EK-78032-UG-PRE

MicroVAX 78032
32-Bit Central Processing Unit
User's Guide

PRELIMINARY

digital

EK-78032-UG-PRE

MicroVAX 78032
32-Bit Central Processing Unit
User's Guide

PRELIMINARY

digital

Preliminary, June 1985

Printed in U.S.A.

The material in this document is for informational purposes and is subject to change without notice.

Digital Equipment Corporation assumes no responsibility for any errors which may appear in this manual.

The following are trademarks of Digital Equipment Corporation:

Digital Logo	MASSBUS	TOPS-10
DEC	MicroVAX	TOPS-20
DECnet	MicroVMS	ULTRIX
DECUS	MINC-11	UNIBUS
DECsystem-10	OMNIBUS	VAX
DECSYSTEM-20	OS/8	VAXELN
DECwriter	PDP	VMS
DIBOL	PDT	VT
Edusystem	RSTS	78032
IAS	RSX	78132

CONTENTS

FIGURES

FIGURES (Cont)

FIGURES (Cont)

TABLES

PREFACE

This user's guide is intended to familiarize the reader with the hardware and software characteristics of the MicroVAX 78032 32-bit Central Processing Unit chip. It is assumed that the reader has had experience with microprocessor design. Familiarity with the VAX architecture will also be helpful.

Chapter 1, Introduction - Provides the reader with an overview and brief functional description of the MicroVAX 78032 CPU single chip microproceesor.

Chapter 2, Architecture - Describes the implementation of the VAX architecture by the MicroVAX 78032 CPU. It covers such areas as: data types, registers, memory management, stacks, interrupts, faults and exceptions, and the restart process.

Chapter 3, Instruction Formats and Addressing Modes - Provides a detailed description of the instruction formats and addressing modes used by the MicroVAX 78032 CPU.

Chapter 4, Instruction Set - Describes the instruction set for the MicroVAX 78032 CPU and its companion floating point unit (FPU). This chapter also describes the emulation process for the VAX instructions that are not directly implemented in the CPU or its companion FPU.

Chapter 5, Bus Transactions - Describes the various bus cycles and external processor/register protocols used by the MicroVAX 78032 CPU.

Chapter 6, Pin Description - Describes the function of each MicroVAX 78032 CPU pin.

Chapter 7, Interfacing - Provides some introductory information on interfacing external logic to the MicroVAX 78032 CPU.

Appendix A, DC and AC Characteristics - Provides power, environmental, and detailed timing information.

Appendix B, Instruction Set Summary - Provides a summary of the VAX instructions implemented by the MicroVAX 78032 CPU and 78132 FPU. This appendix also summarizes the instructions that receive emulation assistance from the CPU.

Appendix C, Console Entry and Exit Routines - Sample console entry, exit and memory management emulation routines.

Appendix D, Mechanical Specifications - Provides package dimensions for the two different packages that the MicroVAX 78032 CPU comes in.

CHAPTER 1

INTRODUCTION

1.1 GENERAL DESCRIPTION

The MicroVAX 78032 Central Processing Unit (CPU) is a single chip 32-bit microprocessor. Its principal design features are:

- Instruction set, data type, and memory management compatibility with the VAX-11 superminicomputer.
- 4 Gbyte virtual address space, 1 Gbyte physical address space.
- 32-bit internal and external data paths.
- High performance
- On chip, tightly integrated, demand paged virtual memory management.
- On chip clock generator and interrupt controller.
- Simple external interface.

The MicroVAX 78032 CPU implements a compatible subset of the VAX-11 architecture. Visible machine state consists of sixteen general purpose registers, a processor status word, and eighteen privileged registers. The instruction set architecture supports all 304 native-mode VAX instructions. Of these, 175 are implemented in the MicroVAX 78032 CPU, and 70 in the MicroVAX 78132 Floating Point Unit (FPU). The remaining 59 instructions may be implemented via software emulation, of which 27 are assisted by microcode. All VAX data types are supported. Of these, six are implemented in the MicroVAX 78032 CPU: byte, word, longword, and quadword integers; variable length bit fields; and variable length character strings. Three are implemented in the MicroVAX 78132 FPU: single precision, double precision, and extended double precision floating point numbers. The remaining data types are supported via software emulation.

The memory management architecture provides demand paged virtual memory management. Virtual memory is 4 Gbyte, divided into four 1 Gbyte regions of 2**21 512 byte pages each: P0, P1, system, and reserved (Figure 1-1). The P0 and P1 regions are intended for user programs and are mapped through double level page tables, that is, the page tables reside in system virtual address space and can be paged. The system region is used for the operating system and is mapped through a single level page table. Four hierarchical access modes (kernel, executive, supervisor, user) are provided, with kernel the most privileged. Each 512 byte page can be set for read/write, read only, or no access from each of the four modes. Physical address space is 1 Gbyte, divided into 512 Mbyte for memory, and 512 Mbyte for Input/Output (I/O) devices or other special uses (Figure 1-1).

