri-state drivers to allow the data being read to be driven from the memory controller to the data path.

The microcode does not expect a response from the memory controller until the microcycle following the next microcycle; the memory controller error status signals are in an undefined state until then. After this intervening microcycle, a Move or Moveout microinstruction to read or write the data may be executed, and a microbranch taken to test the status of the operation. (The data to be read or written are the data currently on the memory data bus; this is specified by MEMORY.DATA in the long operand of the Move or Moveout microinstruction.)

Byte displacements are read from the instruction stream by enabling the IBYTE register onto the ID bus; the Memory Request microinstruction is not used. For this case, the microcode must always test whether the byte in the IBYTE register is valid, to insure that valid data has been read. The byte in the IBYTE register is valid when the signal DAPR IB INVALID is not asserted.

Stalls

Stalls are caused by one of three situations. If the microcode executes a Move or Moveout microinstruction following a Memory Request or an I-stream Request and REQ ACK has not been received from the

memory controller, the data path hardware stalls the operation for one full cycle (250 ns) by delaying the clock edges to the data path control logic. At the end of this cycle, REQ ACK is tested again and the stall repeated if REQ ACK is still not asserted. Thus, the DAP hardware stalls the execution of the microprogram by continuously repeating the Move or Moveout microinstruction until REQ ACK is asserted. Note that this type of stall only occurs when a microinstruction with MEMORY.DATA specified in the long operand is executed following a memory request microinstruction.

The second situation causing a stall occurs when the memory controller asserts the signal MCTN MEM BUSY. If the memory controller is unable to deliver status or data in the cycle in which the information is expected, the memory controller asserts MEM BUSY. Upon receiving MEM BUSY, the DAP hardware causes a stall until MEM BUSY is negated.

A stall also occurs when a microinstruction selects one of the console UART registers. A stall condition is generated for a single cycle. This is because of the long write pulse and read time needed by the UART.

MD Bus Latches

The 32 signals on the data bus are latched into four latches, collectively named the MD bus latch. From this latch, the signals are driven onto the memory data bus and then to the memory controller. Similarly, signals coming into the DAP module from the memory controller on the memory data bus are latched into four more latches, collectively named the MD bus input latch. From the MD bus input latch, the signals are driven onto the data bus.

The MD bus latch and the MD bus input latch are needed because the memory controller uses a 125 ns

cycle and may not be in the correct half of the 250 ns DAP cycle when data are being sent to the memory controller or received by the data path.

The signal DAPK OPEN LATCHES controls the MD bus latch, opening it to capture the data that are to be driven from the data bus onto the memory data bus. The memory controller module controls the MD bus input latch with the signal MCTN MD BUS IN LE, opening it to capture the data that are to be driven from the memory data bus onto the data bus.

Memory Function Latches

The memory function latches are part of the memory function control block shown in Figure 6-1. There are two latches: the first one holds the current memory function code and the second holds a previous memory function code.

The bits saved in the first memory function latch are microinstruction bits DAPA CS <38:37> and <28:23> when a Memory Request microinstruction is decoded. Bits <38:37> actually come from the size register but still represent the data type. Bit <28> is the data flow bit, and <27:23> are the memory function code (see Figure 5-4).

These eight bits are saved in the first memory function latch until the memory controller is available. The latch is normally open and is closed when a memory request is started. When the memory controller is ready for the function code, the latched bits are driven from the first memory function latch onto the memory control bus as BUS MEM CTL <7:0>.

If the latch bit, bit <31>, of a Memory Request microinstruction is set when the microinstruction is decoded, bits <38:37> and <28:23> are also saved in the second memory function latch. If a page crossing or

memory management fault occurs when this microinstruction is executed, the microcode retries the microinstruction after it fixes the condition that caused the failure. The memory request information needed by the microcode to retry the microinstruction that failed is the information latched in the second memory function latch.

Thus, when a Memory Request microinstruction is repeated, the contents of this second memory function latch are driven onto the memory control bus as BUS MEM CTL <7:0> instead of the contents of the first memory function latch. (The first memory function latch contains memory request information from the most recent Memory Request microinstruction in the microroutine invoked to fix the failure condition.)

Memory Function Control

When a Memory Request or I-stream Request microinstruction is decoded and executed, twelve bits of control information are sent to the memory controller from the data path module. These twelve bits inform the memory controller about the requested memory function.

Eight of the twelve bits are the microinstruction bits latched in the memory function latch and driven over the memory control bus as BUS MEM CTL <7:0>. These eight bits consist of the microinstruction data type field (bits <38:37>), the 5-bit memory function field (bits <27:23>), and the data flow bit (<28>).

The other four bits of control information are sent to the memory controller over the backplane. They are: DAPT MEM REQ MODE <1:0>, DAPT MODIFY, and DAPT SECOND PART.

The two MEM REQ MODE bits indicate the access mode, which is used for protection checking. If the

access mode bit in the Memory Request microinstruction (bit <30>) is set, the value of MEM REQ MODE <1:0> is 0 to indicate kernel. If the access mode bit in the Memory Request microinstruction is clear, MEM REQ MODE <1:0> have the same value as the current mode bits (DAPR CUR MODE <1:0>) in the current mode register (PSL.MODE).

The signal DAPT MODIFY is asserted when the modify intent is write; that is, when bit <29> in the Memory Request microinstruction is a one.

The signal DAPT SECOND PART is the second part flag. This signal is used to inform the memory controller that it has already completed part of a function that is being repeated. This occurs when the data being read or written do not all reside in the same page of memory.

If a page crossing or memory management fault occurs when a microinstruction is executed, the microcode traps to a routine to fix the condition that caused the failure. The microroutines that fix these conditions contain Memory Request microinstructions with REPEAT.FIRST or REPEAT.SECOND memory functions.

When a REPEAT.FIRST Memory Request is executed, the signal DAPR REPEAT is asserted. When DAPR REPEAT is asserted, the previous memory function bits latched in the second memory function latch are driven onto the memory control bus as BUS MEM CTL <7:0>.

When a REPEAT.SECOND Memory Request is executed, DAPR REPEAT is also asserted and with the same effect. But in addition, the second part flag is set; that is, the signal DAPT SECOND PART is asserted. The second part flag is cleared when a Memory Request

microinstruction is executed with the latch bit (<31>) set.

When the memory controller receives these twelve signals, it reassembles them into a control word and uses this control word to access its own control store. The selected memory controller microcode routine then carries out the requested memory function.

PSL.MODE Register

PSL.MODE is the current mode register, address 6A. The current mode bits of the processor status longword (PSL bits <25:24>) are stored here. The current mode register is used to inform the memory controller of the access mode of the current memory request.

PSL.MODE can be read or written. The register is written when an REI (return from exception or interrupt), or a CHM (change mode) macroinstruction is executed. The PSL.MODE register is also written for interrupts and some exceptions. The new current mode is computed in the data path chip and written to the PSL.MODE register from the low two bits of the internal data bus, BUS ID <01:00>.

The output from the PSL.MODE register is the signals DAPR CUR MODE <1:0>. These signals carry the encoding for the current mode and are sent to a memory request control PAL. Other inputs to the memory request control PAL include:

- the signal DAPR REPEAT, which is asserted by the IBYTE control PAL when a Memory Request function code is REPEAT.FIRST or REPEAT.SECOND,
- the latch bit, <31>, from a Memory Request or I-stream Request microinstruction, and

- the mode bit, <30>, from a Memory Request or I-stream Request microinstruction.

When bit <31> of a Memory Request or I-stream Request microinstruction is set, indicating that the memory function is to be latched, the memory request control PAL saves the state of microinstruction bit <30> as the signal DAPT SAVED MODE.

When the signal DAPR REPEAT is not asserted, the memory request control PAL examines microinstruction bit <30>. If bit <30> is clear, indicating current mode, the PAL sends the input signals from the PSL.MODE register (DAPR CUR MODE <1:0>) to the memory controller as the signals DAPT MEM REQ MODE <1:0>.

When DAPR REPEAT is not asserted and bit <30> is set, the signals DAPT MEM REQ MODE <1:0> carry the encoding 00 (binary) to the memory controller to indicate an access mode of kernel.

If DAPR REPEAT is asserted, the memory request control PAL examines the SAVED MODE bit. If SAVED MODE is set, DAPT MEM REQ MODE <1:0> carry the encoding for kernel access mode. If SAVED MODE is clear, DAPT MEM REQ MODE <1:0> carry the encoding from the PSL.MODE register.

Sign-Extenders

If an I-stream Request microinstruction is executed and the long operand specifies IB.WORD, a word of data is read from the instruction stream and returned to the data path module over the memory data bus.

The signal MCTN SEXT from the memory controller informs the data path module that the data needs to be sign-extended. The state of bit <15> is copied into bits

<31:16> and driven by the sign-extenders onto the data bus.

For more information about sign-extension, see the paragraphs titled "Sign-Extension" in the "Data Transfers" section of this chapter.

Memory Request Timing

Figure 6-12 shows the timing of a read from memory, and Figure 6-13 shows the timing of a write to memory. Both diagrams assume a cache hit. The signal DAPK OPEN LATCHES is asserted to open the MD bus latch. This latch is opened once for a read, to capture the contents of the location specified by the long operand before those contents are driven onto the memory data bus. (The contents are usually a virtual address, but can also be a physical address or the actual data.)

DAPK OPEN LATCHES is asserted twice for a write to memory; first to capture the virtual address (or physical address or data) to be driven onto the memory data bus, and second to capture the data to be written to memory before the data are driven onto the memory data bus. The data to be written appear at the output pins of the data path chip 80 ns into the EXECUTE cycle of the Moveout microinstruction.

Table 6-10 summarizes the DAP/MCT interface signals and briefly describes the functions of the signals.

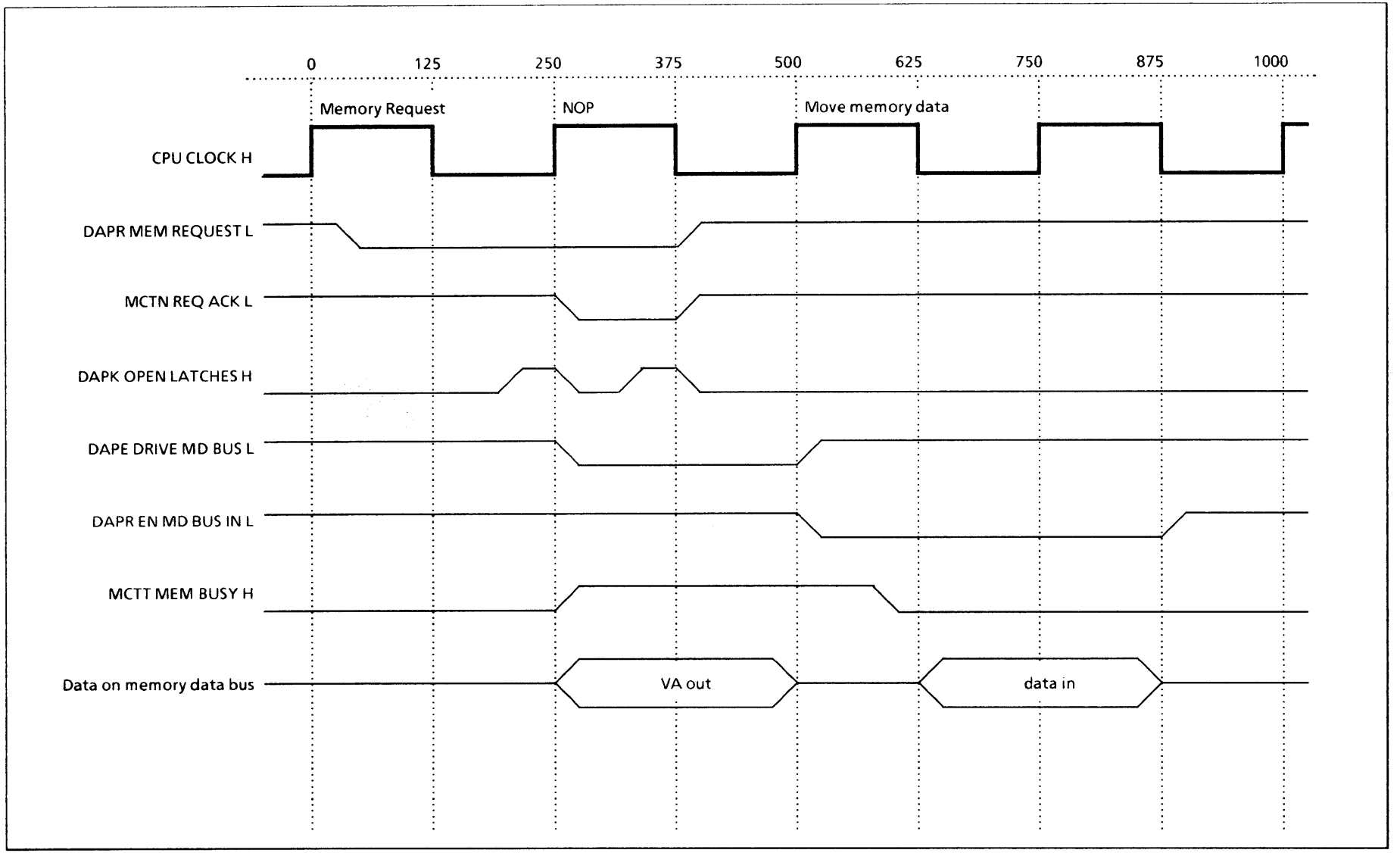


Figure 6-12. Timing of a Read from Memory

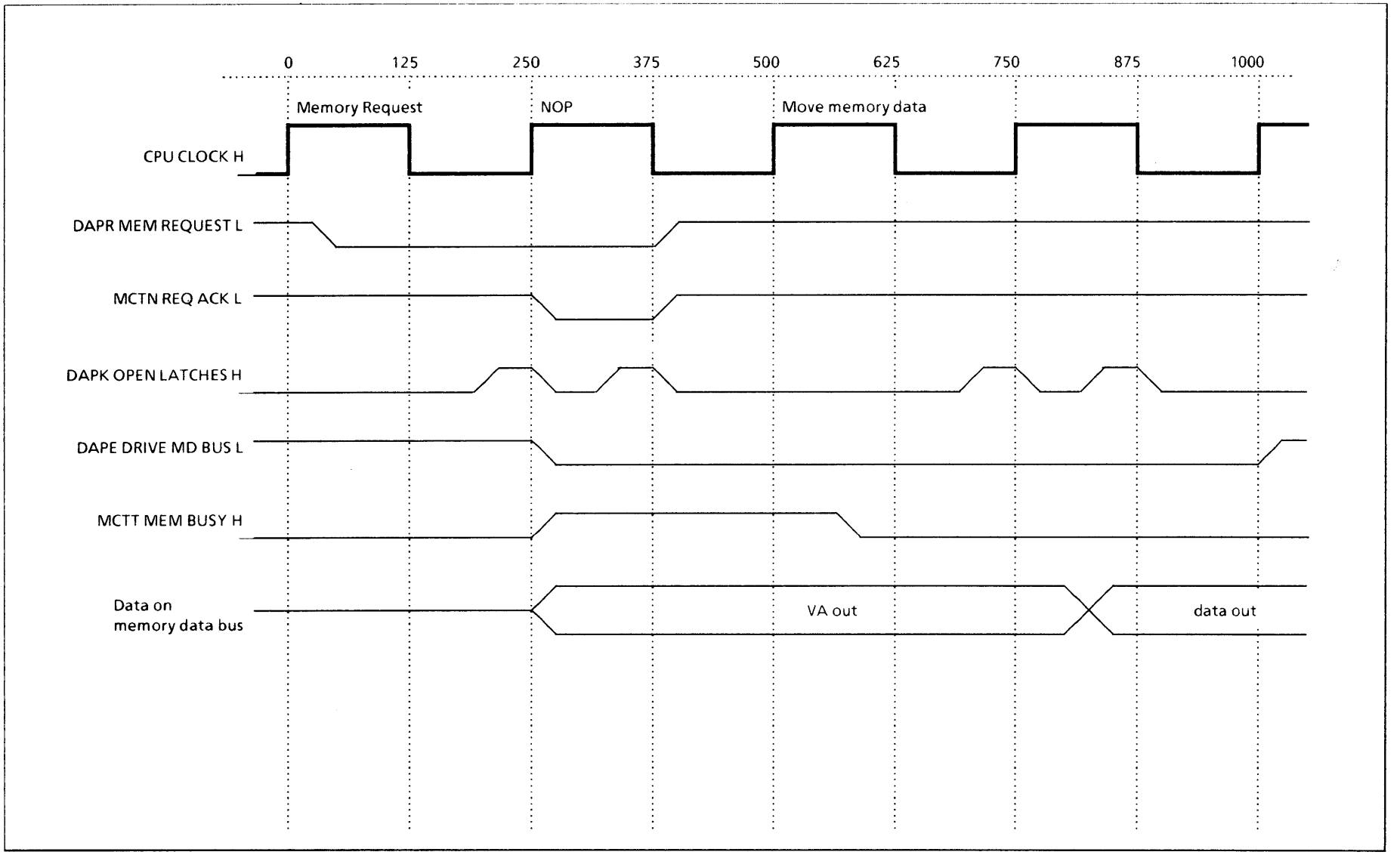


Figure 6-13. Timing of a Write to Memory

Table 6-10. DAP/MCT Interface Signals

Signal Name	Function
DAPL MCT INIT L	Initialize system to a known state; asserted asynchronously, negated 12 ns following low–high edge of 16 MHz clock.
DAPL MCT 250 L	Memory controller copy of the CPU clock.
MCTM BASE CLOCK H	Basic clock source used on DAP module.
MCTM DPC SRC L	Inverted version of clock needed for data path chip.
BUS MEM DATA <31:00> H	Data or address from DAP to MCT. Data from MCT to DAP.
BUS MEM CTL <7:0> H	Memory function code from DAP to MCT. Instruction stream byte from MCT to DAP.
DAPR MEM REQUEST H	Informs memory controller of new function code on control bus.
DAPT MEM REQ MODE <1:0> H	Access mode used for protection check; encoding as defined in PSL.
DAPT MODIFY H	A write will be attempted to the current address.
DAPT SECOND PART H	The first part of this request has already been attempted. This signal is asserted when a Memory Request microinstruction with the REPEAT.SECOND function is executed. The signal is deasserted when a Memory Request microinstruction with the latch bit (<31>) set is executed.
MCTN REQ ACK L	Request acknowledged; the MCT has accepted the command and the address or data.
MCTT MEM ERR H	A memory error has occurred.

Table 6-10. Continued

Signal Name	Function
MCTN MEM BUSY H	The memory controller is busy.
MCTN MD BUS IN LE H	MD bus input latch control for incoming data; needed to keep the data stable across a 250 ns edge for the data path chip. This signal is asserted as long as the MCT memory data bus transceivers are directed out.
MCTS TB MISS H	Translation buffer miss.
MCTS MOD REF H	Modify request refused.
MCTT NXT VALID REG H	IBYTE valid; the IBYTE register may be loaded when this signal is active.
MCTE PAGE CROSS H	A memory reference across a page boundary has occurred.
MCTB WRT TMO H	A Q22 bus timeout occurred on the last write operation; valid for a single cycle.
MCTT IB ERROR H	MCT cannot supply the next byte from the I-stream due to an error or a page crossing.
MCTN SEXT WORD H	Sign-extend control; this signal is asserted when a word displacement is read from the I-stream.
DAPR IB TAKEN L	The microsequencer has used the current contents of the IBYTE register; asserted during decode microinstructions, I-stream requests, IB refills and microinstructions specifying IB.BYTE as the source.
DAPE CONSOLE MODE H	The system has entered console mode; this signal is used to prevent write timeouts in console mode.
DAPL TINIT H	The Q22 bus initialization flag has been asserted; this signal is sent to the MCT to initialize the Q22 bus controller.

Microprogram Level Flow: ADDW3

Now that all of the hardware components and control signals of the data path module have been described in detail, this section takes an ADDW3 (add word 3 operand) macroinstruction and describes the decoding and execution of this instruction at the microprogram level.

An ADDW3 macroinstruction adds the word (16 bits) at the address specified by the first operand specifier to the word at the address specified by the second operand specifier, and stores the sum in the location specified by the third operand specifier. A sample ADDW3 instruction is:

ADDW3 B↑5(R0)[R1],(R5),R2

This instruction uses several addressing modes. The first operand specifier uses byte displacement indexed addressing; the second operand specifier uses register deferred, and the third operand specifier uses register mode.

At some virtual address (VA) in memory, this instruction looks like this:

52	65	05	A0	41	A1	:VA

where A1 is the opcode, 41 specifies index mode using R1, 05A0 is byte displacement mode using R0 (05 is the displacement), 65 is register deferred mode using R5, and 52 is register mode using R2. The 05A0 specifies the base address of an array of words, and the content of R1 is an index into this array. Before the microprogram level description begins, the next few

paragraphs summarize the steps needed to execute this ADDW3 instruction.

Step 1. Evaluate the opcode to select the proper microroutine for this macroinstruction.

Step 2. Evaluate the first operand specifier and obtain the first operand. This is accomplished as follows:

a. Add 5 to the contents of R0; the sum is called the base operand address.

b. Multiply the contents of R1 by 2 (because there are two bytes per word).

c. Add the result from step b to the base operand address from step a to get the address of the first operand.

d. Use the address computed in step c to read the first operand from memory and store it in a working register on the data path chip.

Step 3. Evaluate the second operand specifier and obtain the second operand. This is accomplished as follows:

a. R5 contains the longword address of the operand.

b. Read the operand and store it in a working register on the data path chip.

Step 4. Add the first operand to the second operand, set the PSL condition codes, and store the sum in a result register.

Step 5. Evaluate the third operand specifier to determine where the result is to be stored.

Step 6. Move the contents of the result register to R2.

The remainder of this chapter describes the microprogram steps necessary to decode and execute the ADDW3 macroinstruction. Assume that the six bytes of the instruction are already in the prefetch buffer,

that all virtual addresses are in the translation buffer, and that all requested data are located in the data cache. Figure 6-14 at the end of this chapter diagrams all of the microinstructions.

Evaluating the Opcode: Decode A1

As a result of the execution of the previous macroinstruction, the current conditions are: the PC on the data path chip contains the virtual address of the first byte (A1) of the ADDW3 instruction, the microprogram counter (μPC) contains the microaddress of a Decode microinstruction that decodes macroinstruction opcodes (IRD), and the IBYTE register contains A1 (the ADDW3 opcode).

The contents of μPC are driven onto the NμA bus, selected by the NμA MUX and latched into the CSA register at T2 (125 ns into the microcycle). Control store is accessed with this microaddress and the IRD microinstruction is the output. (IRD means a Decode microinstruction that decodes macroinstruction opcodes.) The microinstruction bits are distributed on the data path: bits <36:16> are sent to the data path chip; bits <24:23> (the IFUNC field, see Table 5-7) are sent to the decode ROMs; bits <15:08> and <24> are sent to the OR MUX control logic; bits <36:32> (the microinstruction opcode) and bit <24> are sent to the ID bus address decode logic; bits <28:24> are sent to the IBYTE control logic.

Bits <24:23> are available at the input to the decode ROMs 20 ns before the next clock edge (T0). The IBYTE control logic detects that an IRD is executing (as opposed to an operand specifier decode) because DAPA CS 24 is asserted; that is, bit <24> of the current microinstruction is set. The bits in the IBYTE register are accessing the decode ROMs as soon as the

new byte is loaded into the IBYTE register, but the decode ROMs are not enabled until the rising edge of DLYD CPU CLOCK (62.5 ns into the current microcycle). The decode ROMs are enabled by the signal DAPC EN ROMS, which is generated from the OR MUX control logic. Bits <15:08> and <24> as inputs to the OR MUX control logic PAL determine that an IRD microinstruction is executing and assert DAPC EN ROMS as the output.

When the decode ROMs are enabled, the contents of the location being accessed by the byte in the IBYTE register are driven onto the NμA bus. The contents are 12 bits of microaddress (<11:0>) because this is an IRD (an opcode decode). The NμA MUX selects these 12 bits and forces a zero as the high-order bit, bit <13> (see Table 6-1). These 13 bits are the microaddress of the first microinstruction in the microroutine for ADDW3. This microaddress is latched into the CSA register at T2 (125 ns).

In addition to 12 bits of microaddress, the output from the decode ROMs includes 2 bits of condition code class, and 2 bits of data type. The condition code class bits are sent to the condition code class register; the data type bits are sent to the size register.

The condition code class is arithmetic because the two condition code class bits contain the encoded value 1 (see Table 6-2).

The data type bits from the decode ROMs for this IRD are 01 (binary) to indicate word. This value is loaded into the size register at the next T0. Thus, the size register contains a value of 1, specifying word.

Meanwhile, the signal DAPR LOAD I BYTE is asserted by the IBYTE control logic because a Decode microinstruction has just been executed. At the next rising edge of CPU CLOCK, (the next T0), DAPR CLOCK I

BYTE is generated from DAPR LOAD I BYTE and the next byte in the instruction stream is clocked into the IBYTE register from the memory control bus. The PC on the data path chip is incremented by one because a Decode microinstruction was just executed.

At the next rising edge of DLYD CPU CLK (T1), the signal DAPR IB TAKEN is generated from LOAD I BYTE. This signal when asserted informs the memory controller that the next instruction stream byte is needed, so the memory controller drives the third byte of the ADDW3 instruction (A0) from the prefetch logic onto the memory control bus.

At this point, the PC on the data path chip contains the virtual address of the second byte (41) of ADDW3, the IBYTE register contains the second byte, 41, and the microaddress of the first microinstruction in the ADDW3 microroutine is latched in the CSA register.

Step 1 is now complete; the macroinstruction opcode has been evaluated and the proper microroutine selected.

Evaluating the First Operand Specifier

Decode 41

The contents of the CSA register select a microinstruction in control store; the microinstruction selected is the first microinstruction in the microroutine for ADDW3. This microinstruction is an operand specifier Decode. The Decode microinstruction bits are distributed to the proper data path elements. Bits <24:23> of this microinstruction (the IFUNC field) have the value 0, indicating an operand specifier decode type 1.

The IFUNC field and the contents of the IBYTE register (41) are used to access the decode ROMs. Since this is an operand specifier decode, the output from the

ROMs to the NμA bus is the low eight bits of the microaddress. The high five bits are driven onto the NμA bus from the jump register. The NμA MUX selects the bits on the NμA bus and latches them into the CSA register.

The size register is loaded at T0 from the CC/DT field of the microinstruction when operand specifier Decodes are executed, unless the CC/DT field contains the encoding 2 to specify the size register. Bits <38:37> of this Decode do contain the value 2, so the size register is unaffected; that is, the size register still contains the value 01, specifying word.

Any time an operand specifier decode is executed, bits <5:0> of the IBYTE register are passed through the IBYTE buffer, driven onto the ID bus, then to the data bus, and into one of the two pointer registers on the data path chip. Bit <26> (the pointer register select bit) of the Decode microinstruction just decoded is zero, so bits <5:0> of the IBYTE register (= 000001) are saved in pointer 1 on the data path chip. Thus, pointer 1 is pointing to R1. Assume that R1 contains the value 3; that is, the contents of R1 will select the third entry in the array of words defined by the base address.

Another result of this operand specifier decode is that the current microaddress plus 1 is pushed on the microstack. This happens for every operand specifier decode when the addressing mode is not register mode, and the content of the IBYTE register is valid.

The PC on the data path chip is incremented by one because a Decode was just executed.

Shift by 2

Next, the 13 microaddress bits latched in the CSA register select a Shift microinstruction from control store. Bits <36:16> of this Shift microinstruction are

latched in the control store register on the data path chip. The CC/DT field of this Shift is 00, so data path chip pins SIZE1 and SIZE0 are both zero. This encoding means that the chip operation (shift) uses data type long. (The data path chip size pins are determined by the CC/DT field of the current microinstruction when the microinstruction is not a Memory Request or an I-stream Request and the CC/DT field does not contain the value 3.)

This Shift microinstruction causes the contents of R1 to be shifted left by two bits, and stores the result in the RESULT2 register. A left shift by two effectively multiplies the contents of R1 by 4. This Shift is executed in case the array to be indexed is an array of longwords. But the next address control field of this Shift microinstruction uses the CASE format; this Shift microinstruction cases on the contents of the size register. The result is that the next microaddress is the address of another Shift microinstruction, but one that multiplies by 2 instead of by 4.

To further explain how this happens, assume that the first Shift microinstruction is located at control store address 1603, and that the next address control field (bits <15:0>) of this Shift microinstruction is 7C30, or 0111/1100/0011/0000. Bits <15:13> have the value 011, which specifies the CASE format (see Figure 5-3). Bits <9:8> are defined as the jump control field (JC<1:0>); the value of 0 in this field specifies that the output of the OR MUX is to be ORed with the low four bits on the NμA bus to obtain the next microaddress. Bits <12:10> are defined as the OR<2:0> field; the value of 7 in this field selects the OR MUX input line with these four signals: 0, 0, SIZE1, SIZE0. SIZE1 and SIZE0 are signals from the size register (DAPE SIZE 1 H and DAPE SIZE 0 H) and have the value 01.

The microsequencer computes the next microaddress as follows. The control logic determines from bits <15:13> that the next address control field format is a CASE, and enables the output of the OR MUX because of the value in the jump control field. Bits <7:0> of the Shift microinstruction (30 hex) are driven onto the NμA bus from the jump register. The output from the OR MUX: 0001 (binary), is ORed with <7:0> (=30 hex) from the jump register; thus, the value of the low eight bits on the NμA bus is 31 (hex). The NμA MUX selects these eight bits off the NμA bus, and combines them with the bits in the page register to generate the next microaddress. The page register contains the value 16 from the high-order five bits of the current Shift microinstruction; thus, the next microaddress is 1631.

Shift by 1

Control store location 1631 contains the second Shift microinstruction. This Shift microinstruction shifts the contents of the register pointed to by pointer 1, left by 1. Pointer 1 is still pointing at R1, which still contains the value 3. Shifting the value 3 left by one bit effectively multiplies by 2; the result 6 is stored in the RESULT2 register on the data path chip. This is now the correct index value because in the array of words (that is, each array entry is two bytes wide) that will be accessed shortly, the sixth byte from the base address of the array is the address of the third entry.

The CC/DT field of this Shift is also 00, so data path chip pins SIZE1 and SIZE0 are zero, and therefore the chip operation uses data type long.

While these two Shift microinstructions were executing, the IBYTE control logic has caused the third byte (A0) of the ADDW3 instruction to move off the memory

control bus into the IBYTE register, and the memory controller has driven the next instruction byte, 05, onto the memory control bus. Thus, the IBYTE register contains A0, the PC contains the virtual address of the third byte of ADDW3 (A0; the PC was incremented by one when the Decode for 41 was executed), and the microcode is ready to compute the base address of the array of words.

Decode A0

The next microaddress generated from the execution of the second Shift microinstruction selects another Decode microinstruction in control store. This Decode is part of a microroutine used to calculate base addresses. As this Decode microinstruction is evaluated and executed, the same steps that happened when 41 was decoded are repeated:

- The IFUNC field and the contents of the IBYTE register (A0) are used to access the decode ROMs.
- The size register is unaffected because the CC/DT field of this Decode contains the value 2; thus, the contents of the size register is still 01, specifying word.
- Bits <5:0> of the IBYTE register are latched into the IBYTE buffer, driven onto the ID bus, then to the data bus, and into pointer 1 on the data path chip (bit <26> of this Decode microinstruction is also a zero); pointer 1 now contains the value 0, that is, pointer 1 now points to R0.
- The current microaddress plus 1 is pushed on the microstack. This happens for every operand specifier decode when the addressing mode is not register mode, and the content of the IBYTE register is valid.

- The PC on the data path chip is incremented by one because a Decode was just executed.
- The microsequencer calculates the address of the next microinstruction using the low eight bits from the decode ROMs and the high five bits from the jump register. The NμA MUX selects these combined 13 bits off the NμA bus and latches them into the CSA register.

Since the Decode just completed, LOAD I BYTE is asserted, the next instruction byte, 05, is loaded into the IBYTE register, and the memory controller drives the fifth byte of ADDW3 (65) onto the memory control bus. Thus, the IBYTE register contains 05, the PC contains the virtual address of 05, and the CSA register contains the microaddress of the next microinstruction.

Add

The microaddress in the CSA register selects an Add microinstruction in control store. This Add computes the base operand virtual address. The short operand of this Add microinstruction specifies *pointer 1; that is, use the contents of the register pointed to by pointer 1. The long operand of the Add specifies IB.BYTE; that is use the contents of the IBYTE register.

Pointer 1 points to R0; R0 contains a virtual address, say, 0200. The IBYTE register contains the byte displacement, 05. Any time the long operand of a microinstruction specifies IB.BYTE, the byte currently in the IBYTE register is driven over the ID bus, to the data bus, and into the data path chip, having been sign-extended on the data bus. IB.BYTE as the long operand also causes the data path chip to increment the PC by one.

The CC/DT field of this Add is 00 (binary), so data path chip pins SIZE1 and SIZE0 are zero, and therefore the chip operation uses data type long.

The execution of the Add microinstruction within the data path chip happens as follows. The internal data path chip logic decodes the 21 bits of the Add microinstruction stored in the chip's CSR. As a result, the contents of the register pointed to by pointer 1 (00000200) are driven from the register file (where R0 is) over bus A to the ALU. The sign-extended byte displacement (05) from the IBYTE register is driven from the data bus over the internal chip bus B, and to the ALU. The ALU adds 00000200 and 00000005, and stores the result in the RESULT1 register. The sum is stored in the RESULT1 register because bit <31> in the Add microinstruction is set.

While the data path chip is executing the Add, the data path microsequencer uses the next address control field of the microinstruction to compute the next microaddress. This field of the Add has the hex value A601; that is, the next address control field format is TRAP. The OR MUX condition that would cause a trap is IB invalid. Since the signal IB INVALID is not asserted at this time, no trap occurs, and the NμA MUX selects the contents of the μPC (microprogram counter), which is the microaddress of the Add microinstruction plus 1, as the next microaddress.

When IB.BYTE is specified as the long operand, the IBYTE control logic asserts the same series of signals as when a Decode has just been executed, to load the next instruction stream byte into the IBYTE register from the memory control bus. So at this point, the IBYTE register contains the next byte of ADDW3: 65, and the PC contains its virtual address; the last byte of ADDW3 (52) is on the memory control bus, and the

CSA register contains the microaddress of the Add microinstruction plus 1.

Move

The microinstruction following the Add is a Move. This Move stores the computed base operand address in a temporary register. The CC/DT field of this Move is 10 (binary), so data path chip pins SIZE1 and SIZE0 are 1 and 0, respectively. Therefore, the chip operation uses data type long.

A Move microinstruction moves the contents of the location specified by the long operand to the location specified by the short operand. The long operand of this Move is RESULT1, and the short operand specifies a temporary register, labeled VIRTUAL. When bits <36:16> of this microinstruction are clocked into the CSR on the data path chip, decoded and executed, 00000205 (the contents of RESULT1), is driven over bus B and stored in VIRTUAL in the register file.

Meanwhile, the data path microsequencer computes the address of the next microinstruction from the next address control field of the Move, which is a return. The microaddress at the top of the microstack is the address of the last Decode microinstruction plus 1. So the data path microsequencer pops this microaddress off the stack to generate the address of the next microinstruction. The microaddress now at the top of the microstack is the microaddress plus 1 of the Decode microinstruction that decoded 41 (the second byte of ADDW3).

The IBYTE register still contains 65, 52 is still on the memory control bus, the PC still contains the virtual address of 65, and the CSA register contains the microaddress that was just popped off the top of the microstack.

Add

The microinstruction following the Move is another Add. This Add computes the final effective address of the first operand by adding the base address to the scaled index value. The short operand of this Add microinstruction specifies RESULT2, which contains the value 6 from the second Shift operation. The long operand specifies VIRTUAL, which contains 0205.

The CC/DT field of this Add is 00, so data path chip pins SIZE1 and SIZE0 are zero, and therefore the chip operation uses data type long. The result registers are all 32 bits wide, so this Add operation is manipulating longwords of data.

The value 6 (actually 00000006) is driven over bus A to the ALU, 00000205 is driven over bus B to the ALU, and the sum 0000020B is stored in RESULT1 because bit <31> is set in the Add microinstruction.

The data path microsequencer computes the address of the next microinstruction using the next address control field of the Add, which is a branch. The specified branch condition being tested for is register mode. Since this condition is not met, the branch is not taken, and the next microaddress generated is the current microaddress plus 1.

The IBYTE register still contains 65, 52 is still on the memory control bus, the PC still contains the virtual address of 65, and the CSA register latches the contents of μPC, which is the microaddress of the Add plus 1.

Memory Request

The microinstruction following the Add is a Memory Request. This microinstruction sends the computed address of the first operand to the memory controller.

The memory controller will then return the data at that address.

The memory function specified in bits <27:23> of the microinstruction is VREAD.RCHECK. The data flow bit is a zero (bit <28>) as the data flow will be from the memory controller to the data path (a read). Thus, a value of 01 (hex) is assembled in the low-order six bits of the memory function latch. The other two latch bits are set by signals from the size register. The last time the size register was loaded was during the Decode for A1; the size register still contains the value 01, which is therefore also the value of the two high-order memory function latch bits. Thus, the value of the output signals BUS MEM CTL <7:0> from the memory function latch is 41 (hex).

Four additional signals are sent to the memory controller over the backplane: DAPT MEM REQ MODE <1:0>, DAPT MODIFY, and DAPT SECOND PART. For this Memory Request, DAPT MEM REQ MODE <1:0> have the value of the current access mode from the PSL.MODE register, and neither MODIFY or SECOND PART is asserted.

The CC/DT field of this Memory Request is 10 (binary). A value of 2 in the CC/DT field of a Memory Request causes the data path chip size control pins to carry the encoding from the size register. Since the size register contains 01 indicating word, the data path chip pins SIZE1 and SIZE0 are 0 and 1, respectively. Therefore, the memory controller will return a word of data at the specified address.

The long operand specifies the address of the RESULT1 register, so the virtual address 0000020B is driven from RESULT1, over bus B, latched into the MD bus latch, and driven over the memory data bus as BUS MEM DATA <31:00> to the memory controller.

The memory controller asserts the signal MCTN REQ ACK when it accepts the virtual address 0000020B off the memory data bus and the memory function request information off the memory control bus.

The data path microsequencer computes the address of the next microinstruction using the next address control field of the Memory Request, which is a jump; that is, the next microaddress is supplied in bits <12:0> of the Memory Request microinstruction.

The IBYTE register still contains 65, 52 is still on the memory control bus, the PC still contains the virtual address of 65, and the CSA register latches bits <12:0> of the Memory Request microinstruction, which were driven onto the NμA bus from the jump register.

Move

The microinstruction following the Memory Request is a Move. This Move sets a register number in pointer 1. The CC/DT field of this Move is 10 (binary), so data path chip pins SIZE1 and SIZE0 are 1 and 0, respectively. Therefore, the chip operation uses data type long.

A Move microinstruction moves the contents of the location specified by the long operand to the location specified by the short operand. The long operand of this Move is hex 43, which is a location in the constants ROM. The contents of location 43 is the value 14 (hex); hex 14 is the address of a temporary register, labeled OPERAND1. The short operand specifies the address of pointer 1. When bits <36:16> of this microinstruction are clocked into the CSR on the data path chip, decoded and executed, 14 (the contents of location 43), is driven over bus B and stored in pointer 1. Thus, pointer 1 points to OPERAND1.

The data path microsequencer computes the address of the next microinstruction from the next address control field of the Move, which is a jump; that is, the next microaddress is supplied in bits <12:0> of the Move microinstruction.

The IBYTE register still contains 65, 52 is still on the memory control bus, the PC still contains the virtual address of 65, and the CSA register latches bits <12:0> of the Move microinstruction, which were driven onto the NμA bus from the jump register.

Move

Bits <12:0> of the Move microinstruction are the microaddress of another Move. The previous Move was the one intervening cycle between the Memory Request and the availability of the requested data; this Move microinstruction moves the data supplied by the memory controller into the data path chip. The CC/DT field of this Move is 10 (binary), so data path chip pins SIZE1 and SIZE0 are 1 and 0, respectively. Therefore, the chip operation uses data type long.

The long operand of this Move is MEMORY.DATA which is essentially the address of the memory data bus. The requested data (the first operand) are currently on the memory data bus and latched in the MD bus input latch. The short operand specifies OPERAND1. When bits <36:16> of this microinstruction are clocked into the CSR on the data path chip, decoded and executed, the first operand is driven onto the data bus, into the data path chip over bus B, and stored in OPERAND1.

The data path microsequencer computes the address of the next microinstruction from the next address control field of the Move, which is a return; the return is executed if no memory error occurred. The microad-

dress currently at the top of the microstack is the one that was stored when the Decode for 41 was executed, which is the microaddress of that Decode microinstruction plus 1. (The microaddress that was stored when the Decode for A0 was executed, was popped for the return from the Move microinstruction that stored the base address in VIRTUAL.)

The data path microsequencer pops the microaddress off the top of the microstack to generate the address of the next microinstruction. The microstack is now empty.

The IBYTE register still contains 65, 52 is still on the memory control bus, the PC still contains the virtual address of 65, and the CSA register latches the microaddress from the top of the microstack.

Step 2 is now complete; the first operand of the macroinstruction has been evaluated and fetched from memory.

Evaluating the Second Operand Specifier

Decode 65

The popped microaddress selects an operand specifier Decode microinstruction. This Decode is for the current contents of the IBYTE register: 65. As this Decode microinstruction is evaluated and executed, the same steps that happened when 41 was decoded are repeated:

- The IFUNC field and the contents of the IBYTE register (65) are used to access the decode ROMs.
- Bits <38:37> of this Decode have the value 2; that is, use the size register, which still contains the value 01 for word.
- Bits <5:0> of the IBYTE register are latched into the IBYTE buffer, driven onto the ID bus, then to

the data bus, and into pointer 2 on the data path chip (bit <26> of this Decode microinstruction is a one). Pointer 1 still points to OPERAND1, and pointer 2 points to R5.

- The current microaddress plus 1 is pushed on the microstack. This happens for every operand specifier decode when the addressing mode is not register mode, and the content of the IBYTE register is valid.
- The PC on the data path chip is incremented by one because a Decode was just executed.
- The microsequencer calculates the address of the next microinstruction using the low eight bits from the decode ROMs and the high five bits from the jump register. The NμA MUX selects these combined 13 bits off the NμA bus and latches them into the CSA register.

Since the Decode just completed, LOAD I BYTE is asserted, the next instruction byte, 52, is loaded into the IBYTE register, and the memory controller drives the next byte in the instruction stream onto the memory control bus. (The next byte is the opcode of the next macroinstruction in the I-stream.) Thus, the IBYTE register contains 52, the PC contains the virtual address of 52, and the CSA register contains the microaddress of the next microinstruction.

Memory Request

The contents of the CSA register select a Memory Request microinstruction from control store. The second operand is located at the virtual address contained in R5. This microinstruction sends the virtual address in R5 to the memory controller.

The memory function specified in bits <27:23> of the microinstruction is VREAD.RCHECK. The data flow bit is a zero (bit <28>) as the data flow will be from the memory controller to the data path (a read). Thus, a value of 01 (hex) is assembled in the low-order six bits of the memory function latch. The other two latch bits are set by signals from the size register. The size register still contains the value 01, which is therefore also the value of the two high-order memory function latch bits. Thus, the value of the output signals BUS MEM CTL <7:0> from the memory function latch is 41 (hex).

Four additional signals are sent to the memory controller over the backplane: DAPT MEM REQ MODE <1:0>, DAPT MODIFY, and DAPT SECOND PART. For this Memory Request, DAPT MEM REQ MODE <1:0> have the value of the current access mode from the PSL.MODE register, and neither MODIFY or SECOND PART is asserted.

The CC/DT field of this Memory Request is 10 (binary). A value of 2 in the CC/DT field of a Memory Request causes the data path chip size control pins to carry the encoding from the size register. Since the size register contains 01 indicating word, the data path chip pins SIZE1 and SIZE0 are 0 and 1, respectively. Therefore, the memory controller will return a word of data at the specified address.

The long operand specifies *pointer 2, so the longword virtual address contained in R5 is driven over bus B, latched into the MD bus latch, and driven over the memory data bus as BUS MEM DATA <31:00> to the memory controller.

The memory controller asserts the signal REQ ACK when it accepts the virtual address off the memory data

bus and the memory function request information off the memory control bus.

The data path microsequencer computes the address of the next microinstruction using the next address control field of the Memory Request, which is a jump; that is, the next microaddress is supplied in bits <12:0> of the Memory Request microinstruction.

The IBYTE register still contains 52 and the PC still contains its virtual address, the next byte in the I-stream is still on the memory control bus, and the CSA register latches bits <12:0> of the Memory Request, which were driven onto the NμA bus from the jump register.

Move

Bits <12:0> of the Memory Request microinstruction are the microaddress of a Move. The purpose of the Move is to store a new address in pointer 2.

The CC/DT field of this Move is 10 (binary), so data path chip pins SIZE1 and SIZE0 are 1 and 0, respectively. Therefore, the chip operation uses data type long.

The long operand of this Move is hex 44, which is a location in the constants ROM. The contents of location 44 is the value 16 (hex); hex 16 is the address of a temporary register, labeled OPERAND2. The short operand specifies pointer 2. When bits <36:16> of this microinstruction are clocked into the CSR on the data path chip, decoded and executed, 16 (the contents of location 44) is driven over bus B and stored in pointer 2. Thus, pointer 2 now points to OPERAND2.

The data path microsequencer computes the address of the next microinstruction using the next address control field of the Move, which is a jump; the next

microaddress is supplied in bits <12:0> of the Move microinstruction.

The IBYTE register still contains 52 and the PC still contains its virtual address, the next byte in the I-stream is still on the memory control bus, and the CSA register latches bits <12:0> of the Move, which were driven onto the NμA bus from the jump register.

Move

Bits <12:0> of the Move microinstruction select another Move. The previous Move was the one intervening cycle between the Memory Request and the availability of the requested data; this Move microinstruction moves the data supplied by the memory controller into the data path chip.

The CC/DT field of this Move is also 10 (binary), so data path chip pins SIZE1 and SIZE0 are 1 and 0, respectively. Therefore, the chip operation uses data type long.

The long operand of this Move is MEMORY.DATA which represents the "address" of the memory data bus. The requested data (the second operand) are currently on the memory data bus and latched in the MD bus input latch. The short operand specifies OPERAND2. When bits <36:16> of this microinstruction are clocked into the CSR on the data path chip, decoded and executed, the second operand is driven onto the data bus, into the data path chip over bus B, and stored in OPERAND2.

The next address control field format of this Move is a return; the return is executed if no memory error occurred. The microaddress currently at the top of the microstack is the one that was stored when the Decode for 65 was executed, which is the microaddress of that Decode microinstruction plus 1.

The data path microsequencer pops the microaddress off the top of the microstack to generate the address of the next microinstruction. The microstack is now empty.

The IBYTE register still contains 52 and the PC still contains its virtual address, the next byte in the I-stream is still on the memory control bus, and the current microaddress plus 1 is latched in the CSA register.

Step 3 is now complete; the second operand of the macroinstruction has been evaluated, read from memory, and stored in a working register (OPERAND2) on the data path chip.

Adding the Operands

Add

The popped microaddress selects an Add microinstruction. This Add handles the actual addition of the operands.

The CC/DT field of this Add is 11 (binary). A value of 3 in the CC/DT field of an Add means use the data type in the size register. Thus, data path chip pins SIZE1 and SIZE0 reflect the contents of the size register, which is still 01 to indicate word. Therefore, the chip operation uses data type word, which is appropriate since this is the add operation of the Add Word macroinstruction.

The short operand of the Add specifies *pointer 1; that is, use the contents of the register pointed to by pointer 1. Pointer 1 is still pointing to OPERAND1, which contains the first operand. The long operand specifies *pointer 2; that is, use the contents of the register pointed to by pointer 2. Pointer 2 is still pointing to OPERAND2, which contains the second operand.

When bits <36:16> of this microinstruction are clocked into the CSR on the data path chip, decoded and executed, the first operand is driven over bus A to the ALU, the second operand is driven over bus B to the ALU, and the sum is stored in RESULT0 because bit <31> is clear in the Add microinstruction.

When the opcode for ADDW3 was decoded, the signals DAPF CC CLASS <1:0> were part of the output from the decode ROMs; the value of this field was 1, meaning arithmetic (see Table 6-2). The CC/DT field value of 3 and CC CLASS value of 1 are combined and encoded in the condition code PALs to generate the field DAPE CC <F3:F0>. The result is a value for <F3:F0> that means load ALU and PSL CCs arithmetic. Consequently, when this Add microinstruction is executed in the data path chip, the ALU condition codes are set depending on the result, and the PSL condition codes are set from the ALU condition codes. Thus, step 4 is completed: adding the operands, storing the sum in a result register, and setting the condition codes.

The data path microsequencer computes the address of the next microinstruction using the next address control field of the Add, which is a jump; the next microaddress is supplied in bits <12:0> of the Add microinstruction.

The IBYTE register still contains 52 and the PC still contains its virtual address, the next byte in the I-stream is still on the memory control bus, and bits <12:0> of the Add microinstruction are latched in the CSA register.

Decode 52

Bits <12:0> of the Add microinstruction are the microaddress of an operand specifier Decode. This Decode is for the current contents of the IBYTE

register: 52. As this Decode microinstruction is evaluated and executed, the following steps occur:

- The IFUNC field and the contents of the IBYTE register (52) are used to access the decode ROMs.
- Bits <38:37> of this Decode have the value 2; that is, use the size register, which still contains the value 01 for word.
- Bits <5:0> of the IBYTE register are latched into the IBYTE buffer, driven onto the ID bus, then to the data bus, and into pointer 2 on the data path chip (bit <26> of this Decode microinstruction is a one). Pointer 1 still points to OPERAND1, and pointer 2 now points to R2.
- The current microaddress plus 1 is **not** pushed on the microstack because the addressing mode is register mode. Thus, the microstack remains empty.
- The PC on the data path chip is incremented by one because a Decode was just executed, and now contains the virtual address of the next byte in the instruction stream.
- The microsequencer calculates the address of the next microinstruction as μPC plus 1 because the operand specifier, 52, specifies register mode. The NμA MUX selects μPC plus 1 off the NμA bus and latches this address into the CSA register.

Since the Decode just completed, LOAD I BYTE is asserted, the next byte in the I-stream is loaded into the IBYTE register (the opcode of the next macroinstruction), and the memory controller drives the next byte in the instruction stream onto the memory control bus. Thus, step 5 is completed: evaluating the third operand specifier.

Move

The microaddress latched in the CSA register selects a Move. The purpose of the Move is to move the sum computed in the Add to the location specified by the instruction byte, 52.

The CC/DT field of this Move is 11 (binary). A value of 3 in the CC/DT field of a Move selects the data type specified in the size register, which is still 01, indicating word. Therefore, the chip operation uses data type word.

The long operand of this Move specifies RESULT0, which is where the sum from the Add microinstruction is stored. The short operand specifies *pointer 2; pointer 2 is pointing to R2 because of the operand specifier decode just executed. When bits <36:16> of this Move microinstruction are clocked into the CSR on the data path chip, decoded and executed, the sum in RESULT0 is driven over bus B and stored in R2. Step 6 is now complete: the sum of the operands is stored in the destination register.

The data path microsequencer computes the address of the next microinstruction using the next address control field of the Move, which is a jump; the next microaddress is supplied in bits <12:0> of the Move microinstruction.

The microaddress supplied in bits <12:0> of the Move selects an opcode Decode microinstruction (an IRD), and the decoding and execution of the next macroinstruction in the I-stream begins.

Figure 6-14 summarizes the microinstructions used to decode and execute the ADDW3 macroinstruction. It also completes this discussion of the data path microprogram level flow.

Chapter 5 describes the data path microcode and this chapter describes the data path hardware. Similarly, the next two chapters describe the memory controller microcode and hardware.

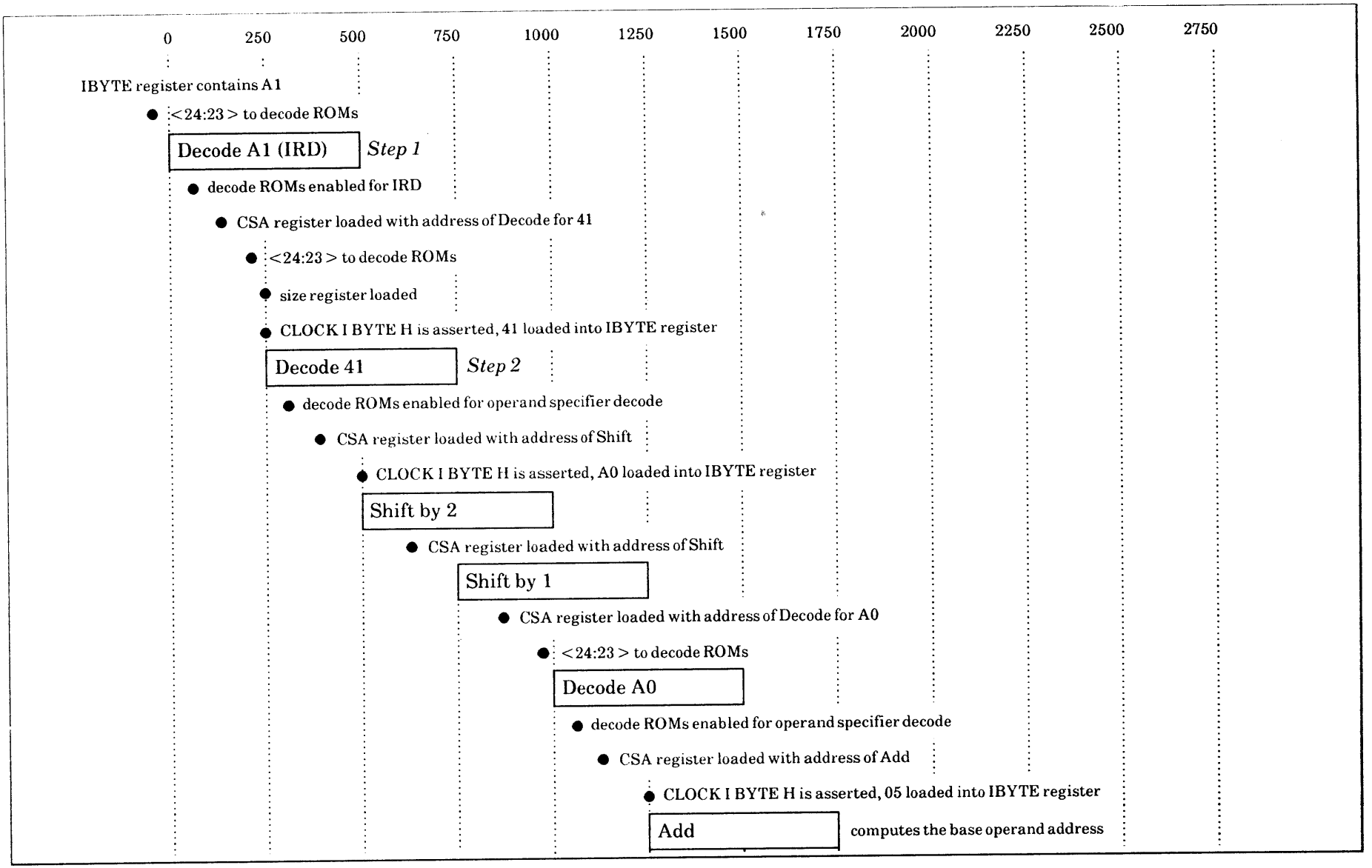


Figure 6-14. ADDW3 Microinstructions

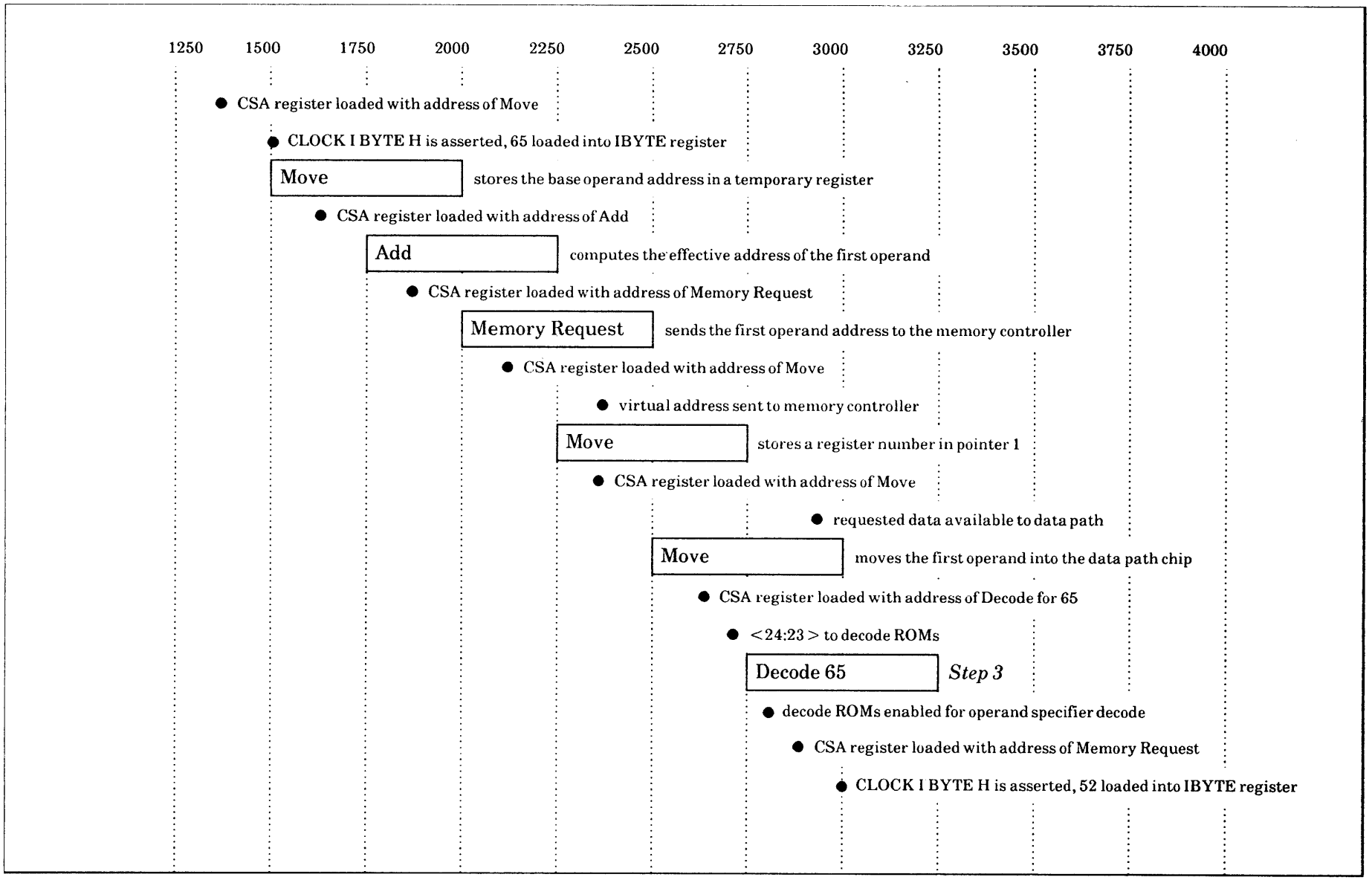


Figure 6-14. Continued

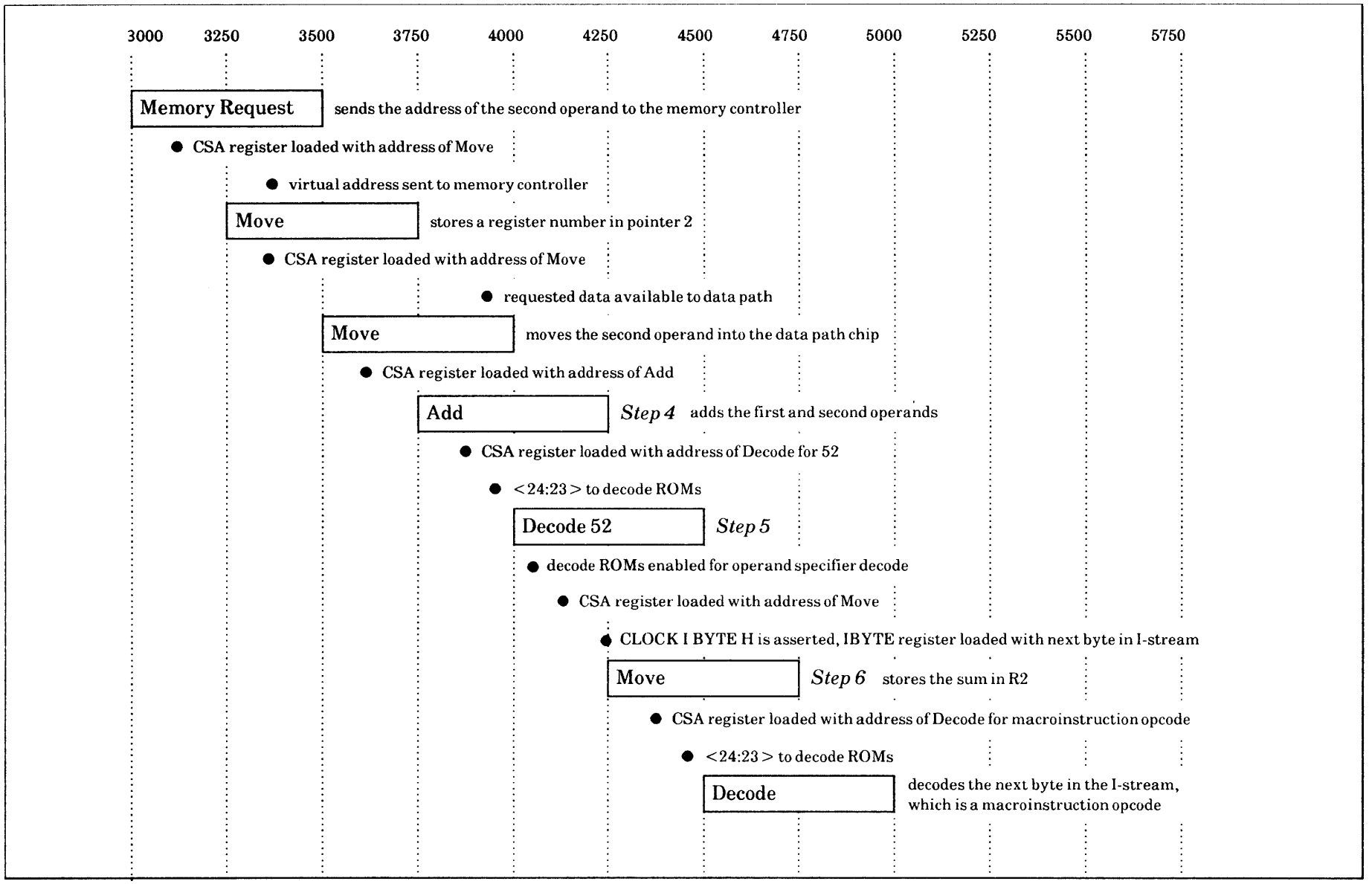


Figure 6-14. Continued

Chapter 7
Memory Controller Microcode

The memory controller module accepts memory request commands issued by the data path micromachine and sequences the necessary functional blocks to carry out the command. The memory controller has its own set of microcode, stored in the MCT control store, to implement the commands from the data path module. Each memory controller microinstruction allows simultaneous activity of several functional blocks. This chapter describes these memory controller microinstructions.

Memory Controller Function Parameters

Every memory controller function involves a set of parameters; that is, the memory controller must know the following information to carry out the memory request from the data path module:

- Address — A virtual or physical memory address, or sometimes the actual data to be written.
- Access Mode — The mode used to check if the operation can be performed. The access mode is specified as either the current mode or kernel mode.
- Data Flow — The direction that data will flow on the memory data bus during the memory operation; that is, whether the operation is a read or a write.
- Data Type — The size of the data to be read or written. The size is specified as byte, word, or

longword, or determined by the contents of the size register.

- Map Enable — A memory controller state flag that specifies whether the translation buffer is to be used. It is set via an MTPR instruction at the macromachine level.
- Modify Intent — Indicates whether the data access is to be checked for read or write access intent. Modify intent does not signify whether data are read or written, but rather which access intent is checked.
- Previous Function — The previously latched memory function, data flow, and data type bits. These bits are saved in the second memory function latch when bit <31>, the latch bit, of a Memory Request is set.
- Second Part Flag — A flag that specifies whether the first or second part of a function is being executed.

When the data path module executes a Memory Request or I-stream Request microinstruction, eight bits of information are latched in the memory function latches and delivered to the memory controller over the memory control bus: the 5-bit memory function code, two bits of data type, and one data flow bit. An additional four bits are delivered over the backplane: two access mode bits, a modify intent bit, and the second part bit.

These twelve bits are recombined on the memory controller module. The following eight bits are presented to the MCT control store as the low-order bits of the 10-bit microaddress: second part flag, two bits of data type, and the 5-bit memory function code. Figure

7-1 shows the format of the memory controller microaddress.

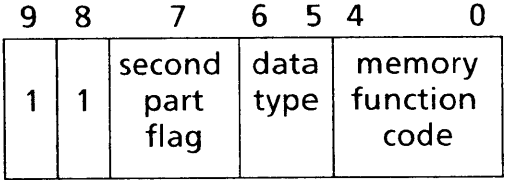


Figure 7-1. MCT Microaddress

The MCT microsequencer forces the two high-order bits of the microaddress to ones. The remaining four bits now are the data flow bit, the two access mode bits (MEM REQ MODE <1:0>), and the modify intent bit (MODIFY).

The signals generated by the data flow and modify intent bits are two of the inputs to the MCT branch MUX. The modify intent bit is also one input to the access violation logic, and the two access mode bits are inputs to the access violation logic.

The address (or data) needed by the memory controller is specified by the long operand of the data path microinstruction that is making the memory request. Thus, the address is delivered to the memory controller over the memory data bus.

The memory controller stores the state of the map enable flag in the MAP.ENABLE control and status register (CSR). This bit is cleared or set when the data path issues a WRITE.MAP.ENABLE Memory Request microinstruction. (The map enable flag is generally set during system bootstrap to enable memory management, and then memory management is left enabled.)

So for each memory request from the data path, the memory controller receives the proper function parameters: four bits from the backplane, eight bits from the memory control bus, 32 bits of address or data over the memory data bus, and has access to the MAP.ENABLE CSR.

When the memory controller receives these function parameters, the proper bits are presented to control store as the next microaddress, and the first microinstruction is accessed in the microroutine that handles the requested memory function.

Microinstruction Format

The memory controller microinstruction is 64 bits wide, but only 63 bits are used. The bits are divided into four major functional fields:

- a 6-bit Q22 bus interface control field that allows communications with and control of the asynchronous Q22 bus interface,
- a 39-bit functional block control field that provides clocking and output enables for each functional block in the memory controller,
- a 3-bit status control field that controls two status signals, and
- a 15-bit microprogram control field consisting of 10 bits of next address, 3 bits of branch control, and 2 bits of branch, dispatch and trap control.

Figure 7-2 shows the format of the memory controller microinstruction. Bit <55> is unused and causes no action in the memory controller module. The Q22 bus interface control field consists of bit <63> and bits <50:46>. The functional block control field consists of bits <62:51> and <45:18>. The status control field is

bits <17:15>. The microprogram control field is bits <14:0>. The sections following Figure 7-2 describe these major functional fields in more detail.

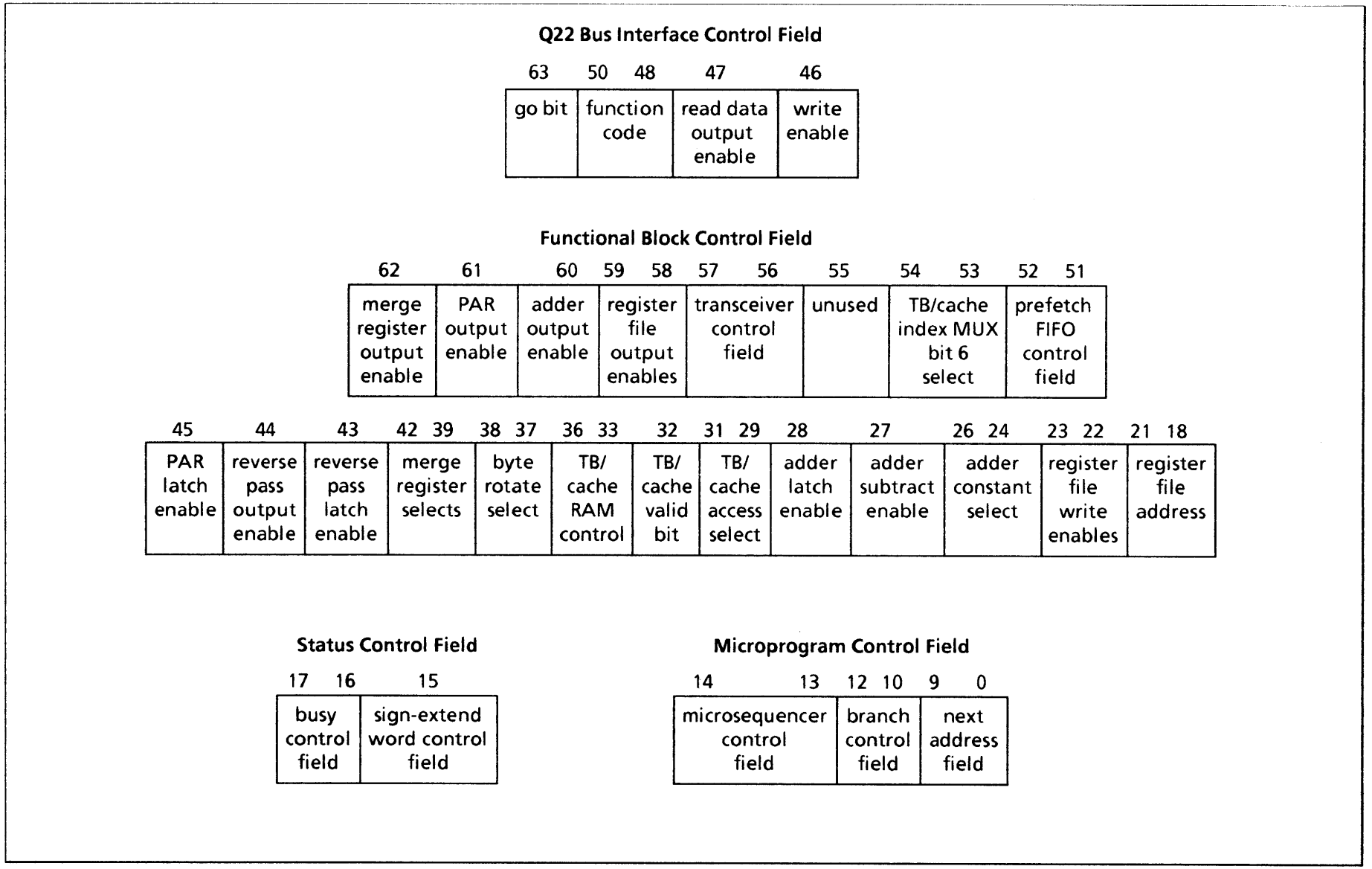


Figure 7-2. MCT Microinstruction

Q22 Bus Interface Control Field

The Q22 bus interface control field allows the memory controller to communicate with and control the Q22 bus interface. This field consists of:

- the go bit, <63>,
- three bits of function code, <50:48>,
- the read data output enable bit, <47>, and
- the write enable bit, <46>.

Q22 Bus Go Bit

Bit <63> of each memory controller microinstruction is the go bit, active high. This bit allows the writing of a function code (MCT microinstruction bits <50:48>) to the Q22 bus interface. (Referencing Figure 4-1 in Chapter 4, the Q22 bus interface is the Q22 bus controller and the Q22 bus registers.)

For all Q22 bus operations, bus arbitration begins with the posting of the function code. When the Q22 bus controller acquires the bus, it asserts the address provided by the memory controller and waits for the go bit before proceeding with the Q22 bus cycle.

For operations such as I/O space accesses, interrupt vector reads, and memory writes, the go bit may be posted at the same time as the function code to cause a bus cycle to begin immediately. For operations such as a read from memory following a cache miss, the go bit may be sent as late as two cycles after the function code. The go bit remains asserted for the entire bus operation.

Q22 Bus Function Code

Bits <50:48> select the type of Q22 bus operation to be performed. The encoding for this field is shown in Table 7-1. When the function code is delivered to the Q22 bus controller, the controller begins arbitration for the bus. When the controller gains control of the bus, it drives an address onto the bus and holds the bus until it receives the go bit or a function code of 000, which specifies no operation. (The go bit must be asserted no later than two cycles after the function code is posted.)

Table 7-1. Function Code Field

<50:48>	Operation	Mnemonic
000	no operation	
001	write word	DATO
010	write byte	DATOB
011	write block	DATBO
100	read word	DATI
101	read interlocked	DATBI
110	read interrupt vector	
111	read block	DATIO

Q22 Bus Read Data Output Enable

Bit <47> is active high and allows data from the Q22 bus interface to be driven onto the memory controller data bus as MCD <21:00>. These 22 bits may represent a 16-bit datum read from the Q22 bus, a 9-bit Q22 bus interrupt vector, or a 22-bit cache invalidate address.

Q22 Bus Write Enable

Bit <46> is active high and causes the data stable on MCA <29> and MCA <21:00> to be written to the Q22 bus write register. (See Figure 8-1 in Chapter 8 for the location of the write register). The data written may represent a 22-bit physical address, a 13-bit I/O space address (if MCA <29> is a one), or a 16-bit datum to be written to Q22 bus memory or an I/O device.

Functional Block Control Field

The functional block control field, shown in Figure 7-2, consists of microinstruction bits <62:51> and <45:18>. This field provides clocking and output enables for each functional block in the memory controller. In addition, seven of the bits (<62:56>) select which MCA bus sources drive the MCA bus.

The following sections describe the bits in the functional block control field in more detail. The sections are organized by functional blocks; that is, all of the bits that control a particular functional block are discussed together, even though they may not be contiguous in the microinstruction. For example, the first section below, "Rotate/Merge Block Control," describes bits <62>, <42:39>, and <38:37> as all of these bits control the rotate/merge logic.

Rotate/Merge Block Control

The merge register output enable (bit <62>), the merge register selects (bits <42:39>), and the byte rotate select (bits <38:37>) control the rotate/merge logic. This logic consists of a byte rotator, and a merge register.

Merge Register Output Enable. Bit <62> of each memory controller microinstruction is the merge register output enable bit, active high. This bit enables the current contents of the merge register onto the memory controller address (MCA) bus (see Figure 4-1). The merge register is the only MCA bus source that can return a full longword to the data path module.

Merge Register Selects. Bits <42:39> of each memory controller microinstruction are the merge register select bits, active low. These bits control the clocking of data into the four bytes of the rotator and merge register. (See Figure 8-1 in Chapter 8 for the location of these components). Since each byte is individually enabled, bytes from separate sources can be merged to accomplish nonlongword-aligned reads and writes to memory.

Bits <42:39> correspond to merge register bytes 3 through 0, respectively; that is, bit 42 controls the clocking of data into merge register byte 3, and so on. Byte 3 of the merge register is the most significant byte of the output longword. Table 7-2 shows the encoding for the merge register selects.

Table 7-2. Merge Register Selects

<42:39>	Action
1111=0F	load no bytes
1110=0E	load byte 0
1101=0D	load byte 1
1100=0C	load bytes 1 and 0
1011=0B	load byte 2
1010=0A	load bytes 2 and 0
1001=09	load bytes 2 and 1
1000=08	load bytes 2, 1, and 0
0111=07	load byte 3
0110=06	load bytes 3 and 0
0101=05	load bytes 3 and 1
0100=04	load bytes 3, 1, and 0
0011=03	load bytes 3 and 2
0010=02	load bytes 3, 2, and 0
0001=01	load bytes 3, 2, and 1
0000=00	load all bytes

Byte Rotate Select. Bits <38:37> of each memory controller microinstruction are the byte rotate select bits. This two bit field controls the circular byte shift performed on the data from the MCD bus that are presented to the merge register. The encoding is shown in Table 7-3.

Table 7-3. Byte Rotate Select

<38:37>	Action
00	circulate longword 0 bytes right
01	circulate longword 1 byte right
10	circulate longword 2 bytes right
11	circulate longword 3 bytes right

Physical Address Register Control

The physical address register (PAR) output enable (bit <61>), and the PAR latch enable (bit <45>) are the two bits that control the physical address register.

PAR Output Enable. Bit <61> of each memory controller microinstruction is the physical address register output enable bit, active high. This bit enables the current contents of the physical address register (PAR) onto the memory controller address bus as MCA <29:28> and MCA <21:09>.

MCA <29> when set indicates that the physical address is located in the I/O space portion of physical memory. MCA <28> when set indicates that the physical address is not to be saved in the cache because it is an address in a shared physical memory.

MCA <21:09> are the translated physical address bits after a TB/cache access. (MicroVAX I physical addresses are 22 bits long plus the I/O space flag; the remaining nine bits—the page offset bits—are supplied from either the 9-bit adder or the page offset portion of the register file. See Figure 8-1 for the locations of the adder and the register file. MCA <31:30> and <27:22> are pulled high by bus pull-up resistors.)

PAR Latch Enable. Bit <45> of each memory controller microinstruction is the physical address register latch enable bit, active low. This bit enables the PAR to latch the page table entry (PTE) information or test data on its inputs. This signal also latches the 4-bit protection field and the modify bit. The protect field and the modify bit are read from the PTE in the translation buffer (TB) and are used to perform the access violation and modify refused checks.

Adder Logic Control

The adder logic consists of a 9-bit adder and a register. These components are controlled by the adder output enable (bit <60>), the adder latch enable (bit <28>), the adder subtract enable (bit <27>), and the adder constant select (bits <26:24>).

Adder Output Enable. Bit <60> of each memory controller microinstruction is the adder output enable bit, active high. This bit enables the current contents of the 9-bit adder (some previously incremented and saved value) onto the memory controller address bus as MCA <8:0>.

Adder Latch Enable. Bit <28> of each memory controller microinstruction is the adder latch enable bit, active low. This bit enables the adder register to latch the output of the 9-bit adder. (See Figure 8-1 in Chapter 8 for the location of the adder and register.)

Part of the adder logic also contains the page crossing flag. A page cross occurs when an adder operation results in a carry into the tenth bit. An asserted adder latch enable bit also causes the page cross flag to be captured when a page crossing occurs.

Adder Subtract Enable. Bit <27> of each memory controller microinstruction is the adder subtract enable bit. This bit selects whether the 9-bit adder adds or subtracts, in effect.

If bit <27> is a zero, a 3-bit value between 0 and +7 inclusive is supplied to the adder by bits <26:24> of the microinstruction; the adder adds this supplied value to MCA <8:0>. (The supplied 3-bit value is zero-extended to nine bits.) A carry into the tenth bit of the adder is used as the page crossing flag.

If bit <27> is a one, a 4-bit value between −1 and −8 inclusive is supplied to the adder by bits <26:24> of the microinstruction; the adder adds this supplied value to MCA <3:0>. MCA <8:4> are unchanged, and the page crossing flag is negated.

Adder Constant Select. Bits <26:24> of each memory controller microinstruction are the adder constant select bits; that is, they provide the 9-bit adder with a value between −8 and +7 (inclusive) to be added to the bits on the MCA bus. Only a 9-bit value is incremented; a carry into the tenth bit is flagged as the page crossing branch condition. Table 7-4 lists the effective values added to MCA <8:0> or MCA <3:0> for the various states of microinstruction bits <27> and <26:24>.

Table 7-4. Adder Control

<27>	<26:24>	Effective Value	Added To
0	111	+7	MCA <8:0>
0	110	+6	MCA <8:0>
0	101	+5	MCA <8:0>
0	100	+4	MCA <8:0>
0	011	+3	MCA <8:0>
0	010	+2	MCA <8:0>
0	001	+1	MCA <8:0>
0	000	0	MCA <8:0>
1	111	−1	MCA <3:0>
1	110	−2	MCA <3:0>
1	101	−3	MCA <3:0>
1	100	−4	MCA <3:0>
1	011	−5	MCA <3:0>
1	010	−6	MCA <3:0>
1	001	−7	MCA <3:0>
1	000	−8	MCA <3:0>

Register File Control

Eight microinstruction bits control the operation of the register file which is used for storing memory controller working addresses and partially assembled data. Additionally, these eight bits control the reading and writing of the memory controller CSRs (control and status registers). These eight bits are the register file output enables (<59:58>), the register file write enables (<23:22>), and the register file address bits (<21:18>).

Register File Output Enables. Bits <59:58> of each memory controller microinstruction are the register file output enable bits, active high. These bits cause the addressed location of the register file to be driven onto the memory controller address (MCA) bus.

The register file is divided into a 23-bit page portion and a 9-bit offset portion. The page portion, the offset, or both may be driven onto the MCA bus. Microinstruction bit <59> enables the offset portion onto the MCA bus; bit <58> enables the page portion onto the MCA bus.

The register file can be used as the source of physical and virtual addresses. When addresses are supplied from the register file to the MCA bus to access the translation buffer or cache, the register file must be addressed the cycle before the TB/cache access is made and the address maintained into the next cycle.

Microinstruction bit <59> is also the output enable for the CSRs. The four CSRs control memory management functions and reflect error status; they share the register file address space but are read and written over the memory controller data (MCD) bus. When bit <59> is a zero, the CSRs are enabled as well as the offset portion of the register file.

Register File Write Enables. Bits <23:22> of each memory controller microinstruction are the register file write enables, active low. These bits cause the data that are stable on the MCA bus to be written into the addressed location of the register file.

Bit <23> is the offset register file write enable; bit <22> is the page register file write enable. When bit <23> is asserted, data on the MCA bus are written into the 9-bit offset portion of the register file. When bit <22> is asserted, data from the MCA bus are written into the page portion of the register file.

Additionally, if the register file address is 8 through F inclusive, and the offset write enable is asserted (<23>), the datum MCD <0> is also written into the selected control and status register (CSR).

Register File Address Field. Bits <21:18> of each memory controller microinstruction are the register file address field, active high. These bits specify a 4-bit register file address which defines an address space shared by the register file and the control and status registers (CSRs). The encoding is listed in Table 7-5.

Table 7-5. Register File Address Space

<21:18> (hex)	Register File Location	Content
00	0	virtual address
01	1	physical address
02	2	I-stream PC
03	3	error code
04	4	zero
05	5	unused
06	6	unused
07	7	unused
08	do not use	
09	do not use	
0A	do not use	
0B	do not use	
0C	map enable control register	
0D	cache enable control register	
0E	error flag status register	
0F	instruction prefetch error (IB.ERROR) status register	

When a CSR is read, its register file address is specified in microinstruction bits <21:18> and its content enabled onto the memory controller data bus as MCD <0>. The microcode guarantees that no source is enabled on the MCD bus for one microcycle before, and one microcycle after, a CSR read.

For addresses 04 through 0F, bit <8> of the register file is not implemented, and always reads 0. The remaining 31 bits are read and written as usual from the MCA bus.

Transceiver Control

Bit <57> of each memory controller microinstruction is the transceiver enable bit, and bit <56> specifies the transceiver direction. These two bits control the operation of the MCT transceiver, which isolates the memory data bus (the 32-bit bus between the two modules) from the MCA bus (the 32-bit bus internal to the MCT; see Figure 4-1). The MCT transceiver is the data communication port to the DAP module for all data transfers except bytes from the I-stream. Table 7-6 shows the encoding for the transceiver control field of the MCT microinstruction.

Table 7-6. Transceiver Control Field

Transceiver Enable <57>	Transceiver Direction <56>	Result
1	0	DAP to MCT
1	1	MCT to DAP
0	x	no operation

TB/Cache Control

The translation buffer and the cache share a 4K location by 48-bit-wide RAM. Ten bits control whether the cache or the translation buffer is accessed, and how. These bits are the TB/cache index MUX bit <6> select (<54:53>), TB/cache RAM control (<36:33>), TB/cache valid bit (<32>), and the TB/cache access select (<31:29>).

TB/Cache Index MUX Bit <6> Select. Bits <54:53> of each memory controller microinstruction determine the source for index MUX bit <6>.

The index MUX selects the correct bits from the MCA bus to access the desired TB or cache location. In other words, the index MUX forms an address that selects a location in the translation buffer or the cache.

Bit 6 of the address supplied by the index MUX is sourced from various places; microinstruction bits <54:53> control the selection of the source for index MUX bit <6>. In addition, the register file output enable bit for the offset portion of the register file (microinstruction bit <59>), affects the selection of the source for index MUX <6>. Table 7-7 summarizes the sources for index MUX bit <6> and under what conditions each source is selected.

Table 7-7. Index MUX <6> Select

Bit <59>	Bits <54:53>	Source for Index MUX Bit <6>
0	00	bit <8> from the adder
1	00	bit <8> from register file location 0
0	01	MCA<8>
1	01	bit <8> from register file location 1
0	10	unused
1	10	bit <8> from register file location 2
0	11	MCA<15>
1	11	bit <8> from register file location 3

TB/Cache RAM Control. Bits <36:33> of each memory controller microinstruction are the TB/cache RAM control bits. These four bits control the read and write operations of the TB/cache RAM. All data are read from, and written to, the MCD bus. The encoding is shown in Table 7-8.

Table 7-8. TB/Cache RAM Control

<36:33>	Action
1111	no operation (and power down)
1100	translation buffer write
1010	cache write (normal or conditional)
0101	translation buffer read
0011	cache read

TB/Cache Valid. Bit <32> of each memory controller microinstruction is the TB/cache valid bit, active high. This bit directly controls a hardware TB/cache valid bit that is written into the TB/cache as part of the tag.

If bit <32> is a one, the hardware valid bit in the tag is written high to indicate that the associated information being stored in the TB or cache is a valid copy of the same information in memory. If bit <32> is a zero, the hardware valid bit is written low to indicate that the associated TB or cache entry is invalid.

If the hardware valid bit is written high, it enables a comparison between the tag for the cached entry and the presented address; this comparison may then produce a TB/cache hit. If the valid bit is written low, the comparison is disabled and a TB/cache miss is forced.

TB/Cache Access Select. Bits <31:29> of each memory controller microinstruction are the TB/cache access select bits. These bits select the type of TB/cache access to be performed. The encoding is shown in Table 7-9.

Table 7-9. TB/Cache Access Select

<31:29>	Access Type
000	normal TB access read or write
101	normal cache read or write
110	TB invalidate via adder
111	conditional cache invalidate

When <31:29> are 110, a translation buffer invalidate all (TBIA) operation occurs. For a TBIA, the translation buffer is accessed by MCA <10:2>, which select one of the 512 entries in the translation buffer. MCA <8:2> are supplied by the adder and <10:9> are read from the page portion of a register file location. (MCA <10:9> are supplied to a register file location by the data path microcode. MCA <31:11> and <1:0> are ignored for a TBIA.)