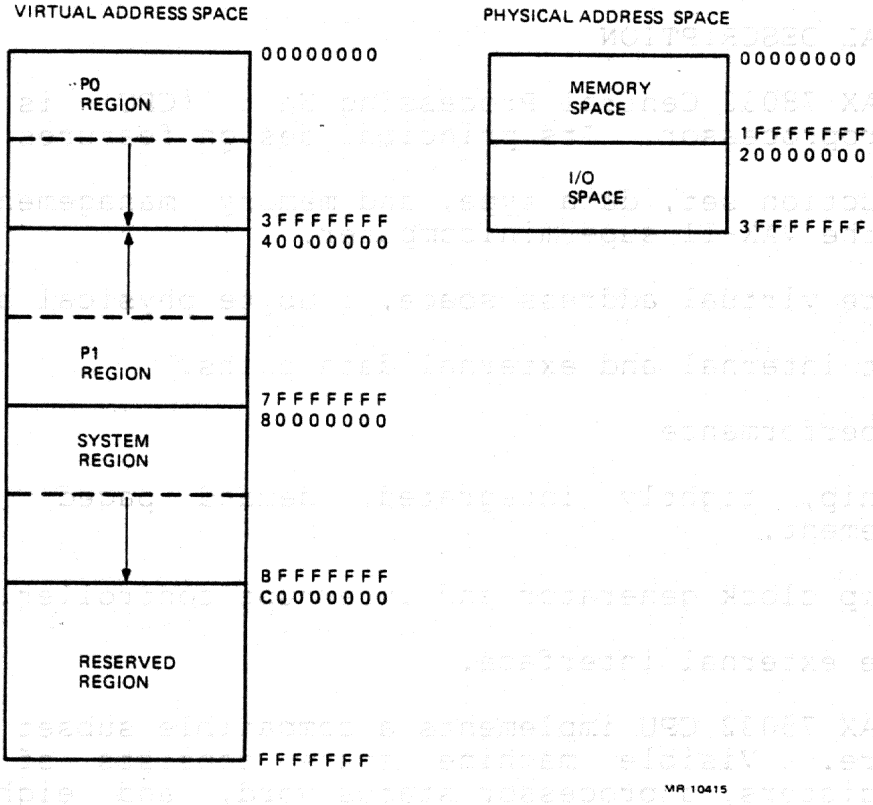


Figure 1-1 Address Space

The MicroVAX 78032 CPU provides a simple and efficient external interface which minimizes support devices without impacting performance (Figure 1-2). The primary communications path is the multiplexed Data and Address Bus (DAL<31:00>). This bus is used to transmit address from and data to and from the MicroVAX 78032 CPU. The Address Strobe ($\overline{AS}$) and Data Strobe ($\overline{DS}$) signals provide timing information for addresses and data, respectively. The Byte Masks ($\overline{BM}$<3:0>) control byte selection within the 32-bit DAL bus. The Control Status (CS<2:0>) lines and Write ($\overline{WR}$) signal provide status information and data direction. Bus cycles are terminated

asynchronously when external logic asserts either Ready (RDY) or Error (ERR). The timing of read and write bus cycles permit standard dynamic RAMs to be interfaced without cycle slips. An eight-byte prefetch buffer, together with a four-byte write buffer, permit instruction fetches and data writes to be overlapped with other operations.