When the TBIA operation begins, MCA <10:2> select the first entry in the process portion of the translation buffer, or the first entry in the system portion of the translation buffer. The selected entry is in the process TB if MCA <10> is 0, and in the system TB if MCA <10> is 1. MCA <9> is set to zero and 128 entries are then accessed by using the adder to increment the value in MCA <8:2>. As each entry is accessed, it is marked invalid by clearing the hardware valid bit in its associated tag.

When the page crossing flag is set, the first 128 entries have been invalidated and the memory controller sets MCA <9> to 1 to select the second block of 128 entries. When these entries are invalidated the TBIA function is complete.

When <31:29> are 111, and a cache write operation is selected (microinstruction bits <36:33> = 1010), the

cache is accessed for a conditional cache invalidate. If a cache hit occurred in the previous cycle, the hardware valid bit is cleared as specified by bit <32>, the TB/cache valid bit; that is, if a cache hit occurred, that cache entry is marked invalid. All other bits in that cache entry are written to an undefined state. If a cache hit did not occur in the previous cycle, this cycle is a no operation.

Prefetch FIFO Control

Bits <52:51> of each memory controller microinstruction are the instruction prefetch FIFO control bits. The RAM and the associated control logic provide first-in-first-out storage for up to 16 bytes of prefetched data from the instruction stream. The control logic asserts the prefetch enable flag whenever less than 8 bytes of I-stream data are contained in the prefetch FIFO.

Prefetch FIFO Clear. Microinstruction bit <52> is active low and causes the entire contents of the I-stream prefetch FIFO to be cleared. This direct clear function is used to synchronize the FIFO to the instruction stream after program flow changes.

Prefetch FIFO Load Clock. Microinstruction bit <51> is active high and controls the clocking of data from the low byte of the MCA bus into the I-stream prefetch FIFO. This clock occurs at the end of the current MCT microcycle so that data are fetched and written into the FIFO in one cycle.

Reverse Pass Latch Control

The reverse pass latch allows data on the MCA bus to be driven onto the MCD bus. (See Figure 4-1 in Chapter 4 for the location of the reverse pass latch.) The reverse pass output enable (bit <44>) and the

reverse pass latch enable (bit <43>) are the two bits that control the reverse pass latch.

Reverse Pass Output Enable. Bit <44> of each memory controller microinstruction is the reverse pass output enable bit, active low. This bit causes the current contents of the reverse pass latch to be enabled onto the MCD bus.

Reverse Pass Latch Enable. Bit <43> of each memory controller microinstruction is the reverse pass latch enable bit, active low. This bit causes the data on the MCA bus to be latched into the 32-bit-wide reverse pass latch. The reverse pass latch is transparent, allowing data from the merge register to be passed back to the MCD bus, rotated, and presented to the merge register inputs in one cycle.

Status Control Field

The 3-bit status control field is shown in Figure 7-2 and consists of:

- the busy control field, bits <17:16>, and
- the sign-extend bit, <15>.

Busy Control

Bits <17:16> of each memory controller microinstruction are the busy control field bits. These bits determine the state of the MEM BUSY signal to the data path module. Together, these bits synchronize data transfers between the memory controller module and the data path module. The encoding is shown in Table 7-10.

Table 7-10. Busy Control Field Encoding

<17:16>	Function
00	unconditionally clear busy
01	no operation
10	not used
11	conditionally clear busy

The conditionally clear busy state causes the MEM BUSY signal to be cleared if a TB miss, modify refused error, or access violation occurs during a TB access. Similarly, the conditionally clear busy state causes the MEM BUSY signal to be cleared if a cache hit occurs during a cache access.

The unconditionally clear busy state is not dependent on any conditions; if bits <17:16> are both zeros, the MEM BUSY signal is explicitly cleared. Unconditionally clear busy is used, for example, when the microcode determines that it has reached the end of an extended memory request.

Sign-Extend Word Control Field

This single bit field causes the sign-extend word flag to be asserted to the sign-extend logic on the data path module. This bit (<15>) indicates that the memory request from the data path was an I-stream Request with IB.WORD specified so that a word is read from the instruction stream, and should therefore be sign-extended to a longword.

The particular case for which this bit is needed is when the requested instruction stream word is a word displacement, and will be added directly to the program counter on the data path chip.

Once the sign-extend bit is set in a memory controller microinstruction, the sign-extend word flag remains set

until the memory controller accepts the next memory request dispatch from the data path module.

Microprogram Control Field

The 15-bit microprogram control field provides the information for the MCT microsequencer to determine the address of the next MCT microinstruction. The generated microaddress is then used to access control store and retrieve the next MCT microinstruction.

The control store address space consists of 1,024 locations; each location contains a 64-bit microinstruction. A 10-bit microaddress is presented to the MCT control store to access the next microinstruction.

Figure 7-2 shows the microprogram control field divided into three subfields: microsequencer control, branch control, and next address. The following sections describe these subfields in more detail.

Microsequencer Control

Bits <14:13> of each memory controller microinstruction are the microsequencer control field bits. These bits control the next microaddress to be executed by the MCT micromachine. The encoding for these bits is shown in Table 7-11.

Table 7-11. Microsequencer Control Field Encoding

<14:13>	Function
00	enable trap, disable dispatch, jump
01	disable trap, disable dispatch, jump
10	enable trap, enable dispatch, jump
11	enable trap, disable dispatch, return from trap

Trap is defined as a cache invalidate trap to address 3FF; dispatch means dispatch on the memory request supplied by the data path; jump is the microaddress created by bits <9:0> of the microinstruction, with bits <3:0> modified by branch conditions.

Branch Control

Bits <12:10> of each memory controller microinstruction are the branch control field bits. This field selects one of eight groups of status conditions that can be ORed with the four least significant bits of the next address field to cause conditional microprogram branches. The branch control field influences the next microaddress in the same way that the OR <2:0> field does in the data path microcode. Figure 7-3 shows the encodings for the branch control field.

Next Address

Bits <9:0> of each memory controller microinstruction are the next address field. These bits specify the next microaddress that the microsequencer will execute if no conditional branching, dispatch, or trap occurs.

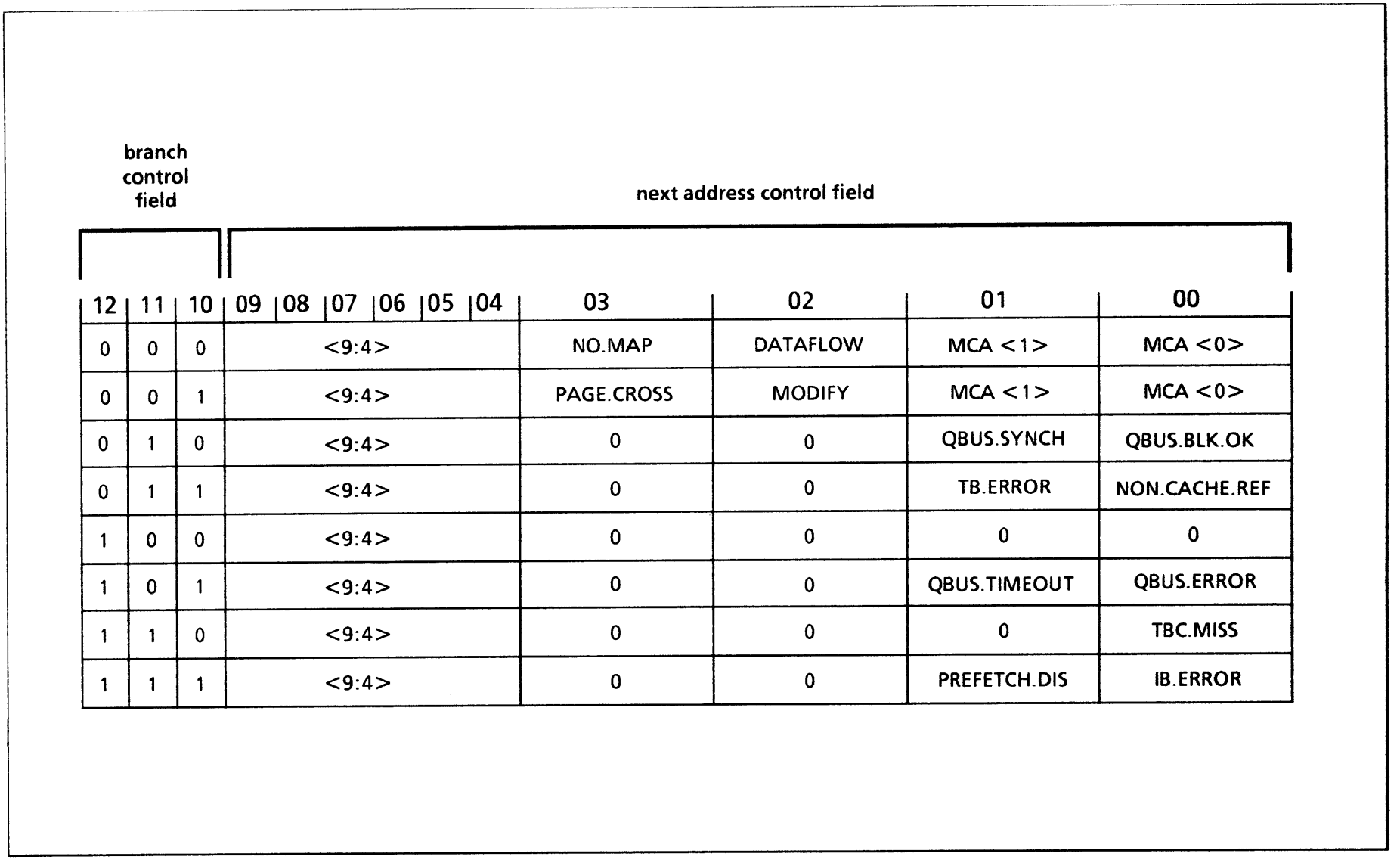
The four lowest bits, <3:0>, can be modified by the branch conditions as selected by the branch control field. Figure 7-3 shows the branch control field and corresponding branch conditions. The result is that two, four, eight, or sixteen-way branching can occur.

When a two-way branch is coded in a memory controller microinstruction, one of the four lowest bits in the microinstruction is zero; when a four-way branch is coded, two of the four lowest bits in the microinstruction are zero; when an eight-way branch is coded, three of the four lowest bits in the microinstruction are zero; when a sixteen-way branch is coded, the low four bits

are all zeros. The bits that are zeros can be changed by selected branch conditions.

Branch Conditions

When a branch condition signal is asserted, it causes a branch to the address of the memory controller microroutine that handles that branch condition. There are sixteen branch conditions that can influence the address of the next MCT microinstruction; these branch conditions are listed in Figure 7-3, and described further in the following paragraphs.



12	11	10	09 | 08 | 07 | 06 | 05 | 04	03	02	01	00
0	0	0	<9:4>	NO.MAP	DATAFLOW	MCA <1>	MCA <0>
0	0	1	<9:4>	PAGE.CROSS	MODIFY	MCA <1>	MCA <0>
0	1	0	<9:4>	0	0	QBUS.SYNCH	QBUS.BLK.OK
0	1	1	<9:4>	0	0	TB.ERROR	NON.CACHE.REF
1	0	0	<9:4>	0	0	0	0
1	0	1	<9:4>	0	0	QBUS.TIMEOUT	QBUS.ERROR
1	1	0	<9:4>	0	0	0	TBC.MISS
1	1	1	<9:4>	0	0	PREFETCH.DIS	IB.ERROR

Figure 7-3. Branch Control Field and Next Address Field Formats

NO.MAP

When this signal is asserted, memory management is turned off; that is, no address translation takes place so all addresses are treated as physical addresses, no access checking is performed and there is no memory protection.

The map enable bit is one bit in an internal processor register (IPR) on the data path chip; this bit is set and cleared by executing a MTPR macroinstruction. The data path then places a copy of the map enable bit in the memory controller map enable control register (one of the CSRs in the register file) by executing a WRITE.MAP.ENABLE Memory Request; the address of the CSR is specified as part of the function code.

DATAFLOW

This branch condition signal is asserted directly from bit <28> in the data path microinstruction that is making the memory request. If bit <28> is a zero, the requested memory operation is a read; a one indicates a write.

This bit is latched in the memory function latch on the data path module, and transmitted to the memory controller as BUS MEM CTL 5. On the memory controller module, this signal is one of the inputs to the branch MUX.

MCA <1> and MCA <0>

These bits are the low two bits of the virtual or physical address currently on the MCA bus. They are used to indicate what data alignment must be performed.

PAGE.CROSS

If the adder is enabled and the add operation results in a carry into the tenth bit, the page crossing signal is asserted.

MODIFY

This branch condition signal is asserted directly from bit <29> in the data path microinstruction that is making the memory request. If bit <29> is a zero, the access intent is read; a one indicates an access intent of write.

The modify intent signal (DAPT MODIFY) is transmitted from DAP to MCT over pin DP1 in the backplane. Once on the memory controller module, this signal is one of the inputs to the branch MUX.

QBUS.SYNCH

The Q22 bus controller sends the SYNCREADY signal to the memory controller branch condition logic to indicate Q22 bus controller status.

On a read operation, SYNCREADY means that the requested data are available and can be driven onto the memory controller MCD bus. On a write, SYNCREADY means that the Q22 bus controller is ready to receive the write-to address or the data to be written.

QBUS.BLK.OK

The Q22 bus controller sends the BLOCK MODE OK signal to the memory controller branch condition logic when the physical memory on the Q22 bus supports block mode; that is, when data can be read or written in blocks of 1 to 16 words (each word is 16 bits) within one Q22 bus cycle. The words are transferred one at a time

but the Q22 bus controller only needs to drive one address onto the bus at the beginning of the operation, and the cycle does not end until the desired number of words are transferred.

If the physical memory does not support block mode, the Q22 bus controller must drive a new address onto the Q22 bus for each word to be read or written.

TB.ERROR

This signal is asserted when anything goes wrong during a translation operation. It is actually the OR of the signals that indicate a page crossing, a TB miss, an access violation, or modify refused.

NON.CACHE.REF

This branch condition is generated if either bit <29> or bit <28> of a physical address is set.

Bit <29> is the high-order bit of the physical address currently on the MCA bus. MicroVAX I physical addresses are 23 bits long, with the address specified in <21:00> and the I/O space flag, bit <29>, appended as bit <22>. Bit <29> becomes part of the physical address in the address translation procedure.

When bit <22> of a physical address is a one, that address is located in I/O space, and is therefore not in the cache. (No I/O space address is cached.)

Bit <28> is one of the fifteen bits latched in the physical address register after an address translation operation. When set, it indicates that the address is located in a physical memory that can be shared by Q22 bus processors, and therefore, the address should not be cached. Although bit <28> is driven on the MCA bus as MCA <28>, it is an internal flag and is not sent over the Q22 bus as part of the physical address.

Thus, if either bit <29> or bit <28> is set, the address should not be cached, and the branch condition input signal NON CACHE REF is generated.

QBUS.TIMEOUT

The Q22 bus controller sends this signal to the memory controller branch condition logic when a Q22 bus device does not reply within the allowed time limit of 10 microseconds. This signal generally means that a read or write to a nonexistent memory location was attempted.

QBUS.ERROR

The Q22 bus controller sends this signal to the memory controller branch condition logic when either a timeout or a parity error occurs on the Q22 bus. This signal causes the current memory request to be aborted.

TBC.MISS

This signal is asserted when an address translation or cache access fails. It is not available until the cycle after the one in which the actual translation or cache access took place.

PREFETCH.DIS

This is the disable prefetch signal. It is asserted when the prefetch FIFO is full. It is deasserted when the FIFO content falls below the target value of eight bytes. This deassertion causes the memory controller to reload the prefetch FIFO from the I-stream.

IB.ERROR

When a page crossing occurs during prefetch, a bit is set in the IB.ERROR CSR. The IB.ERROR signal is then

asserted to the memory controller branch condition logic. The assertion of this signal causes the prefetch operation to halt until the correct page address can be supplied by the data path module.

Q22 Bus Controller Interface

Just as the memory controller module executes commands delivered by the data path, the Q22 bus controller executes commands delivered by the memory controller. The memory controller microcode communicates with the Q22 bus controller via four microcode bits: three bits of function code and the go bit. The Q22 bus controller sends back five status flags to communicate its state to the memory controller microcode. All of the microcode bits and most of the status flags are described earlier in this chapter, but are also summarized here for convenience.

Interface Microcode

The memory controller sends four microcode bits to the Q22 bus controller. They are:

- the go bit, which is microinstruction bit <63>. After the Q22 bus controller receives the function code in bits <50:48>, it acquires the bus and asserts an address. It then waits for the go bit before proceeding with the bus cycle.
- the function code, microinstruction bits <50:48>. These bits select the Q22 bus operation to be performed: no operation, write word, write byte, write block, read word, read block, read interrupt vector, or read interlocked. These Q22 bus operations are described in Chapter 9. See Table 7-1 for the encoding of the function code field.

The Q22 bus controller uses the 3-bit function code to determine the sequence of microstates needed to accomplish the given function.

Q22 Bus Controller Status

The Q22 bus controller communicates its state to the memory controller through five status flags. They are:

- QBUS.BLK.OK. This signal is asserted during a write block or read block operation to signify that the memory will be able to handle the next data transfer as a block mode transfer.
- QBUS.SYNCH. The signal SYNCREADY is asserted when data are available on a read from a Q22 bus device, and when data or address is needed for a write to a Q22 bus device.
- QBUS.TIMEOUT. This signal is asserted when the address of a nonexistent memory location is driven on the bus.
- QBUS.ERROR. This signal is the OR of the two error signals, bus timeout and parity error. It causes the current memory request to be aborted.
- Cache Invalidate. This signal is asserted whenever a bus device writes to physical memory. It alerts the memory controller to invalidate its copy of the written-to memory location if that address is in the cache.

This chapter describes the memory controller microcode and the memory controller/Q22 bus controller interface. The next chapter describes the hardware that implements the memory controller microcode.

Chapter 8
Memory Controller Module

This chapter is a detailed description of the components on the memory controller module and how they interact. First, the major logic elements and their hardware components are described. Then, the basic transfers of data between the logic elements are described on a microprogram level.

Overview of MCT Functions

The memory controller module contains hardware to perform the following eight functions:

- generate clock signals
- control MCT microinstruction flow
- translate virtual addresses
- access the data cache
- transfer data within the memory controller module
- prefetch instruction stream bytes
- track and report status
- communicate with the Q22 bus controller to read and write data

The next eight sections describe these functions, and the hardware components that implement them, in detail. The hardware components are illustrated in the MCT block diagram, Figure 8-1.

Generating the Clock Signals

All of the clocks for the MicroVAX I CPU are generated from a single 64 MHz clock on the memory controller module. The clock generator logic produces clocks for the data path module, the memory controller module, and the Q22 bus interface. The major clocks for the memory controller module are described in the next section.

MCT Clocks

The master clock on the memory controller module is a 16 MHz clock MCTM BASE CLOCK (62.5 ns period). This clock signal goes to the data path module over backplane pin CN2, and is the source for all the clock signals on the data path module. BASE CLOCK also synchronizes DAPL DCOK to generate the DAPL INIT signals. One of the DAPL INIT signals is DAPL MCT INIT. DAPL MCT INIT initializes the memory controller module to a known state and synchronizes the clock signals between the DAP and MCT modules.

MCTM CLK125 is the 125 ns period clock that controls the memory controller microcycle. Thus, two memory controller microcycles occur for every one data path microcycle. (DAPL CPU CLOCK is the 250 ns period clock that controls the data path microcycle.)

MCTM CLK62 simply divides the CLK125 signal in half, providing clocking control for each half of a memory controller microcycle.

MCTM MEMCLK is asserted for the first 31 ns of a microcycle, and deasserted for the remaining 94 ns. This clock signal controls TB and cache reads.

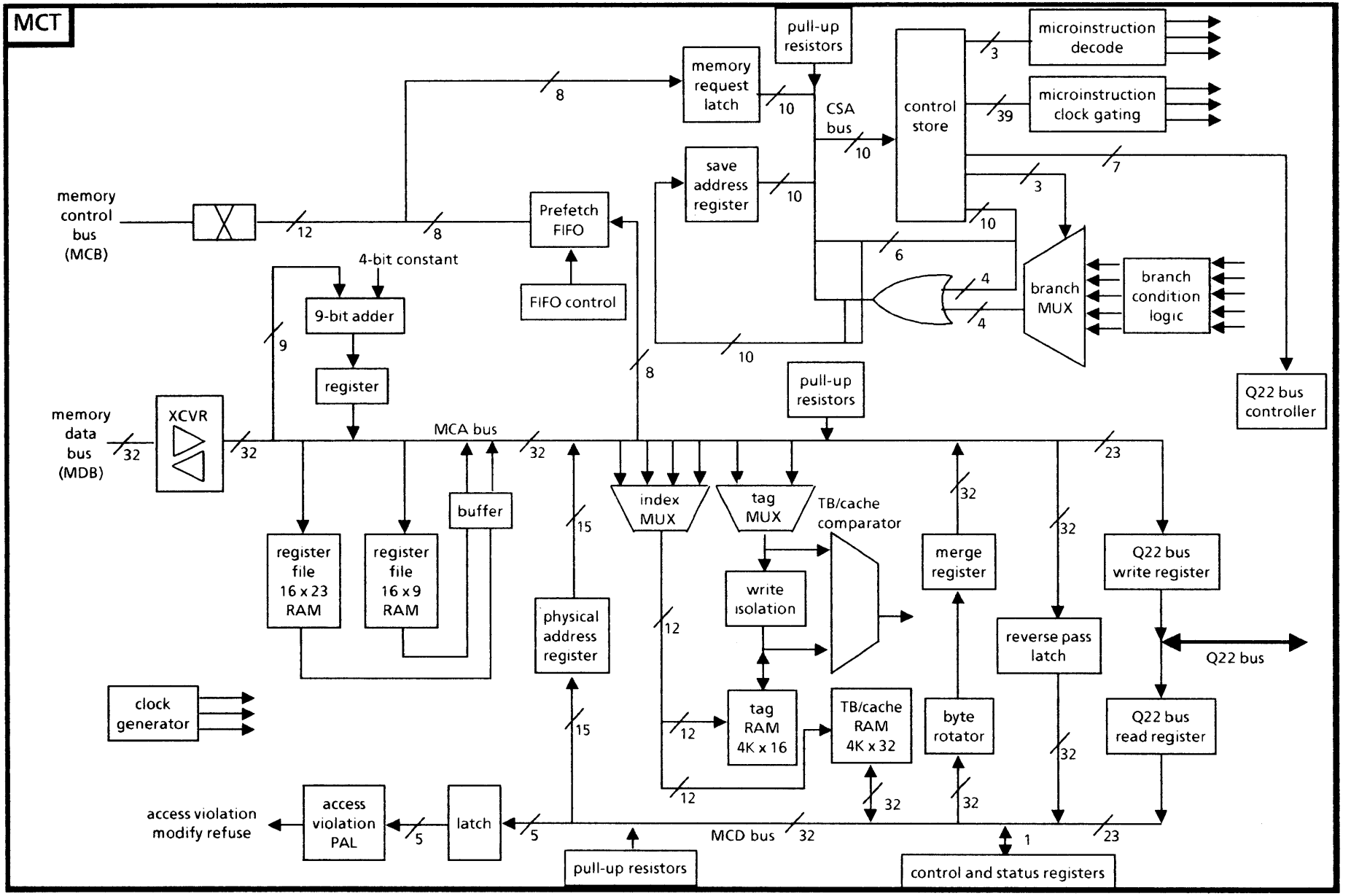


Figure 8-1. Memory Controller Block Diagram

MCTM DPC SRC is sent to the DAP module over backplane pin CP2. This clock signal is inverted to generate the signal DAPL DPC CLK. DPC CLK is the input clock signal to the data path chip.

MCTM DLYD CLK125 delays the CLK125 signal by 31 ns. This delayed clock enables writes to the register file and latches the reverse pass latch.

MCTM ADV CLK125 advances the CLK125 signal by 15 ns. This advanced clock is used to clock or latch data into MCA bus destinations.

Timing

A memory controller microcycle begins at the rising edge of MCTM CLK125 when the MCT microinstruction is available as the output of the control store. At this point, the microinstruction clock gating logic takes over to distribute the microinstruction control bits to their respective functional blocks.

Branch conditions are available to the MCT microsequencer by 50 ns into the current microcycle.

For reads from the translation buffer or the cache, the access address becomes available on the MCA bus between 0 and 28 ns into a microcycle, and the read occurs between 28 and 125 ns.

All MCA bus destinations are written by 110 ns into the current microcycle. If the bus destination is a latch part, the latch is open between 47 ns and 110 ns. If the bus destination is an edge sensitive part, data are clocked into the destination at 110 ns, which is the rising edge of MCTM ADV CLK125.

Controlling the MCT Microinstruction Flow

The memory controller module has its own set of hardware components to sequence the flow of memory controller microinstructions. These components are the memory request latch, the CSA bus, pull-up resistors, the control store, microinstruction clock gating, branch condition logic, and the microsequencer. The following paragraphs describe each of these components in turn.

Memory Request Latch

When the data path module executes a Memory Request or I-stream Request microinstruction, eight bits of control information are sent to the memory controller over the memory control bus, and four additional bits are sent over the backplane. These twelve bits are recombined on the memory controller module and the following eight bits are latched in the memory request latch:

- the second part flag, bit <7>
- the data type, bits <6:5>
- the memory function code, bits <4:0>.

The control store address (CSA) PAL in the MCT microsequencer supplies ones for bits <9:8> on a memory request dispatch.

The memory request latch is a tri-state latch located at the memory controller end of the memory control bus. The signal DAPR MEM REQUEST is asserted by the data path to inform the memory controller when a new memory function code is on the memory control bus.

This signal is synchronized with two clock signals to generate the signal MCTN MRL LE (memory request latch, latch enable). This signal causes the proper eight

bits to be latched into the memory request latch. If dispatches are enabled in the current MCT microinstruction, the contents of the memory request latch, plus the two high-order ones from the CSA PAL, are presented to the control store as the next microaddress.

CSA Bus

The control store address bus conveys the next microaddress to be executed to the control store. The CSA bus is 10 bits wide.

The low eight bits of the CSA bus are sourced from the next address buffer, the save address register, or the memory request latch, or they are passively asserted by pull-up resistors. The upper two bits are multiplexed through the CSA PAL.

The CSA bus destination is the address inputs to the control store.

Pull-up Resistors

When no other source is driving the bus, the pull-up resistors cause the default condition on the bus to be a logical high; that is, they pull up the bus. Thus, CSA bus bits <7:0> are all ones when the next address buffer, the save address register, and the memory request latch are not enabled.

MCT Control Store

The control store for the memory controller microinstructions is 1K deep by 64 bits wide. Each of the 1K locations contains one MCT microinstruction. Only 63 of the 64 bits are used.

The input to the control store is a 10-bit microaddress, which selects one location, and therefore one microinstruction.

The output from the control store is a 64-bit microinstruction that controls the MCT microsequencer, the Q22 bus interface, and all functional blocks of the memory controller. The encoding of the microinstruction bits is detailed in Chapter 7, but basically:

- Bit <63> and bits <50:46> control the Q22 bus interface.
- Bits <62:51> and <45:18> are the clocking and output enables for every functional block in the memory controller.
- Bits <17:15> control two status signals.
- Bits <14:0> control the memory controller microsequencer.

Microinstruction Clock Gating

This block of logic uses 39 of the 64 microinstruction bits from control store as input, and gates the appropriate latch and output enables to the proper functional blocks. Thus, the microinstruction clock gating logic controls when the various bus sources are enabled onto the MCA and MCD buses.

Branch Condition Logic

The branch condition logic monitors a group of conditions that affect the MCT status as reported to the data path, and can affect the MCT microprogram flow. Part of the logic is a group of flip-flops that act as a pipeline to save status for one additional cycle before it is discarded. The signals saved are:

- MCTL TBC HIT, which indicates a translation buffer or cache hit,
- MCTN MEM BUSY, which is asserted when the MCT is busy processing a memory request,

- MCTP NEXT IB VALID, which is the signal from the prefetch logic indicating the next byte in the prefetch FIFO is valid,
- MCTP PREFETCH EN, which is another signal from the prefetch logic indicating the prefetch FIFO contains less than eight bytes, and
- DAPR MEM REQUEST, which is the signal from the data path indicating a new memory function code is on the memory control bus.

The branch condition logic sends these three signals to the branch MUX in the MCT microsequencer:

- MCTT NO MAP, which indicates when memory management is disabled and sets up the branch condition NO.MAP,
- MCTT REG PREFETCH EN, which indicates the prefetch FIFO contains less than the desired number of bytes; when REG PREFETCH EN is deasserted, it sets up the branch condition PREFETCH.DIS, and
- MCTT IB ERROR, which indicates to the data path that the MCT cannot supply the next instruction stream byte because a page crossing has occurred, and sets up the branch condition IB.ERROR.

MCT Microsequencer

The memory controller microsequencer generates the next 10-bit microaddress every 125 ns. It provides conditional branching based on MCT internal and external conditions. Branch conditions are assumed to be stable no later than 50 ns before the next clock edge.

The MCT microsequencer consists of these components: the microinstruction decode logic, CSA PAL, the branch MUX, save address register, and next address

buffer. These components are described in the following paragraphs. Figure 8-2 is a block diagram of the MCT microsequencer.

Microinstruction Decode Logic

The 10-bit microaddress used to access the control store comes from one of the following sources: the memory request latch, the save address register, or the next address buffer. The next microaddress can also be 3FF which is the address of the first microinstruction in the trap microroutine. The microinstruction decode logic selects which of these sources provides the next microaddress.

There are basically three inputs to this decode logic: the microseqencer control field (microinstruction bits <14:13>), the signal MCTT MEM REQ DLYD which is generated from a pipelined version of DAPR MEM REQUEST, and the signal MCTB CACHE INV which is a signal from the Q22 bus.

Microinstruction bits <14:13> enable traps, dispatches on memory requests from the data path, and returns from traps. (See Table 7-11 in Chapter 7 for the encoding of this field.)

MCTT MEM REQ DLYD is asserted when the data path drives a new memory function code onto the memory control bus.

MCTB CACHE INV is asserted when a Q22 bus device writes to physical memory.

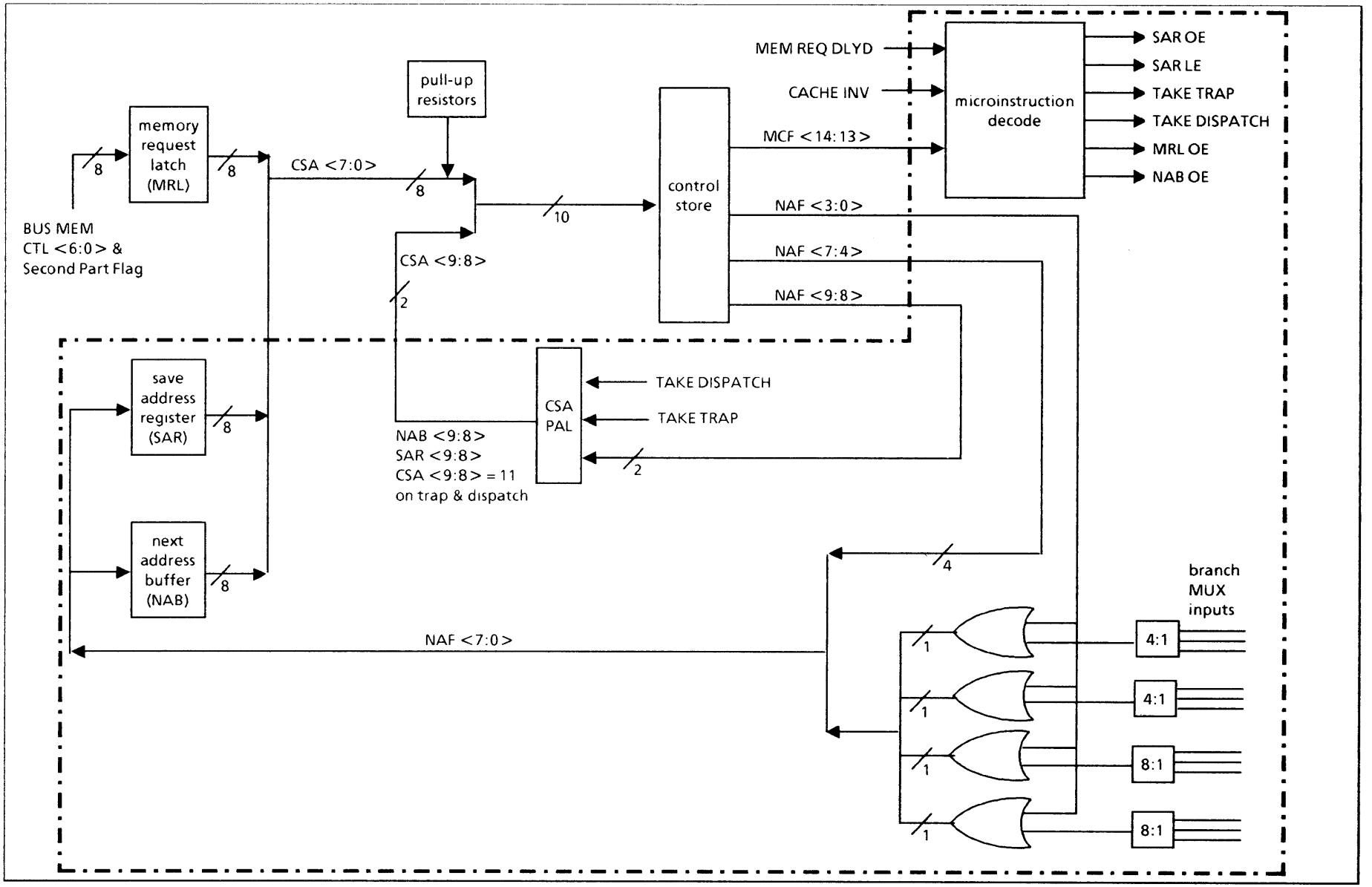


Figure 8-2. MCT Microsequencer Block Diagram

When dispatches are enabled and MEM REQ DLYD is asserted, the microinstruction decode logic asserts the signals MCTJ TAKE DISPATCH, and MCTJ MRL OE (memory request latch output enable).

When MCTJ MRL OE is asserted, the eight memory function bits in the memory request latch are presented to the control store as the low-order eight bits of the next microaddress. The high-order two bits are both ones, and they are driven onto the CSA bus by the CSA PAL in the microsequencer whenever the signal TAKE DISPATCH is asserted.

When traps are enabled and CACHE INV is asserted, the microinstruction decode logic asserts the signals MCTJ TAKE TRAP and MCTJ SAR LE (save address register latch enable), and the pull-up resistors control the bus. The pull-up resistors force next microaddress bits <7:0> to ones. TAKE TRAP also causes the CSA PAL in the microsequencer to force the high-order two bits of the next microaddress to ones. Therefore, the next microaddress is 3FF.

The signal MCTJ SAR LE enables the save address register (SAR). So when a trap is taken, the low eight bits of the microaddress that would have been presented to control store next if the trap had not occurred, are saved in the SAR. The two high-order bits are saved in the CSA PAL.

Bits <14:13> of an MCT microinstruction are 11 (binary) to enable returns when a return from trap is needed. For example, the last microinstruction in the trap microroutine located at 3FF has bits <14:13> set.

The signal SAR OE is asserted as the output of the microinstruction decode logic when bits <14:13> are set. SAR OE enables the contents of the save address register onto the CSA bus to be presented to the control

store as the low eight bits of the next microaddress. Bits <14:13> set also causes the CSA PAL to drive the two high-order bits that it saved when the trap was taken, onto the CSA bus. Thus, a return from trap is executed.

When jumps are enabled, and traps and dispatches are either not enabled or don't occur, the next microaddress is bits <9:0> of the current microinstruction from control store. If any branch conditions are in effect, they are ORed with the low four bits <3:0>. This microaddress, modified as appropriate, is passed through the next address buffer and presented to control store as the next microaddress.

CSA PAL

The next address buffer, save address register, and memory request latch drive microaddress bits <7:0> onto the CSA bus. The CSA PAL provides the two high-order bits, MCTN CSA <09:08>.

On a cache invalidate trap or memory request dispatch, the CSA PAL forces bits <09:08> to ones.

When a trap is taken, the two high-order bits of what would have been the next microaddress are saved in the CSA PAL. On a return from trap, the CSA PAL drives these saved bits onto the CSA bus. The low eight bits are driven onto the CSA bus from the save address register, which is enabled by the microinstruction decode logic.

For a normal operation where the next microaddress is provided by bits <9:0> of the current microinstruction, bits <9:8> are driven onto the CSA bus from the CSA PAL. Bits <7:0> are provided by the next address buffer; the low four bits are modified by any asserted branch conditions.

Save Address Register

When a trap occurs, the save address register stores microaddress bits <7:0> of the microinstruction that would have executed next. The signal MCTJ SAR LE asserted by the microinstruction decode logic causes the save address register to latch the microaddress bits.

When a return from trap is executed, the microinstruction decode logic asserts the signal MCTJ SAR OE to enable the save address register to drive the saved bits onto the CSA bus.

Next Address Buffer and Latch

The next address buffer latches bits <7:0> of the next microaddress by 94 ns into the current microcycle.

Bits <7:4> come from the current microinstruction, and bits <3:0> are the low four bits of the current microinstruction after they have passed through the branch MUX and been modified by any asserted branch conditions. These eight bits are then presented to control store as the low-order bits of the next microaddress if a trap or a dispatch does not occur.

When trapping and dispatching are not enabled or do not occur, the microinstruction decode logic generates the signal MCTJ NAB OE (next address buffer output enable). MCTJ NAB OE causes the bits that are latched in the next address buffer (next microaddress bits <7:0>) to be presented to control store.

Branch MUX

The branch MUX consists of two 4:1 multiplexers, two 8:1 multiplexers, and four OR gates. The branch conditions described in Chapter 7 and listed in Figure 7-3 are the inputs to the branch MUX.

Each 4:1 and 8:1 MUX selects one branch condition that is ORed with one of the low four bits of the microinstruction from the control store. The ORed signals are then passed through the next address buffer and become the low four bits of the next microaddress if the next address buffer is enabled by the microinstruction decode logic.

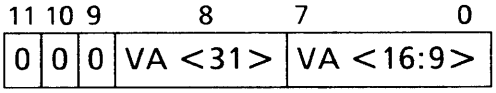
Translating Virtual Addresses

One of the main functions of the memory controller is to translate virtual addresses supplied by the data path module into physical addresses. The components used to do this are the index MUX, the tag MUX, the tag RAM, the TB/cache RAM, the write isolation buffer, the TB/cache comparator, the physical address register, the register file, and the 9-bit adder. The following paragraphs describe each of these components in turn, and the different kinds of TB accesses.

Index MUX

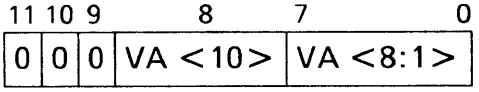
The index MUX selects the correct bits from the MCA bus to access the TB location that is to be read, written or invalidated. The twelve bits selected from the MCA bus by the index MUX form an address that accesses a location in the tag RAM; the same address is used to access a location in the translation buffer.

If a single translation buffer entry is being read, written, or invalidated, the index MUX selects virtual address bits <31> and <16:9> off the MCA bus. The index MUX supplies zeros for the high-order three bits, thus selecting the translation buffer portion of the tag RAM and of the TB/cache RAM. The address then presented to the tag RAM and to the TB/cache RAM by the index MUX is:



If VA <31> is a zero, the presented address selects a process space tag entry and translation buffer entry. If VA <31> is a one, the presented address selects a system space tag entry and translation buffer entry.

If the entire translation buffer is being invalidated by a Memory Request microinstruction with the INVALID.MULTIPLE function code, the index MUX selects virtual address bits <10> and <8:1> off the MCA bus. The index MUX again supplies zeros for the high-order three bits. The address then presented to the tag RAM and to the translation buffer by the index MUX is:



MCA <8> is not bused as the other MCA bits are; it is implemented via a hardwired multiplexer. The output of the multiplexer is bit <6> of the index MUX (MCTF INDEX 06). Thus, for any translation buffer access, index MUX <6> comes from one of seven possible sources: adder bit <8>, register file locations 0, 1, 2, or 3 bit <8>, MCA <8>, or MCA <15>. MCT microinstruction bits <59> and <54:53> select the source for index MUX <6>. (See Table 7-7 in Chapter 7 for the encoding of these microinstruction bits.)

Tag MUX

The tag MUX selects virtual address bits <30:17> from the MCA bus to form the low-order fourteen tag bits for the translation buffer entry being read, written or invalidated.

When the data path issues a Memory Request microinstruction with a function code that requires a translation buffer read or invalidate, the tag MUX passes bits <30:17> of the virtual address on the MCA bus to the TB/cache comparator.

When the data path issues a Memory Request microinstruction with a function code that requires a translation buffer write, the tag MUX passes bits <30:17> of the virtual address on the MCA bus to the write isolation buffer, which in turn passes them to the tag RAM, where they are written into the location accessed by the index MUX.

Tag RAM

The tag RAM is a 4K locations by 16-bit-wide memory array that stores the address tag bits associated with each translation buffer and cache entry. The first 512 locations in the tag RAM contain the translation buffer tags for process and system space, the next 1.5K locations are unused, and the last 2K locations contain the translation buffer tags for the data and instruction cache. Figure 8-3 shows the organization of the tag RAM.

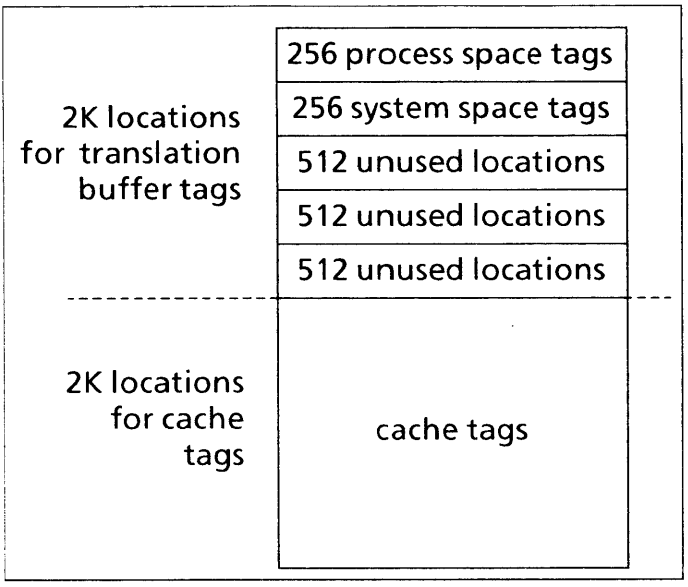


Figure 8-3. Organization of Tag RAM

Each translation buffer tag is 16 bits wide, consisting of one valid bit controlled by bit <31> of every memory controller microinstruction, one spare bit, and 14 tag bits which are virtual address bits <30:17>. Figure 8-4 shows the organization of one tag entry in the translation buffer:

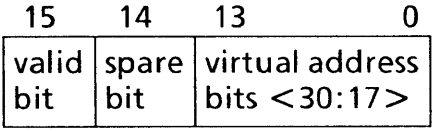


Figure 8-4. Translation Buffer Tag

The tag RAM is written whenever the TB/cache RAM is written. When a new TB tag is written into the tag RAM, bits <30:17> of the virtual address on the MCA bus are stripped off by the tag MUX, passed through the write isolation buffer, and written into the tag RAM location selected by the index MUX. Tag bit <15>, the tag valid bit, is set or cleared by the memory controller microinstruction executing the write to the translation buffer, to mark the entry as valid or invalid.