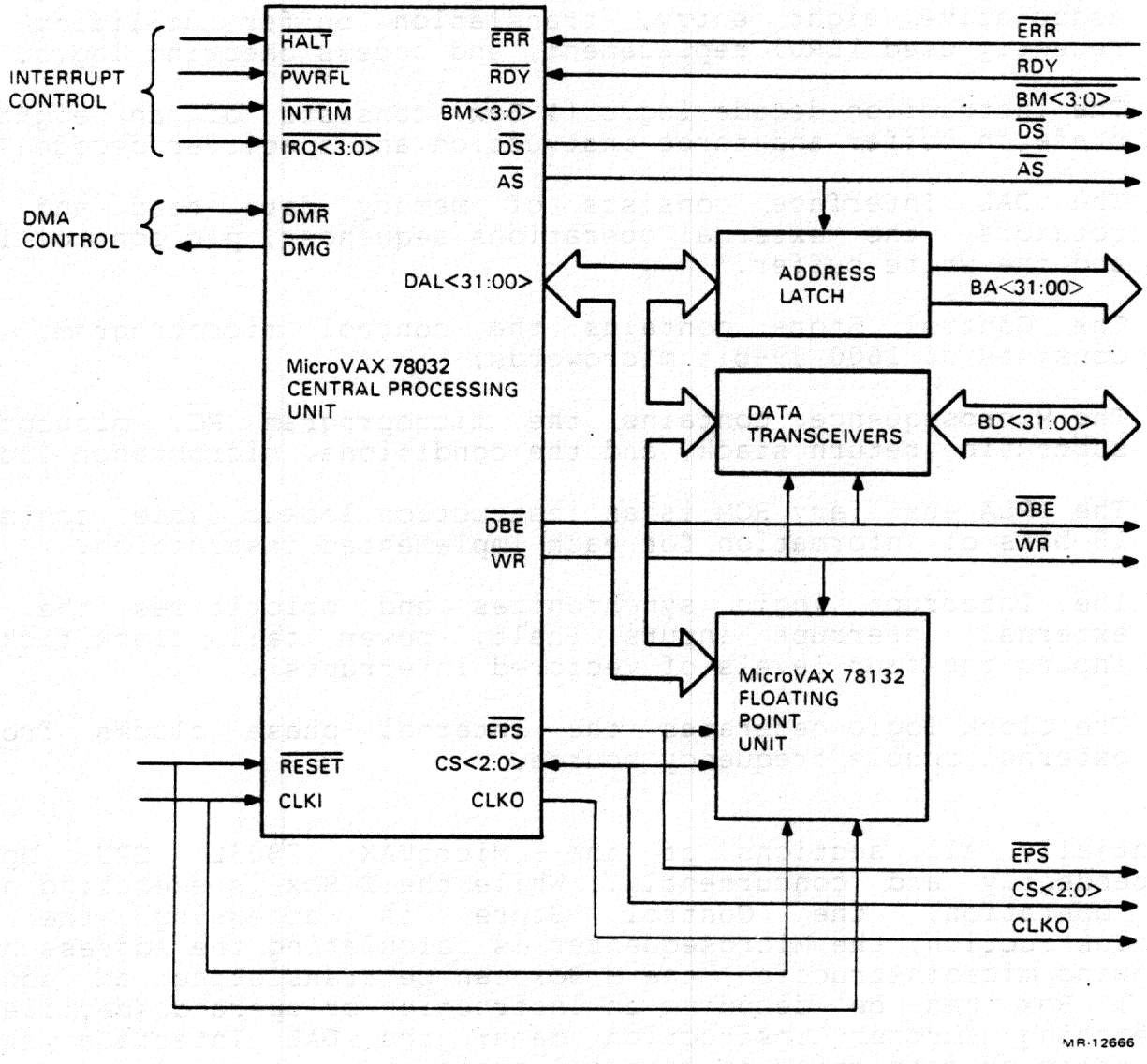


Figure 1-2 Bus Connections

1.2 FUNCTIONAL OVERVIEW

The MicroVAX 78032 CPU utilizes a pipelined, microprogrammed 32-bit implementation. The principal sections are shown in Figure 1-3.

- The data path (E Box) contains the 16 architecturally-specified general registers, 20 scratch registers used by microcode, a 32-bit ALU for arithmetic and logical operations, and a 32-bit barrel shifter for shifts and bit field operations.
- The memory management unit (M Box) contains three address registers (two for data, one for instructions), a fully associative, eight entry, translation buffer utilizing least recently used (LRU) replacement, and access checking logic.
- The instruction decode logic (I Box) consists of an eight-byte prefetch buffer and three instruction and specifier decode PLAs.
- The DAL Interface consists of memory data read and write rotators, the external operations sequencer, pin control logic, and the write buffer.
- The Control Store contains the control microprogram, which consists of 1600 39-bit microwords.
- The Microsequencer contains the microprogram PC, microprogram subroutine return stack, and the conditional microbranch logic.
- The IPLA auxiliary ROM is an instruction lookup table containing 19 bits of information for each implemented instruction.
- The Interrupt logic synchronizes and prioritizes the seven external interrupt inputs (halt, power fail, clock tick, and inputs for four levels of vectored interrupts).
- The Clock logic generates the internal phase clocks from an external double frequency source.

Essentially all sections of the MicroVAX 78032 CPU operate independently and concurrently. While the E Box is executing a data path operation, the Control Store is accessing the next microinstruction; the Microsequencer is calculating the address of the following microinstruction; the M Box can be translating an address; the I Box can be decoding an instruction or operand specifier and prefetching further instruction data; the DAL Interface can be initiating or completing an external access.