The tag RAM is read whenever the TB/cache RAM is read. For a read, the tag MUX passes bits <30:17> of the virtual address on the MCA bus to the TB/cache comparator. The tag at the tag RAM location selected by the index MUX is also sent to the comparator. If tag bit <15> is clear, indicating an invalid tag, the comparator generates a TB miss indication. If tag bit <15> is set and tag bits <13:0> match virtual address bits <30:17> stripped off the MCA bus by the tag MUX, the comparator generates the signal MCTL TBC HIT.

When a Memory Request microinstruction from the data path requests a translation buffer invalidate function, the tag MUX selects virtual address bits <30:17> from the MCA bus and passes them through the write isolation buffer to the tag RAM, where they are written into the tag RAM location accessed by the index MUX bits. As the tag is written, the valid bit is cleared marking the tag invalid.

TB/Cache RAM

The TB/cache RAM is a 4K locations by 32-bit wide memory array. It is organized the same way the tag RAM is organized, with the lower 2K locations containing process and system space page table entries (PTEs), and the upper 2K locations containing data and in-

struction cache entries. Figure 8-5 shows the organization of the TB/cache RAM.

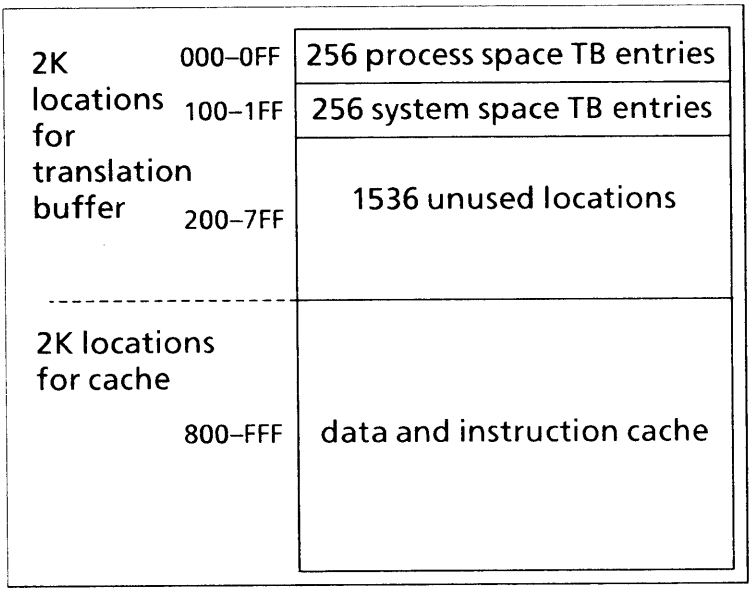


Figure 8-5. Organization of TB/cache RAM

Each translation buffer location contains a PTE (page table entry). Figure 8-6 shows the organization of one page table entry in the translation buffer:

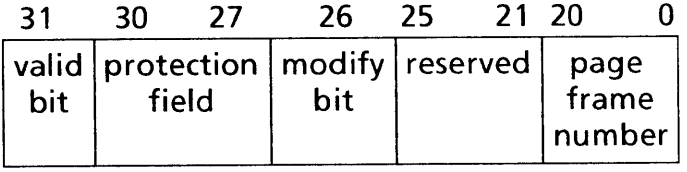


Figure 8-6. Translation Buffer PTE

PTEs are read from or written to the translation buffer from the MCD bus when a Memory Request from the data path specifies a function requiring a translation buffer access. Bits <36:33> of the MCT microinstruction that is executed to implement the requested function, determine whether the current TB access is a read or a write. (See Table 7-8 for the encoding of these bits). Bits <31:29> of the MCT microinstruction determine whether the current TB access is a normal access (that is, a read or a write), or a translation buffer invalidate. (See Table 7-9 for the encoding of these bits).

The TB/cache RAM address selected by the index MUX bits on a TB access is the same address selected in the tag RAM by these same index MUX bits. For example, if the address presented to the tag RAM by the index MUX is OCD (hex), then location OCD in the TB/cache RAM is accessed at the same time.

Write Isolation Buffer

This 16-bit-wide tri-state isolator allows the tag portion of a TB entry to be written into the tag RAM during a translation buffer write access or a translation buffer invalidate operation. For a read from the translation buffer, the write isolation buffer is disabled to allow the tag from the tag RAM to be read to the comparator.

TB/Cache Comparator

The TB/cache comparator is a 16-bit comparator that asserts a TB/cache hit indication for a translation buffer read when the stored address tag from the tag RAM equals the search address tag supplied by the tag MUX. When the comparison results in a match, the comparator asserts the signal MCTL TBC HIT. If this signal is not asserted, the compared tags did not match.

Physical Address Register

The physical address register is hardwired to form the physical page address from the page frame number (PFN) of the PTE read during a translation buffer access.

A total of fifteen bits are latched in the physical address register (PAR) following an address translation. Bits <12:0> of the PTE PFN are latched as the low-order thirteen bits of the PAR; these bits form physical address bits <21:09> when driven onto the MCA bus.

Bit <19> from the PTE is the next highest-order bit in the PAR; it is driven onto the MCA bus as MCA <28>. This bit indicates whether the address is located in shared memory.

The logical AND of PFN <20> and the TB/cache hit signal is latched as the high-order bit in the PAR; this bit forms physical address bit <29> when driven onto the MCA bus. When <29> is a one, the physical address is located in I/O space.

When the PAR output enable bit (bit <61>) is a zero in the current MCT microinstruction, the PAR contents are driven onto the MCA bus as MCA <29:28> and MCA <21:09>.

Register File

The register file is a 16-location by 32-bit-wide block of general storage within the memory controller. It is organized into a 9-bit-wide offset address portion and a 23-bit-wide (virtual or physical) page address portion; that is, the 23 bits select the memory page, and the 9 bits select the byte offset within the memory page. The register file address space is shared with the control

and status registers (CSRs). Table 7-5 shows the register file address space.

Although there are sixteen locations, only the first four are accessible as longwords. Bit 8 of locations 5 through 15 (addresses 04 through 0F) is not implemented and reads as 0.

Virtual addresses are stored at register file address 0, which is the first of the register file locations, and physical addresses are stored at address 1, the second location. (These addresses are assigned by the microprogram, not fixed by the hardware.) Virtual and physical addresses can be sourced onto the MCA bus from the register file.

The third location stores the instruction stream prefetch program counter, which always points to the last instruction stream byte that was stored in the prefetch FIFO.

Location four stores error codes, location five is the longword constant zero, and the next seven locations are not used. The addresses of the last four locations overlap the addresses of the CSRs and are not used.

The register file is written from the MCA bus when the register file write enable bits (<23:22>) in the current MCT microinstruction are asserted; microinstruction bits <21:18> determine the register file location that is written.

The contents of the addressed location in the register file are driven on the MCA bus when the register file output enable bits (<59:58>) in the current MCT microinstruction are asserted; microinstruction bits <21:18> determine the register file location that is addressed. A buffer/isolator passes the output from the register file to the MCA bus. Like the register file, the

buffer/isolator is divided into a 23-bit page portion, and a 9-bit offset portion.

Adder and Adder Register

The 9-bit adder contains page-crossing detection, and has a tri-state register associated with it. The adder and register provide 9-bit counts by successive increments of −8 through +7 inclusive, and the means for modifying virtual or physical page offset addresses. The adder and register are controlled by six bits in the MCT microinstruction: the three adder constant select bits (<26:24>), the adder subtract enable bit (<27>), the adder output enable (<60>), and one latch enable bit (<28>) for the register.

The source for the adder is always data bits <8:0> from the MCA bus. The output from the adder is the modified data bits <8:0> which are driven onto the MCA bus.

The adder is used for generating the multiple memory addresses required by unaligned data accesses, and for probing the last bytes of word and longword data types to insure access privilege for the entire datum during writes. For translation buffer accesses, the adder provides successive TB entry addresses when the entire translation buffer must be invalidated.

Translation Buffer Operations

Each translation buffer entry can be read, written, or invalidated. The following paragraphs describe how these accesses are accomplished.

Address Sources

Virtual addresses for translation buffer operations can come from any of the following: the data path, the

register file, or the register file modified by the 9-bit adder.

TB Reads

There are two kinds of translation buffer reads. The translation buffer is read for an address translation operation, and it is read when the data path issues a Memory Request with the READ.TB function specified.

For an address translation TB read, the selected PTE is driven onto the MCD bus and fourteen of the page frame number (PFN) bits are latched into the physical address register to form physical address bits <21:09>. The contents of the PAR are then driven onto the MCA bus and to the data path via the memory data bus. The TB hit or miss status is available to the data path the same cycle and the data on the memory data bus are used or ignored accordingly.

When a READ.TB Memory Request is issued by the data path, the PTE is read out of the addressed location in the translation buffer and latched in the merge register, without any rotation. From the merge register, the PTE is delivered directly to the data path over the memory data bus.

TB Writes

For a TB write operation, the PTE is driven onto the MCD bus from the reverse pass latch and written into the addressed location in the translation buffer.

Meanwhile, the corresponding tag bits are selected from the virtual address on the MCA bus by the tag MUX and written into the tag RAM at the same address that the PTE is written into in the TB/cache RAM.

TB Invalidates

The translation buffer can also be accessed to invalidate a single entry, or to invalidate the entire buffer. A TB invalidate function (single or multiple) writes the valid bit, bit <15>, invalid. The remaining tag bits and the associated PTE in the translation buffer are undefined.

For an invalidate multiple operation, the index MUX selects bits <10:2> from the virtual address on the MCA bus. Bits <8:2> are supplied by the 9-bit adder. Bits <10:9> are supplied by a register file location.

One INVALID.MULTIPLE Memory Request invalidates 256 translation buffer entries at a time. The INVALID.MULTIPLE request specifies an address; bit <10> of this address selects the process space TB entries or the system space TB entries. If bit <10> is 0, the process space TB entries are invalidated. If bit <10> is 1, the system space TB entries are invalidated. The MCT microcode enables the adder and supplies it with a constant, and from that point on, the adder provides the low-order seven bits of the virtual address on the MCA bus. The 256 tags in the tag RAM are invalidated by clearing bit <15> in each tag using sequential TB writes.

Thus, one Invalidate Multiple request marks all of the TB locations in process or system space invalid.

Accessing the Cache

Another main function of the memory controller is to supply the data path with the requested data. The memory controller accesses the data and instruction cache first to try to supply the requested data. If the

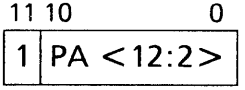
data is not in the cache, the memory controller initiates a Q22 bus cycle.

A cache access uses much of the same hardware as an address translation operation: the index MUX, the tag MUX, the tag RAM, the TB/cache RAM, the write isolation buffer, and the TB/cache comparator. The following paragraphs describe each of these components in turn, and the different kinds of cache accesses.

Index MUX

The index MUX selects the correct bits from the MCA bus to access the location in the cache that is to be read, written or invalidated. The twelve bits selected from the MCA bus by the index MUX form an address that accesses a location in the cache portion of the tag RAM, and the same location in the cache.

For any cache access, the index MUX selects physical address bits <12:2> off the MCA bus. The index MUX supplies a one for the high-order bit. The address then presented to the tag RAM and to the cache by the index MUX is:



Again, index MUX bit <6> is the output of a multiplexer that has the following inputs: bit <8> of register file locations 0, 1, 2, and 3, adder <8>, MCA <8>, and MCA <15>. Microinstruction bits <59> and <54:53> select one of these seven sources as index MUX bit <6>.

Tag MUX

The tag MUX selects physical address bits <21:13> from the MCA bus and supplies zeros for the five high-

order bits to form the low-order fourteen tag bits for the cache entry being read, written or invalidated.

When the cache is accessed for a read or an invalidate operation, the tag MUX passes bits <21:13> of the physical address on the MCA bus, plus the five high-order zeros, to the comparator.

For a cache write, the tag MUX passes bits <21:13> of the physical address on the MCA bus, plus the five high-order zeros, to the write isolation buffer. The write isolation buffer in turn passes them onto the cache portion of the tag RAM, where they are written into the location accessed by the index MUX.

Tag RAM

The tag RAM stores the address tag bits associated with each translation buffer and cache entry. The first 2K locations in the tag RAM contain the tags for process and system space addresses, and the last 2K locations contain the tags for the data and instruction cache. Figure 8-3 shows the organization of the tag RAM.

Each cache tag is sixteen bits wide, consisting of one valid bit controlled by bit <32> of every memory controller microinstruction, one spare bit, five zeros, and nine tag bits which are physical address bits <21:13>. Figure 8-7 shows the organization of one tag entry in the cache portion of the tag RAM.

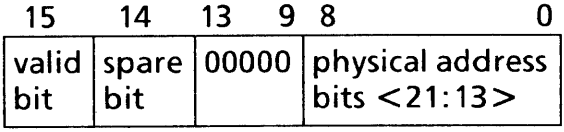


Figure 8-7. Cache Tag

The tag RAM is written whenever the TB/cache RAM is written. When a new tag is written into the cache portion of the tag RAM, the tag is assembled by the tag MUX, passed through the write isolation buffer, and written into the tag RAM location selected by the index MUX. Tag bit <15> is set or cleared by the memory controller microinstruction executing the write to the cache.

The tag RAM is read whenever the TB/cache RAM is read. For a read, the cache tag assembled by the tag MUX is passed to the comparator. The cache tag at the tag RAM location selected by the index MUX is also sent to the comparator. If tag bit <15> is clear, indicating an invalid tag, the comparator generates a cache miss indication. If tag bit <15> is set and tag bits <8:0> match physical address bits <21:13> stripped off the MCA bus by the tag MUX, the comparator generates the signal MCTL TBC HIT.

When the cache invalidate signal is sent to the memory controller from the Q22 bus, the tag MUX passes physical address bits <21:13> from the MCA bus to the comparator. The cache tag at the tag RAM location accessed by the index MUX is also sent to the comparator. If the two tags match, the cache tag at the location accessed by the index MUX is marked invalid by clearing bit <15>.

TB/Cache RAM

The cache portion of the TB/cache RAM is the upper 2K locations. Figure 8-5 shows the organization of the TB/cache RAM. Each cache location contains data or an instruction.

Data are read from or written to the cache from the MCD bus when a Memory Request from the data path specifies a function requiring a cache access. Bits

<36:33> of the MCT microinstruction that is executed to implement the requested function, determine whether the current cache access is a read or a write. (See Table 7-8 for the encoding of these bits). Bits <31:29> of the MCT microinstruction determine whether the current cache access is a normal access (that is, a read or a write), or a conditional cache invalidate. (See Table 7-9 for the encoding of these bits).

Write Isolation Buffer

The write isolation buffer is used the same way for cache operations as for translation buffer accesses: it allows the tag portion of a cache entry to be written into the tag RAM during a cache write, and is disabled to allow the tag from the tag RAM to be read into the comparator during a cache read or a cache invalidate.

TB/Cache Comparator

The TB/cache comparator asserts a TB/cache hit indication for a cache read or invalidate operation when the stored address tag from the tag RAM equals the search address tag supplied by the tag MUX. When the comparison results in a match, the comparator asserts the signal MCTL TBC HIT. If this signal is not asserted, the compared tags did not match.

Cache Operations

Each cache entry can be read, written, or conditionally invalidated. The following paragraphs describe how these accesses are accomplished.

Address Sources

Physical addresses for cache operations can come from any of the following: the data path, the register file, the

register file modified by the adder, the physical address register, or the merge register.

For cache accesses, the adder and its register are used to supply modified physical addresses for unaligned data accesses, and for probes to check access privilege (RCHECK and WCHECK Memory Request functions).

Cache Reads

For a cache read operation, the selected data are driven onto the MCD bus, through the byte rotator, and latched into the merge register. From the merge register, the data are driven over the memory data bus to the data path. The TB/cache hit or miss status is available to the data path in the same cycle and the data on the memory data bus are used or ignored accordingly.

Cache Writes

If the physical address presented to the TB/cache RAM during a cache read access results in a miss, the physical address is driven on the Q22 bus, and the data at that address obtained from physical memory. The Q22 bus delivers the data to the MCD bus for a cache write operation, and the data are written into the addressed location in the cache. Meanwhile, the corresponding tag bits are selected from the physical address on the MCA bus by the tag MUX and written into the tag RAM at the same address that the data are written into in the TB/cache RAM.

If either bit <29> or bit <28> is set in the physical address on the MCA bus, a cache write does not occur. Bit <29> set means that the physical address is located in I/O space, and no I/O space addresses are cached.

Physical address bit <28> is part of the PTE stored in the translation buffer. It is latched in the access violation PAL from the MCD bus after a translation buffer cycle. (See Figure 8-1 for the location of the access violation PAL). When bit <28> is set, the physical address is located in that part of physical memory that can be shared by another processor on the Q22 bus. The addresses in the shared portion of physical memory are not cached either. Thus, no cache write occurs if either bit <29> or <28> is set.

The cache is a write-through cache: any macroinstruction that causes a write updates physical memory. The write to physical memory is handled by a Q22 bus cycle.

Conditional Cache Invalidates

A cache invalidate operation is conditional: the addressed data cache location is invalidated if there is a match between the stored tag and the search tag. Only one cache entry at a time can be invalidated. For a cache invalidate operation, the valid bit, bit <15>, is cleared in the associated cache tag; the actual data in the cache entry are undefined.

Since the cache invalidate signal arrives from the Q22 bus asynchronously, conditional logic is implemented in hardware to enable the write to cache as soon as possible after the cache invalidate signal arrives. The cache invalidate signal is generated whenever an I/O device on the bus writes to physical memory and causes a cache invalidate trap in the memory controller. When bits <31:29> of the current MCT microinstruction are all ones, the TB/cache access selected is conditional cache invalidate. This condition plus the signal MCTT TBC HIT DLYD enables the cache write.

Transferring Data

Transferring data within the memory controller module is the fifth of the eight functions that the memory controller module performs. The hardware components are the MCA bus, the MCD bus, the memory data bus transceiver, the memory control bus, the merge register and byte rotator, and the reverse pass latch. The following paragraphs describe each of these components in turn.

MCA Bus

The memory controller address (MCA) bus is completely contained on the MCT module. It is a 32-bit, tri-state bus that normally supplies virtual and physical addresses to the TB/cache and to the Q22 bus write register. It is also the path used to transfer read data to the data path module, and write data from the data path module.

Addresses or data are asserted onto the MCA bus by 28 ns into the MCT microcycle, and are held until the end of the microcycle. The MCA bus destinations are clocked or latched by 110 ns into the microcycle. The MCT microcode guarantees that only one driver is enabled onto the bus during every cycle.

Table 8-1 lists the possible MCA bus sources and Table 8-2 lists the possible MCA bus destinations. When the physical address register (PAR) is driving the MCA bus, it only drives bits MCA <29:28> and <21:09>. MCA bits <31:30> and <27:22> are passively asserted by pull-up resistors, and <8:0> are provided by the offset register file or the adder.

Table 8-1. MCA Bus Sources

Sources	Data written to bus:
MDB transceiver	MCA <31:00>
page register file	MCA <31:09>
offset register file	MCA <08:00>
adder	MCA <08:00>
PAR	MCA <29:28>, <21:09>
merge register	MCA <31:00>
pull-up resistors	MCA <31:30>, <27:22>

Table 8-2. MCA Bus Destinations

Destinations	Data written from bus:
MDB transceiver	MCA <31:00>
page register file	MCA <31:09>
offset register file	MCA <08:00>
adder	MCA <08:00>
TB/cache MUXs	MCA <31:00>
reverse pass latch	MCA <31:00>
Q22 bus write register	MCA <29>, <21:00>
prefetch FIFO	MCA <07:00>

MCD Bus

The memory controller data (MCD) bus is also completely contained on the MCT module. It is normally used to transfer data to and from the TB/cache RAMs, transfer data from the Q22 bus read register, and transfer data to the PAR. It is also passively asserted by pull-up resistors. Table 8-3 lists the possible sources

for the MCD bus and Table 8-4 lists the possible destinations.

Table 8-3. MCD Bus Sources

Sources	Data written to bus:
TB/cache RAM	MCD<31:00>
reverse pass latch	MCD<31:00>
Q22 bus read register	MCD<15:00> or MCD<29>, <21:00>
control and status registers (CSRs)	MCD<00>
pull-up resistors	MCD<07:00>

Table 8-4. MCD Bus Destinations

Destinations	Data written from bus:
merge register via rotator	MCD<31:00>
TB/cache RAM	MCD<31:00>
control and status registers (CSRs)	MCD<00>
PAR	MCD<20:19>, <12:00>
access violation latch	MCD<30:26>

Memory Data Bus Transceiver

The MDB transceiver is located between the memory controller and the data path modules. It isolates the interboard memory data bus from the memory controller MCA bus and consists of both receive and transmit buffers.

Bits <57:56> in the MCT microinstruction control the transceiver direction, and enable the output. Table 7-6 shows the transceiver control field encoding.

Memory Control Bus

This 8-bit bidirectional bus is physically part of the CD interconnect on the backplane. The memory control bus conveys memory function requests from the data path module to the memory controller. The memory function requests are then latched in the MCT memory request latch. In addition, instruction stream bytes from the memory controller prefetch FIFO are time-multiplexed over the memory control bus and latched into the IBYTE register on the DAP module.

Merge Register and Rotate Logic

The merge register actually consists of four registers, each one byte wide. Each byte-wide register is enabled by one bit in the MCT microinstruction (the merge register selects, bits <42:39>). The entire 32 bits in the merge register are driven onto the MCA bus when bit <62> in the current MCT microinstruction is asserted.

The byte rotator consists of eight shifters; each shifter handles four bits. The 32 bits from the MCD bus are the inputs to the shifters and are driven onto the MCD bus from the cache, the reverse pass latch, or from the Q22 bus read register. The shifters are controlled by the byte rotate select field (bits <38:37>) in the MCT microinstruction. The byte rotator shifts the longword into the desired alignment (see Table 7-3).

Data to be read or written across byte or word boundaries are rotated by 0, 1, 2, or 3 bytes in the byte rotator, and latched in the merge register. Since the byte-wide registers in the merge register are individu-

ally enabled, any number and combination of the rotated bytes can be latched (see Table 7-2). If data from a previous rotate/merge operation are still in the selected register, the data are written over. This is exactly how data are updated on write-through cache operations, for example.

Reverse Pass Latch

The reverse pass latch allows data from the MCA bus to be latched and presented to the MCD bus. From the MCD bus, the data may be written to the translation buffer, the cache, or presented to the rotate/merge logic for alignment.

The input to the reverse pass latch is MCA BUS <31:00> and the output is MCD bus <31:00>. The reverse pass latch is controlled by one latch enable bit (<43>) and one output enable bit (<44>) in the MCT microinstruction.

Prefetching Instruction Stream Bytes

Prefetching bytes from the instruction stream is the sixth of the eight functions that the memory controller module performs. Three hardware components handle the prefetching: the prefetch FIFO, the FIFO control logic, and the prefetch program counter. The following paragraphs describe these components and how the prefetching operates.

Prefetch FIFO

The prefetch FIFO is a 16-locations-deep by 8-bits-wide RAM that holds the next 0 to 16 bytes from the instruction stream. The prefetch FIFO is loaded from the low byte of the MCA bus. The FIFO is loaded at the end of the current MCT microcycle if bit <51>, the

prefetch FIFO load clock bit, is asserted in the current microinstruction.

The output from the FIFO is the next byte in the instruction stream; it is sent to the IBYTE register on the DAP module via the memory control bus.

When a change in the program flow occurs (for example, because a Memory Request with IB.REFILL is executed), an MCT microinstruction with bit <52> asserted is executed to clear the entire contents of the FIFO.

Prefetch FIFO Control Logic

The control logic is a counter that keeps track of how many bytes have been loaded into the prefetch FIFO. When the FIFO contains less than eight bytes, the control logic asserts the prefetch enable flag, MCTP PREFETCH EN. This signal is one of the inputs to the branch MUX in the MCT microsequencer. When it is asserted, the MCT microcode branches to a micro-routine that refills the prefetch FIFO.

Prefetch Program Counter

The I-prefetch program counter is maintained in location 2 of the register file. This PC always contains the physical address of the last instruction stream byte loaded into the FIFO.

The prefetch PC is incremented whenever a byte from the MCA bus is loaded into the prefetch FIFO. The PC is also incremented when an I-stream Request microinstruction is executed by the data path. When the data path executes a Memory Request with IB.REFILL, the data path sends a 32-bit virtual address over the memory data bus; this address is translated and saved in location 2 of the register file as the new prefetch PC.

Prefetch Operation

The IBYTE control logic on the DAP module asserts the signal DAPR IB TAKEN to inform the memory controller that the instruction stream byte that was on the memory control bus is now latched in the IBYTE register. DAPR IB TAKEN causes the prefetch control logic to decrement the count by one, and causes the prefetch FIFO to drive the next I-stream byte onto the memory control bus.

When the prefetch FIFO drives another I-stream byte onto the memory control bus, it asserts the signal MCTP NEXT IB VALID to inform the DAP module that a valid I-stream byte is on the memory control bus. (This signal is transmitted over backplane pin DH2 as MCTT NXT VALID REG.)

When the prefetch enable flag is asserted because the prefetch FIFO contains less than eight bytes, a memory controller microroutine is invoked to refill the prefetch FIFO. The microroutine uses the physical address in the prefetch PC and performs a cache access.

If there is a cache hit, the retrieved data are driven into the byte rotator via the MCD bus, and rotated so that the desired byte (that is, the next byte in the I-stream) is in the low-byte position, and latched in the merge register. All 32 bits are then driven onto the MCA bus, but only the low eight bits are loaded into the prefetch FIFO. The prefetch PC is incremented by one.

The 32 bits are then driven off the MCA bus, through the reverse pass latch, to the MCD bus. From there, they are driven into the rotator and shifted again so that the next byte in the I-stream is in the low-byte position, and latched in the merge register. From the merge register, the rearranged 32 bits are driven onto

the MCA bus, and the low-order eight bits loaded into the next location in the prefetch FIFO.

This process continues until the prefetch FIFO contains more than eight bytes and the prefetch enable flag is negated. If there is a cache miss instead of a cache hit, the data are retrieved from physical memory, written to the cache, and then the same procedure carried out.

Thus, the prefetching is handled entirely by the memory controller. The result is that the memory controller always has the next byte of instruction stream data ready to be latched into the IBYTE register from the memory control bus.

Tracking and Reporting Status

Since the memory controller operates essentially as a slave responding to commands from the data path module, one of the memory controller's functions is to track status and report it to the data path. The hardware components that are involved with tracking and reporting status are the four CSRs (control and status registers), the access protection latch, the access violation PAL, the busy control logic, and the sign-extend logic. The next paragraphs describe these components and their activities.

Control and Status Registers

The four single-bit CSRs control memory management functions performed by the MCT, and reflect error status. The CSRs share the register file address space, but they are read and written over the MCD bus as MCD <0>.

The CSRs are selected by the MCT microinstruction register file address bits <21:18>. (See Table 7-5 for the encoding.) MCD <0> is loaded into the selected

CSR when bit <23>, the offset register file write enable, is asserted in the current MCT microinstruction. The content of the selected CSR is driven onto the MCD bus as bit <0> when bit <59>, the offset register file output enable, is asserted in the current MCT microinstruction.

The four CSRs are the map enable control register, the cache enable control register, the error flag status register, and the IB error status register. Each of these is described further in the following paragraphs.

Map Enable Control Register

This single-bit register is a copy of the Memory Management Enable Register, as defined by VAX architecture. (For more information about VAX architecture, see the *VAX Architecture Handbook*, order number EB-19580-20.)

When the bit in the map enable control register is set, address translation and access checking are enabled. When this bit is clear, all addresses are assumed to be physical addresses.

Cache Enable Control Register

The data and instruction cache is enabled when the bit in this CSR is set.

Error Flag Status Register

The bit in this CSR is set when an error has been detected during the execution of a Memory Request from the DAP module which the hardware is unable to report to the DAP micromachine directly. A code indicating the cause of the error is stored in location 3 of the register file. The code is created by reading a zero from location four of the register file, and adding a

constant to it in the adder. Errors and their corresponding codes are shown in Table 8-5.

Table 8-5. MCT Error Codes

Error Code	Error
0	Invalid State
1	Parity Error
2	Q22 Bus Timeout
3	Illegal Operation
4	Access Violation
5	Translation Check Error

When one of the errors listed in Table 8-5 is detected, the MCT microcode writes the corresponding error code in the error code location in the register file, and sets the error flag status bit by writing a 1 to the error flag status register via MCD <0>.

The error flag status bit is set directly by the hardware when an access violation is detected. The signal MCTT MEM ERR is sent to the DAP module over the backplane indicating a memory error has occurred. This signal is one of the inputs to the data path OR MUX.

Instruction Prefetch Error Status Register

The bit in this CSR is set by the prefetch microroutine when an error has been detected during an I-stream prefetch operation such that prefetching cannot continue without intervention by the DAP micromachine; for example, the prefetching crosses a page boundary. The signal MCTT IB ERROR is sent to the DAP module over the backplane. MCTT IB ERROR is ANDed with the signal DAPR IB INVALID to generate

the signal DAPR IB ERROR; DAPR IB ERROR is one of the inputs to the jump MUX.

The memory controller clears this CSR by an explicit write during an IB.REFILL Memory Request.

Access Protection Latch

The memory controller checks for access violations after every translation buffer access. The 8-bit access protection latch captures memory access information for determining access violations. The latch closes at the same time as the PAR; it stores PTE bits <30:26> (the protection field and the modify bit) from the MCD bus at the end of a translation buffer access. The latch also saves the TB hit or miss indication from the TB access. The PAR latch enable bit in the MCT microinstruction (bit <45>) also enables the access protection latch. The four protection field bits, the modify bit, and the TB hit or miss bit are passed on as the inputs to the access violation PAL.

Access Violation PAL

The access violation PAL decodes the necessary memory management data and asserts an access violation or modify refused indication when one of these errors occurs.

There are ten inputs to the access violation PAL:

- the six bits from the latch,
- the two access mode bits DAPT MEM REQ MODE <1:0> which are sent over the backplane on a Memory Request and specify the mode (kernel, executive, supervisor, or user) of the current Memory Request,
- the modify intent bit, DAPT MODIFY, also sent over the backplane, which specifies the modify

intent (read or write) of the current Memory Request, and

- the signal MCTS EN ACC CHECK; this signal is asserted after each translation buffer access to enable the access violation PAL.

The logic in the PAL does two comparisons. First, it compares the protection field from the PTE with the access mode bits and the modify intent bit from the current Memory Request microinstruction. The PAL asserts the signal MCTS ACC VIOL if the process that issued the Memory Request does not have sufficient privilege to access the page (corresponding to the PTE just retrieved from the translation buffer) for the intended read or write operation. MCTS ACC VIOL generates the signal MCTT MEM ERR to the DAP module, and causes an error code of 4 to be loaded in register file location 3.

For convenience, Table 8-6 shows the encoding of the PTE protection field, reproduced from the *VAX Architecture Handbook*.

The second comparison is between the modify bit from the PTE and the modify intent bit from the Memory Request. If the PTE modify bit is a 0 (meaning the associated page has not yet been modified), and the modify intent bit is a 1 indicating a write intent, the PAL asserts the signal MCTS MOD REF to indicate modify refused. MCTS MOD REF is sent over the backplane and is also one of the inputs to the data path OR MUX.

If neither an access violation or a modify refused is found, the next microcycle is the cache access.

If an access violation occurs, the DAP microcode traps to a memory management fault service routine, and generally, the process is aborted.

If modify refused occurs, the DAP microcode takes a modify refused trap and a new Memory Request micro-instruction is issued to rewrite that translation buffer entry (the PTE) with the modify bit set. Then a return is executed to the microroutine containing the original Memory Request, and the cache is accessed in the next microcycle.

The translation buffer miss signal that is one of the inputs to the access violation PAL, is passed through the PAL and sent over the backplane to the data path OR MUX as MCTS TB MISS.

Table 8-6. Protection Codes

Protection Field	Mnemonic	Kernel	Executive	Supervisor	User	Comment
0000	NA	no access	no access	no access	no access	No Access
0001		unpredictable	unpredictable	unpredictable	unpredictable	Reserved
0010	KW	read/write	no access	no access	no access	
0011	KR	read only	no access	no access	no access	
0100	UW	read/write	read/write	read/write	read/write	All Access
0101	EW	read/write	read/write	no access	no access	
0110	ERKW	read/write	read only	no access	no access	
0111	ER	read only	read only	no access	no access	
1000	SW	read/write	read/write	read/write	no access	
1001	SREW	read/write	read/write	read only	no access	
1010	SRKW	read/write	read only	read only	no access	
1011	SR	read only	read only	read only	no access	
1100	URSW	read/write	read/write	read/write	read only	
1101	UREW	read/write	red/write	read only	read only	
1110	URKW	read/write	read only	read only	read only	
1111	UR	read only	read only	read only	read only	

Busy Control Logic

A group of AND and OR gates act together as a latch to capture the current state of the memory controller; the current state is either busy or not busy. The latch is set and cleared by the busy control logic.

When the memory controller accepts a Memory Request from the data path, the microsequencer asserts the TAKE DISPATCH signal. TAKE DISPATCH asserted causes the CSA PAL in the microsequencer to assert the signal MCTN REQ ACK, and to deassert a signal named MCTN DONE HOLD.

The REQ ACK signal is sent over the backplane to the data path module to indicate that the memory controller has accepted the Memory Request.

The DONE HOLD signal is one input to the busy control logic. The deassertion of DONE HOLD causes the busy control logic to assert the signal MCTN MEM BUSY. This signal prevents the data path from issuing a memory request.

When the memory controller completes the function requested by the data path, the busy control logic clears MEM BUSY, thereby asserting the signal MCTN DONE. (The DONE signal is simply the MEM BUSY signal inverted.) When MEM BUSY is deasserted, data and status pertaining to the requested function are available to the data path.

The MEM BUSY signal can be cleared several ways. If bits <17:16> of the current microinstruction are 11 (binary) to allow the busy signal to be cleared conditionally, any one of the following events causes the busy control logic to deassert MEM BUSY during a translation buffer access:

- a translation buffer miss

- a modify refused error
- an access violation

If a translation buffer hit occurs and no modify refused or access violation errors are found, MEM BUSY remains asserted because the memory controller continues with a cache access.

If bits <17:16> = 11 (binary) when the memory controller does a cache access, a cache hit causes the busy control logic to deassert MEM BUSY. A cache miss causes the memory controller to initiate a Q22 bus cycle, and MEM BUSY remains asserted.

If bits <17:16> of a microinstruction are 00 (binary), the busy control logic immediately clears the MEM BUSY signal.

In summary, the signal MCTN DONE HOLD from the CSA PAL causes the busy control logic to assert the signal MCTN DONE as long as the memory controller is not servicing a memory request from the data path. The signal MCTJ TAKE DISPATCH from the microsequencer causes the busy control logic to deassert DONE and assert MEM BUSY. MEM BUSY is then cleared conditionally or unconditionally depending on microinstruction bits <17:16>, and DONE reasserted.

Sign-Extend Word Flag

The sign-extend word flag is the signal MCTH USEXT WORD, which is asserted when MCT microinstruction bit <15> is set. This signal is sent to the data path as MCTN SEXT WORD. Once the SEXT WORD signal is asserted, it remains asserted until cleared by the signal MCTN MRL LE at the next memory request dispatch.

MCTN SEXT WORD tells the data path to sign-extend the word being delivered over the memory data bus to the data path from the memory controller. SEXT

WORD is asserted whenever the data path has issued an I-stream Request to read a word displacement from the instruction stream.

In most cases, the data path knows it issued an I-stream Request specifying IB.WORD, and the SEXT WORD signal is unnecessary information. However, if the word displacement in the instruction stream crosses a page boundary, a page crossing error occurs and the memory controller returns control to the data path microcode.

The data path microcode determines the correct address in the next page for the second byte of the desired word, and issues a memory request with the REPEAT.SECOND function specified; that is, the data path simply reissues the memory request function parameters that are stored in the previous memory function latch.

In this case, the data path does not know it is issuing an IB.WORD I-stream Request. Therefore, the SEXT WORD signal from the memory controller is necessary information because the word displacement from the instruction stream must be sign-extended before it can be added to the program counter in the data path chip.

Communicating with the Q22 Bus Interface

Communicating with the Q22 bus interface is the eighth of the eight functions that the memory controller performs. The Q22 bus interface consists of the controller, the Q22 bus write register, and the Q22 bus read register. This section describes these components and how they interact with the memory controller.

Q22 Bus Controller

The controller handles the Q22 bus protocol, freeing the memory controller from this task. It is implemented as a programmable state machine and consists of several logic PALs and registers. These Q22 bus controller hardware components are described in more detail in Chapter 9.

The Q22 bus controller handles all bus arbitration, and is capable of executing block mode transfers to and from memory, if block mode is supported by the memory. If the Q22 bus is free, the arbitration logic in the controller sets up the bus address while waiting for the cache hit or miss signal.

The memory controller communicates with the Q22 bus controller via four bits of the MCT microinstruction. The controller reports status to the memory controller via five status flags. The microcode bits and the status flags are described in the section titled "Q22 Bus Controller Interface" in Chapter 7. The Q22 bus controller accepts function requests from the memory controller microcode and carries them out through its own set of microstates.

Q22 Bus Write Register

The write register latches addresses and data from the MCA bus that need to be driven onto the Q22 bus. The write register is actually one side of the bus transceivers that separate the memory controller module from the Q22 bus.

The Q22 bus write register is controlled by bit <46> in the current MCT microinstruction. When this bit is asserted, the data stable on MCA <29> and <21:00> are latched in the write register. The data written can be a 22-bit physical address, a 13-bit I/O space address,

or 16 bits of data to be written to physical memory or an I/O device.

The outputs from the write register are BDAL <21:00>; these are the Q22 bus data/address lines. If MCA <29> is set, it is driven onto the Q22 bus as the signal BBS7 to indicate that BDAL <12:00> represent a physical adddress in I/O space.

Q22 Bus Read Register

The read register latches addresses and data from the Q22 bus that need to be driven onto the MCD bus. The read register is actually the other side of the bus transceivers that separate the memory controller module from the Q22 bus.

The read register is controlled by bit <47> in the current MCT microinstruction. When this bit is asserted, the data on BDAL <21:00> are latched in the read register. The data latched can be a 22-bit cache invalidate address, a 9-bit Q22 bus interrupt vector, or 16 bits of data read from physical memory or an I/O device.

The outputs from the read register are MCD bus bits <21:00>. If the signal BBS7 is asserted, it is driven onto the MCD bus as MCD <29> to indicate that MCD <12:00> represent a physical adddress in I/O space.

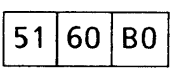
Microprogram Level Flow: MOVW

Now that the hardware components of the memory controller module have been described, this section takes a MOVW (move word) macroinstruction and describes how the memory controller accomplishes the translation buffer accesses, cache accesses and data retrieval necessary to return the requested data to the data path.

A MOVW macroinstruction replaces the destination operand with the source operand. A sample MOVW instruction is:

MOVW (R0), R1

The first operand specifier, "(R0)," uses register deferred address mode, and the second operand specifier, "R1," uses register mode. At some virtual address in memory, this instruction looks like this:



where B0 is the opcode, 60 specifies register deferred mode using R0, and 51 specifies register mode using R1. For this example, R0 contains the virtual address 00000211 (hex). This instruction obtains the word of data located at 00000211 and moves it into R1.

The next few paragraphs summarize the data path steps needed to decode and execute this MOVW macroinstruction.

Step 1. Evaluate the opcode to select the proper DAP microroutine for this macroinstruction.

Step 2. Evaluate the first operand specifier (the source) and obtain the first operand.

Step 3. Evaluate the second operand specifier (the destination) and write the first operand to the destination.

The remainder of this chapter describes the microprogram steps executed by the memory controller to translate the virtual address in R0, obtain the data located at the corresponding physical address, and return the data to the data path.

Whenever the memory controller is not doing anything, it loops in an "idle state" microroutine. In the idle state, the memory controller monitors the prefetch branch condition. If the branch condition is asserted, a jump is taken to the microroutine that refills the prefetch FIFO.

Dispatches are enabled during portions of the idle routine so that the correct location in control store can be accessed if a memory function request is received from the data path. Assume that the memory controller is in the idle state, that the prefetch FIFO is full, and that the entire MOVW macroinstruction is located in the prefetch FIFO.

Evaluating the Opcode

The MOVW opcode, B0, is clocked into the IBYTE register and decoded. The decode causes a dispatch to the microroutine that handles MOVW macroinstructions in the DAP control store.

The data path asserts the signal DAPR IB TAKEN, the prefetch FIFO drives the first operand specifier, 60, onto the memory control bus, and the prefetch counter is decremented by one. The memory controller asserts the signal MCTT NXT VALID REG to inform the data path that the next instruction stream byte is on the memory control bus. The data path asserts DAPR LOAD I BYTE and 60 is clocked into the IBYTE register.

Evaluating the First Operand Specifier

Next, the first operand specifier is decoded. The decode causes a dispatch to the DAP microroutine that handles the evaluation of operand specifiers with register deferred mode.

Once the decode has completed, the data path asserts IB TAKEN and the prefetch FIFO drives 51 (the second operand specifier) onto the memory control bus. The memory controller asserts NXT VALID REG, the data path asserts LOAD I BYTE, and 51 is clocked into the IBYTE register. So far, the memory controller has remained idle except for supplying the next instruction stream byte.

Meanwhile, the DAP microinstructions in the register deferred mode routine begin executing. The first microinstruction is a Memory Request with the function VREAD.RCHECK specified; the microinstruction in hex is 1E80B61628, where 1E is the opcode. This function asks the memory controller to perform a virtual read operation, with a check for read access. The data path assembles the twelve bits of information to send the memory controller. It latches these eight bits in both memory function latches because bit <31> in the Memory Request microinstruction is set:

- the two high-order bits are 01 from the size register, indicating a data type of word;
- the next bit is the data flow bit (<28>) from the Memory Request microinstruction which is a 0, indicating the data flow will be from MCT to DAP;
- the low-order five bits are the function code from the Memory Request microinstruction (bits <27:23>), and they are 00001 (binary) indicating the function VREAD.RCHECK.

The data path sends an additional four bits over the backplane: two access mode bits, one modify intent bit, and the second part flag. Since bit <30> (the mode bit) in the microinstruction is a 0, the two access mode bits sent over the backplane reflect the contents of the PSL.MODE register. The MOVW is part of the pro-

gram running in a User process, so the PSL.MODE contains 11 (binary).

The modify intent bit in the Memory Request microinstruction (bit <29>), is a 0 indicating read intent. The second part flag is also a 0 because this is the first part of this Memory Request to be executed.

The long operand of the Memory Request (bits <22:16>), specifies *pointer 1; that is, use the contents of the register pointed to by pointer 1. When 60 was decoded, pointer 1 was loaded with the value 0, pointing to R0. R0 contains the virtual address 00000211. Therefore, 00000211 is driven over the memory data bus to the memory controller.

Meanwhile, the eight bits in the first memory function latch are driven over the memory control bus, and the additional four bits of memory request information are driven over the backplane. The memory function control logic on the DAP module asserts the signal DAPR MEM REQUEST to inform the memory controller that a new function code is on the memory control bus. This signal generates the signal MCTN MRL LE which is the latch enable for the memory request latch, causing the following eight bits to be latched in the memory request latch on the memory controller:

- the high-order bit is the second part flag from the backplane, so it is a zero;
- the next two high-order bits are the data type bits, and they are 01, indicating data type word;
- the low-order five bits are the function code: 00001 (binary), indicating VREAD.RCHECK.