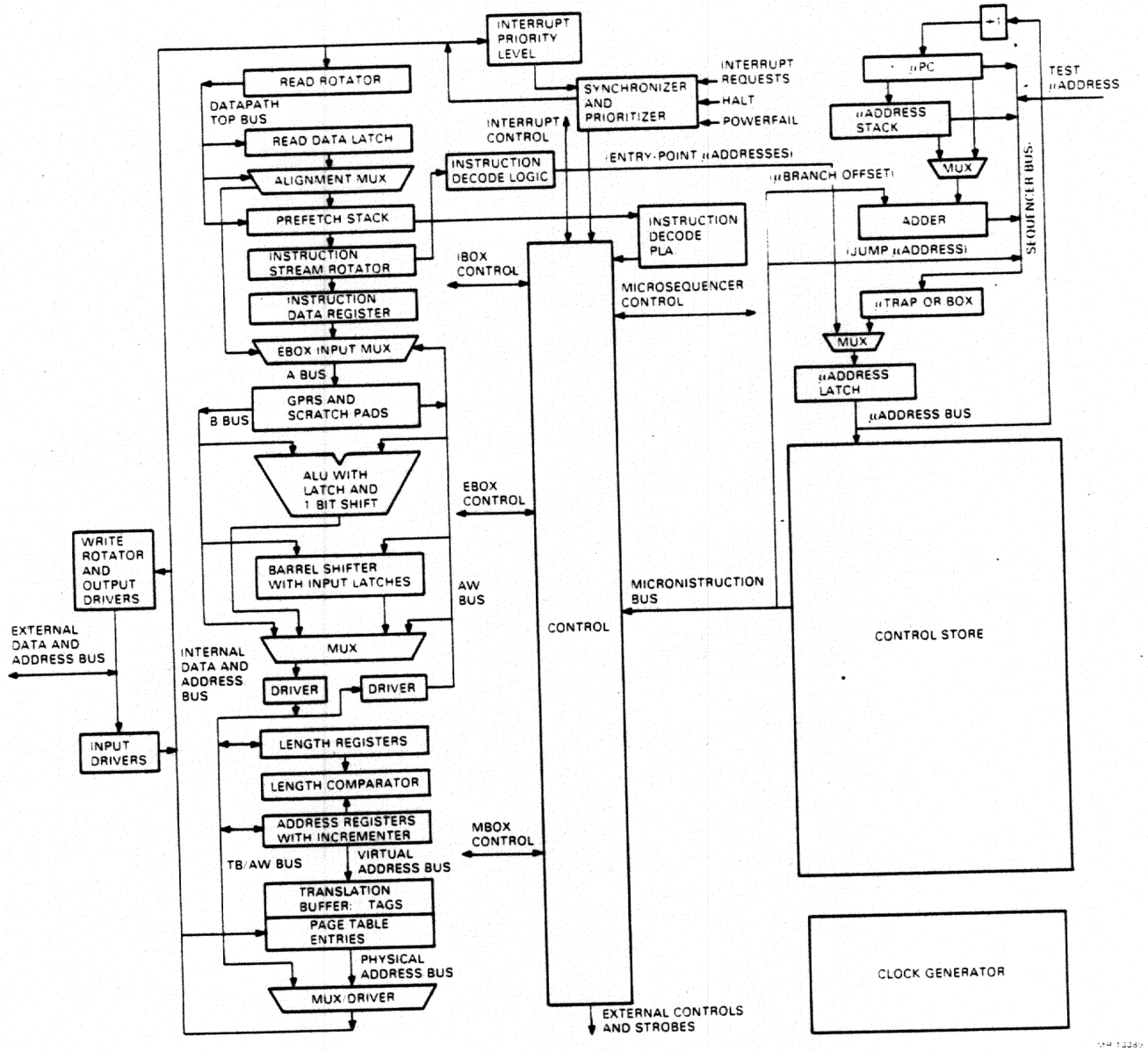


Figure 1-3 MicroVAX 78032 CPU Block Diagram

CHAPTER 2

ARCHITECTURE

2.1 INTRODUCTION

This chapter describes the MicroVAX 78032 Central Processing Unit's implementation of the VAX architecture. This chapter is divided into the following major sections:

- Data Types
- Registers
- Memory Management
- Exceptions and Interrupts
- Process Structure
- Stacks
- Restart Process

The MicroVAX 78032 CPU instruction formats and instruction set are discussed in Chapter 3 and Chapter 4.

2.2 DATA TYPES

The MicroVAX 78032 CPU supports nine data types: byte, word, longword, quadword, character string, variable length bit field, and through the companion MicroVAX 78132 FPU, F_floating, D_floating, and G_floating.