The data flow bit from the DAP memory function latch is one input to the branch MUX in the MCT microsequencer. The modify intent bit from the backplane is an input to the branch MUX, and an input to the access

violation PAL. The two access mode bits from the backplane are also inputs to the access violation PAL.

Up to now, the memory controller has remained in the idle state microroutine. Those microinstructions within the idle microroutine that have dispatch enabled also have 00 in the transceiver control field to enable the transceiver to drive data from the memory data bus onto the MCA bus. Thus, the virtual address 00000211 is now stable on the MCA bus.

The memory controller asserts the signal MCTN REQ ACK to inform the data path that the 32 bits on the memory data bus have been accepted, and the data path stops driving 00000211 over the memory data bus.

At this point, dispatches are enabled, the correct eight bits are latched in the MCT memory request latch, and a virtual address is stable on the MCA bus. The microinstruction decode logic in the MCT microsequencer sends 11 (binary) as the two high-order bits over the CSA bus, and control store is accessed with the 10-bit address 321 (hex). This is the address of the microroutine that handles the reading and writing of data words. The dispatch causes the MEM BUSY signal to be asserted.

Obtaining the Operand

At the rising edge of MCTM CLK125, the first microinstruction in the read/write data words microroutine is available at the output of the MCT control store. This first microinstruction causes:

- the data stable on the MCA bus (the virtual address 00000211) to be written into the register file at location 0 (the virtual address location),

- the low nine bits of the virtual address to be written into the adder, the constant 1 added to them, and the sum latched in the adder register,
- the MEM BUSY signal to remain asserted,
- a translation buffer read access, and
- the physical address register latch enable signal to be asserted.

TB Access

To accomplish the translation buffer read access, the tag MUX selects VA <30:17> from the MCA bus, which is 0000 (hex), and passes it to the TB/cache comparator.

At the same time, the index MUX selects VA <31> and VA <16:9> from the MCA bus, and provides zeros for the three high-order bits. For any normal read or write translation buffer access, the source for index MUX bit <6> is MCA <15>.

Thus, the output from the index MUX is the value 001 (hex), and so location 001 is accessed in the tag RAM and in the TB/cache data RAM. The 16-bit tag at location 001 in the tag RAM is passed to the TB/cache comparator. The tag is not 0000 so it does not match the search tag provided by the tag MUX, and the comparator does not assert the signal MCTL TBC HIT.

Simultaneously, the PTE at location 001 in the TB/cache RAM is driven onto the MCD bus. PTE bits <12:0> from the MCD bus are latched into the PAR as physical address bits <21:09>. PFN <19> is 0, so 0 is latched in the PAR as physical address bit <28>. The AND of the TB/cache hit signal and PFN <20> is 0, so 0 is latched in the PAR as physical address bit <29>.

The four-bit protection field and the modify bit from the PTE are also latched in the access protection latch.

The microprogram control field of this first microinstruction disables traps and dispatches, so the MCT microsequencer selects bits <9:0> from the first microinstruction, modified by any asserted conditions in the branch MUX, as the next microaddress.

Bits <9:0> are 0F0. The branch MUX input that modifies bit 0 is MCA <0> (see Figure 7-3), which is a 1 because of the VA 00000211 currently on the MCA bus. Thus, 0F1 is passed through the next address buffer and used to access control store as the next microaddress, and the first MCT microcycle ends.

Cache Access

At the rising edge of MCTM CLK125, the microinstruction at 0F1 is available at the output of the control store. This microinstruction causes:

- the 9-bit offset portion of the virtual address location in the register file to be driven onto the MCA bus as MCA <8:0>; that is, 011 hex,
- the fifteen bits in the PAR to be driven onto the MCA bus as MCA <29:28> and <21:09>,
- MCA <31:09> to be written into the page portion of the virtual address location in the register file (MCA <31:30> and <27:22> are asserted by pull-up resistors, MCA <29:28> and <21:09> are from the PAR),
- MEM BUSY to remain asserted,
- a cache read access,
- the data on the MCD bus from the cache access to be rotated 1 byte to the right and all four bytes latched into the merge register,

- the physical address on the MCA bus to be latched into the Q22 bus write register, and
- the Q22 bus function code for a read block operation (DATBI) to be sent to the Q22 bus controller; the go bit is not sent.

To accomplish the cache read access, the tag MUX selects physical address bits <21:13> from the MCA bus, supplies five zeros for the high-order bits and passes these fourteen bits to the TB/cache comparator.

At the same time, the index MUX selects PA <12:2> from the MCA bus, and provides a one for the high-order bit. The source for index MUX <6> is bit <8> from register file location 0. The 16-bit tag at the accessed location in the tag RAM is passed to the TB/cache comparator. The tag does not match the search tag provided by the tag MUX, and the comparator does not assert the signal MCTL TBC HIT.

Simultaneously, the data at the accessed location in the TB/cache RAM are driven onto the MCD bus, rotated one byte to the right and latched into the merge register.

The microprogram control field of this second microinstruction also disables traps and dispatches, so the MCT microsequencer selects bits <9:0> from the first microinstruction, modified by any asserted conditions in the branch MUX, as the next microaddress.

Bits <9:0> are 0D0. The branch MUX input that modifies bit 1 is TB.ERROR (see Figure 7-3), which is a 1 because of the miss on the translation buffer access. Thus, 0D2 is passed through the next address buffer and used to access control store as the next microaddress, and the second MCT microcycle ends.

Q22 Bus NOP

At the rising edge of MCTM CLK125, the microinstruction at 0D2 is available at the output of the control store. This microinstruction causes:

- the offset and page portions of location four in the register file, which contain zeros, to be driven onto the MCA bus as MCA <31:0>,
- the constant 4 to be added to the low nine bits on the MCA bus (currently zeros), and latched in the adder register,
- no cache or TB access,
- the Q22 bus function code for no operation to be sent to the Q22 bus controller.

The microprogram control field of this third microinstruction disables dispatches, but enables traps and jumps. The branch control field is 100 (binary), so the MCT microsequencer selects bits <9:0> from the microinstruction as the next microaddress (see Figure 7-3).

Bits <9:0> are 379 (hex). This is the microaddress of a routine that sets error codes. Thus, 379 is passed through the next address buffer and used to access control store as the next microaddress, and the third MCT microcycle ends.

Set Error Code

At the rising edge of MCTM CLK125, the microinstruction at 379 is available at the output of the control store. This microinstruction causes:

- the value 4 from the adder register to be written into the error code location of the register file, and
- the MEM BUSY signal to be cleared.

The value 4 in the error code location of the register file indicates an access violation (see Table 8-5). This code is always set for TB.ERROR in case an access violation is the cause.

Three conditions in addition to access violation cause TB.ERROR: modify refused, page crossing, and TB miss. The data path microcode is notified of these conditions through inputs to the OR MUX. If TB.ERROR occurs in the MCT and none of these inputs is asserted in the DAP OR MUX, then access violation is the cause and the DAP microcode examines the error code location of the register file.

So even though the access violation error code is set by this microinstruction, no access violation has occurred—only a TB miss—and the TB MISS signal is asserted as an input to the DAP OR MUX.

The microprogram control field of this microinstruction causes a jump back to the MCT idle state microroutine to wait for the next function request from DAP.

Servicing the TB Miss

While the memory controller was executing the five MCT microinstructions described above, the data path executed a Move microinstruction to set a register number in a pointer register, then another Move to try to move the requested data into the data path chip from the memory controller. The requested data are not available because of the TB miss. The DAP microcode branches to a microroutine that handles TB misses because of the TB miss input signal to the OR MUX.

The DAP microcode determines that the PTE for the VA 00000211 must be read from cache or physical memory since it wasn't in the translation buffer. So the DAP microcode completes an entire P0 virtual to

physical translation operation to obtain the PTE (full memory management is enabled). Briefly, the steps performed by the DAP microcode are:

- check the virtual address that was sent to the memory controller (00000211) to determine whether it is a system or process space address, and since it is a process space address, whether it is in P0 or P1;
- rotate the virtual address to obtain the virtual page number (VPN);
- check that the VPN is within the P0LR limits;
- add the VPN to the virtual address in the P0 base register (P0BR)—this sum is the virtual address of the desired PTE;
- ask the memory controller to do a virtual read using this computed virtual address (this reference is made as a kernel read).

The virtual address of the PTE is located in system space (where process page tables reside), so the memory controller accesses the system space portions of the tag and TB/cache RAMs for a translation buffer read.

Assuming a TB hit and a subsequent cache hit (that is, the desired PTE was found in the cache), the memory controller returns the PTE to the data path via the memory data bus. The PTE is A4000000 (hex). Now the data path has the PTE for the virtual address 00000211. (If the PTE was not in the cache, the memory controller would have asked the Q22 bus controller to perform a Q22 bus read, and the PTE would have been returned from physical memory.)

TB Write

Next, the PTE needs to be written into the translation buffer. The memory management routine in the DAP microcode sends the memory controller a Memory Request with WRITE.TB specified. These eight bits are latched in the first memory function latch:

- the two high-order bits are 00 (binary) indicating a data type of byte (byte is always specified for WRITE.TB even though a longword is written into the translation buffer);
- the next bit is the data flow bit (<28>) from the Memory Request microinstruction which is a 1, indicating the data flow will be from DAP to MCT;
- the low-order five bits are the function code; they are 01001 (binary) indicating WRITE.TB.

The following four bits are sent over the backplane:

- two access mode bits which indicate the mode of the current process (access mode is ignored on writes to the translation buffer);
- one modify intent bit from the microinstruction; (It is a 0, indicating read intent even though the intended function is a write; this is because the modify intent does not matter on a WRITE.TB.)
- a second part flag of 0 because this is the first part of this Memory Request to be executed.

The long operand of the WRITE.TB Memory Request specifies the virtual address 00000211. The eight bits in the first memory function latch are driven over the memory control bus, and the additional four bits of Memory Request information over the backplane. The data path asserts the signal DAPR MEM REQUEST

and these eight bits are latched in the memory request latch on the memory controller:

- the second part flag, which is a 0,
- the data type bits, 00, and
- the function code, 01001.

The data flow and modify intent bits are sent to the MCT branch MUX, and the modify intent bit and the access mode bits are sent to the access violation PAL.

Since the memory controller is in the idle microroutine and dispatches are enabled during certain portions of the loop, the MCT microsequencer causes a dispatch on the contents of the memory request latch plus two high-order ones from the CSA PAL. Thus, the control store address 309 (hex) is accessed. This is the address of the MCT microroutine that handles writes to the translation buffer. The dispatch causes the MEM BUSY signal to be asserted.

The transceiver between the memory data bus and the MCA bus is also enabled in the idle routine, so the virtual address 00000211 is now stable on the MCA bus. The memory controller asserts REQ ACK to inform the data path that the VA has been accepted.

Save Virtual Address. At the next rising edge of MCTM CLK125, the first microinstruction in the MCT write TB microroutine is available at the output of the control store. This first microinstruction saves the virtual address on the MCA bus in the virtual address location of the register file. MEM BUSY remains asserted, and traps and dispatches are disabled. The microprogram control field specifies a jump to microaddress 138 (hex).

Wait. At the next rising edge of MCTM CLK125, the next microinstruction in the MCT write TB microrou-

tine (the microinstruction at address 138) is available at the output of the control store. This microinstruction turns off the transceiver. MEM BUSY remains asserted, and traps and dispatches are disabled. The microprogram control field specifies a jump to microaddress 13A.

Clear Busy. At the next rising edge of MCTM CLK125, the microinstruction at microaddress 13A in the MCT write TB microroutine is available at the output of the control store. This microinstruction unconditionally clears busy. Traps and dispatches are disabled, and the microprogram control field specifies a jump to microaddress 164 (hex).

After the DAP microcode issues the WRITE.TB Memory Request, it executes a Moveout microinstruction which puts the PTE--A4000000--for the VA 00000211 on the memory data bus.

Accept PTE. At the next rising edge of MCTM CLK125, the microinstruction at microaddress 164 in the MCT write TB microroutine is available at the output of the control store. This microinstruction enables the transceiver to pass data from the memory data bus to the MCA bus, so the PTE to be written is now stable on the MCA bus.

The microinstruction also enables the reverse pass latch, so the PTE is captured there. Traps and dispatches are disabled, and the microprogram control field specifies a jump to microaddress 166 (hex).

Write PTE. At the next rising edge of MCTM CLK125, the microinstruction at microaddress 166 is available at the output of the control store. This microinstruction causes:

- the data in location 0 of the register file to be driven onto the MCA bus (location 0 currently contains the VA 00000211),
- the content of the reverse pass latch, which is the PTE, to be driven onto the MCD bus, and
- a translation buffer write access.

To accomplish the translation buffer write access, the index MUX selects VA <31> and VA <16:9> from the MCA bus, and provides zeros for the three high-order bits. The source for index MUX <6> is MCA <15>. Thus, location 001 (hex) is accessed in the tag RAM and in the TB/cache data RAM.

The tag MUX selects VA <30:17> from the MCA bus, which are 0000 (hex), and passes these fourteen bits through the write isolation buffer to the tag RAM location accessed by the index MUX—001. The 14 bits are written as the low-order bits of the tag at location 001. The fifteenth bit in the tag is a spare, and the high-order bit, the valid bit, is written as a 1 because microinstruction bit <32> is asserted, indicating that this is a valid tag.

Simultaneously, the PTE from the MCD bus is written into location 001 in the TB/cache data RAM, and the TB write is complete. Thus, location 001 in the TB/cache now contains A4000000 (hex).

The microprogram control field of this microinstruction disables dispatches, but enables traps and jumps. The branch control field is 100 (binary), so the MCT microsequencer selects bits <9:0> from the microinstruction as the next microaddress (see Figure 7-3). Bits <9:0> are 002, so the memory controller returns to the idle state.

The last microinstruction executed by the DAP microcode was the Moveout microinstruction that put the PTE for the VA 00000211 on the memory data bus.

Next, the DAP microcode executes another Memory Request; this time, the function code is REPEAT.FIRST. This causes the contents of the second memory function latch to be driven onto the memory control bus. The second memory function latch still contains the function code for the initial VREAD.RCHECK that failed. Thus, the memory controller is asked to retry the virtual read with read check, and the virtual address 00000211 is driven over the memory data bus again.

The data path asserts the signal DAPR MEM REQUEST and these eight bits are latched in the memory request latch on the memory controller:

- the second part flag, which is a 0,
- the data type bits, 01, indicating a word of data is to be read,
- the function code, 00001 (binary), indicating VREAD.RCHECK.

The data flow and modify intent bits are sent to the MCT branch MUX, and the modify intent bit and the access mode bits are sent to the access violation PAL.

Since the memory controller is in the idle microroutine and dispatches are enabled during certain portions of the loop, the MCT microsequencer causes a dispatch on the contents of the memory request latch plus two high-order ones from the CSA PAL. Thus, control store address 321 (hex) is accessed again; this is the beginning address of the MCT read/write data words microroutine. The dispatch causes the MEM BUSY signal to be asserted.

The transceiver between the memory data bus and the MCA bus is also enabled in the idle routine, so the virtual address 00000211 is now stable on the MCA bus. The memory controller asserts REQ ACK to inform the data path that the VA has been accepted.

TB Access Retried

At the next rising edge of MCTM CLK125, the first microinstruction in the MCT read/write data words microroutine is available at the output of the control store. Once again, this microinstruction causes:

- the VA 00000211 from the MCA bus to be stored in register file location 0,
- the low nine bits of the VA to be incremented by 1 in the adder and the sum latched in the adder register,
- MEM BUSY to remain asserted,
- a translation buffer read access, and
- the physical address register latch enable signal to be asserted.

This time, the translation buffer access is successful; the index MUX accesses location 001 in the tag RAM, the fourteen low-order bits of the tag are sent to the comparator (0000 hex), and they match MCA bits <30:17> supplied by the tag MUX. The comparator asserts the signal MCTL TBC HIT.

Simultaneously, the PTE at location 001 in the TB/cache RAM (A4000000) is driven onto the MCD bus. PTE bits <12:0> from the MCD bus are latched into the PAR as physical address bits <21:09>. PFN <19> is 0, so 0 is latched in the PAR as physical address bit <28>. The AND of the TB/cache hit signal and PFN <20> is 0, so 0 is latched in the PAR as

physical address bit <29>. So the PAR now contains the 14-bit value 0000 (hex). The four-bit protection field and the modify bit from the PTE are also latched in the access protection latch.

The microprogram control field of this microinstruction is decoded by the MCT microsequencer and 0F1 is passed through the next address buffer and used to access control store as the next microaddress.

Cache Access Retried

At the rising edge of MCTM CLK125, the microinstruction at 0F1 is available at the output of the control store. Once again, this microinstruction causes:

- the value 011 (hex) to be driven from the offset portion of the virtual address location in the register file onto the MCA bus as MCA <8:0>;
- the fifteen bits in the PAR to be driven onto the MCA bus as MCA <29:28> and <21:09> (MCA <31:30> and <27:22> are ones via the pull-up resistors); thus, the 32 bits on the MCA bus form the hex value DFC00011, and the physical address is the low 22 bits plus MCA <29>, or the hex value 000011.
- MCA <31:09> (DCF000) to be written into the page portion of the virtual address location in the register file,
- MEM BUSY to remain asserted,
- a cache read access,
- the data on the MCD bus from the cache access to be rotated 1 byte to the right and all four bytes latched into the merge register (the data are whatever happened to be in the accessed location),

- the physical address on the MCA bus (000011) to be latched into the Q22 bus write register, and
- the Q22 bus function code for a read block operation (DATBI) to be sent to the Q22 bus controller; the go bit is not sent.

To accomplish the cache read access, the tag MUX selects physical address bits <21:13> from the MCA bus, supplies five zeros for the high-order bits and passes these fourteen bits, which have the value 0000, to the TB/cache comparator.

At the same time, the index MUX selects PA <12:2> from the MCA bus, and provides a one for the high-order bit. The source for index MUX <6> is bit <8> from register file location 0. Thus, location 804 (hex) is accessed in the tag RAM and in the TB/cache RAM. The 16-bit tag at location 804 in the tag RAM is passed to the TB/cache comparator. The tag does not match the search tag 0000 provided by the tag MUX, and the comparator does not assert the signal MCTL TBC HIT.

However, the data at location 804 in the TB/cache RAM are driven onto the MCD bus anyway, rotated one byte to the right and latched into the merge register.

The microprogram control field of the microinstruction is decoded and the MCT microsequencer selects bits <9:0> from the first microinstruction, modified by any asserted conditions in the branch MUX, as the next microaddress. Bits <9:0> are 0D0. This time, no branch conditions are set, so 0D0 is passed through the next address buffer and used to access control store as the next microaddress.

Incorrect Data Returned

At the rising edge of MCTM CLK125, the microinstruction at 0D0 is available at the output of the control store. This microinstruction causes:

- the contents of the merge register (whatever was stored in location 804 in the TB/cache RAM) to be driven onto the MCA bus as MCA <31:0>, and
- the transceiver to drive the data on the MCA bus over the memory data bus to the data path, even though it is not the desired data.

The microprogram control field of this microinstruction disables dispatches, but enables traps and jumps. The branch control field is 110 (binary), so the MCT microsequencer selects bits <9:0> from the microinstruction, modified by the branch condition TBC.MISS which is asserted because of the cache miss in the previous microcycle, as the next microaddress (see Figure 7-3).

Bits <9:0> are 090 and modified by TBC.MISS, the next address is 091. Thus, 091 is passed through the next address buffer and used to access control store as the next microaddress.

The cache miss signal is also sent to the data path, so the incorrect data on the memory data bus are ignored.

Q22 Bus Go

At the rising edge of MCTM CLK125, the microinstruction at 091 is available at the output of the control store. This microinstruction sends the go bit to the Q22 bus controller. This causes the controller to begin the DATBI (read block) cycle using the physical address (000011) saved in the Q22 bus write register two microcycles earlier.

Note that at this point, each successive memory controller microinstruction must assert the DATBI function code and the go bit until SYNCREADY is received. Once SYNCREADY is received, the next memory controller microinstruction asserts a no operation function code.

The microprogram control field of this microinstruction disables dispatches, but enables traps and jumps. The branch control field is 100 (binary), so the MCT microsequencer selects bits <9:0> from the microinstruction as the next microaddress. Bits <9:0> are 1C5. Thus, 1C5 is passed through the next address buffer and used to access control store as the next microaddress.

Read Block

At the rising edge of MCTM CLK125, the microinstruction at 1C5 is available at the output of the control store. This microinstruction causes:

- the contents of the virtual address location in the register file to be driven onto the MCA bus; the virtual address location currently contains the hex value DCF00011,
- the adder to add the constant 1 to the low nine bits on the MCA bus and latch the sum in the adder register; so the adder register now contains 012, and
- the contents of the Q22 bus read register to be driven onto the MCD bus, rotated 1 byte to the right in the rotator, and latched in the merge register.

The microprogram control field of this microinstruction causes the microprogram to loop on this instruction until the SYNCREADY signal is received from the Q22

bus, indicating that the data are available and latched in the Q22 bus read register.

Q22 bus block reads use word-aligned addresses, so although the physical address 000011 was latched in the Q22 bus write register, the low byte is ignored, and a word of data is read at address 000010.

Suppose the data at physical address 000010 in memory is:

DD	CC	BB	AA	:000010

where the data requested by the data path is the word CCBB. (For this example, each letter represents one hex digit; AA, for instance, is one byte.) The first block transfer from the Q22 bus read returns the bytes BBAA and latches them in the Q22 bus read register.

This microinstruction moves the contents of the MCD bus into the rotator. The value FFFFBBAA is stable on the MCD bus at this point; the FFFF is provided by the pull-up resistors, and the BBAA is from the Q22 bus read register. When FFFFBBAA is rotated one byte to the right and latched in the merge register, the merge register contains AAFFFFBB, where BB is the low-byte.

SYNCREADY is received as an input to the MCT branch MUX, so when it is asserted, it causes a jump to microaddress 1C7. Thus, 1C7 is passed through the next address buffer and used to access control store as the next microaddress.

Change Q22 Bus Function

At the rising edge of MCTM CLK125, the microinstruction at 1C7 is available at the output of the control store. This microinstruction asserts the function code for read word (DATI) instead of read block (DATBI) to

indicate that this is the last Q22 bus cycle of the block read. The microinstruction causes:

- a read word bus cycle to begin,
- the page portion of the virtual address location in the register file to be driven onto the MCA bus; that is, the hex value DCF000,
- the contents of the adder register to be driven onto the MCA bus; that is, the hex value 012,
- the physical address on the MCA bus (000012) to be latched into the Q22 bus write register, and
- the adder to subtract the constant 2 from the low nine bits on the MCA bus and latch the sum in the adder register; so the adder register now contains 010.

The microprogram control field of this microinstruction disables dispatches, but enables traps and jumps. The branch control field is 100 (binary), so the MCT microsequencer selects bits <9:0> from the microinstruction as the next microaddress, modified by the branch conditions QBUS.TIMEOUT and QBUS.ERROR (see Figure 7-3). Bits <9:0> are 0DA and neither branch condition is asserted. Thus, 0DA is passed through the next address buffer and used to access control store as the next microaddress.

At the rising edge of MCTM CLK125, the microinstruction at 0DA is available at the output of the control store. This microinstruction causes the contents of the merge register (AAFFFFBB) to be driven onto the MCA bus. This is an intermediate cycle to allow a bus timeout error to be detected. The microprogram control field causes a jump to address 1CD.

Read Word

At the rising edge of MCTM CLK125, the microinstruction at 1CD is available at the output of the control store. This microinstruction causes:

- the contents of the merge register (AAFFFFBB) to be driven onto the MCA bus,
- the contents of the Q22 bus read register to be driven onto the MCD bus, rotated 3 bytes to the right in the rotator, and bytes 2 and 1 latched in the merge register.

The microprogram control field of this microinstruction causes the microprogram to loop on this instruction until the SYNCREADY signal is received from the Q22 bus, indicating that the second word of data is available and latched in the Q22 bus read register. This read word bus cycle returns the bytes DDCC and latches them in the Q22 bus read register.

This microinstruction also moves the contents of the MCD bus into the rotator. The value FFFFDDCC is stable on the MCD bus at this point; the FFFF is provided by the pull-up resistors, and the DDCC is from the Q22 bus read register.

When FFFFDDCC is rotated three bytes to the right and bytes 1 and 2 latched in the merge register, the merge register contains AADDCCBB, where BB is the low-byte. (The AA and BB are still in the merge register from before.) Notice that the requested data CCBB are now aligned as the low-order word of the merge register.

SYNCREADY is received as an input to the MCT branch MUX, so when it is asserted, it causes a jump to microaddress 1CF. Thus, 1CF is passed through the

next address buffer and used to access control store as the next microaddress.

At the rising edge of MCTM CLK125, the microinstruction at 1CF is available at the output of the control store. This microinstruction causes the contents of the merge register (AADDCCBB) to be driven onto the MCA bus, and asserts a no operation Q22 bus function code. This is also an intermediate cycle to allow bus timeout errors to be detected. The microprogram control field causes a jump to address 0E2 if no bus errors occur.

Return Correct Data

At the rising edge of MCTM CLK125, the microinstruction at 0E2 is available at the output of the control store. This microinstruction causes:

- the contents of the merge register, AADDCCBB, to be driven onto the MCA bus as MCA <31:0>,
- the transceiver to drive the data on the MCA bus over the memory data bus to the data path,
- the reverse pass latch to be enabled for input and output, so AADDCCBB is latched there from the MCA bus and driven onto the MCD bus,
- the AADDCCBB from the reverse pass latch to be rotated three bytes to the right and all four bytes latched in the merge register; the merge register now contains DDCCBBAA, where AA is the low-order byte, and
- the MEM BUSY signal to be cleared.

The longword AADDCCBB is returned to the data path over the memory data bus. Because the data path is executing a macroinstruction with data type word (MOVW), the data path will only read the low-order

word off the memory data bus. The low-order word is the desired data CCBB and it is, in fact, the first operand.

After being rotated again by this microinstruction and latched in the merge register as DDCCBBAA, the longword is now restored to the correct order for a cache write.

The microprogram control field of this microinstruction disables dispatches, but enables traps and jumps. The branch control field is 100 (binary), so the MCT microsequencer selects bits <9:0> from the microinstruction as the next microaddress. Bits <9:0> are 382 (hex). Thus, 382 is passed through the next address buffer and used to access control store as the next microaddress.

Prepare for Cache Write

At the rising edge of MCTM CLK125, the microinstruction at 382 is available at the output of the control store. This microinstruction causes:

- the contents of the merge register, DDCCBBAA, to be driven onto the MCA bus as MCA <31:0>, and
- the reverse pass latch to be enabled for input, so DDCCBBAA is once again written into the reverse pass latch.

The microprogram control field of this microinstruction only enables jumps. The branch control field is 100 (binary), so the MCT microsequencer selects bits <9:0> from the microinstruction as the next microaddress. Bits <9:0> are 35C. Thus, 35C is passed through the next address buffer and used to access control store as the next microaddress.

Cache Write

At the rising edge of MCTM CLK125, the microinstruction at 35C is available at the output of the control store. This microinstruction causes:

- the contents of the adder register, 010, to be driven onto the MCA bus as MCA <8:0>,
- the page portion of the virtual address location in the register file to be driven onto the MCA bus; so MCA <31:9> are the hex value DCF000,
- the reverse pass latch to be enabled for output, so DDCCBBAA is driven onto the MCD bus, and
- a cache write access.

To accomplish the cache write access, the index MUX selects PA <12:2> from the MCA bus, and provides a one for the high-order bit. The source for index MUX <6> is bit <8> from the adder. The twelve index MUX bits form the hex value 804. Thus, location 804 is accessed in the tag RAM and in the TB/cache data RAM.

At the same time, the tag MUX selects physical address bits <21:13> from the MCA bus, supplies five zeros for the high-order bits and passes these fourteen bits, which have the hex value 0180, to the write isolation buffer. From the write isolation buffer, 0180 hex is written into the low-order fourteen bits of tag RAM location 804. The high-order bit—the valid bit—is written as a one, indicating that this is a valid tag, because bit <32> in the microinstruction is a one. The next high-order bit is a spare. Tag RAM location 804 now contains the hex value 8180.

Simultaneously, the data on the MCD bus, DDCCBBAA, are written into location 804 in the

TB/cache RAM, and the cache write operation is complete.

The microprogram control field of this microinstruction disables dispatches, but enables traps and jumps. The branch control field is 010 (binary), so the MCT microsequencer selects bits <9:0> from the microinstruction as the next microaddress, modified by the branch conditions QBUS.SYNCH and QBUS.BLK.OK (see Figure 7-3). Bits <9:0> are 0A6 and none of the branch conditions are asserted. Thus, 0A6 is passed through the next address buffer and used to access control store as the next microaddress.

The microinstruction at 0A6 enables dispatches, starts the prefetch sequence to refill the prefetch FIFO if the FIFO contains less than eight bytes, and then jumps to the idle state microroutine.

Move Data

After the memory controller clears MEM BUSY and returns the data CCBB on the memory data bus, the data path executes a MOVL microinstruction, which moves CCBB off the memory data bus and into temporary storage in the data path chip. The data path has now obtained the first operand.

Evaluating the Second Operand Specifier

The second operand specifier, 51, was loaded into the IBYTE register many cycles ago—soon after the decode of the first operand specifier, 60, was completed. Now the second operand specifier is decoded. The decode causes the DAP microsequencer to use the current microaddress plus one as the next microaddress since the operand specifier 51 indicates register mode.

Once the decode has completed, the data path asserts IB TAKEN and the prefetch FIFO drives the next byte in

the instruction stream onto the memory control bus. The memory controller asserts NXT VALID REG, the data path asserts LOAD I BYTE, and the next I-stream byte is clocked into the IBYTE register.

The microinstruction that follows the Decode is a Move to *pointer 2. Pointer 2 is currently pointing to R1 because bit <26> in the Decode microinstruction just executed is a 1, causing bits <5:0> from the IBYTE register to be stored in pointer 2. When this Move microinstruction is executed, the word of data, CCBB, in temporary storage in the data path chip, is moved into R1. The entire MOVW macroinstruction is now complete.

This completes the description of the memory controller hardware and microprogram level flow example. The next chapter describes the Q22 bus controller. Although the Q22 bus controller hardware is physically located on the memory controller module, it is described separately because it is an independent micromachine.

Chapter 9
Q22 Bus Controller

The Q22 bus controller handles the Q22 bus protocol, freeing the memory controller microcode from this task. The Q22 bus controller accepts function requests from the memory controller microcode and carries them out through its own set of microstates.

The first part of this chapter describes the hardware components on the memory controller module that make up the Q22 bus controller, and how they interact. The second part of this chapter describes the Q22 bus itself, and the bus operations that are handled by the Q22 bus controller.

Overview of Q22 Bus Controller Functions

The Q22 bus controller performs the following five functions.

- It services function requests from the memory controller.
- It sequences bus cycles.
- It arbitrates the bus.
- It monitors direct memory accesses.
- It communicates with the memory controller and the data path module.

The Q22 bus controller is the default bus master. It remains in an idle state unless it is servicing a request from the Q22 bus or the memory controller. The Q22 bus controller uses the same 125 ns microcycle as the memory controller.

The next sections describe the five Q22 bus controller functions, and the hardware components that implement them, in detail.

Servicing MCT Function Requests

One of the main functions of the Q22 bus controller is carrying out requests from the memory controller. To do this, the Q22 bus controller must receive the following information from the memory controller.

- function code — A three bit field from the memory controller microinstruction that defines the function requested.
- go bit — One bit from the memory controller microinstruction that informs the Q22 bus controller to proceed with the current function request.

At 0 ns of each MCT microcycle, the MCT microinstruction is available at the output of the MCT control store. If the microinstruction contains the go bit or a 3-bit function code, these bits are sent to the Q22 bus controller within the current microcycle; call it microcycle 1. The address for the data to be read from or written to is also latched in the Q22 bus write register during microcycle 1.

The Q22 bus controller uses the 3-bit function code to determine the sequence of microstates needed to accomplish the given function. A new microstate is entered every 125 ns. The function code also determines the control signals needed to accomplish the requested function. The 3-bit function code is bits <50:48> from the MCT microinstruction. The encoding is shown in Table 9-1.

Table 9-1. Function Code Field

<50:48>	Operation	Mnemonic
000	no operation	NOP
001	write word	DATO
010	write byte	DATOB
011	write block	DATBO
100	read word	DATI
101	read interlocked	DATIO
110	read interrupt vector	
111	read block	DATBI

As soon as the Q22 bus controller receives the function code from the memory controller, it places the address that is latched in the Q22 bus write register on the Q22 bus, if the bus is free. The memory controller microcode then sends the go bit during any cycle up to the second cycle following the one in which the request was initiated. The go bit is bit <63> of the memory controller microinstruction.

When the Q22 bus controller receives the go bit, it generates the signal MCTA TSYNC. TSYNC tells the slave device that a valid address is on the bus and a bus cycle for the function requested by the memory controller is in progress.

The memory controller microcode can cancel the requested function up to the third microcycle by sending the function code for no operation instead of the go bit. The memory controller microcode does this, for example, when a cache read access results in a hit, and a read from memory is not necessary. (For every cache read, the physical address used to access the cache is also saved in the Q22 bus write register. A read from

memory bus operation is started if a cache miss occurs.)

Thus, the Q22 bus controller begins bus operations for the requested function as soon as it receives a function code from the memory controller. By the second microcycle after the one in which the request was initiated, the memory controller microcode must send the go bit to tell the Q22 bus controller to continue the bus cycle, or a NOP function code to cancel the request.

Two PALs in the Q22 bus controller are the main components that handle function requests from the memory controller. These PALs are the function decoder PAL and the sequencer PAL. Other components involved are the two transceivers and a receiver that form part of the Q22 bus interface, the cache invalidate pipeline register, and the registered transceivers that form the data/address interface to the Q22 bus. (The data/address interface includes the Q22 bus read and write registers.) These components are described in the next paragraphs.

Function Decoder PAL

The function decoder PAL determines which operation the memory controller is requesting. The 3-bit function code and the go bit from the memory controller microcode are received at the input side of the function decoder PAL. The three signals for the function code are MCTH UQIF FUN2, MCTH UQIF FUN1, and MCTH UQIF FUN0. The go bit is signal MCTN QBUS GO.

If any one of the function code signals is asserted and no bus device has requested a direct memory access, the function decoder PAL generates the signal MCTA MFUN GO, which is one input to the sequencer PAL. MFUN GO causes the sequencer PAL to begin the set up for the desired bus operation. If the function code is

000 for no operation, the function decoder PAL deasserts MFUN GO.

If a direct memory access request has been made when the function decoder PAL receives a function request from the memory controller, the PAL inhibits the MFUN GO signal and the function code is ignored. The memory controller must continue to assert the function code and the go bit until it receives the SYNCREADY signal from the Q22 bus controller. SYNCREADY informs the memory controller that the requested Q22 bus operation is underway.

Sequencer PAL

The sequencer PAL generates the control signals that sequence the bus cycles for the desired bus operation. The function decoder PAL starts the sequencer PAL by sending it MCTA MFUN GO. Bit <50> from the memory controller microinstruction is also passed to the sequencer PAL as the signal MCTH UQIF FUN2. This signal when asserted indicates that the requested function is a read operation rather than a write.

When the sequencer PAL receives MFUN GO and UQIF FUN2, it generates the correct signals to control the desired bus operation. For example, the sequencer PAL generates the signals MCTA ENDALADD and MCTA ENDAL to cause the contents of the Q22 bus write register to be driven onto the Q22 bus.

Once the memory controller microcode sends the go bit, the sequencer PAL receives the signal MCTB RRPLY several microcycles later and continues asserting the necessary control signals for the desired bus operation. If the memory controller does not send the go bit by the second microcycle after the one in which the function request was initiated, it sends the no operation function code and the function decoder PAL deasserts MFUN

GO. As a result, the sequencer PAL returns the bus control signals to their default states.

For more information about the sequencer PAL and the control signals for the various bus operations, see the section titled "Sequencing Bus Cycles" in this chapter.

Q22 Bus Interface

Two quad transceivers and one receiver are the components that make up the interface between the control signals asserted by the Q22 bus controller and the Q22 bus signals.

The letter "T" in front of a Q22 bus signal name indicates that the signal is generated by the Q22 bus controller and transmitted to the Q22 bus. The letter "R" in front of a Q22 bus signal name indicates that the signal is generated by a Q22 bus device and received by the Q22 bus controller. The letter "B" in front of a Q22 bus signal name indicates that the signal is the actual bus signal on the backplane. Thus, the Q22 bus interface transmits "T" signals from the controller onto the bus as "B" signals, and receives "B" signals off the bus as "R" signals.

For example, the controller signal MCTA TDIN is one input to an interface transceiver and is sent out the bus as the signal BDIN. TDIN is the data in control signal transmitted from the Q22 bus controller to a bus device. BDIN is the backplane version of TDIN. Similarly, the bus signal BDMR is received by the interface receiver and sent to the controller as MCTC RDMR. RDMR is the request for a direct memory access (DMA) from a bus device and is received by the Q22 bus controller. BDMR is the backplane version of RDMR.

Cache Invalidate Pipeline Register

The cache invalidate pipeline register is a 22-bit register located between the MCD bus and the Q22 bus read register. The cache invalidate pipeline register latches DMA addresses and is a transparent latch for data read from a Q22 bus device. Thus, the cache invalidate pipeline register is always loaded from the Q22 bus read register. The Q22 bus read register is loaded with the address or data on the Q22 bus during bus read operations.

Q22 Bus Transceivers

Six quad registered transceivers are the hardware components that make up the Q22 bus read and write registers. The transceivers are bidirectional, allowing data and memory addresses to be transmitted to and received from the memory controller.

The memory controller microcode controls the input side of the write register transceivers, and the output side of the cache invalidate pipeline register. Thus, the memory controller microcode causes addresses and data to be latched in the write register, and to be driven from the cache invalidate pipeline register onto the MCD bus.

The sequencer PAL controls the output side of the write register with the signal MCTA ENDAL, which controls bits <15:0>, and the signal MCTA ENDALADD, which controls bits <21:16>.

For a read operation, data are latched in the Q22 bus read register from the Q22 bus on the falling edge of the signal MCTA TDIN, which is generated by the sequencer PAL.

For direct memory access (DMA) activity on the bus, the read register is open when the signal RSACK is asserted. (A bus device asserts RSACK to accept and retain bus mastership.) The read register closes, latching the address on the bus data/address lines, when the function decoder PAL has the signal DMA IN PROG asserted and the bus master asserts RSYNC. (The function decoder PAL asserts DMA IN PROG when bus mastership has been granted to a bus device; a bus device asserts RSYNC to indicate that it has placed an address on the bus.)

Sequencing Bus Cycles

Sequencing the bus cycles is another major function of the Q22 bus controller. This section describes the control signals that sequence the bus cycles. The sequencer PAL in the Q22 bus controller maintains state and generates most of the signals that control the Q22 bus protocol. The function decoder PAL supplies the remaining bus control signals.

ENDAL and ENDALADD

When the Q22 bus controller is in an idle state, the sequencer PAL continuously asserts these two signals. MCTA ENDAL asserted causes bits <15:0> of the Q22 bus write register to be driven onto the Q22 bus as DAL <15:0>. MCTA ENDALADD asserted causes bits <21:16> of the write register to be driven onto the bus as DAL <21:16>. Bit <29>, the I/O space flag, is also loaded into the write register. If bit <29> is asserted, the assertion of ENDALADD also causes bit <29> to be driven onto the bus as the signal BBS7.

The sequencer PAL asserts ENDAL when the write register contains data; it asserts ENDALADD and

ENDAL when the write register contains an address. Thus, the sequencer PAL deasserts ENDALADD for the data cycle of a bus write operation, and it deasserts ENDAL and ENDALADD for the data cycle of a bus read operation.

The signal DMA IN PROG is asserted by the function decoder PAL when bus mastership has been granted to a bus device. When the sequencer PAL receives DMA IN PROG asserted, it deasserts ENDALADD and ENDAL.

Similarly, the function decoder PAL asserts the signal EN IAKO when an interrupt acknowledge bus operation has started. This signal also causes the sequencer PAL to deassert ENDALADD and ENDAL.

PRESYNC

The signal MCTA PRESYNC is asserted by the sequencer PAL as an internal state bit. PRESYNC marks when the memory controller changes the function code. When the sequencer PAL receives MFUN GO asserted, it asserts PRESYNC in the next microcycle.

In addition to marking when the function code changes, PRESYNC asserted also causes the signal TSYNC EN to be asserted. When TSYNC EN is asserted and the go bit is received from the memory controller microcode, the signal TSYNC is asserted. TSYNC is driven onto the bus as BSYNC. BSYNC is asserted by the bus master (in this case, the Q22 bus controller) to indicate that it has placed an address on the bus and a transfer is in progress.

SYNC HOLD

The sequencer PAL asserts MCTA SYNC HOLD to keep the BSYNC signal asserted on the bus. This is a

necessary part of the bus protocol; BSYNC must remain asserted to allow the bus master to keep the addressed slave device selected. BSYNC stays asserted until the entire bus operation is completed.

TDIN

The signal DIN stands for data input. The sequencer PAL asserts the signal MCTA TDIN during bus read operations to inform the slave bus device that the Q22 bus controller wants the requested data. The device must respond with RRPLY to indicate that it is ready to place the data on the bus. The assertion of RRPLY causes the negation of TDIN; when TDIN negates, the data on the bus data/address lines (DAL) are latched in the Q22 bus read register.

TDOUT

The signal DOUT stands for data output. The sequencer PAL asserts the signal MCTA TDOUT during bus write operations to inform the slave bus device that valid data are available on the bus data/address lines. The device must respond with RRPLY to indicate that it is ready to accept the data on the bus. The assertion of RRPLY causes the negation of TDOUT; when TDOUT negates, the controller removes the data from the bus data/address lines after one cycle of hold time.

TWTBT

If the 3-bit function code from the memory controller specifies a write word, write byte or write block operation, the function decoder PAL asserts the signal MCTA TWTBT (transmit write byte) during the address portion of the cycle to indicate that an output cycle is to follow rather than an input cycle. During the

data portion of a write byte (DATOB) or a read/modify/write byte (DATIOB) bus cycle, the function decoder PAL asserts TWTBT to indicate a byte rather than a word transfer is to take place.

BM TBS7

If the 3-bit function code from the memory controller specifies a block mode read operation, the function decoder PAL asserts the signal MCTA BM TBS7 (block mode transmit bank select 7). The assertion of BM TBS7 causes the assertion of BBS7 on the bus. BBS7 asserted while TDIN is asserted informs the Q22 bus memory that the next word in memory is also desired. The Q22 bus controller asserts BBS7 with the first data transfer until the start of the last transfer to indicate to the memory that there will be subsequent transfers.

EN IAKO

If the 3-bit function code from the memory controller specifies read vector, the function decoder PAL asserts the signal MCTA EN IAKO (enable interrupt acknowledge output). EN IAKO is pipelined and ANDed with TDIN to generate the signal MCTA TIAKO. TIAKO is driven onto the bus as BIAKO which informs the bus device with the highest priority interrupt request that its interrupt is acknowledged.

Arbitrating the Q22 Bus

Another major function of the Q22 bus controller is to arbitrate the Q22 bus; that is, the Q22 bus controller monitors the bus as well as the memory controller and decides when to assume bus mastership to execute a function request from the memory controller, and when to grant bus mastership to allow DMA transfers.

The hardware involved in arbitrating the bus is the function decoder PAL and the bus error logic. These components are described in the next paragraphs.