2.2.1 Byte

A byte is 8 contiguous bits starting on an addressable byte boundary. The bits are numbered from the right 0 through 7, Figure 2-1

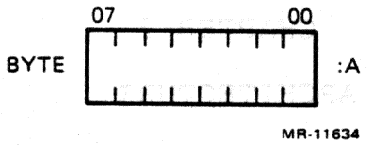


Figure 2-1 Byte Data Type

A byte is specified by its address A. When interpreted arithmetically, a byte is a twos complement integer with bits of increasing significance going 0 through 6 and bit 7 the sign bit. The value of the integer is in the range -128 through 127. For the purposes of addition, subtraction, and comparison, VAX-11 instructions also provide direct support for the interpretation of a byte as an unsigned integer with bits of increasing significance going 0 through 7. The value of the unsigned integer is in the range 0 through 255.

2.2.2 Word

A word is 2 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right 0 through 15, Figure 2-2.

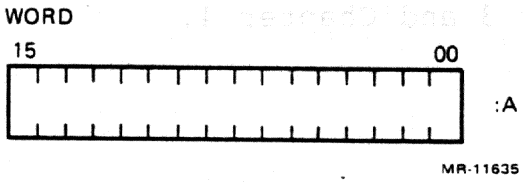


Figure 2-2 Word Data Type

A word is specified by its address A, the address of the byte containing bit 0. When interpreted arithmetically, a word is a twos complement integer with bits of increasing significance going 0 through 14 and bit 15 the sign bit. The value of the integer is in the range -32,768 through 32,767. For the purposes of addition, subtraction and comparison, VAX-11 instructions also provide direct support for the interpretation of a word as an unsigned integer with bits of increasing significance going 0 through 15. The value of the unsigned integer is in the range 0 through 65,535.

2.2.3 Longword

A longword is 4 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right 0 through 31, Figure 2-3.

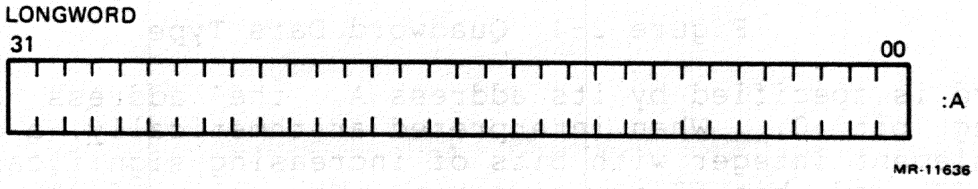


Figure 2-3 Longword Data Type

A longword is specified by its address A, the address of the byte containing bit 0. When interpreted arithmetically, a longword is a twos complement integer with bits of increasing significance going 0 through 30 and bit 31 the sign bit. The value of the integer is in the range -2,147,483,648 through 2,147,483,647. For the purposes of addition, subtraction, and comparison, VAX-11 instructions also provide direct support for the interpretation of a longword as an unsigned integer with bits of increasing significance going 0 through 31. The value of the unsigned integer is in the range 0 through 4,294,967,295.

2.2.4 Quadword

A quadword is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right 0 through 63, Figure 2-4.

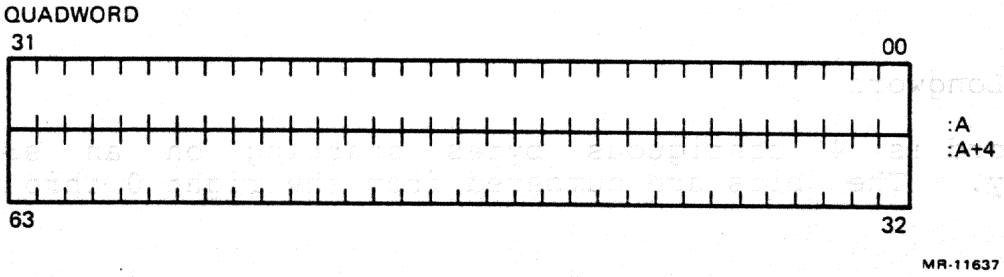


Figure 2-4 Quadword Data Type

A quadword is specified by its address A, the address of the byte containing bit 0. When interpreted arithmetically, a quadword is a twos complement integer with bits of increasing significance going 0 through 62 and bit 63 the sign bit. The value of the integer is in the range -2**63 to 2**63-1.

2.2.5 Variable Length Bit Field

A variable bit field is 0 to 32 contiguous bits located arbitrarily with respect to byte boundaries. A variable bit field is specified by 3 attributes: the address A of a byte, a bit position P which is the starting location of the field with respect to bit 0 of the byte at A, and a size S of the field. The specification of a bit field is indicated by the following where the field is the shaded area, Figure 2-5.