Function Decoder PAL

When the Q22 bus controller is in an idle state, the function decoder PAL asserts the signal MCTA EN DMA. Thus, the default state of the controller is that requests for direct memory accesses are enabled. A Q22 bus device requests bus mastership for a direct memory access by asserting the bus signal BDMR.

BDMR is pipelined, synchronized, ANDed with EN DMA, and presented to the input side of the function decoder PAL as the signal MCTA DMGO EN (direct memory access grant output enable). This signal causes the PAL to inhibit the signal MCTA MFUN GO and assert the signal MCTA DMA IN PROG. As a result, the sequencer PAL disables the data/address lines by deasserting MCTA ENDALADD and MCTA ENDAL.

Two microcycles after BDMR is received, the signal TDMGO goes out on the Q22 bus. TDMGO informs the device that its DMA request is granted, and the device then becomes bus master. The bus device cannot proceed with the direct memory access until it receives the signal TDMGO from the Q22 bus controller.

The bus device acknowledges the TDMGO signal and assumes bus mastership by asserting RSACK. As long as the bus device asserts RSACK, it retains bus mastership. The bus device relinquishes bus mastership by negating RSACK.

When the function decoder PAL receives the RSACK signal, it deasserts the EN DMA (enable DMA) signal. As a result, both function requests from the memory

controller and DMA requests from other Q22 bus devices are inhibited.

Thus, when the Q22 bus controller is in the idle state, a DMA request takes precedence over a function request from the memory controller if the DMA request is received first, or at the same time. Once a bus device assumes bus mastership for DMA activity, function requests from the memory controller are locked out until the bus device relinquishes bus masterhip. Similarly, once the Q22 bus controller assumes bus mastership to execute a function request from the memory controller, DMA requests are locked out until the function request is completed.

Bus Error Logic

The Q22 bus controller contains hardware to respond to memory parity errors and bus timeouts. The sequencer PAL asserts the signal MCTA QBUS ERROR if a parity error or a bus timeout occurs. QBUS ERROR is one input to the memory controller branch MUX. The error conditions are cleared at the start of any function request from the memory controller.

Parity

On the Q22 bus, DAL <17> and <16> are used for parity error detection. During the portion of the data transfer bus cycles in which data is being placed on the bus by the slave for the bus master, DAL <17> is asserted to enable parity error detection logic, and DAL <16> is asserted when a parity error occurs.

DAL <17> and <16> are ANDed to form an input to the sequencer PAL; when they are both asserted and TDIN negates, the sequencer PAL asserts MCTB QBUS ERROR. The bus operation then completes and the Q22 bus controller returns to the idle state.

Meanwhile, the memory controller microcode branches to the post parity error microroutine. The memory controller detects that this is a parity error because only the QBUS ERROR signal is asserted. If a bus timeout occurs, both QBUS ERROR and a signal called TIMEOUT are asserted.

Bus Timeout

The bus timeout logic limits the length of time the Q22 bus controller waits for a reply from a slave or a DMA device. The timeout limit is 10 microseconds. The timer is started by the assertion of TDIN, TDOUT, or TDMGO. The timer is reset by the assertion of RRPLY (in response to TDIN or TDOUT) or by the assertion of RSACK (in response to SDMGO).

If the expected response, RRPLY or RSACK, does not appear within 10 microseconds, the timeout logic asserts MCTA BUSERR, which locks out RRPLY and RSACK. BUSERR is synchronized, then asserted as MCTB EN TIMEOUT. EN TIMEOUT is passed to the function decoder PAL as the signal MCTA ABORT and causes MFUN GO to negate.

EN TIMEOUT is also an input to another PAL in the Q22 bus controller called the control and status PAL. When the control and status PAL receives EN TIMEOUT, it asserts the signal MCTA TIMEOUT, which is one input to the memory controller branch MUX.

The negation of MFUN GO while either TDIN or TDOUT is asserted causes the sequencer PAL to assert QBUS ERROR and SYNCREADY, and to negate TDIN or TDOUT, whichever was set. The sequencer PAL also negates SYNC HOLD and asserts ENDAL and ENDALADD. Essentially, the Q22 bus controller returns to the idle state except that the QBUS ERROR

and TIMEOUT signals are asserted and received by the memory controller branch MUX, and SYNCREADY is still asserted.

At the next microcycle, the sequencer PAL negates SYNCREADY, which causes the control and status PAL to assert the signal CLR TM ERR. This signal resets the timeout detect circuit. QBUS ERROR and TIMEOUT remain asserted until the memory controller microcode sends a new function code.

Monitoring Direct Memory Accesses

The Q22 bus controller also contains logic that detects direct memory access activity on the Q22 bus, and posts the cache invalidate flag. The main components of this logic are a binary counter, a MUX and a PAL. The following paragraphs explain the process for a cache invalidate because of a direct memory access (DATO, DATOB or DATBO), and the process for a cache invalidate because of a read interlocked (DATIO) operation.

DMA Cache Invalidates

The Q22 bus controller grants bus mastership to a bus device by asserting TDMGO in response to the device's request for a direct memory access (RDMR). The device assumes and retains bus mastership by asserting RSACK. Once the device assumes bus mastership, it begins a direct memory access. It first places the address on the bus and asserts RSYNC and RWTBT. The assertion of RSYNC indicates that an address is on the bus and a transfer is in progress. RWTBT is asserted during this address portion of the cycle to indicate that an output cycle is to follow, rather than an input cycle.

When the monitoring logic detects the assertion of RSYNC with DMA IN PROG from the function decoder PAL, the input side to the Q22 bus read register closes and the PAL negates the signal MCTB OPEN LATCH. Negating OPEN LATCH causes the address in the Q22 bus read register to be latched in the cache invalidate pipeline register. At the same time, bits <4:2> of the address are latched in the binary counter that is part of the DMA monitoring logic. In the next microcycle, the counter increments bits <4:2> by one.

When RWTBT is asserted on the rising edge of RSYNC, the signal MCTB CINV is asserted. When CINV is asserted with RSACK, the PAL in the DMA monitoring logic also asserts the signal MCTB CACHE INV, which is the cache invalidate flag. The cache invalidate flag is one input to the MCT microinstruction decode logic, and causes the MCT to trap to the microroutine at control store address 3FF. The microroutine reads the address out of the cache invalidate pipeline register, checks if that address is in the cache, and marks the cache entry invalid if it is.

Next, the bus device asserts the signal RDOUT to indicate that the data to be written into memory are now on the Q22 bus. When RDOUT asserts, the PAL in the DMA monitoring logic asserts SEL BLK (select block).

The cache invalidate pipeline register holds the DMA address until it is read by the memory controller microcode. The microcode reads the cache invalidate pipeline register by asserting MCTN QBUS RD OE. Once the address is read and SEL BLK is asserted, the PAL asserts OPEN LATCH and negates the cache invalidate flag. On the next microcycle, OPEN LATCH negates, causing the cache invalidate pipeline register to latch bits <29>, <21:5>, and <1:0> from the read register, and the incremented bits <4:2> from the

binary counter. Thus, the first incremented address, which is the original DMA address plus four, is latched in the cache invalidate pipeline register. This is done in case the DMA is a block write, and successive addresses need to be invalidated. The binary counter again increments bits <4:2> of the original DMA address.

At this point, if the bus device asserts RDOUT a second time, the DMA is indeed a block write. If BDAL <1> of the original address is a 1, the data being written are the high-order word of a longword, and the cache invalidate flag is posted again; that is, when RDOUT is asserted for the second time. If BDAL <1> of the original address is a 0, the data being written are the low-order word of a longword, and the cache invalidate flag does not need to be posted again until RDOUT is asserted for the third time.

From the time the second cache invalidate flag is posted, the following increment-address cycle continues for every other RDOUT until the bus device negates RSYNC.

- The PAL asserts the cache invalidate flag and the PAL sets up to skip this increment-address cycle for the next RDOUT.
- The memory controller reads the cache invalidate pipeline register which contains the incremented address.
- The PAL asserts OPEN LATCH and negates the cache invalidate flag.
- The PAL negates OPEN LATCH capturing the next incremented address.
- The binary counter increments bits <4:2> again.

Figure 9-1 is a timing diagram of a cache invalidate operation when the DMA is a block write and BDAL <1> is 0. Figure 9-2 is the cache invalidate operation timing when BDAL <1> is 1.

If the bus device does not assert RDOUT a second time, then the operation is not a block write, the bus device negates RSYNC, and the DMA monitoring logic returns to an idle state when the last cache invalidate is acknowledged by the memory controller.

DATIO Cache Invalidates

The cache invalidate flag is also posted for DATIO (read interlocked) bus cycles. A DATIO operation is a read followed by a write. During the address portion of a DATIO operation, RSYNC and RSACK are asserted, but RWTBT is not asserted.

When the DMA monitoring logic detects the assertion of RSYNC with DMA IN PROG from the function decoder PAL, the input side to the Q22 bus read register closes and the PAL in the monitoring logic negates the signal MCTB OPEN LATCH. Negating OPEN LATCH causes the address in the Q22 bus read register to be latched in the cache invalidate pipeline register.

Next, RDOUT is asserted. This signal indicates that the data to be written to the address that was just read from are now on the bus. The assertion of RDOUT causes the PAL in the monitoring logic to assert the cache invalidate flag. The address to be invalidated in the cache is the address that is latched in the cache invalidate pipeline register.

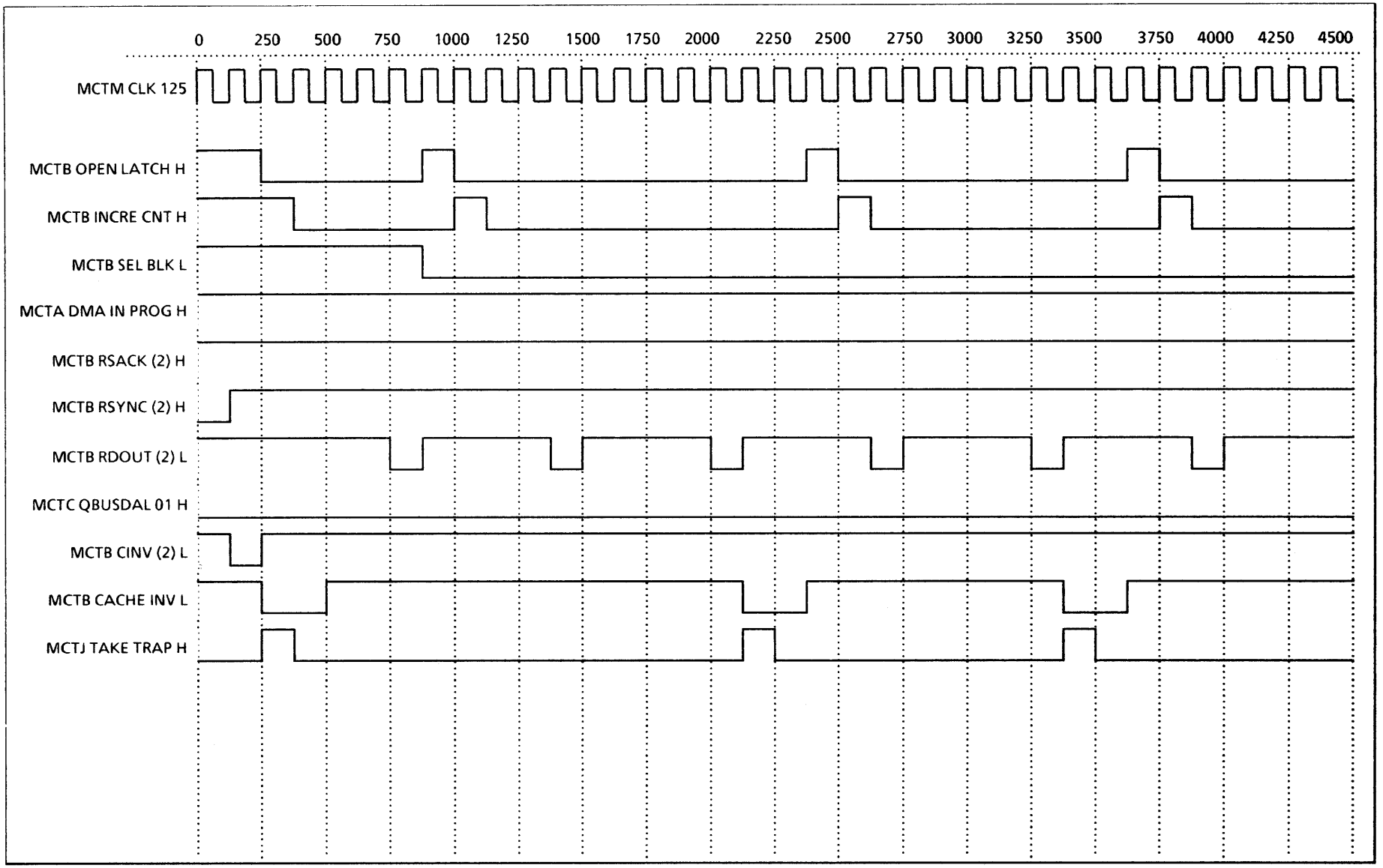


Figure 9-1. Block Write Cache Invalidate Timing Diagram: BDAL<1> = 0

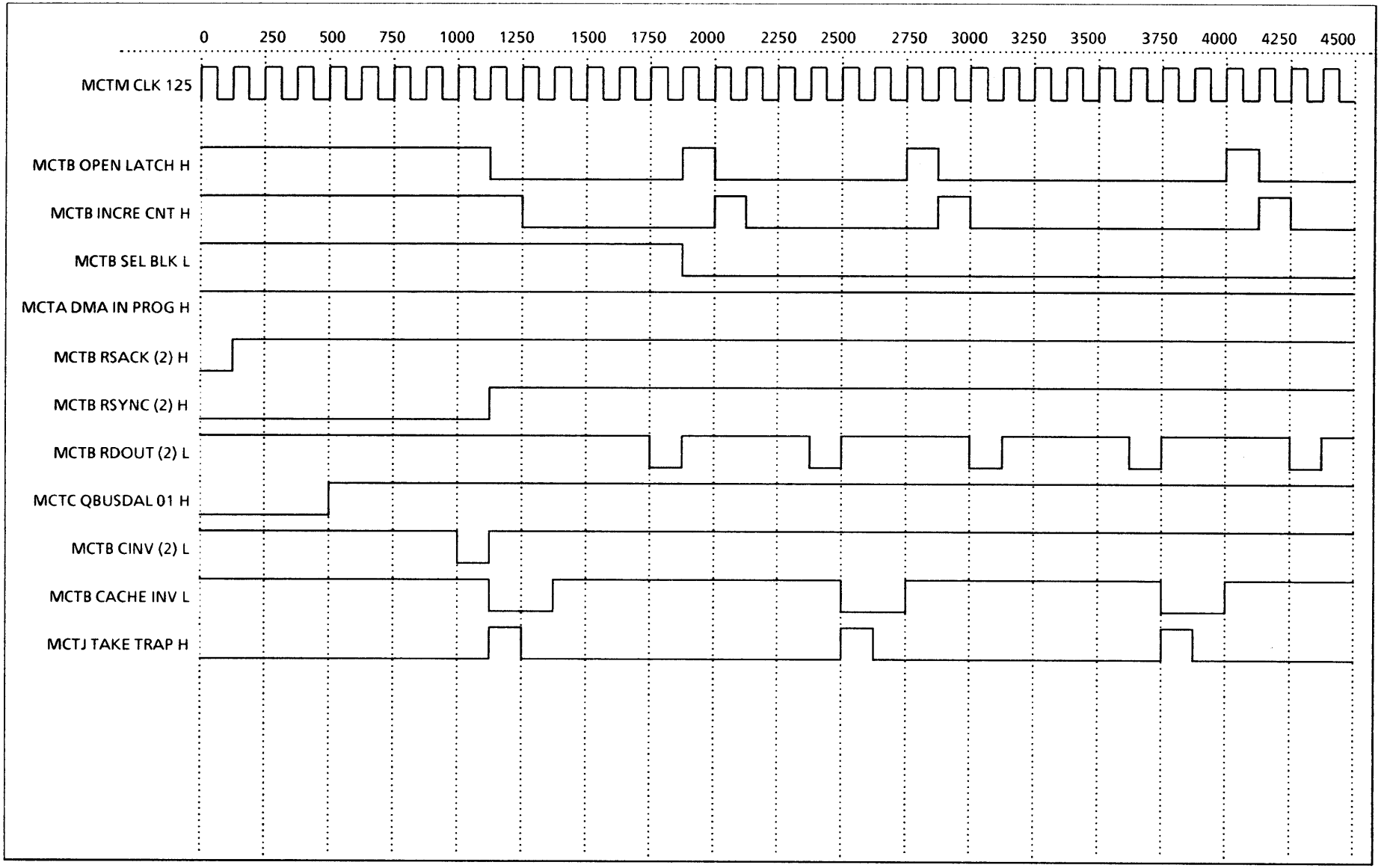


Figure 9-2. Block Write Cache Invalidate Timing Diagram: BDAL<1> = 1

Communicating with MCT and DAP

There are six status signals generated by various pieces of the Q22 bus controller hardware. The Q22 bus controller uses five of these signals to communicate its progress to the memory controller during a function request. The Q22 bus controller can also send a write timeout status flag to the data path module (DAP). These six signals are described in the following paragraphs.

Block Mode

The control and status PAL asserts the signal MCTA BLOCK MODE OK during a write block or read block bus operation to signify that the memory is able to handle the next data transfer as a block mode transfer. This signal is one input to the memory controller branch MUX.

During a read block transfer, the block mode OK flag is only used by the Q22 bus controller since the memory controller microcode always loads the next address after the data from the previous transfer are latched.

SYNCREADY

The signal MCTA SYNCREADY is true 30 ns maximum after the rising edge of the 125 ns clock to allow the Q22 bus controller to coordinate read and write events on the bus with the memory controller. SYNCREADY is asserted when data are available on a read from the Q22 bus, when data or an address is needed for a write to a Q22 bus device, and when an error is posted by the Q22 bus controller. The signal is generated by the sequencer PAL and is one input to the memory controller branch MUX.

For block read transfers, SYNCREADY is asserted during the cycle that data are loaded in the Q22 bus read register and remains asserted for one 125 ns microcycle. If block mode is not asserted with SYNCREADY, the next address that is always loaded by the memory controller microcode for block read operations is used to start another microcycle.

During a write cycle, the word to be written is placed in the Q22 bus write register during the microcycle following the first assertion of SYNCREADY. If any function code from the memory controller is still asserted at this time, the address of the next word is loaded during the cycle following the next assertion of SYNCREADY. The cycle is then repeated for each subsequent word transfer.

During block write operations, the block mode OK signal is used with SYNCREADY to coordinate the loading of data or address into the Q22 bus write register. The memory controller microcode loads the first word in the Q22 bus write register during the microcycle following the assertion of SYNCREADY. If block mode is set when SYNCREADY is reasserted, the next word is loaded into the write register during the next microcycle. If block mode is not asserted, the next address is loaded into the Q22 bus write register.

Q22 Bus Timeout

The signal MCTA TIMEOUT is true 40 ns maximum after the rising edge of the 125 ns clock, when a nonexistent memory request bus timeout occurs. This signal can be read at the assertion of SYNCREADY. The signal remains asserted until another function code is received from the memory controller. MCTA TIMEOUT is generated by the timeout logic and is one input to the memory controller branch MUX.

Q22 Bus Error

The signal MCTA QBUS ERROR is the OR of the bus timeout and parity error signals. Once set, it is cleared at the start of a new function request. This signal causes any single or multiple memory function request to abort. MCTA QBUS ERROR is generated by the sequencer PAL and is one input to the memory controller branch MUX.

A Q22 bus memory parity error (MCTA QBUS ERROR and not MCTA TIMEOUT) can be detected 25 ns after the start of the microcycle following SYNCREADY for a bus read transfer.

Cache Invalidate

The signal MCTB CACHE INV is asserted during DMA write or read/write cycles. This signal is synchronized to the 125 ns clock and remains asserted until the memory controller microcode acknowledges the cache invalidate signal by reading the address in the cache invalidate pipeline register. (The cache invalidate pipeline register contains the address written to the Q22 bus read register from the Q22 bus.) MCTB CACHE INV is generated by the DMA monitoring logic and is one input to the memory controller microinstruction decode logic. This signal causes the memory controller microcode to trap to control store address 3FF for a cache invalidate trap.

Write Timeout

The Q22 bus controller asserts the signal MCTB WRT TMO when a bus timeout occurs during a write function. This signal is one input to the interrupt control logic on the data path module. When WRT

TMO is asserted, an interrupt request is posted at IPL 1D (hex).

The data path module can disable write timeout interrupts by asserting the signal DAPE CONSOLE MODE.

Q22 Bus Operations

This section contains a brief summary of the bus signals and the master/slave relationship. This summary is followed by descriptions of the bus operations that are handled by the Q22 bus controller as the result of function requests from the memory controller.

Q22 Bus Signals

The 42 signal lines used in the Q22 bus are:

- Sixteen data/address lines—BDAL<15:0>
- Two address/parity lines—BDAL<17:16>
- Four extended address lines—BDAL<21:18>
- Six data transfer control lines—BBS7, BDIN, BDOUT, BRPLY, BSYNC, BWTBT
- Six system control lines—BHALT, BREF, BEVNT, BINIT, BDCOK, BPOK
- Eight interrupt control and direct memory access control lines—BIAKO/BIAKI, BIRQ4, BIRQ5, BIRQ6, BIRQ7, BDMGO/BDMGI, BDMR, BSACK

All Q22 bus signals are asserted low and negated high, except BPOK and BDCOK, which are asserted high and negated low to indicate an event such as impending loss of power.

With the exception of DMA grant and interrupt acknowledge signals, Q22 bus signals are bidirectional; that is, they can be driven or received at any point

along the signal line. When driven, bidirectional signals travel from the driver to the near end terminator, and from the driver to the far end terminator. The exceptions are BIAKO, BIAKI, BDMGO, and BDMGI.

BIAKI (interrupt acknowledge) is received by a bus device on one pin and conditionally transmitted out on a different pin as BIAKO to the next bus device. (The signal is not transmitted to the next bus device if the receiving bus device has the highest priority interrupt pending.)

Bus wiring connects BIAKO as output from one device to BIAKI as input to the next device on the bus. BDMGI and BDMGO form a similar priority daisy chain for Bus Mastership Grant. Appendix A of this manual contains functional descriptions of the Q22 bus signals.

Devices connect to all Q22 bus lines through high impedance receivers and gated, high current, open-collector drivers. Receivers and drivers are actually considered part of the bus.

Master/Slave Relationship

Communication between devices on the bus is asynchronous. A master/slave relationship exists throughout each bus transaction. At any time, there is one device that has control of the bus. This controlling device is called the bus master. The master device controls the bus when communicating with another device on the bus, called the slave. The bus master (typically the Q22 bus controller or a DMA device) initiates a bus transaction. The slave device responds by acknowledging the transaction in progress and by receiving data from, or transmitting data to, the bus master. Q22 bus control signals transmitted or received by the bus

master or bus slave device must complete the sequence according to bus protocol.

Read Word

The mnemonic for a read word bus operation is DATI. The memory controller microcode latches an address in the Q22 bus write register and sends the function code for DATI to the Q22 bus controller. The read word operation addresses the proper bus device and reads one aligned word from the location specified in the address. When the Q22 bus controller asserts SYNCREADY, the word read at the specified address is stable in the cache invalidate pipeline register. The following paragraphs describe the sequence of bus cycles and control signals that happen for a read word bus operation.

In the idle state, the Q22 bus controller asserts the signals ENDALADD, ENDAL, and EN DMA. The memory controller asserts the signal MCTN WR LE to latch the address for the read in the Q22 bus write register. Once the address is latched and the Q22 bus controller receives the DATI function code from the memory controller, the function decoder PAL in the Q22 bus controller negates EN DMA and asserts MFUN GO. The address in the write register is now on the bus because ENDALADD and ENDAL were asserted.

In the next microcycle, receiving MFUN GO causes the sequencer PAL to assert PRESYNC. Receiving PRESYNC causes the control and status PAL to assert TSYNC EN.

In the third microcycle, the go bit arrives from the memory controller. The assertion of TSYNC EN plus the go bit causes the assertion of TSYNC. TSYNC informs the addressed bus device that a valid address is on the bus. Also in this third microcycle, the sequencer PAL asserts SYNC HOLD.

In the fourth microcycle, the sequencer PAL asserts TDIN, and negates ENDALADD, ENDAL and PRESYNC. Now the sequencer PAL waits for the addressed bus device to assert RRPLY.

Some number of microcycles later, the bus device has accessed the desired data and placed it on the bus data/address lines. The device informs the Q22 bus controller of this by asserting RRPLY.

Once the sequencer PAL receives RRPLY (2), which is a synchronized version of RRPLY, it negates TDIN. This causes the data to be latched in the Q22 bus read register. The sequencer PAL also negates SYNC HOLD and TSYNC EN in this cycle, and asserts SYNC-READY. SYNCREADY is one input to the memory controller and informs the memory controller that the requested data are available in the read register.

The sequencer PAL negates SYNCREADY one microcycle later. In the same microcycle, it asserts ENDAL and ENDALADD, then waits for the bus device to negate RRPLY. When RRPLY negates, TSYNC negates, and the Q22 bus controller returns to the idle state. Figure 9-3 is a timing diagram of a typical read word bus operation.

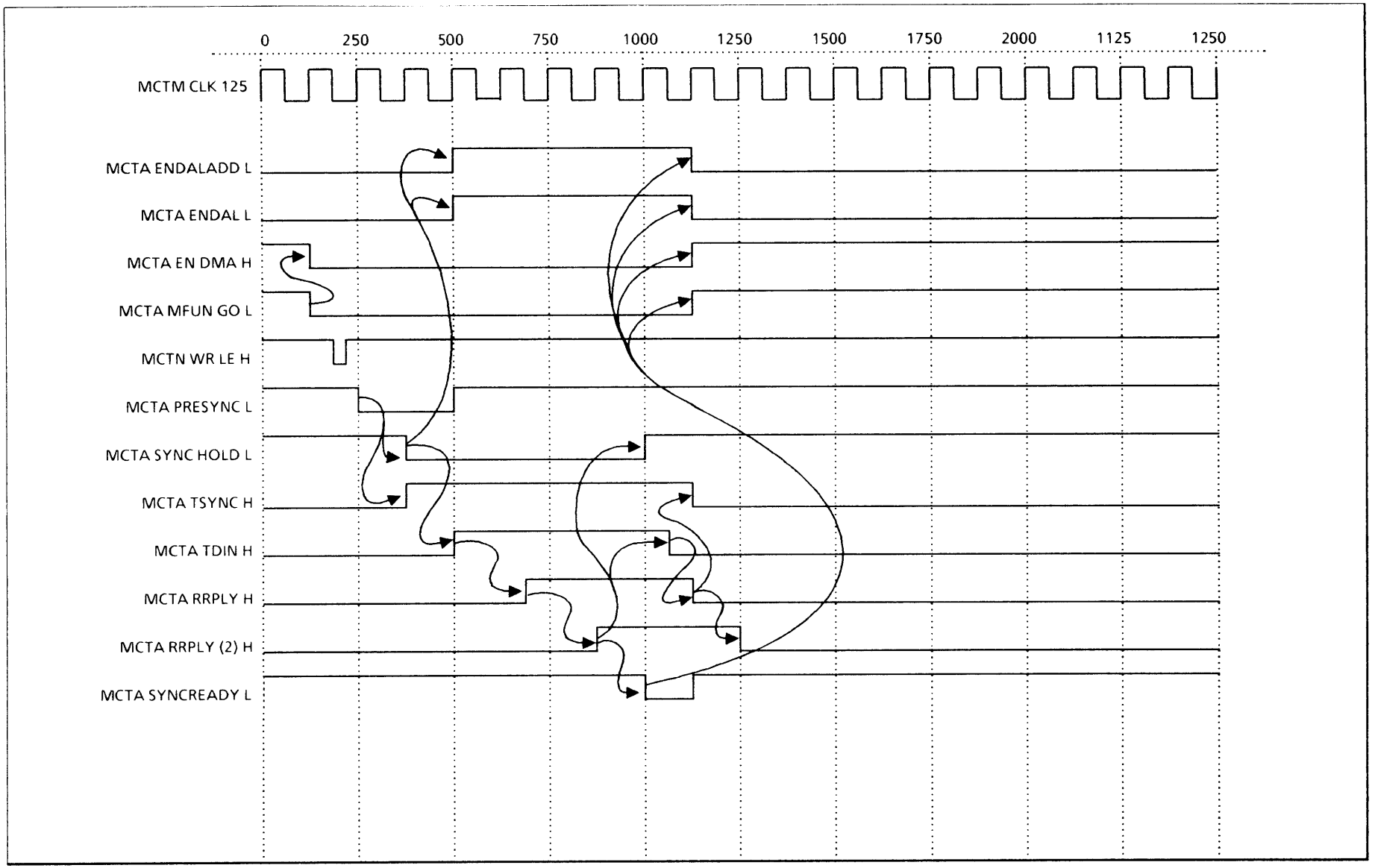


Figure 9-3. Read Word Timing Diagram

Read Block

The mnemonic for a read block bus operation is DATBI. A read block operation is the same as a read word operation except that from one to four sequential words can be read from memory. The memory controller microcode sends the function code for DATBI to the Q22 bus controller and latches an address in the Q22 bus write register for each word it wants read in case the memory does not support block mode. However, if the memory does support block mode transfers, the Q22 bus controller only uses the first address posted in the write register.

The read block operation reads one aligned word from the location specified by the first address in the write register. When the Q22 bus controller asserts SYNC-READY, the word read at the specified address is stable in the cache invalidate pipeline register. The block mode OK flag is asserted and the next sequential word in memory is returned. The following paragraphs describe the differences between the bus cycles and control signals that happen for a read word bus operation and those that happen for a read block bus operation.

The first three microcycles of a read block operation are the same as those for a read word. The fourth microcycle is also the same except in addition, the function decoder PAL generates the signal MCTA BM TBS7. This signal in turn generates the bus signal BBS7 which informs the Q22 bus memory that the current operation is a block mode read and therefore the next word in memory is also desired.

If the memory supports block mode transfers, it asserts the signal MCTC RREF in addition to RRPLY. When the control and status PAL receives MCTC RREF, it asserts MCTA BLOCK MODE OK. For a read word

operation, the sequencer PAL negates SYNC HOLD in the microcycle after it receives RRPLY (2). For a read block operation, the sequencer PAL keeps SYNC HOLD asserted. As a result, TSYNC remains asserted.

The other difference between read word and read block operations is that as long as the memory controller continues to supply the DATBI function code, the Q22 bus controller repeats the microcycles and control signals starting from the fourth microcycle on. The controller returns to the state described earlier as that of the fourth microcycle each time that RRPLY (2) negates, indicating the end of a word read.

In the microcycle before the last read transfer, the memory controller changes the function code to DATI instead of DATBI. This causes the function decoder PAL to negate BM TBS7. The Q22 bus controller then completes the remaining microcycles as it would for any read word operation. Figure 9-4 is a timing diagram of a typical read block bus operation.

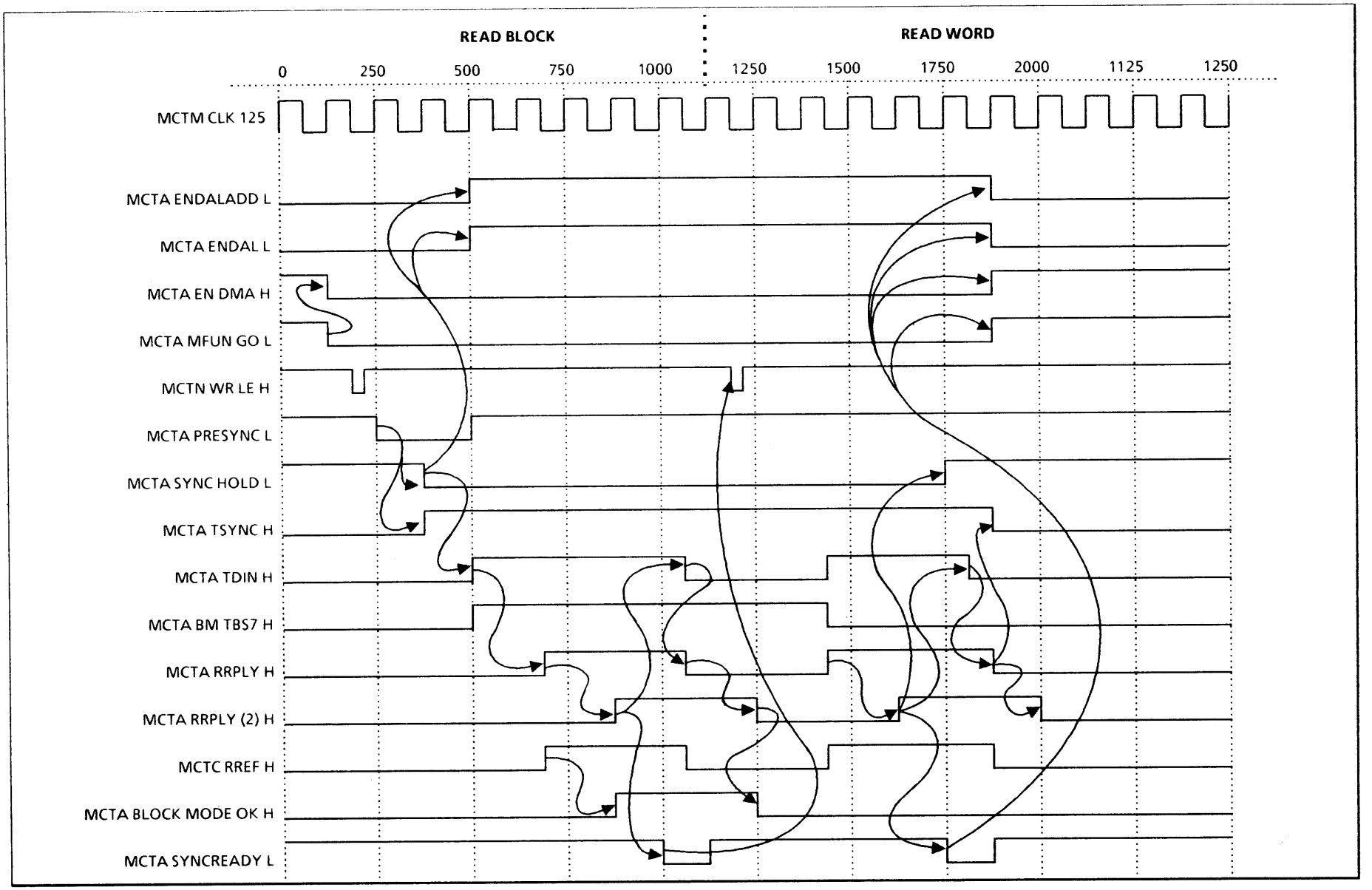


Figure 9-4. Read BlockTiming Diagram

Write Byte and Write Word

The mnemonic for a write byte bus operation is DATOB; the mnemonic for write word is DATO. Write byte writes a single byte of data to a bus device. Write word writes one aligned word of data to a bus device. The only other difference between these two bus operations is that for write byte, the signal TWTBT remains asserted during the entire operation; that is, it is asserted for both the address and data portions of the bus operation. For write word, TWTBT is negated in the fourth microcycle; TWTBT is only asserted during the address portion of the bus operation.

For both operations, the memory controller microcode latches an address in the Q22 bus write register and sends the function code to the Q22 bus controller. When the Q22 bus controller asserts SYNCREADY, the memory controller latches the byte or word to be written in the Q22 bus write register. The memory controller can request up to four successive write byte or write word operations before it must allow the Q22 bus controller to rearbitrate the bus. The following paragraphs describe the sequence of bus cycles and control signals that happen for a write byte or write word bus operation.

In the idle state, the Q22 bus controller asserts the signals ENDALADD, ENDAL, and EN DMA. The memory controller asserts the signal MCTN WR LE to latch the address in the Q22 bus write register. Once the address is latched and the Q22 bus controller receives the write byte or write word function code from the memory controller, the function decoder PAL in the Q22 bus controller negates EN DMA and asserts MFUN GO. The address in the write register is now on the bus because ENDALADD and ENDAL were

asserted. In addition, the function decoder PAL asserts MCTA TWTBT.

In the next microcycle, receiving MFUN GO causes the sequencer PAL to assert PRESYNC. Receiving PRESYNC causes the control and status PAL to assert TSYNC EN.

In the third microcycle, the go bit arrives from the memory controller. The assertion of TSYNC EN plus the go bit causes the assertion of TSYNC. TSYNC informs the addressed bus device that a valid address is on the bus. Also in this third microcycle, the sequencer PAL asserts SYNC HOLD and SYNCREADY. The assertion of SYNCREADY informs the memory controller that the address has been taken, and the Q22 bus controller is ready for the data that is to be written.

In the fourth microcycle, the sequencer PAL negates PRESYNC and ENDALADD but continues to assert ENDAL. The memory controller latches the data to be written in the write register. If this is a write byte operation, the function decoder PAL keeps TWTBT asserted; if not, the function decoder PAL negates TWTBT.

In the fifth microcycle, the sequencer PAL asserts TDOUT to inform the bus device that valid data are on the bus. The fact that ENDAL is asserted causes the contents of the write register (the data to be written) to be driven onto the bus. Now the sequencer PAL waits for the addressed bus device to assert RRPLY.

Some number of microcycles later, the bus device has written the data to the addressed location and asserts RRPLY. Once the sequencer PAL receives RRPLY (2) (the synchronized version of RRPLY), it negates TDOUT. The sequencer PAL also negates SYNC HOLD which causes TSYNC EN to negate, and it asserts SYNCREADY for one cycle only. SYNCREADY

informs the memory controller that the data was written to the addressed location.

In the next microcycle, the sequencer PAL negates SYNCREADY, asserts ENDALADD, and waits for the bus device to negate RRPLY.

Once RRPLY negates, the Q22 bus controller returns to the idle state by negating TSYNC. Figure 9-5 is a timing diagram of a typical write byte/write word bus operation.

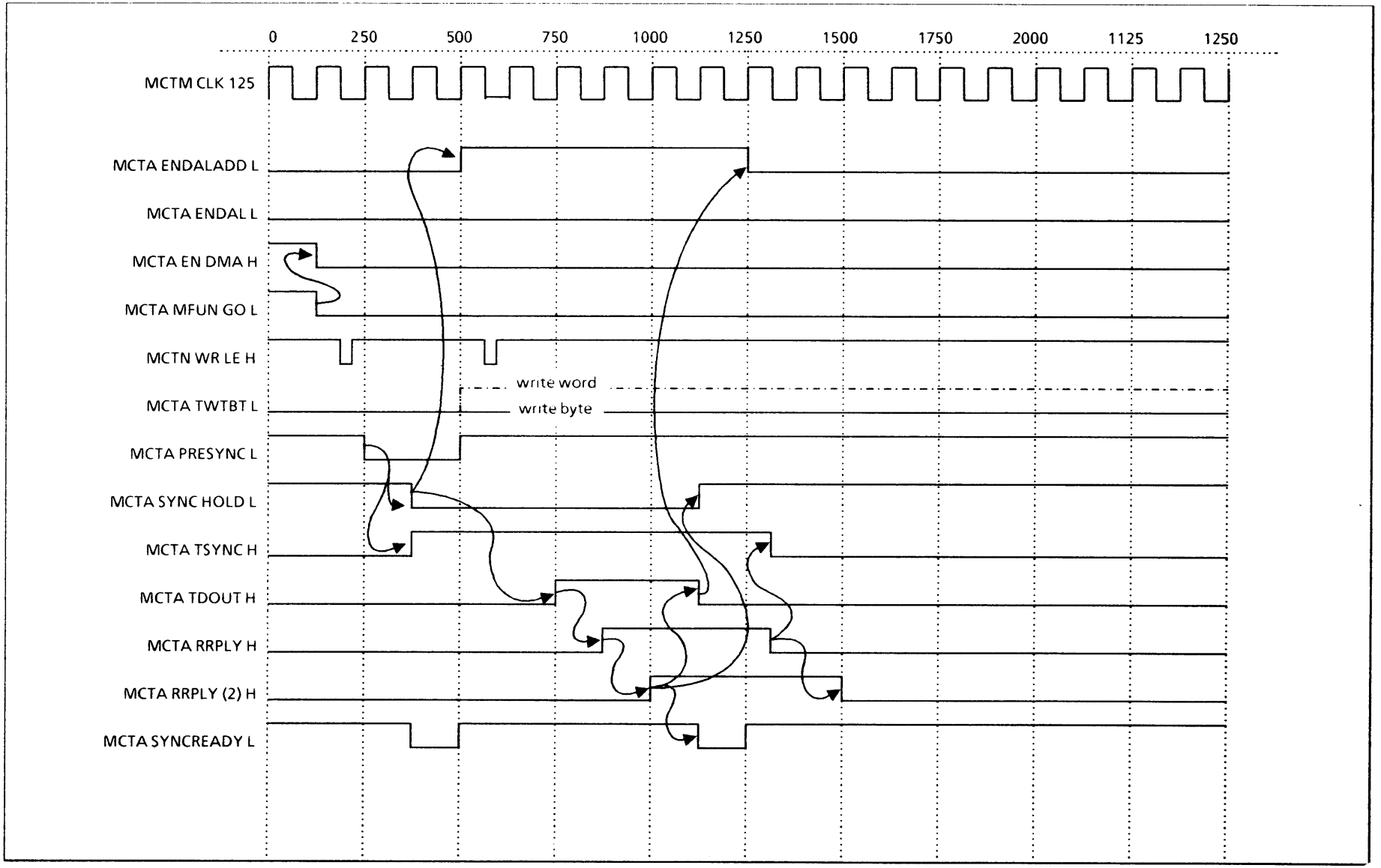


Figure 9-5. Write Byte/Write Word Timing Diagram

Write Block

The mnemonic for a write block bus operation is DATBO. A write block operation is the same as a write word operation except that from one to four sequential words can be written to memory. The memory controller microcode sends the function code for DATBO to the Q22 bus controller and latches an address in the Q22 bus write register for the first word it wants written.

All of the requirements for the first word transfer are the same as for a write word bus operation. During the microcycle after the one in which the Q22 bus controller first asserts SYNCREADY, the memory controller loads the Q22 bus write register with the first word of data to be written. If the block mode OK flag is asserted when the Q22 bus controller asserts SYNCREADY the second time, the memory controller latches the next word of data to be written into the write register.

If the block mode OK flag is not asserted when the controller asserts SYNCREADY the second time, the memory controller loads the address of the next word to be written in the write register. The following paragraphs describe the differences between the bus cycles and control signals that happen for a write word bus operation and those that happen for a write block bus operation.

The first five microcycles of a write block operation are the same as those for a write word. The first difference is that when the bus device asserts RRPLY, it also asserts RREF to indicate it can accept another block mode transfer. When the control and status PAL receives MCTC RREF, it asserts MCTA BLOCK MODE OK.

For a write word operation, the sequencer PAL negates SYNC HOLD in the microcycle after it receives RRPLY (2). For a write block operation, the sequencer PAL keeps SYNC HOLD asserted. As a result, TSYNC remains asserted.

The other difference between write word and write block operations is that as long as the memory controller continues to supply the DATBO function code, the Q22 bus controller repeats the microcycles and control signals starting from the fourth microcycle on. The controller returns to the state described earlier as that of the fourth microcycle each time that RRPLY (2) negates.

In the microcycle before the last write transfer, the memory controller changes the function code to DATO instead of DATBO (write word instead of write block). The Q22 bus controller then completes the remaining microcycles as it would for any write word operation. Figure 9-6 is a timing diagram of a typical write block bus operation.