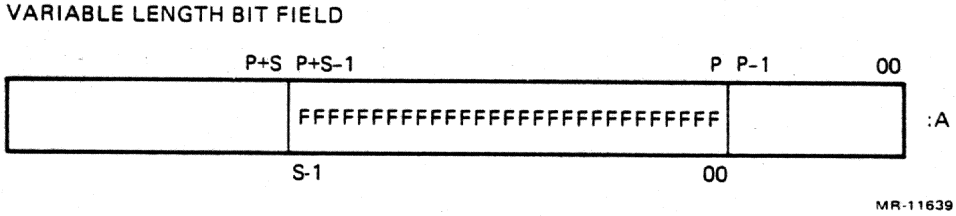


Figure 2-5 Variable Length Bit Field

For bit strings in memory, the position is in the range -2**31 through 2**31-1 and is conveniently viewed as a signed 29-bit byte offset and a 3-bit bit-within-byte field, Figure 2-6.

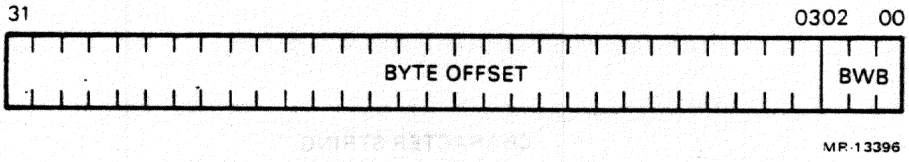


Figure 2-6 Variable Length Bit Field Specifier

The sign extended 29-bit byte offset is added to the address A and the resulting address specifies the byte in which the field begins. The 3-bit bit-within-byte field encodes the starting position (0 through 7) of the field within that byte. The VAX-11 field instructions provide direct support for the interpretation of a field as a signed or unsigned integer. When interpreted as a signed integer, it is twos complement with bits of increasing significance going 0 through S-2; bit S-1 is the sign bit. When interpreted as an unsigned integer, bits of increasing significance go from 0 to S-1. A field of size 0 has a value identically equal to 0.

A variable bit field may be contained in 1 to 5 bytes. From a memory management point of view only the minimum number of aligned longwords necessary to contain the field are actually referenced.

For bit fields in registers, the position is in the range 0 through 31. The position operand specifies the starting position (0 through 31) of the field in the register. A variable bit field may be contained in 2 registers if the sum of position and size exceeds 32, Figure 2-7.

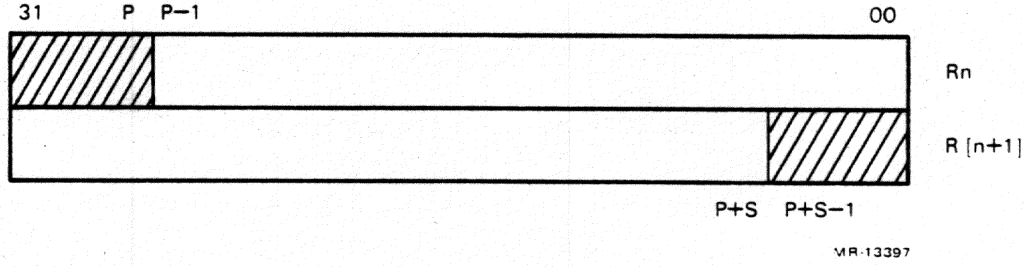


Figure 2-7 Variable Length Bit Field Across Registers

2.2.6 Character String

A character string is a contiguous sequence of bytes in memory. A character string is specified by two attributes: the address A of the first byte of the string, and the length L of the string in bytes. The format of a character string is shown in Figure 2-8. The length L of a string is in the range 0 through 65,535.

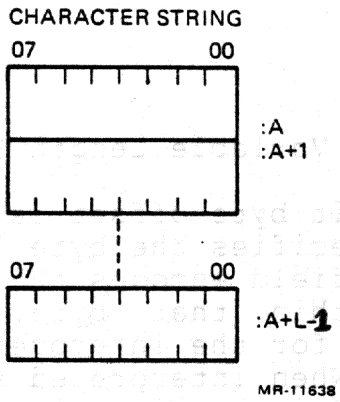


Figure 2-8 Character String Data Type

2.2.7 Floating Point

The MicroVAX 78032 CPU supports the following floating point data types through the MicroVAX 78132 Floating Point Unit.

2.2.7.1 F_floating -

An F_floating datum is 4 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from the right 0 through 31, Figure 2-9.