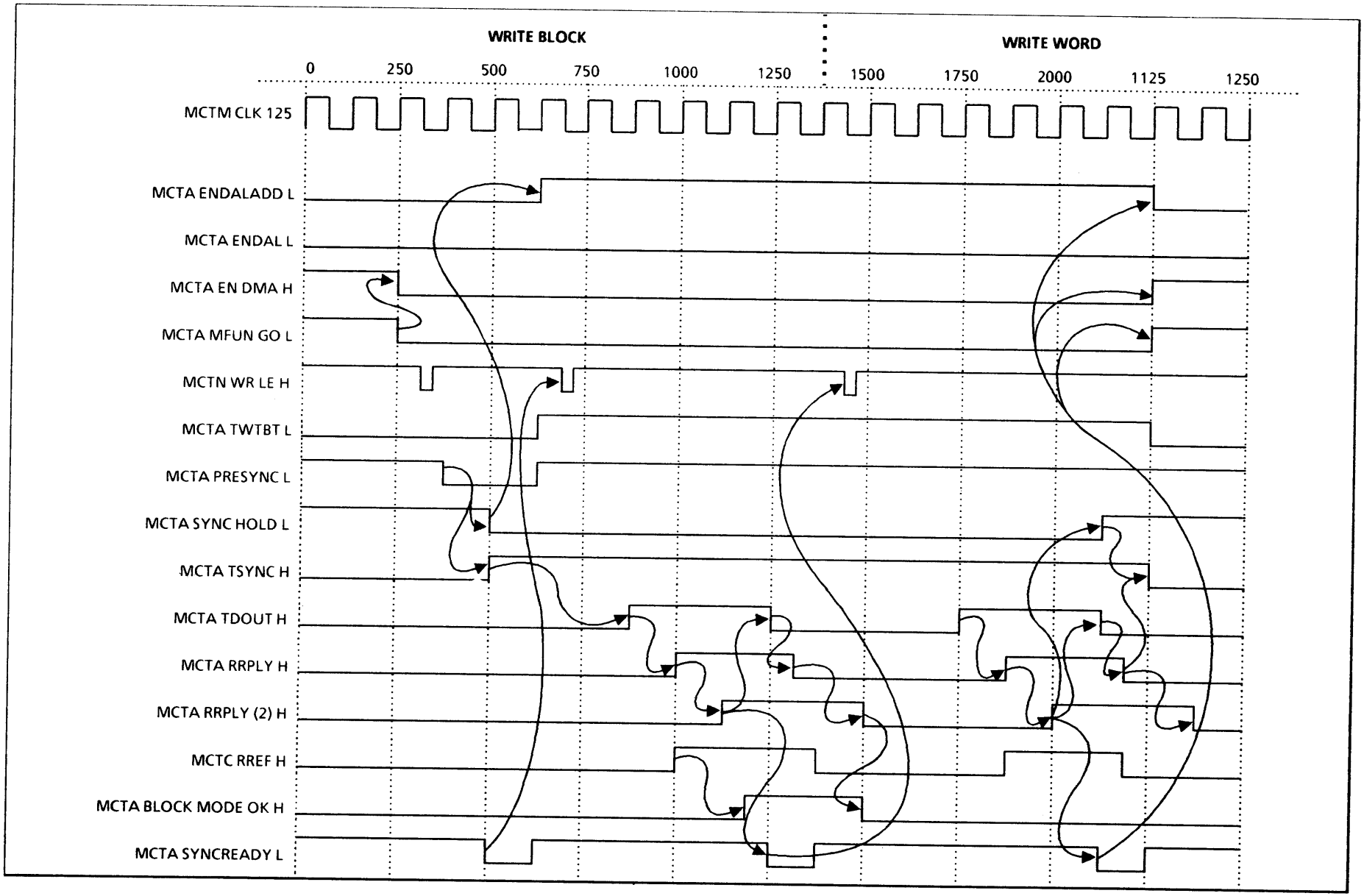


Figure 9-6. Write BlockTiming Diagram

Read Interlocked

The mnemonic for a read interlocked bus operation is DATIO. A read interlocked operation is simply a read followed by a write. The memory controller microcode sends the function code for DATIO to the Q22 bus controller and latches an address in the Q22 bus write register. The Q22 bus controller initiates a read word bus operation and a word of data is read from the addressed bus device.

After reading the data, the Q22 bus controller retains bus mastership and idles. It continues to hold the bus until the memory controller microcode is ready to write data back to the previously addressed bus device. To do this, the memory controller microcode requests a write word or write byte function. When the Q22 bus controller receives the write function request, it asserts SYNCREADY. The memory controller latches the data to be written in the Q22 bus write register during the next microcycle. The Q22 bus controller then continues with the write word or write byte bus operation.

The sequence of bus cycles and control signals for a read interlocked bus operation are identical to those for a read word bus operation followed by a write word or write byte bus operation. The following paragraphs describe the additional signals involved for DATIO.

In the fourth microcycle of the read word portion, the function decoder PAL asserts the signal MCTA READ LOCK because of the DATIO function code from the memory controller and because of the assertion of the signal RSYNC (2). RSYNC (2) is simply a synchronized version of TSYNC.

When the read word portion finishes, the control and status PAL continues to assert TSYNC EN because READ LOCK is asserted. TSYNC EN asserted causes

TSYNC to remain asserted, allowing the Q22 bus controller to retain bus mastership.

Some number of microcycles later, the memory controller sends the write byte or write word function code to start the write portion of DATIO. As always, it also latches an address for the write in the Q22 bus write register. Since ENDAL and ENDALADD are asserted, the address is driven on the Q22 bus. However, the fact that TSYNC has remained asserted causes the addressed bus device to ignore the address from the write register. The sequencer PAL asserts SYNCREADY so that the memory controller latches the data to be written in the write register.

As for a normal write byte or write word bus operation, ENDALADD negates in the fourth microcycle of the write portion. This causes READ LOCK to negate. TSYNC still remains asserted though because the sequencer PAL asserted SYNC HOLD in the previous microcycle. ENDAL remains asserted and the data to be written are driven from the write register onto the Q22 bus. The remaining bus cycles are carried out just as they would be for any write byte or write word bus operation.

Read Interrupt Vector

When a bus device posts an interrupt request, the request is received by the data path module. The data path microcode decides if the interrupt request is to be granted. When the request is granted, the data path sends the READ.VECTOR Memory Request microinstruction to the memory controller. The memory controller in turn sends the read interrupt vector function code to the Q22 bus controller.

The Q22 bus controller acknowledges the interrupt request and reads the interrupt vector off the bus. It

places the vector in the low-order word of the Q22 bus read register and also in the low-order word of the cache invalidate pipeline register. The sequencer PAL asserts SYNCREADY to notify the memory controller that the vector it requested is available in the cache invalidate pipeline register. The vector remains in the cache invalidate pipeline register until 33 ns into the microcycle in which SYNCREADY is asserted. The following paragraphs describe the sequence of bus cycles and control signals that happen for a read interrupt vector bus operation.

In the idle state, the Q22 bus controller asserts the signals ENDALADD, ENDAL, and EN DMA. The memory controller posts the function code for read interrupt vector, and generally sends the go bit in the same microcycle. As soon as the function decoder PAL receives the function code, it negates EN DMA and asserts MFUN GO. The function decoder PAL also asserts the signal MCTA EN IAKO (enable interrupt acknowledge output).

In the second microcycle, the sequencer PAL asserts TDIN, and negates ENDAL and ENDALADD.

In the third microcycle, the combination of TDIN and EN IAKO asserted causes the signal EN IAKO BUF to be asserted.

In the fourth microcycle, the combination of TDIN and EN IAKO BUF asserted causes the signal MCTA TIAKO to be asserted. TIAKO is the interrupt acknowledge signal. The Q22 bus controller now waits for the device with the highest priority interrupt request pending to respond by asserting RRPLY.

Some number of microcycles later, the interrupting device with the highest priority asserts RRPLY and places its vector on the bus. Since TDIN is asserted, the vector is latched in the cache invalidate pipeline

register. When the sequencer PAL receives RRPLY (2) (the synchronized version of RRPLY), it negates TDIN and asserts SYNCREADY to inform the memory controller that the vector is available in the cache invalidate pipeline register. In this same microcycle, TIAKO is negated.

Finally, the sequencer PAL asserts ENDAL and ENDALADD in the next microcycle and the read interrupt vector bus operation completes when the interrupting device negates RRPLY. Figure 9-7 is a timing diagram of a typical read interrupt vector bus operation.

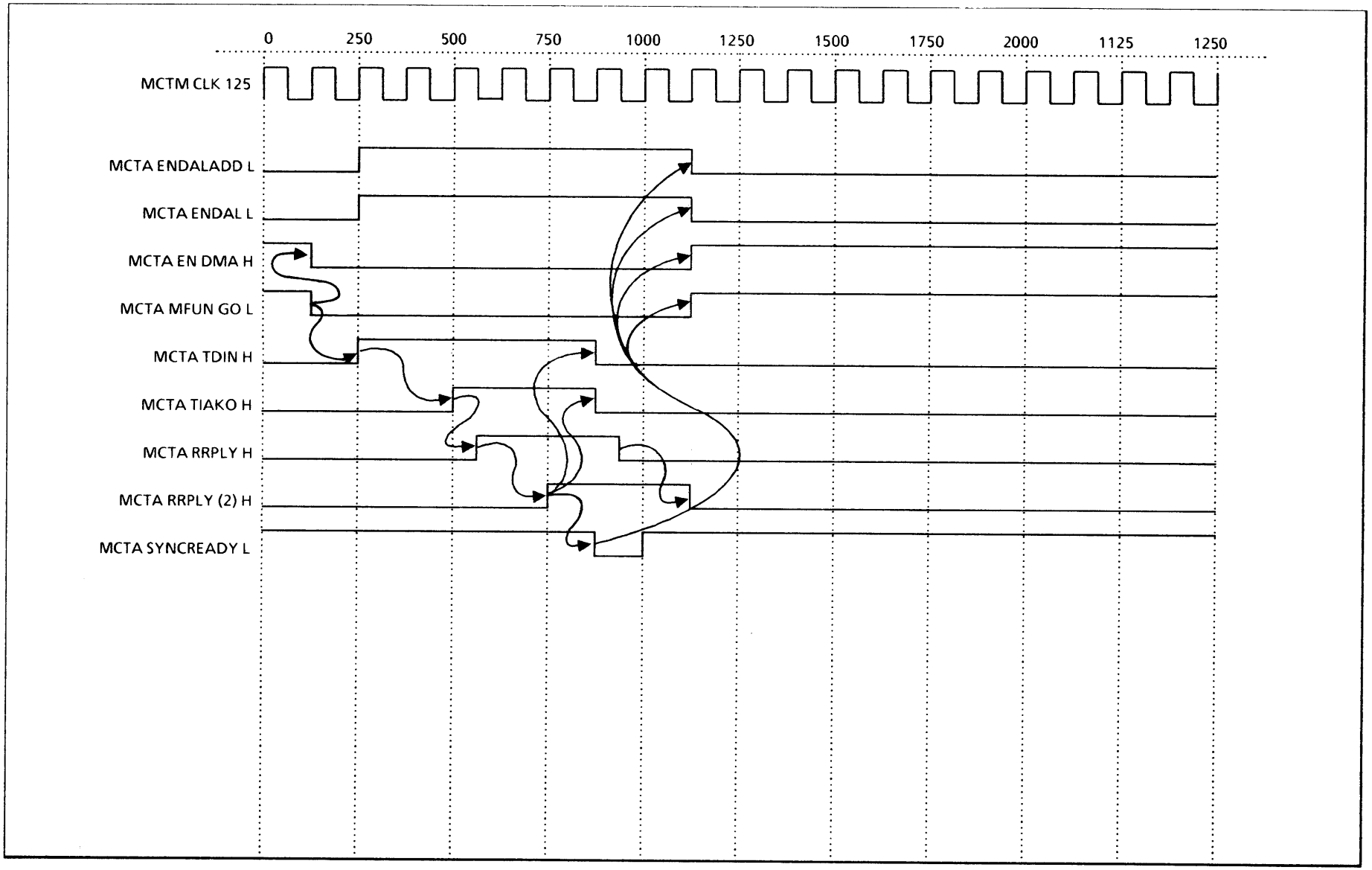


Figure 9-7. Read Interrupt Vector Timing Diagram

Appendix A
Q22 Bus Signals

This appendix describes the Q22 bus signals and their functions, the associated bus pins, and the signal mnemonics.

The backplane used in the MicroVAX I system is the H9278-A backplane. Those Q22 bus signals not used in the H9278-A backplane are noted.

Bus Pin	Signal Mnemonic	Signal Function
AA1	BIRQ 5 L	Interrupt request priority level 5
AB1	BIRQ 6 L	Interrupt request priority level 6
AC1	BDAL 16 L	Address line 16 during addressing protocol; parity control line during data transfer protocol
AD1	BDAL 17 L	Address line 17 during addressing protocol; parity control line during data transfer protocol
AE1	SSPARE1	Special spare; not assigned or bussed in DIGITAL cable or backplane assemblies; available for user connection. (continued)

Bus Pin	Signal Mnemonic	Signal Function
AE1	SSPARE1 continued	Optionally, this pin may be used for +5 V battery backup power to keep critical circuits alive during power failures. A jumper is required on LSI-11 bus options to open (disconnect) the +5 V battery circuit in systems that use this line as SSPARE1. This spare is not used in the MicroVAX I system.
AF1	SSPARE2	Special spare; not assigned or bussed in DIGITAL cable or backplane assemblies. The memory controller module (M7136) in the MicroVAX I uses this pin in slot 1 to indicate its Run state. This pin is unused in other slots and is available for user connection.
AH1	SSPARE3	Special spare; not assigned or bussed in DIGITAL cable or backplane assemblies; available for user interconnection. This spare is not used in the MicroVAX I system.
AJ1	GND	Ground; system signal and dc return

Bus Pin	Signal Mnemonic	Signal Function
AK1 AL1	MSPAREA MSPAREB	Maintenance spares; normally connected together on the backplane at each option location (not a slot-to-slot bussed connection). The H9278-A backplane connects these together, but they are unused in the MicroVAX I system.
AM1	GND	Ground; system signal and dc return
AN1	BDMR L	Direct memory access (DMA) request; a device asserts this signal to request bus mastership.
AP1	BHALT L	Processor halt; when BHALT is asserted, the processor responds by going into its halt state.
AR1	BREF L	Memory refresh; used during refresh protocol to override memory bank selection decoding and cause all banks to be selected. Asserted or negated with BRPLY L by block mode slave devices to indicate to the bus master if the slave can accept another block mode DIN or DOUT transfer.

Bus Pin	Signal Mnemonic	Signal Function
AS1	+5B or +12B battery	+5 or +12 V dc battery backup power to keep critical circuits alive during power failures. A jumper is required on all LSI-11 bus options to open (disconnect) the backup circuit from the bus in systems that use this line at the alternate voltage. This signal is not bussed to BS1 in the H9278-A backplane, and is unused in the MicroVAX I system.
AT1	GND	Ground; system signal and dc return
AU1	PSPARE1	Power spare 1; not assigned a function; not used in the H9278-A backplane; not recommended for use. If a backplane is bussing −12 V (on pin BB2) and a module is inserted upsidedown in the backplane, −12 V dc appears on pin AU1. If AU1 is unused on the module, no damage occurs.
AV1	+5B	+5 V battery backup power; to keep critical circuits alive during power failures. The H9278-A backplane does not bus this signal, and it is unused in the MicroVAX I system.

Bus Pin	Signal Mnemonic	Signal Function
BA1	BDCOK H	DC power OK; power supply generated signal that is asserted when there is sufficient dc voltage available to sustain reliable system operation.
BB1	BPOK H	AC power OK; asserted by the power supply when primary power is normal. When negated during processor operation, a power fail interrupt is initiated.
BC1	BDAL 18 L	Data/address line 18
BD1	BDAL 19 L	Data/address line 19
BE1	BDAL 20 L	Data/address line 20
BF1	BDAL 21 L	Data/address line 21
BH1	SSPARE8	Special spare; not assigned or bussed in DIGITAL cable or backplane assemblies; available for user interconnection. This spare is not used in the MicroVAX I system.
BJ1	GND	Ground; system signal and dc return

Bus Pin	Signal Mnemonic	Signal Function
BK1 BL1	MSPAREB MSPAREB	Maintenance spares; normally connected together on the backplane at each option location (not a slot-to-slot bussed connection). The H9278-A backplane connects these together, but they are unused in the MicroVAX I system.
BM1	GND	Ground; system signal and dc return
BN1	BSACK L	This signal is asserted by a DMA device in response to the processor's BDMGO L signal, indicating the DMA device is accepting bus mastership. The device remains bus master until it negates BSACK L.
BP1	BIRQ 7 L	Interrupt request priority level 7
BR1	BEVNT L	External event interrupt request; this signal is not used by the MicroVAX I.
BS1	+12B	+12 V dc battery backup power; this signal is not bussed to AS1 in the H9278-A backplane, and it is unused in the MicroVAX I system.
BT1	GND	Ground; system signal and dc return

Bus Pin	Signal Mnemonic	Signal Function
BU1	PSPARE2	Power spare 2; not assigned a function; not used in the H9278-A backplane; not recommended for use. If a backplane is bussing −12 V (on pin AB2) and a module is inserted upsidedown in the backplane, −12 V dc appears on pin BU1. If BU1 is unused on the module, no damage occurs.
BV1	+5	Normal +5 V dc system power
AA2	+5	Normal +5 V dc system power
AB2	−12	−12 V dc power for devices requiring this voltage. The MicroVAX I power supply (H7864) does not provide −12 V dc and this signal is not bussed on the H9278-A backplane.
AC2	GND	Ground; system signal and dc return
AD2	+12	Normal +12 V dc system power
AE2	BDOUT L	Data output; BDOUT, when asserted, implies that valid data are available on BDAL <15:00> and that an output transfer, with respect to the bus master, is taking place.

Bus Pin	Signal Mnemonic	Signal Function
AF2	BRPLY L	Reply; BRPLY L is asserted in response to BDIN or BDOUT and during interrupt acknowledge transactions. It is generated by a slave device to indicate that it will place its data on the bus, or that it will accept data from the bus, according to the appropriate protocol.
AH2	BDIN L	Data input; BDIN is used for two types of bus operations: 1. When asserted with BSYNC, BDIN implies an input transfer with respect to the current bus master and requires a response (BRPLY) from the addressed slave. 2. The interrupt fielding processor initiates interrupt service by asserting TDIN L followed by TIACK L.
AJ2	BSYNC L	Synchronize; BSYNC is asserted by the bus master device to indicate that it has placed an address on the bus. The transfer is in process until BSYNC is negated. In block mode, BSYNC remains asserted until the last transfer cycle is completed.

Bus Pin	Signal Mnemonic	Signal Function
AK2	BWTBT L	Write/byte; BWTBT is used two ways to control a bus cycle: 1. It is asserted during the address portion of a cycle to indicate that an output cycle is to follow (DATO, DATOB, DATBO) rather than an input cycle. 2. It is asserted during the data portion of a DATOB or DATIOB bus cycle to indicate a byte rather than a word transfer is to take place.
AL2	BIRQ 4 L	Interrupt request priority level 4
AM2 AN2	BIAKI L BIAKO L	Interrupt acknowledge; the processor asserts BIAKO to acknowledge an interrupt. The bus transmits this to BIAKI of the next priority device (electrically closest to the processor). This device accepts the interrupt acknowledge under two conditions, as follows. (continued)

Bus Pin	Signal Mnemonic	Signal Function
AM2 AN2	BIAKI L BIAKO L continued	1. The device requested the bus by asserting an interrupt, and 2. The device had the highest priority interrupt request on the bus at the time of the previous BDIN L assertion. If both of these conditions are not met, the device asserts BIAKO L to the next device on the bus. This process continues in a daisy-chain fashion until the device with the highest interrupt priority receives the interrupt acknowledge (IAK) signal and proceeds with the interrupt protocol.
AP2	BBS7 L	Bank 7 select; when the bus master asserts TADDR, it asserts BBS7 to reference the I/O page. The address on BDAL <12:0> when BBS7 is asserted is the address within the I/O page. During DATBI transfers, the bus master asserts this signal with the first data transfer until the last transfer to indicate to the block mode slave that there will be subsequent transfers.

Bus Pin	Signal Mnemonic	Signal Function
AR2 AS2	BDMGI L BDMGO L	Direct memory access grant; the processor asserts this signal to grant bus mastership to a requesting device, according to bus mastership protocol. The signal is passed in a daisy-chain from the processor (as BDMGO) through the bus to BDMGI of the next priority device (electrically closest device on the bus). This device accepts the grant only if it requested bus mastership (by asserting BDMR L). If not, the device passes the grant by asserting BDMGO to the next device on the bus. This process continues until the requesting device acknowledges the grant by asserting BSACK L after BRPLY L and BSYNC L are both negated.
AT2	BINIT L	Initialize; this signal is used for system reset. All devices on the bus are to return to a known, initial state; that is, registers are reset to zero, all bus drivers are disabled and logic is reset to state 0, ready to be addressed for operations.

Bus Pin	Signal Mnemonic	Signal Function
AU2	BDAL 0 L	Data/address line 00; specifies high or low byte during address for DATOB and DATIOB cycles.
AV2	BDAL 1 L	Data/address line 01
BA2	+5	+5 V dc power
BB2	−12	−12 V dc power. The MicroVAX I power supply (H7864) does not provide −12 V dc and this signal is not bussed on the H9278-A backplane.
BC2	GND	Power supply return
BD2	+12	+12 V dc power
BE2	BDAL 2 L	Data/address line 02
BF2	BDAL 3 L	Data/address line 03
BH2	BDAL 4 L	Data/address line 04
BJ2	BDAL 5 L	Data/address line 05
BK2	BDAL 6 L	Data/address line 06
BL2	BDAL 7 L	Data/address line 07
BM2	BDAL 8 L	Data/address line 08
BN2	BDAL 9 L	Data/address line 09
BP2	BDAL 10 L	Data/address line 10
BR2	BDAL 11 L	Data/address line 11
BS2	BDAL 12 L	Data/address line 12

Bus Pin	Signal Mnemonic	Signal Function
BT2	BDAL 13 L	Data/address line 13
BU2	BDAL 14 L	Data/address line 14
BV2	BDAL 15 L	Data/address line 15

Appendix B
Module Finger Pin Assignments

This appendix lists the backplane pin assignments for the data path module (M7135 or M7135-YA) and the memory controller module (M7136) of the MicroVAX I CPU.

Data Path Module Pinout

Connector A

AA1	BIRQ 5 L	AJ2	BSYNC L
AA2	+5V	AK1	
AB1	BIRQ 6 L	AK2	BWTBT L
AB2		AL1	
AC1	BDAL 16 L	AL2	BIRQ 4 L
AC2	GND	AM1	GND
AD1	BDAL 17 L	AM2	BIAKO L
AD2		AN1	BDMR L
AE1		AN2	BIAKO L
AE2	BDOUT L	AP1	BHALT L
AF1		AP2	BBS7 L
AF2	BRPLY L	AR1	BREF L
AH1		AR2	BDMGO L
AH2	BDIN L	AS1	
AJ1	GND	AS2	BDMGO L

AT1	GND	AU2	BDAL 00 L
AT2	BINIT L	AV1	
AU1		AV2	BDAL 01 L

Connector B

BA1	BDCOK H	BL1	
BA2	+5V	BL2	BDAL 07 L
BB1	BPOK H	BM1	GND
BB2		BM2	BDAL 08 L
BC1	BDAL 18 L	BN1	BSACK L
BC2	GND	BN2	BDAL 09 L
BD1	BDAL 19 L	BP1	BIRQ 7 L
BD2	+12V	BP2	BDAL 10 L
BE1	BDAL 20 L	BR1	BEVNT L
BE2	BDAL 02 L	BR2	BDAL 11 L
BF1	BDAL 21 L	BS1	
BF2	BDAL 03 L	BS2	BDAL 12 L
BH1		BT1	GND
BH2	BDAL 04 L	BT2	BDAL 13 L
BJ1	GND	BU1	
BJ2	BDAL 05 L	BU2	BDAL 14 L
BK1		BV1	+5V
BK2	BDAL 06 L	BV2	BDAL 15 L

Connector C

CA1		CB1	DAPL MCT INIT L
CA2	+5V	CB2	

CC1	DAPE CONSOLE MODE H	CM1	BUS MEM CTL 7 H
CC2	GND	CM2	
CD1	BUS MEM CTL 0 H	CN1	MCTM BASE CLOCK H
CD2		CN2	
CE1	BUS MEM CTL 1 H	CP1	MCTM DPC SRC L
CE2		CP2	
CF1	BUS MEM CTL 2 H	CR1	DAPR MEM REQUEST H
CF2		CR2	
CH1	BUS MEM CTL 3 H	CS1	DAPT MEM REQ MODE 0 H
CH2		CS2	
CJ1	BUS MEM CTL 4 H	CT1	GND
CJ2		CT2	
CK1	BUS MEM CTL 5 H	CU1	MCTN REQ ACK L
CK2		CU2	
CL1	BUS MEM CTL 6 H	CV1	MCTT MEM ERR H
CL2		CV2	

Connector D

DA1		DL1	DAPR IB TAKEN L
DA2	+5V	DL2	
DB1	DAPT MEM REQ MODE 1 H	DM1	MCTT IB ERROR H
DB2		DM2	
DC1		DN1	MCTN SEXT WORD H
DC2	GND	DN2	
DD1	MCTN MEM BUSY H	DP1	DAPT MODIFY H
DD2		DP2	
DE1	MCTS TB MISS H	DR1	DAPT SECOND PART H
DE2		DR2	
DF1	MCTS MOD REF H	DS1	MCTN MD BUS IN LE H
DF2		DS2	
DH1	MCTT NXT VALID REG H	DT1	GND
DH2		DT2	S RUN L
DJ1	MCTE PAGE CROSS H	DU1	DAPL MCT 250 L
DJ2		DU2	
DK1	MCTB WRT TMO H	DV1	DAPL TINIT H
DK2		DV2	

Memory Controller Module Pinout

Connector A

AA1		AL1	
AA2	+5V	AL2	
AB1		AM1	GND
AB2		AM2	
AC1	BDAL 16 L	AN1	BDMR L
AC2	GND	AN2	BIAKO L
AD1	BDAL 17 L	AP1	
AD2		AP2	BBS7 L
AE1		AR1	BREF L
AE2	BDOUT L	AR2	
AF1	S RUN L	AS1	
AF2	BRPLY L	AS2	BDMGO L
AH1		AT1	GND
AH2	BDIN L	AT2	
AJ1	GND	AU1	
AJ2	BSYNC L	AU2	BDAL 00 L
AK1		AV1	
AK2	BWTBT L	AV2	BDAL 01 L

Connector B

BA1		BL1	
BA2	+5V	BL2	BDAL 07 L
BB1		BM1	GND
BB2		BM2	BDAL 08 L
BC1	BDAL 18 L	BN1	BSACK L
BC2	GND	BN2	BDAL 09 L
BD1	BDAL 19 L	BP1	
BD2		BP2	BDAL 10 L
BE1	BDAL 20 L	BR1	
BE2	BDAL 02 L	BR2	BDAL 11 L
BF1	BDAL 21 L	BS1	
BF2	BDAL 03 L	BS2	BDAL 12 L
BH1		BT1	GND
BH2	BDAL 04 L	BT2	BDAL 13 L
BJ1	GND	BU1	
BJ2	BDAL 05 L	BU2	BDAL 14 L
BK1		BV1	+5V
BK2	BDAL 06 L	BV2	BDAL 15 L

Connector C

CA1	DAPE CONSOLE MODE H	CL1	
CA2	+5V	CL2	BUS MEM CTL 6 H
CB1		CM1	
CB2	DAPL MCT INIT L	CM2	BUS MEM CTL 7 H
CC1		CN1	
CC2	GND	CN2	MCTM BASE CLOCK H
CD1		CP1	
CD2	BUS MEM CTL 0 H	CP2	MCTM DPC SRC L
CE1		CR1	
CE2	BUS MEM CTL 1 H	CR2	DAPR MEM REQUEST H
CF1		CS1	
CF2	BUS MEM CTL 2 H	CS2	DAPT MEM REQ MODE 0 H
CH1		CT1	GND
CH2	BUS MEM CTL 3 H	CT2	S RUN L
CJ1		CU1	
CJ2	BUS MEM CTL 4 H	CU2	MCTN REQ ACK L
CK1		CV1	
CK2	BUS MEM CTL 5 H	CV2	MCTT MEM ERR H

Connector D

DA1		DL1	
DA2	+5V	DL2	DAPR IB TAKEN L
DB1		DM1	
DB2	DAPT MEM REQ MODE 1 H	DM2	MCTT IB ERROR H
DC1		DN1	
DC2	GND	DN2	MCTN SEXT WORD H
DD1		DP1	
DD2	MCTN MEM BUSY H	DP2	DAPT MODIFY H
DE1		DR1	
DE2	MCTS TB MISS H	DR2	DAPT SECOND PART H
DF1		DS1	
DF2	MCTS MOD REF H	DS2	MCTN MD BUS IN LE H
DH1		DT1	GND
DH2	MCTT NXT VALID REG H	DT2	
DJ1		DU1	
DJ2	MCTE PAGE CROSS H	DU2	DAPL MCT 250 L
DK1		DV1	
DK2	MCTB WRT TMO H	DV2	DAPL TINIT H

Appendix C
Serial Line Cable Pinning

A 10-pin connector is mounted on the data path module of the KD32-AA and KD32-AB CPUs. This connector is for the cable to the console terminal.

In the MicroVAX I system, an internal cable connects the 10-pin connector on the DAP module to the patch panel insert at the rear of the system box. A 25-pin EIA connector is mounted on the patch panel insert for the cable to the console terminal, with 10 pins connected.

The pinout for these connectors is as follows.

1 EIA Data Terminal Ready (always asserted)

2 Ground

3 EIA Received Data (MicroVAX I Transmit Data)

4 Ground

5 Ground

6 Unused (no connection – used for cable key)

7 EIA Signal Ground

8 EIA Transmitted Data (MicroVAX I Receive Data)

9 Ground

10 Unused (no connection)

Appendix D
Microverify

Microverify is a microcoded internal test that runs automatically when the MicroVAX I system is powered on. The "Microverify" section in Chapter 2 of this manual provides a brief description of Microverify. This appendix provides additional detail about Microverify. It describes the two modes of Microverify, the subtests within Microverify, and the two special Memory Request microinstructions used in Microverify.

References throughout this document are to pages in the data path (DAP) and memory controller (MCT) schematic drawings. These pages should point you directly to the chip(s) being tested or to the subsystem under test. Refer to these schematics when you require this level of detail.

Operation

Microverify always runs on system powerup. In addition, the TEST console command runs Microverify. In either case, Microverify runs under control of the main DAP microcode.

Microverify Assumes Nothing Works

Testing proceeds from the most basic elements on the DAP board to the fully integrated two board set. Microverify does not test any Q22 bus functions.

On successful completion of Microverify, the MicroVAX I processor is capable of executing macroinstructions, and control is passed to the console microcode.

If the system was just powered on and is attempting bootstrap, the console microcode locates 64 KB of contiguous good memory, and copies the boot EPROM code (the primary bootstrap) into this area. Control is then passed to the primary bootstrap.

Microverify Operates in Two Modes

A jumper on the DAP board determines whether Microverify runs in "single pass mode" or "infinite loop mode." The differences between these two modes are the exit sequence, console output and error handling. LED output is the same for both operating modes.

Single Pass Mode

If the jumper on the DAP board is in, Microverify runs in single pass mode. The jumper is correctly installed for this mode when the DAP module is shipped from the factory. (See Figure 2-10 in Chapter 2 for the location of this jumper.)

The purpose of single pass mode is to inform the user if the machine is capable of running macrolevel programs. In addition, error status reported in the LEDs isolates module level errors for field service personnel. Microverify always returns error status and control to the main DAP microcode upon completion.

Infinite Loop Mode

If the jumper on the DAP board is out, Microverify runs in infinite loop mode. This mode is used for diagnosing intermittent failures.

Infinite loop mode comes as close as possible to chip level error isolation. In this mode, Microverify never returns control to the main DAP microcode. If a console

is present, a unique character is displayed every time a major test completes successfully.

If no errors are detected in infinite loop mode, Microverify repeats all tests.

If an error is detected, Microverify loops on the test that produced the error, as well as indicating the error in the LEDs. This test continues to repeat even if no error is detected again.

LED Error Codes

The LED error codes and their meanings are as follows. **Note:** A number displayed in the LEDs is meaningful only if the system is in console halt mode, as indicated by the system prompt >>>.

7 (on, on, on):	Failed quick DAP microsequencer test, or Microverify did not begin.
6 (on, on, off):	Error found on DAP module.
5 (on, off, on):	Error found on MCT module.
4 (on, off, off):	Undetermined error in DAP/MCT interface.
3 (off, on, on):	Microverify worked as expected, and control is transferred to the console microcode. If bootstrapping is attempted, control is transferred to the primary bootstrap. If the LEDs remain set to 3 (off, on, on) and bootstrapping was attempted, bad memory was found by either the console microcode or the primary bootstrap.

The patterns fluctuate as Microverify tests different components. An error in either operating mode causes the current pattern to stay lit.

Microverify Console Messages

The following messages are sent to the console as indicated:

- "MICROVERIFY STARTED" when the console is determined usable.
- A character indicating successful completion of each test (infinite loop mode only).
- "MICROVERIFY FAILED" if Microverify fails (single pass mode only).
- "MICROVERIFY PASSED" if Microverify passes.

Note: "MICROVERIFY PASSED (FAILED)" should appear within five seconds after "MICROVERIFY STARTED" appears.

Sample Output, Single Pass Mode

The following messages are displayed on the console terminal when Microverify completes successfully in single pass mode:

```
MICROVERIFY STARTED

MICROVERIFY PASSED
>>>
```

The following messages are displayed on the console terminal when Microverify fails in single pass mode:

```
MICROVERIFY STARTED

MICROVERIFY FAILED
>>>
```

Sample Output, Infinite Loop Mode

If the diagnostic jumper is out, Microverify sends one character to the console every time a test completes successfully. A hexadecimal digit is displayed every

time a major test completes, and a "+" every time a subtest completes. All sixteen hexadecimal digits (0 to F) are used. When the last subtest of a major test completes, the hexadecimal digit is displayed instead of the "+." In the example below, tests 5 and 12 have eight subtests, test 7 has four subtests, and the other major tests have no subtests.

The following messages are displayed on the console terminal when Microverify completes once successfully in infinite loop mode:

```
0
MICROVERIFY STARTED
1234+++++++56+++789AB+++++++CDEF
MICROVERIFY PASSED
```

These messages repeat continually as Microverify loops.

If an error is encountered in test 5, subtest 3, for example, the following messages are displayed on the console terminal:

```
0
MICROVERIFY STARTED
1234++
```

Nothing else appears on the console as Microverify loops on the failing subtest.

Major Functional Areas Tested

The actual testing is divided into seventeen major functional areas, described below. Only those previously untested components are listed in each of the areas. (The seventeen major areas are not necessarily tested in the order they are listed here.)

1. Quick Microsequencer Test

LEDs set to 7 (on, on, on)

- Test the microsequencer jump opcode
  - control store address register (DAPB)
  - control store memory (DAPA)
  - next microaddress (NμA) bus
  - next microaddress (NμA) MUX with select = take branch (DAPB)
  - jump register (DAPB)
  - parity checker with parity = good (DAPA)
  - data path chip parity logic and output line (DAPH)
- Test branch on condition codes
  - jump MUX: all select lines pertaining to ALU condition codes (DAPC)
  - data path from data path chip to internal data (ID) bus (DAPH)
  - path from ALU.CC register to jump MUX (DAPE, DAPC)
  - data path chip Moveout microinstruction with destination = ALU.CC register (DAPH, DAPE)
  - ALU.CC register (read/write/hold) DAPE
  - ID bus path to ALU.CC register (DAPH, DAPE)
  - ID bus address decode (DAPK)
  - page register (DAPB)
  - microprogram counter (DAPB)
  - NμA MUX with select = not take branch (DAPB)

- Test case on index register
  - data path chip Moveout with destination = index register (DAPH, DAPE)
  - ID bus path to index register (DAPH, DAPE)
  - index register (read/write/hold) (DAPE)
  - path from index register to OR MUX (DAPE, DAPC)
  - OR MUX: select = index (DAPC)
  - conditional decrement: no decrement (DAPD)
- Test case on size register
  - data path chip Moveout with destination = size register (DAPH, DAPE)
  - ID bus path to size register (DAPH, DAPE)
  - size register (read/write/hold) (DAPE)
  - path from size register to OR MUX (DAPE, DAPC)
  - OR MUX: select = size (DAPC)
- JSB, Return, BSB
  - microstack (DAPD)
  - microstack pointer (DAPC)
  - OR MUX: select = 0010 (DAPC)

2. Test Most Data Path Chip Functions (DAPH)
LEDs set to 6 (on, on, off)

- Test all ALU opcodes
  - path of size register to data path chip size pins
  - all tested with size of 0, 1, 3
  - path of data path chip to ALU.CC through chip condition code pins
- Test barrel shifter with all MUX combinations and shift opcodes

- Miscellaneous instructions and functions
  - Multiply Step microinstruction
  - short operand register backup
  - Restore microinstruction
  - Clear Save Stack microinstruction
- Verify all ROM constants
- Write/read all states with standard 32-bit diagnostic patterns (shown in hex):
  - AAAAAAAA
  - 55555555
  - 33333333
  - 0000FFFF
  - 00FF00FF
  - 0F0F0F0F
  - FFFFFFFF
  - 00000000
- Internal timer interrupt expected in 10 milliseconds

3. **Test the Console**
   LEDs set to 6 (on, on, off)
   - Write to UART and verify loop back
     - console UART (DAPP)
     - UART buffer (DAPP)
     - interrupt control (DAPN)
     - jump MUX select = interrupt
   - Write "MICROVERIFY STARTED" on console if OK
4. **Simplest OR MUX Selects**
   LEDs set to 6 (on, on, off)
   - Select = zero
   - Select = OR2

5. **Write/Read Diagnostic Patterns Over ID Bus to External Registers**
   LEDs set to 6 (on, on, off)
6. **Basic MCT/DAP Interface**
   LEDs set to 4 (on, off, off)
   - Cache enable function
   - Mapping disable function
   - Memory control bus
   - Stall logic
7. **Test Memory Board and MD Bus Interface, But Not Q22 bus**
   LEDs set to 4 (on, off, off) or 5 (on, off, on) depending on subtest
   - Write/read standard diagnostic patterns through all MCT state
     - adder
     - merge register (MCTK)
     - rotator
     - all cache and translation buffer locations (MCTL, MCTN)
     - all registers in register file (MCTD, MCTU)
     - test MCA and MCD busses
     - test the MD bus and latches (DAPJ, MCTE)
8. **Test the MCT Powerup Sequence**
   LEDs set to 5 (on, off, on)
   - Insure that the cache locations all have the same low eight bits as the index used in a WRITE.CACHE Memory Request.
     - adder carry
     - MCT page incrementer
     - MCT branch MUX

9. **Test the DAP/MCT I-stream Interface**
   This includes I-stream Requests and reading from IB.BYTE.
   - Check expected I-stream bytes
     LEDs set to 4 (on, off, off)
     - prefetch FIFO (MCTP)
     - memory control bus
     - IBYTE register (DAPF)
   - Check data path chip actions performed
     LEDs set to 6 (on, on, off)
     - save old PC
     - increment current PC according to size or long operand
     - save long operand on Memory Request

10. **IB.INVALID Indicator to OR MUX**
    LEDs set to 4 (on, off, off)

    Select = ib.invalid

11. **Decode Instructions**
    LEDs set to 6 (on, on, off)
    - NμA bus from decode ROM (DAPF)
    - CC/DT functions from IRD (DAPE)

12. **Decode Dependent OR MUX Selects**
    LEDs set to 6 (on, on, off)
    - Select = branch.state
      - branch false
      - ib.invalid

- Select = decode.state
  - overflow and check
  - arithmetic trap request
  - interrupt request
  - T-bit
  - console halt
  - ib.invalid

13. **MCT Dependent OR MUX Selects (DAPC)**
    LEDs set to 4 (on, off, off)

- Select = memory.error
  - error summary
  - translation buffer miss
  - page crossing
  - modify refused

14. **Remaining Jump MUX Selects (DAPC)**
    LEDs set to 6 (on, on, off)

- Interrupt
- Stack register
- Register destination
- Console halt

15. **Trap from MCT**
    LEDs set to 6 (on, on, off)

    Conditional decrement (decrement = yes) (DAPD)

16. **Boot ROM (DAPF)**
    LEDs set to 6 (on, on, off)

    Read boot EPROM and calculate/verify checksum

17. **MCT Control and Status Registers**
LEDs set to 5 (on, off, on)

- Write, read and verify these states on the MCT board
  - map enable
  - ib error
  - cache enable

Memory Request Microinstructions for Microverify

Two special memory functions for the Memory Request microinstruction are defined solely for use in Microverify. They are VERIFY.ADDER and VERIFY.MCD. (The section titled "Memory Functions" in Chapter 5 of this manual contains descriptions of all the other memory functions.) Both VERIFY.ADDER and VERIFY.MCD are issued in major functional area 7, described above.

VERIFY.ADDER

This Memory Request microinstruction verifies the operation of the 9-bit adder in the memory controller, and the low eight bits of the MCA bus and the memory data bus. Bits <29:23> of the microinstruction have the hex value 10. This memory function causes an 8-bit data pattern to be passed to the adder and returned to the data path.

The data type field must specify word. The register specified by the long operand contains a 32-bit data pattern. The data pattern is sent to the memory controller, but the data path checks only the low-order eight bits that are returned.

The VERIFY.ADDER Memory Request is actually issued eight consecutive times with a different data pattern each time. The data patterns used are listed in the fifth bullet of major functional area 2, above.

VERIFY.MCD

This Memory Request microinstruction verifies the operation of the MCA bus, the reverse pass latch, the MCD bus, the byte rotator, and the merge register in the memory controller, as well as bits <31:08> of the memory data bus. Bits <29:23> of the microinstruction have the hex value 11.

The data type field must specify word. The register specified by the long operand contains a 32-bit data pattern.

This memory function first passes the low-order eight bits of the issued data pattern through the memory controller components and buses mentioned above, and checks that the low-order eight bits of the data pattern return to the data path unchanged. Next, all 32 bits of the data pattern are passed to these memory controller components.

The data patterns used for the VERIFY.ADDER Memory Request are also used for VERIFY.MCD. Thus, VERIFY.MCD is actually issued sixteen times. The eight data patterns are each used once, and the low-order eight bits checked. Then the eight data patterns are used again, and all 32 bits are checked.

Appendix E
MicroVAX Instruction Set

The MicroVAX instruction set as specified by the MicroVAX architecture is the same as the VAX instruction set, minus PDP-11 compatibility mode instructions. This appendix lists the MicroVAX instruction set in alphabetical order by instruction mnemonic.

The MicroVAX architecture is tailored to facilitate low-end implementations of the VAX family of computers. As such, it specifies a subset of instructions that must be implemented in hardware. Any machine using MicroVAX architecture must implement at least this subset of instructions in hardware, and may optionally implement more in hardware. All remaining instructions are emulated in software.

The MicroVAX I system, for example, implements the specified subset of instructions in hardware, plus these additional instructions: all F_floating point instructions, D_ or G_floating point instructions, CMPC3, LOCC, SCANC, SKPC, and SPANC. All remaining instructions (except PDP-11 compatibility mode) are emulated in software, or in software with a hardware assist.