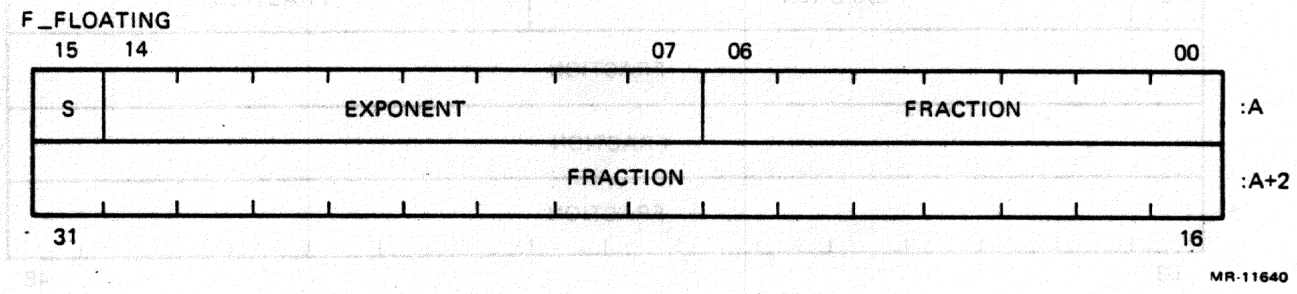


Figure 2-9 F_floating Data Type

An F_floating datum is specified by its address A, the address of the byte containing bit 0. The form of an F_floating datum is sign magnitude with bit 15 the sign bit, bits 14:7 an excess 128 binary exponent, and bits 6:0 and 31:16 a normalized 24-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance go from 16 through 31 and 0 through 6. The 8-bit exponent field encodes the values 0 through 255. An exponent value of 0 together with a sign bit of 0, is taken to indicate that the F_floating datum has a value of 0. Exponent values of 1 through 255 indicate true binary exponents of -127 through +127. An exponent value of 0, together with a sign bit of 1, is taken as reserved. Floating point instructions processing a reserved operand take a reserved operand fault. The value of an F_floating datum is in the approximate range .29*10**-38 through 1.7*10**38. The precision of an F_floating datum is approximately one part in 2**23, i.e., typically 7 decimal digits.

2.2.7.2 D_floating -

A D_floating datum is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from the right 0 through 63, Figure 2-10.

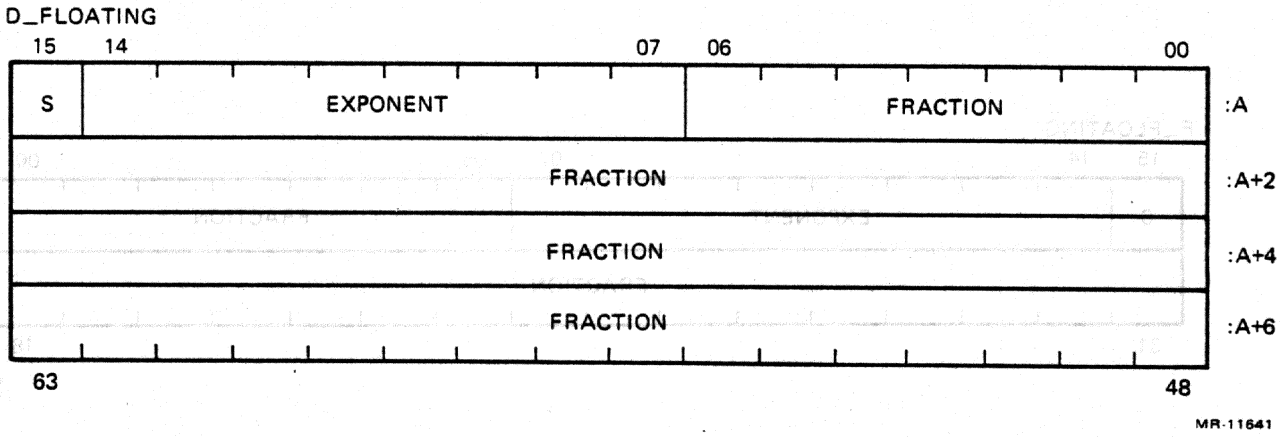


Figure 2-10 D_floating Data Type

A D_floating datum is specified by its address A, the address of the byte containing bit 0. The form of a D_floating datum is identical to an F_floating datum except for an additional 32 low significance fraction bits. Within the fraction, bits of increasing significance go 48 through 63, 32 through 47, 16 through 31, and 0 through 6. The exponent conventions, and approximate range of values is the same for D_floating as F_floating. The precision of a D_floating datum is approximately one part in 2**55, i.e., typically 16 decimal digits.

2.2.7.3 G_floating -

A G_floating datum is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from the right 0 through 63, Figure 2-11.