There are five footnotes that annotate the instructions listed in this appendix. The set of all footnoted instructions are those instructions for which emulation support is specified by the MicroVAX architecture; these are the instructions that are not part of the subset of instructions that must be implemented in hardware.

The footnotes are as follows:

1. Those instructions marked with footnote 1 are the D_floating point instructions that are implemented in hardware in the MicroVAX I KD32-AB processor, and cause a reserved instruction fault in the KD32-AA processor. Any time a floating point instruction causes a reserved instruction fault, the operating system can emulate the instruction in software.
2. Those instructions marked with footnote 2 are implemented in hardware on both MicroVAX I processors, even though the MicroVAX architecture specifies emulation support for these instructions. Those instructions with footnote 2 are the difference between the MicroVAX instruction set as specified by the MicroVAX architecture and the implementation of this instruction set by the MicroVAX I .
3. Those instructions marked with footnote 3 are the G_floating point instructions that are implemented in hardware in the KD32-AA processor, and cause a reserved instruction fault in the KD32-AB processor. Again, the operating system can emulate any floating point instruction that causes a reserved instruction fault in software.

   Note: Those instructions with footnotes 1 or 3 are the difference between the instruction set as implemented by the MicroVAX I KD32-AA processor, and as implemented by the KD32-AB processor.
4. Those instructions marked with footnote 4 are the H_floating point instructions that cause reserved instruction faults in both MicroVAX I processors. As with the D_ and G_floating point instructions, the operating system can emulate H_floating instructions in software. All software supplied by

DIGITAL for the MicroVAX I system emulates D_, G_, and H_floating point instructions.

5. Those instructions marked with footnote 5 cause instruction emulation exceptions in both MicroVAX I processors. These instructions are then handled by software emulation with a hardware assist, for all software supplied by DIGITAL for the MicroVAX I.

Mnemonic	Opcode	Description
ACBB	9D	Add compare and branch byte
[1] ACBD	6F	Add compare and branch D_floating
[2] ACBF	4F	Add compare and branch F_floating
[3] ACBG	4FFD	Add compare and branch G_floating
[4] ACBH	6FFD	Add compare and branch H_floating
ACBL	F1	Add compare and branch longword
ACBW	3D	Add compare and branch word
ADAWI	58	Add aligned word interlocked
ADDB2	80	Add byte 2-operand
ADDB3	81	Add byte 3-operand
[1] ADDD2	60	Add D_floating 2-operand
[1] ADDD3	61	Add D_floating 3-operand
[2] ADDF2	40	Add F_floating 2-operand
[2] ADDF3	41	Add F_floating 3-operand

[1] This instruction generates a reserved instruction fault in the KD32-AA processor. The KD32-AB processor implements this instruction in hardware.

[2] The MicroVAX architecture specifies this as an emulated instruction, but both MicroVAX I processors implement it in hardware.

[3] This instruction generates a reserved instruction fault in the KD32-AB processor. The KD32-AA processor implements this instruction in hardware.

[4] This instruction generates a reserved instruction fault in both MicroVAX I processors.

Mnemonic	Opcode	Description
[3] ADDG2	40FD	Add G_floating 2-operand
[3] ADDG3	41FD	Add G_floating 3-operand
[4] ADDH2	60FD	Add H_floating 2-operand
[4] ADDH3	61FD	Add H_floating 3-operand
ADDL2	C0	Add longword 2-operand
ADDL3	C1	Add longword 3-operand
[5] ADDP4	20	Add packed 4-operand
[5] ADDP6	21	Add packed 6-operand
ADDW2	A0	Add word 2-operand
ADDW3	A1	Add word 3-operand
ADWC	D8	Add with carry
AOBLEQ	F3	Add one and branch on less or equal
AOBLSS	F2	Add one and branch on less
ASHL	78	Arithmetic shift longword
[5] ASHP	F8	Arithmetic shift and round packed
ASHQ	79	Arithmetic shift quad
BBC	E1	Branch on bit clear
BBCC	E5	Branch on bit clear and clear
BBCCI	E7	Branch on bit clear and clear interlocked

[3] This instruction generates a reserved instruction fault in the KD32-AB processor. The KD32-AA processor implements this instruction in hardware.

[4] This instruction generates a reserved instruction fault in both MicroVAX I processors.

[5] This instruction generates an instruction emulation exception in both MicroVAX I processors.

Mnemonic	Opcode	Description
BBCS	E3	Branch on bit clear and set
BBS	E0	Branch on bit set
BBSC	E4	Branch on bit set and clear
BBSS	E2	Branch on bit set and set
BBSSI	E6	Branch on bit set and set interlocked
BCC	1E	Branch on carry clear
BCS	1F	Branch on carry set
BEQL	13	Branch on equal
BEQLU	13	Branch on equal unsigned
BGEQ	18	Branch on greater or equal
BGEQU	1E	Branch on greater or equal unsigned
BGTR	14	Branch on greater
BGTRU	1A	Branch on greater unsigned
BICB2	8A	Bit clear byte 2-operand
BICB3	8B	Bit clear byte 3-operand
BICL2	CA	Bit clear longword 2-operand
BICL3	CB	Bit clear longword 3-operand
BICPSW	B9	Bit clear processor status word
BICW2	AA	Bit clear word 2-operand
BICW3	AB	Bit clear word 3-operand
BISB2	88	Bit set byte 2-operand
BISB3	89	Bit set byte 3-operand
BISL2	C8	Bit set longword 2-operand
BISL3	C9	Bit set longword 3-operand
BISPSW	B8	Bit set processor status word
BISW2	A8	Bit set word 2-operand
BISW3	A9	Bit set word 3-operand

Mnemonic	Opcode	Description
BITB	93	Bit test byte
BITL	D3	Bit test longword
BITW	B3	Bit test word
BLBC	E9	Branch on low bit clear
BLBS	E8	Branch on low bit set
BLEQ	15	Branch on less or equal
BLEQU	1B	Branch on less or equal unsigned
BLSS	19	Branch on less
BLSSU	1F	Branch on less unsigned
BNEQ	12	Branch on not equal
BNEQU	12	Branch on not equal unsigned
BPT	03	Break point fault
BRB	11	Branch with byte displacement
BRW	31	Branch with word displacement
BSBB	10	Branch to subroutine with byte displacement
BSBW	30	Branch to subroutine with word displacement
BUGL	FDFF	VMS bugcheck
BUGW	FEFF	VMS bugcheck
BVC	1C	Branch on overflow clear
BVS	1D	Branch on overflow set
CALLG	FA	Call with general argument list
CALLS	FB	Call with argument list on stack
CASEB	8F	Case byte
CASEL	CF	Case longword

Mnemonic	Opcode	Description
CASEW	AF	Case word
CHME	BD	Change mode to executive
CHMK	BC	Change mode to kernel
CHMS	BE	Change mode to supervisor
CHMU	BF	Change mode to user
CLRB	94	Clear byte
[1] CLRD	7C	Clear D_floating
[2] CLRF	D4	Clear F_floating
[3] CLRG	7C	Clear G_floating
[4] CLRH	7CFD	Clear H_floating
CLRL	D4	Clear longword
[5] CLRO	7CFD	Clear octaword
CLRQ	7C	Clear quad
CLRW	B4	Clear word
CMPB	91	Compare byte
[2] CMPC3	29	Compare character 3-operand
[5] CMPC5	2D	Compare character 5-operand

[1] This instruction generates a reserved instruction fault in the KD32-AA processor. The KD32-AB processor implements this instruction in hardware.

[2] The MicroVAX architecture specifies this as an emulated instruction, but both MicroVAX I processors implement it in hardware.

[3] This instruction generates a reserved instruction fault in the KD32-AB processor. The KD32-AA processor implements this instruction in hardware.

[4] This instruction generates a reserved instruction fault in both MicroVAX I processors.

[5] This instruction generates an instruction emulation exception in both MicroVAX I processors.

Mnemonic	Opcode	Description
[1] CMPD	71	Compare D_floating
[2] CMPF	51	Compare F_floating
[3] CMPG	51FD	Compare G_floating
[4] CMPH	71FD	Compare H_floating
CMPL	D1	Compare longword
[5] CMPP3	35	Compare packed 3-operand
[5] CMPP4	37	Compare packed 4-operand
CMPV	EC	Compare field
CMPW	B1	Compare word
CMPZV	ED	Compare zero-extended field
[5] CRC	0B	Calculate cyclic redundancy check
[1] CVTBD	6C	Convert byte to D_floating
[2] CVTBF	4C	Convert byte to F_floating
[3] CVTBG	4CFD	Convert byte to G_floating
[4] CVTBH	6CFD	Convert byte to H_floating
CVTBL	98	Convert byte to longword

[1] This instruction generates a reserved instruction fault in the KD32-AA processor. The KD32-AB processor implements this instruction in hardware.

[2] The MicroVAX architecture specifies this as an emulated instruction, but both MicroVAX I processors implement it in hardware.

[3] This instruction generates a reserved instruction fault in the KD32-AB processor. The KD32-AA processor implements this instruction in hardware.

[4] This instruction generates a reserved instruction fault in both MicroVAX I processors.

[5] This instruction generates an instruction emulation exception in both MicroVAX I processors.

Mnemonic	Opcode	Description
CVTBW	99	Convert byte to word
[1] CVTDB	68	Convert D_floating to byte
[1] CVTDF	76	Convert D_floating to F_floating
[4] CVTDH	32FD	Convert D_floating to H_floating
[1] CVTDL	6A	Convert D_floating to longword
[1] CVTDW	69	Convert D_floating to word
[2] CVTFB	48	Convert F_floating to byte
[1] CVTFD	56	Convert F_floating to D_floating
[3] CVTFG	99FD	Convert F_floating to G_floating
[4] CVTFH	98FD	Convert F_floating to H_floating
[2] CVTFL	4A	Convert F_floating to longword
[2] CVTFW	49	Convert F_floating to word
[3] CVTGB	48FD	Convert G_floating to byte

[1] This instruction generates a reserved instruction fault in the KD32-AA processor. The KD32-AB processor implements this instruction in hardware.

[2] The MicroVAX architecture specifies this as an emulated instruction, but both MicroVAX I processors implement it in hardware.

[3] This instruction generates a reserved instruction fault in the KD32-AB processor. The KD32-AA processor implements this instruction in hardware.

[4] This instruction generates a reserved instruction fault in both MicroVAX I processors.

Mnemonic	Opcode	Description
[3] CVTGF	33FD	Convert G_floating to F_floating
[4] CVTGH	56FD	Convert G_floating to H_floating
[3] CVTGL	4AFD	Convert G_floating to longword
[3] CVTGW	49FD	Convert G_floating to word
[4] CVTHB	68FD	Convert H_floating to byte
[4] CVTHD	F7FD	Convert H_floating to D_floating
[4] CVTHF	F6FD	Convert H_floating to F_floating
[4] CVTHG	76FD	Convert H_floating to G_floating
[4] CVTHL	6AFD	Convert H_floating to longword
[4] CVTHW	69FD	Convert H_floating to word
CVTLB	F6	Convert longword to byte
[1] CVTLD	6E	Convert longword to D_floating
[2] CVTLF	4E	Convert longword to F_floating
[3] CVTLG	4EFD	Convert longword to G_floating

[1] This instruction generates a reserved instruction fault in the KD32-AA processor. The KD32-AB processor implements this instruction in hardware.

[2] The MicroVAX architecture specifies this as an emulated instruction, but both MicroVAX I processors implement it in hardware.

[3] This instruction generates a reserved instruction fault in the KD32-AB processor. The KD32-AA processor implements this instruction in hardware.

[4] This instruction generates a reserved instruction fault in both MicroVAX I processors.

Mnemonic	Opcode	Description
[4] CVTLH	6EFD	Convert longword to H_floating
[5] CVTLP	F9	Convert longword to packed
CVTLW	F7	Convert longword to word
[5] CVTPL	36	Convert packed to longword
[5] CVTPS	08	Convert packed to leading separate
[5] CVTPT	24	Convert packed to trailing
[1] CVTRDL	6B	Convert rounded D_floating to longword
[2] CVTRFL	4B	Convert rounded F_floating to longword
[3] CVTRGL	4BFD	Convert rounded G_floating to longword
[4] CVTRHL	6BFD	Convert rounded H_floating to longword
[5] CVTSP	09	Convert leading separate to packed

[1] This instruction generates a reserved instruction fault in the KD32-AA processor. The KD32-AB processor implements this instruction in hardware.

[2] The MicroVAX architecture specifies this as an emulated instruction, but both MicroVAX I processors implement it in hardware.

[3] This instruction generates a reserved instruction fault in the KD32-AB processor. The KD32-AA processor implements this instruction in hardware.

[4] This instruction generates a reserved instruction fault in both MicroVAX I processors.

[5] This instruction generates an instruction emulation exception in both MicroVAX I processors.

Mnemonic	Opcode	Description
[5] CVTTP	26	Convert trailing to packed
CVTWB	33	Convert word to byte
[1] CVTWD	6D	Convert word to D_floating
[2] CVTWF	4D	Convert word to F_floating
[3] CVTWG	4DFD	Convert word to G_floating
[4] CVTWH	6DFD	Convert word to H_floating
CVTWL	32	Convert word to longword
DECB	97	Decrement byte
DECL	D7	Decrement longword
DECW	B7	Decrement word
DIVB2	86	Divide byte 2-operand
DIVB3	87	Divide byte 3-operand
[1] DIVD2	66	Divide D_floating 2-operand
[1] DIVD3	67	Divide D_floating 3-operand
[2] DIVF2	46	Divide F_floating 2-operand
[2] DIVF3	47	Divide F_floating 3-operand
[3] DIVG2	46FD	Divide G_floating 2-operand

[1] This instruction generates a reserved instruction fault in the KD32-AA processor. The KD32-AB processor implements this instruction in hardware.

[2] The MicroVAX architecture specifies this as an emulated instruction, but both MicroVAX I processors implement it in hardware.

[3] This instruction generates a reserved instruction fault in the KD32-AB processor. The KD32-AA processor implements this instruction in hardware.

[4] This instruction generates a reserved instruction fault in both MicroVAX I processors.

[5] This instruction generates an instruction emulation exception in both MicroVAX I processors.

Mnemonic	Opcode	Description
[3] DIVG3	47FD	Divide G_floating 3-operand
[4] DIVH2	66FD	Divide H_floating 2-operand
[4] DIVH3	67FD	Divide H_floating 3-operand
DIVL2	C6	Divide longword 2-operand
DIVL3	C7	Divide longword 3-operand
[5] DIVP	27	Divide packed
DIVW2	A6	Divide word 2-operand
DIVW3	A7	Divide word 3-operand
[5] EDITPC	38	Edit packed to character string
EDIV	7B	Extended divide
[1] EMODD	74	Extended modulus D_floating
[2] EMODF	54	Extended modulus F_floating
[3] EMODG	54FD	Extended modulus G_floating
[4] EMODH	74FD	Extended modulus H_floating
EMUL	7A	Extended multiply
ESCD	FD	Escape D
ESCE	FE	Escape E

[1] This instruction generates a reserved instruction fault in the KD32-AA processor. The KD32-AB processor implements this instruction in hardware.

[2] The MicroVAX architecture specifies this as an emulated instruction, but both MicroVAX I processors implement it in hardware.

[3] This instruction generates a reserved instruction fault in the KD32-AB processor. The KD32-AA processor implements this instruction in hardware.

[4] This instruction generates a reserved instruction fault in both MicroVAX I processors.

[5] This instruction generates an instruction emulation exception in both MicroVAX I processors.

Mnemonic	Opcode	Description
ESCF	FF	Escape F
EXTV	EE	Extract field
EXTZV	EF	Extract zero-extended field
FFC	EB	Find first clear bit
FFS	EA	Find first set bit
HALT	00	Halt (kernel mode only)
INCB	96	Increment byte
INCL	D6	Increment longword
INCW	B6	Increment word
INDEX	0A	Index calculation
INSQHI	5C	Insert at head of queue, interlocked
INSQTI	5D	Insert at tail of queue, interlocked
INSQUE	0E	Insert into queue
INSV	F0	Insert field
JMP	17	Jump
JSB	16	Jump to subroutine
LDPCTX	06	Load process context (only legal on interrupt stack)
[2] LOCC	3A	Locate character
[5] MATCHC	39	Match characters
MCOMB	92	Move complemented byte
MCOML	D2	Move complemented longword

[2] The MicroVAX architecture specifies this as an emulated instruction, but both MicroVAX I processors implement it in hardware.

[5] This instruction generates an instruction emulation exception in both MicroVAX I processors.

Mnemonic	Opcode	Description
MCOMW	B2	Move complemented word
MFPR	DB	Move from processor register (kernel mode only)
MNEGB	8E	Move negated byte
[1] MNEGD	72	Move negated D_floating
[2] MNEGF	52	Move negated F_floating
[3] MNEGG	52FD	Move negated G_floating
[4] MNEGH	72FD	Move negated H_floating
MNEGL	CE	Move negated longword
MNEGW	AE	Move negated word
MOVAB	9E	Move address of byte
[1] MOVAD	7E	Move address of D_floating
[2] MOVAF	DE	Move address of F_floating
[3] MOVAG	7E	Move address of G_floating
[4] MOVAH	7EFD	Move address of H_floating
MOVAL	DE	Move address of longword
[5] MOVAO	7EFD	Move address of octaword

[1] This instruction generates a reserved instruction fault in the KD32-AA processor. The KD32-AB processor implements this instruction in hardware.

[2] The MicroVAX architecture specifies this as an emulated instruction, but both MicroVAX I processors implement it in hardware.

[3] This instruction generates a reserved instruction fault in the KD32-AB processor. The KD32-AA processor implements this instruction in hardware.

[4] This instruction generates a reserved instruction fault in both MicroVAX I processors.

[5] This instruction generates an instruction emulation exception in both MicroVAX I processors.

Mnemonic	Opcode	Description
MOVAQ	7E	Move address of quadword
MOVAW	3E	Move address of word
MOVB	90	Move byte
MOVC3	28	Move character 3-operand
MOVC5	2C	Move character 5-operand
[1] MOVD	70	Move D_floating
[2] MOVF	50	Move F_floating
[3] MOVG	50FD	Move G_floating
[4] MOVH	70FD	Move H_floating
MOVL	D0	Move longword
[5] MOVO	7DFD	Move octaword
[5] MOVP	34	Move packed
MOVPSL	DC	Move processor status longword
MOVQ	7D	Move quadword
[5] MOVTC	2E	Move translated characters

[1] This instruction generates a reserved instruction fault in the KD32-AA processor. The KD32-AB processor implements this instruction in hardware.

[2] The MicroVAX architecture specifies this as an emulated instruction, but both MicroVAX I processors implement it in hardware.

[3] This instruction generates a reserved instruction fault in the KD32-AB processor. The KD32-AA processor implements this instruction in hardware.

[4] This instruction generates a reserved instruction fault in both MicroVAX I processors.

[5] This instruction generates an instruction emulation exception in both MicroVAX I processors.

Mnemonic	Opcode	Description
[5] MOVTUC	2F	Move translated until character
MOVW	B0	Move word
MOVZBL	9A	Move zero-extended byte to longword
MOVZBW	9B	Move zero-extended byte to word
MOVZWL	3C	Move zero-extended word to longword
MTPR	DA	Move to processor register (kernel mode only)
MULB2	84	Multiply byte 2-operand
MULB3	85	Multiply byte 3-operand
[1] MULD2	64	Multiply D_floating 2-operand
[1] MULD3	65	Multiply D_floating 3-operand
[2] MULF2	44	Multiply F_floating 2-operand
[2] MULF3	45	Multiply F_floating 3-operand
[3] MULG2	44FD	Multiply G_floating 2-operand
[3] MULG3	45FD	Multiply G_floating 3-operand

[1] This instruction generates a reserved instruction fault in the KD32-AA processor. The KD32-AB processor implements this instruction in hardware.

[2] The MicroVAX architecture specifies this as an emulated instruction, but both MicroVAX I processors implement it in hardware.

[3] This instruction generates a reserved instruction fault in the KD32-AB processor. The KD32-AA processor implements this instruction in hardware.

[5] This instruction generates an instruction emulation exception in both MicroVAX I processors.

Mnemonic	Opcode	Description
[4] MULH2	64FD	Multiply H_floating 2-operand
[4] MULH3	65FD	Multiply H_floating 3-operand
MULL2	C4	Multiply longword 2-operand
MULL3	C5	Multiply longword 3-operand
[5] MULP	25	Multiply packed
MULW2	A4	Multiply word 2-operand
MULW3	A5	Multiply word 3-operand
NOP	01	No operation
[1] POLYD	75	Evaluate polynomial D_floating
[2] POLYF	55	Evaluate polynomial F_floating
[3] POLYG	55FD	Evaluate polynomial G_floating
[4] POLYH	75FD	Evaluate polynomial H_floating
POPR	BA	Pop registers

[1] This instruction generates a reserved instruction fault in the KD32-AA processor. The KD32-AB processor implements this instruction in hardware.

[2] The MicroVAX architecture specifies this as an emulated instruction, but both MicroVAX I processors implement it in hardware.

[3] This instruction generates a reserved instruction fault in the KD32-AB processor. The KD32-AA processor implements this instruction in hardware.

[4] This instruction generates a reserved instruction fault in both MicroVAX I processors.

[5] This instruction generates an instruction emulation exception in both MicroVAX I processors.

Mnemonic	Opcode	Description
PROBER	0C	Probe read access
PROBEW	0D	Probe write access
PUSHAB	9F	Push address of byte
[1] PUSHAD	7F	Push address of D_floating
[2] PUSHAF	DF	Push address of F_floating
[3] PUSHAG	7F	Push address of G_floating
[4] PUSHAH	7FFD	Push address of H_floating
PUSHAL	DF	Push address of longword
[5] PUSHAO	7FFD	Push address of octaword
PUSHAQ	7F	Push address of quadword
PUSHAW	3F	Push address of word
PUSHL	DD	Push longword
PUSHR	BB	Push registers
REI	02	Return from exception or interrupt
REMQHI	5E	Remove from head of queue, interlocked

[1] This instruction generates a reserved instruction fault in the KD32-AA processor. The KD32-AB processor implements this instruction in hardware.

[2] The MicroVAX architecture specifies this as an emulated instruction, but both MicroVAX I processors implement it in hardware.

[3] This instruction generates a reserved instruction fault in the KD32-AB processor. The KD32-AA processor implements this instruction in hardware.

[4] This instruction generates a reserved instruction fault in both MicroVAX I processors.

[5] This instruction generates an instruction emulation exception in both MicroVAX I processors.

Mnemonic	Opcode	Description
REMQTI	5F	Remove from tail of queue, interlocked
REMQUE	0F	Remove from queue
RET	04	Return from procedure
ROTL	9C	Rotate longword
RSB	05	Return from subroutine
Reserved	57	Reserved
Reserved	5A	Reserved
Reserved	5B	Reserved
Reserved	77	Reserved
Reserved	FE	Reserved
Reserved	FF	Reserved
SBWC	D9	Subtract with carry
[2] SCANC	2A	Scan for character
[2] SKPC	3B	Skip character
SOBGEQ	F4	Subtract one and branch on greater or equal
SOBGTR	F5	Subtract one and branch on greater
[2] SPANC	2B	Span characters
SUBB2	82	Subtract byte 2-operand
SUBB3	83	Subtract byte 3-operand
[1] SUBD2	62	Subtract D_floating 2-operand
[1] SUBD3	63	Subtract D_floating 3-operand

[1] This instruction generates a reserved instruction fault in the KD32-AA processor. The KD32-AB processor implements this instruction in hardware.

[2] The MicroVAX architecture specifies this as an emulated instruction, but both MicroVAX I processors implement it in hardware.

Mnemonic	Opcode	Description
[2] SUBF2	42	Subtract F_floating 2-operand
[2] SUBF3	43	Subtract F_floating 3-operand
[3] SUBG2	42FD	Subtract G_floating 2-operand
[3] SUBG3	43FD	Subtract G_floating 3-operand
[4] SUBH2	62FD	Subtract H_floating 2-operand
[4] SUBH3	63FD	Subtract H_floating 3-operand
SUBL2	C2	Subtract longword 2-operand
SUBL3	C3	Subtract longword 3-operand
[5] SUBP4	22	Subtract packed 4-operand
[5] SUBP6	23	Subtract packed 6-operand
SUBW2	A2	Subtract word 2-operand
SUBW3	A3	Subtract word 3-operand
SVPCTX	07	Save process context (kernel mode only)
TSTB	95	Test byte
[1] TSTD	73	Test D_floating
[2] TSTF	53	Test F_floating

[1] This instruction generates a reserved instruction fault in the KD32-AA processor. The KD32-AB processor implements this instruction in hardware.

[2] The MicroVAX architecture specifies this as an emulated instruction, but both MicroVAX I processors implement it in hardware.

[3] This instruction generates a reserved instruction fault in the KD32-AB processor. The KD32-AA processor implements this instruction in hardware.

[4] This instruction generates a reserved instruction fault in both MicroVAX I processors.

[5] This instruction generates an instruction emulation exception in both MicroVAX I processors.

Mnemonic	Opcode	Description
[3] TSTG	53FD	Test G_floating
[4] TSTH	73FD	Test H_floating
TSTL	D5	Test longword
TSTW	B5	Test word
XFC	FC	Extended function call
XORB2	8C	Exclusive OR byte 2-operand
XORB3	8D	Exclusive OR byte 3-operand
XORL2	CC	Exclusive OR longword 2-operand
XORL3	CD	Exclusive OR longword 3-operand
XORW2	AC	Exclusive OR word 2-operand
XORW3	AD	Exclusive OR word 3-operand

[3] This instruction generates a reserved instruction fault in the KD32-AB processor. The KD32-AA processor implements this instruction in hardware.

[4] This instruction generates a reserved instruction fault in both MicroVAX I processors.

Glossary

abort An exception that occurs in the middle of an instruction and potentially leaves the registers and memory in an indeterminate state, such that the instruction cannot necessarily be restarted.

access mode Any of the four processor access modes in which software executes. Processor access modes are, in order from most to least privileged and protected: kernel, executive, supervisor, and user. When the processor is in kernel mode, the executing software has complete control of, and responsibility for, the system. In any other mode, the processor is inhibited from executing privileged instructions. The Processor Status Longword contains the current access mode field. The operating system uses access modes to define protection levels for software executing in the context of a process. For example, the executive runs in kernel and executive mode and is most protected. The command interpreter is less protected and runs in supervisor mode. The debugger runs in user mode and is not more protected than normal users programs.

access type The way in which the processor accesses instruction operands. Access types are: read, write, modify, address, and branch.

access violation An attempt to reference an address that is not mapped into virtual memory or an attempt to reference an address that is not accessible by the current access mode.

ALU Arithmetic and logic unit. A device that performs the basic mathematical and logical operations in a processor.

AND A logic operation with the property that if P and Q are elements, then the AND of P and Q is true if both P and Q are true, and false if P or Q is false.

array An arrangement of elements in one or more dimensions.

ASCII American National Standard Code for Information Interchange. A set of 8-bit binary numbers representing the alphabet, punctuation, numerals, control, and other special symbols, used in text representation and communications protocol.

asynchronous Lacking a regular time relationship; hence, as applied to computer program execution, unexpected or unpredictable with respect to the instruction sequence.

backplane The physical mounting blocks into which modules are inserted; bus signals are connected on the reverse side by wire or etch.

block mode A type of data transfer implemented by some devices in which data can be read or written in blocks of one to sixteen words within one bus cycle; that is, the bus master does not need to arbitrate for control of the bus between word transfers.

boot To boot a computer system is to get it initialized and loaded with a system image so that it is ready to execute user programs. Typically, the computer does most of this by itself, using a bootstrap, so that the operator only has to turn the system power on, or press a button, or enter a command to get the computer started.

bootstrap 1. A set of instructions that cause additional instructions to be loaded until the complete program is in memory. 2. A technique or device designed to bring itself into a desired state by means of its own action; for example, a machine routine whose first few instructions are

sufficient to bring the rest of the routine into memory from an input device such as a disk.

bootstrap block A block in the index file on a system disk that contains a program that can load the operating system into memory and start its execution.

buffer 1. A routine or storage used to compensate for a difference in rate of flow of data, or time of occurrence of events, when transferring data from one device to another. 2. An isolating circuit used to prevent a driven circuit from influencing the driving circuit.

bus One or more conductors used for transmitting signals or power. See control bus and data bus.

byte A byte is eight contiguous bits starting on an addressable byte boundary. Bits are numbered from right to left, 0 through 7. Bit 0 is the low-order bit.

cache Special memory internal to the processor that stores data the processor may need in the immediate future.

central processing unit The functional unit in a computer system that interprets and executes instructions. Also referred to as the processor.

comparator A circuit which compares two signals and supplies an indication of agreement or disagreement.

condition codes Four bits in the processor status word that indicate the results of the previously executed instruction.

console mode Also called console I/O mode. When the system is in this mode, a user can enter commands from the console terminal to start and stop the system, monitor system operation, and run diagnostics.

console terminal A hard copy or video terminal that an operator uses to communicate with and control the system's processor.

control bus A bus carrying the signals that regulate system operations.

controller A functional unit that controls one or more units of peripheral equipment.

CPU See central processing unit.

daisy-chain 1. A bus signal that is broken at each module slot. The signal may be terminated at the slot, or passed along by the insertion of a module with the appropriate circuitry. 2. A signal that is broken by pins on a module. The signal terminates at the first pin, or is passed along by jumpers connecting the pins sequentially.

data bus A bus used to communicate data internally and externally to and from a central processing unit, memory, and peripheral devices.

data type In general, the way in which bits are grouped and interpreted. In reference to the processor instructions, the data type of an operand identifies the size of the operand and the significance of the bits in the operand. Operand data types include: byte, word, longword and quadword integer, floating and double floating, character string, packed decimal string, and variable-length bit field.

default An implicit value or option that is assumed by the system when no value or option is explicitly stated.

device The general name for any physical terminus or link connected to the processor that is capable of receiving, storing, or transmitting data.

diagnostic A program that tests logic and reports any faults it detects.

DIP Dual in-line package. A type of housing, generally molded plastic, for integrated circuits.

direct-mapped cache A cache organization in which only one address comparison is needed to locate any data in the

cache because any block of main memory data can be placed in only one possible position in the cache.

direct memory access (DMA) A method of directly accessing main memory for data transfer without involving the CPU. For example, a disk with its own controller can write data directly into memory, bypassing the processor.

displacement indexed mode An indexed addressing mode in which the base operand specifier uses displacement mode addressing.

displacement mode In displacement mode addressing, the specifier extension is a byte, word, or longword displacement. The displacement is sign-extended to 32 bits and added to a base address obtained from the specified register. The result is the address of the actual operand. If the PC is used as the register, the updated contents of the PC are used as the base address. The base address is the address of the first byte beyond the specifier extension.

DMA See direct memory access.

down-line load A communications procedure between a host computer and a target node where a program image is transferred over a communications link from the host to the node. Typically, this is done to bootstrap the target node when, for example, the node does not have mass storage media such as disks to store the image.

EIA interface A set of signal properties (time duration, voltage, and current) specified by the Electronic Industries Association for machine and data set connections.

EMI Abbreviation for electromagnetic interference. Electromagnetic phenomena which can adversely affect system performance.

EPROM An erasable PROM.

Ethernet A local area network that provides a communication facility for high speed data exchange among

computers and other digital devices located within a moderate-sized geographic area.

exception An event detected by the hardware (other than an interrupt or jump, branch, case, or call instruction) that changes the normal flow of instruction execution. An exception is always caused by the execution of an instruction or set of instructions. (In contrast, an interrupt is caused by an activity in the system independent of the current instruction.) There are three type of hardware exceptions: traps, faults, and aborts. Examples are: attempts to execute a privileged or reserved instruction, trace faults, breakpoint instruction execution, and arithmetic faults such as floating point overflow, underflow, and divide by zero.

extended LSI-11 bus See Q22 bus.

fault A hardware exception condition that occurs in the middle of an instruction and that leaves the registers and memory in a consistent state, such that eliminating the fault and restarting the instruction gives correct results.

FIFO First-in-first-out. A queuing technique in which the next item to be retrieved is the item that has been in the queue for the longest time.

Files-11 The name of the on-disk structure used by the RSX-11, IAS and VAX/VMS operating systems. Volumes created under this structure are transportable between these operating systems.

finger The point of contact between a signal on a module or a cable and the same signal on a backplane; also called a pin contact, connector, or connection point.

flag 1. Any of various types of indicators used for identification. 2. A character that signals the occurrence of some condition, such as the end of a word. 3. Synonym for switch indicator.

floating-point A form of number representation in which quantities are expressed in terms of a bounded number (mantissa) and a scale factor (characteristic or exponent) consisting of a power of the number base. For example, $127.6 = 0.1276 \times 10^3$ where the bounds are 0 and 1.

flip-flop A circuit or device containing active elements, capable of assuming either one of two stable states at a given time.

giga A metric term used to represent the number 1 followed by nine zeros.

halt To stop. Specifically, when the processor halts, it has stopped executing macroinstructions.

Hz Abbreviation for hertz. A unit of frequency equal to one cycle per second.

indexed addressing mode In indexed mode addressing, two registers are used to determine the actual instruction operand: an index register and a base operand specifier. The contents of the index register are used as an index (offset) into a table or array. The base operand specifier supplies the base address of the array (called the base operand address or BOA). The address of the actual operand is calculated by multiplying the contents of the index register by the size (in bytes) of the actual operand and adding the result to the base operand address.

internal processor register A part of the processor used by the operating system software to control the execution states of the computer system. They include the system base and length registers, the program and control region base and length registers, the system control block base register, the software interrupt request register, and many more.

interrupt An event other than an exception or branch, jump, case, or call instruction that changes the normal flow

of instruction execution. Interrupts are generally external to the process executing when the interrupt occurs.

interrupt priority level (IPL) The interrupt level at which the processor executes when an interrupt is generated. There are 31 possible interrupt priority levels. IPL 1 is lowest, 31 highest. The levels arbitrate contention for processor service. For example, a device cannot interrupt the processor if the processor is currently executing at an interrupt priority level greater than the interrupt priority level of the device's interrupt service routine.

interrupt vector See vector.

I/O Input-output.

I/O space The region of physical address space that contains the configuration registers, and device control/status and data registers.

IPR See internal processor register.

jumper A short length of wire used to complete a circuit temporarily or to bypass a circuit.

K When referring to storage capacity, two to the tenth power, or 1024 (decimal).

kernel mode The most privileged processor access mode (mode 0). The operating system's most privileged services, such as I/O drivers and the pager, run in kernel mode.

latch A simple logic storage element.

leading edge That transition of a pulse which occurs first.

LED Light emitting diode. An LED is a pn junction semiconductor device designed to emit light when forward biased.

LIFO Last-in-first-out. A queuing technique in which the next item to be retrieved is the item that has been in the queue for the shortest time.

logical block number (LBN) A number used to identify a block on a mass storage device. The number is a volume-relative address rather than its physical (device-oriented) address or its virtual (file-relative) address. The blocks that constitute the volume are labeled sequentially starting with logical block 0.

longword Four contiguous bytes (32 bits) starting on an addressable byte boundary. Bits are numbered from right to left, 0 through 31. The address of the longword is the address of the byte containing bit 0.

machine check A hardware error detected by the processor and reported to the operating system software.

macroinstruction An instruction of a macroprogram.

macroprogram A computer program consisting of a series of instructions written in a source language.

mass storage control protocol (MSCP) MSCP is the protocol used by a family of mass storage controllers and devices designed and built by Digital. In a system that uses an MSCP storage subsystem, the controller contains intelligence to perform the detailed I/O handling tasks. This arrangement allows the host to simply send requests for reads or writes to the controller and receive response messages back. The host does not concern itself with details such as device type, media geometry, media format, and error recovery.

master The device, module or option that is currently controlling bus transactions. There can only be one bus master at a time.

mega (M) One million.

memory management The system functions that include the hardware's page mapping and protection, and the operating system's image activator and pager.

microcode A sequence of elementary instructions that correspond to a specific computer operation. The microcode is stored in special memory internal to the processor. Execution of the microcode is initiated by the introduction of a computer instruction into an instruction register of the computer.

microinstruction An instruction of microcode.

microprogram See microcode.

microsecond One millionth of a second. Abbreviated μs.

millisecond One thousandth of a second. Abbreviated ms.

mnemonic A symbol or abbreviation chosen to assist the human memory.

module A board, made of plastic covered with an electrical conductor, on which logic devices such as transistors, resistors, and memory chips, are mounted, and circuits connecting these devices are etched.

MSCP See mass storage control protocol.

multiplexer (MUX) A device that can select one of a number of inputs and pass the logic level of that input on as the output.

nanosecond One billionth of a second. Abbreviated ns.

nominal voltage The voltage of a fully charged storage cell when delivering rated current.

octaword Sixteen contiguous bytes (128 bits) starting on an addressable byte boundary. Bits are number from right to left, 0 to 127. An octaword is identified by the address of the byte containing bit 0.

offset A fixed displacement from the beginning of a data structure. System offsets for items within a data structure normally have an associated symbolic name used instead of the numeric displacement. Where symbols are defined, programmers always reference the symbolic names for

items in a data structure instead of using the numeric displacement.

opcode The pattern of bits within an instruction that specify the operation to be performed.

operand specifier The pattern of bits in an instruction that indicate the addressing mode, a register and/or displacement, which, taken together, identify an instruction operand.

OR A logic operation with the property that if P and Q are elements, then the OR of P and Q is true if either P or Q is true, and false if P and Q are both false.

overcurrent/overvoltage protection A device which automatically disconnects the circuit whenever the current or voltage becomes excessive.

page A set of 512 contiguous byte locations used as the unit of memory mapping and protection.

page frame number (PFN) The address of the first byte of a page in physical memory. The high-order 21 bits of the physical address of the base of a page.

page table entry (PTE) The data structure that identifies the location and status of a page of virtual address space. When a virtual page is in memory, the PTE contains the page frame number needed to map the virtual page to a physical page. When it is not in memory, the page table entry contains the information needed to locate the page on secondary storage (disk).

PAL Programmable array logic. A general purpose logic structure consisting of an array of logic circuits. The way in which these circuits are programmed determines how input signals are processed. Programming is done on a custom basis at the factory and permanently establishes the functional operation of the PAL.

parity check A check that tests whether the number of ones (or zeros) in an array of bits is odd or even.

PC See program counter.

pipeline 1. A method of overlapping instructions such that the decode phase of one instruction is executed at the same time as the execute phase of the previous instruction. 2. When a signal is pipelined, it is passed through a flip-flop to delay it one cycle.

processor See central processing unit.

processor status longword (PSL) A system programmed processor register consisting of a word of privileged processor status and the processor status word (PSW). The privileged processor status information includes: the current IPL (interrupt priority level), the previous access mode, the current access mode, the interrupt stack bit, and the trace fault pending bit. The PSW includes: the condition codes (carry, overflow, zero, negative), the arithmetic trap enable bits (integer overflow, decimal overflow, floating underflow), and the trace enable bit.

program counter (PC) General register 15 (R15). At the beginning of an instruction's execution, the PC normally contains the address of a location in memory from which the processor will fetch the next instruction it will execute.

PROM Programmable read-only memory. An integrated circuit memory array that is manufactured with a pattern of either all logical zeros or ones and has a specific pattern wirtten into it by the user. Once the pattern is written, the PROM contents can only be read; writing is locked out.

protocol A set of rules that govern the operation of functional units to achieve communication.

PSL See processor status longword.

Q22 bus Any of a group of backplane and cable systems that implement the QBUS signals and are capable of handling 22 lines for address delimitation.

RAM Random-access memory. A storage device that provides direct access to data.

register A storage location in hardware logic other than main memory.

register deferred mode In register deferred mode addressing, the contents of the specified register are used as the address of the actual instruction operand.

register mode In register mode addressing, the contents of the specified register are used as the actual instruction operand.

sign-extension An operation where less than 32 bits of data are expanded to a longword by copying the state of the high-order bit of the data into the high-order bytes of the longword. For example, the word E030 (hex) becomes the longword FFFFE030 when sign-extended.

slave A bus device that can be addressed by, and participate in, bus transactions with a bus master. It has the subordinate role in a data transfer.

slot One of several locations into which a modular interface with the bus may be physically inserted.

stack An area of memory reserved for storing working program data. Under control of a processor register called the stack pointer, data are referred to as being pushed onto and popped off of the stack, in a last-in, first-out sequence.

system image The image that is read into memory from secondary storage when the system is started up; for example, an operating system.

tag One or more characters, attached to a set of data, that contains information about the set, including its identification.

trailing edge The transition of a pulse that occurs last, such as the high-to-low transition of a high clock pulse.

transceiver A device that can transmit and receive data.

transfer To send data from one place and to receive the data at another place. Synonymous with move.

transient suppression Transients are large voltage spikes that occur on power lines when heavy electrical equipment is switched. Transient suppression is a feature of a computer system's power supply that prevents these spikes from interfering with the operation of the computer system.

translation buffer An internal processor cache containing translations for recently used virtual addresses.

trap An exception condition that occurs at the end of the instruction that caused the exception. The PC saved on the stack is the address of the next instruction that would normally have been executed. All software can enable and disable some of the trap conditions with a single instruction.

tri-state Logic systems using three conditions on one line: a definitely applied high voltage (logic 1), a definite low voltage (logic 0), and an open circuit or undefined state, permitting another part of the circuit to determine whether the line will be high or low.

UART Universal asynchronous receiver transmitter. A device that connects a word-parallel controller or data terminal to a bit-serial communication network.

vector An interrupt or exception vector is a storage location known to the system that contains the starting address of a procedure to be executed when a given interrupt or

exception occurs. The system defines separate vectors for each interrupting device controller and for classes of exceptions. Each system vector is a longword.

virtual page number (VPN) The virtual address of a page of virtual memory.

virtual memory The set of storage locations in physical memory and on disk that are referred to by virtual addresses. From the programmer's viewpoint, the secondary storage locations appear to be locations in physical memory. The size of virtual memory in any system depends on the amount of physical memory available and the amount of disk storage used for non-resident virtual memory.

VLSI Very large scale integration. A method of chip design and layout such that thousands of gates are packaged on one chip.

volume A mass storage medium such as a disk, diskette, or reel of magnetic tape.

word Two contiguous bytes (16 bits) starting on an addressable byte boundary. Bits are numbered from right to left, 0 through 15. A word is identified by the address of the byte containing bit 0.

write protect To protect programs recorded on tape, disks or diskettes from being written over. When a recording device is write protected, writing to that device is locked out. Write protection can be achieved with software, or as with diskettes, by taping or untaping the notch in the diskette jacket.

write through A cache management technique in which data from a write operation are copied in both cache and main memory. Cache and main memory data are always consistent.

zero-extension The process of expanding less than 32 bits of data to a longword by supplying zeros for the high-order bits.

Index

A

B

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

READER'S COMMENTS

MicroVAX I CPU Technical Description
EK-KD32A-TD-002

Your comments and suggestions will help us in our continuous effort to improve the quality and usefulness of our publications.

What is your general reaction to this manual? In your judgement is it complete, accurate, well organized, well written, etc.? Is it easy to use?

_____

_____

_____

What features are most useful? _____

_____

_____

What faults or errors have you found in the manual?

_____

_____

_____

Does this manual satisfy your needs? _____

Why? _____

_____

_____

Additional copies of this document are available from:

Digital Equipment Corporation
444 Whitney Street
Northboro, MA 01532

Attn: Printing and Circulation Services (NR02/M15)
Customer Services Section

digital™

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO 33 MAYNARD, MA

POSTAGE WILL BE PAID BY ADDRESSEE

Digital Equipment Corporation
SASE, ML03-5/U26
146 Main Street
Maynard, MA 01754