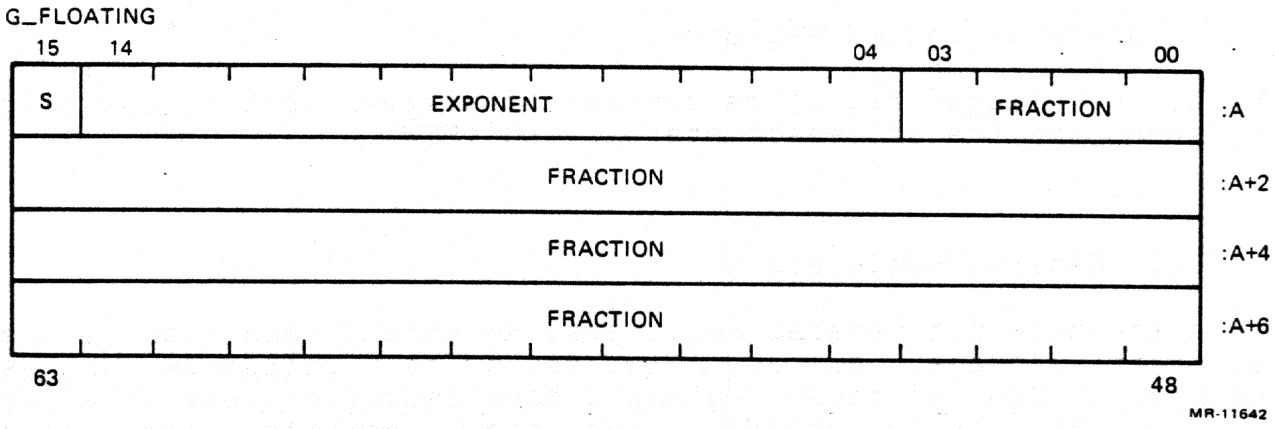


Figure 2-11 G_floating Data Type

A G_floating datum is specified by its address A, the address of the byte containing bit 0. The form of a G_floating datum is sign magnitude with bit 15 the sign bit, bits 14:4 an excess 1024 binary exponent, and bits 3:0 and 63:16 a normalized 53-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance go 48 through 63, 32 through 47, 16 through 31, and 0 through 3. The 11-bit exponent field encodes the values 0 through 2047. An exponent value of 0 together with a sign bit of 0, is taken to indicate that the G_floating datum has a value of 0. Exponent values of 1 through 2047 indicate true binary exponents of -1023 through +1023. An exponent value of 0, together with a sign bit of 1, is taken as reserved. Floating point instructions processing a reserved operand take a reserved operand fault. The value of a G_floating datum is in the approximate range .56*10**-308 through .9*10**308. The precision of a G_floating datum is approximately one part in 2**52, i.e., typically 15 decimal digits.

2.3 REGISTERS

The MicroVAX 78032 CPU's register set is divided into two sections, as seen in Figure 2-12. The general registers and Processor Status Word (PSW) are accessible to non-privileged software; the remaining registers are reserved for system software.

2.3.1 Non-Priviledged Registers

The non-privileged registers consist of sixteen 32-bit general purpose registers and the processor status word (PSW).

2.3.1.1 General Registers -

The sixteen 32-bit general registers, R0 through R15, can be used for temporary storage, as accumulators, as base registers, and as index registers. Four of these registers have specific uses: the argument pointer, the frame pointer, the stack pointer, and the program counter.

1. Argument Pointer - R12 is the argument pointer (AP). The AP contains the base address of a software data structure called the argument list, which is maintained for procedure calls.

2. Frame Pointer - R13 is the frame pointer (FP). The FP contains the base address of a software data structure called the stack frame, which is maintained for procedure calls.

3. Stack Pointer - R14 is the stack pointer (SP). The SP contains the address of the top of the processor defined stack. There are five stack pointers, one for each operating mode (kernel, executive, supervisor, and user) of the processor and one for use by the system when handling interrupts. The stack pointer currently in use is determined by the operating mode of the processor. The operating mode is selected by the current mode bits and the IS bit of the Processor Status Longword (PSL).

4. Program Counter - R15 is the program counter (PC). The PC contains the address of the next instruction byte of the program.

A registers special function does not limit its use to that function, with the exception of the PC. The PC cannot be used as an accumulator, as a temporary register, or as an index register. In general, however, most software does not use the stack pointer, argument pointer, or frame pointer for purposes other than those designated.

When a register is used to contain data, the data is stored in the same format that it would be stored in memory. If a quadword or double floating datum is stored in a register, it is actually stored in two adjacent registers, Rn and R[n+1]. Writing a byte or a word to a register writes only bits <7:0> or bits <15:0> respectively, the remaining bits of the register are unaffected.

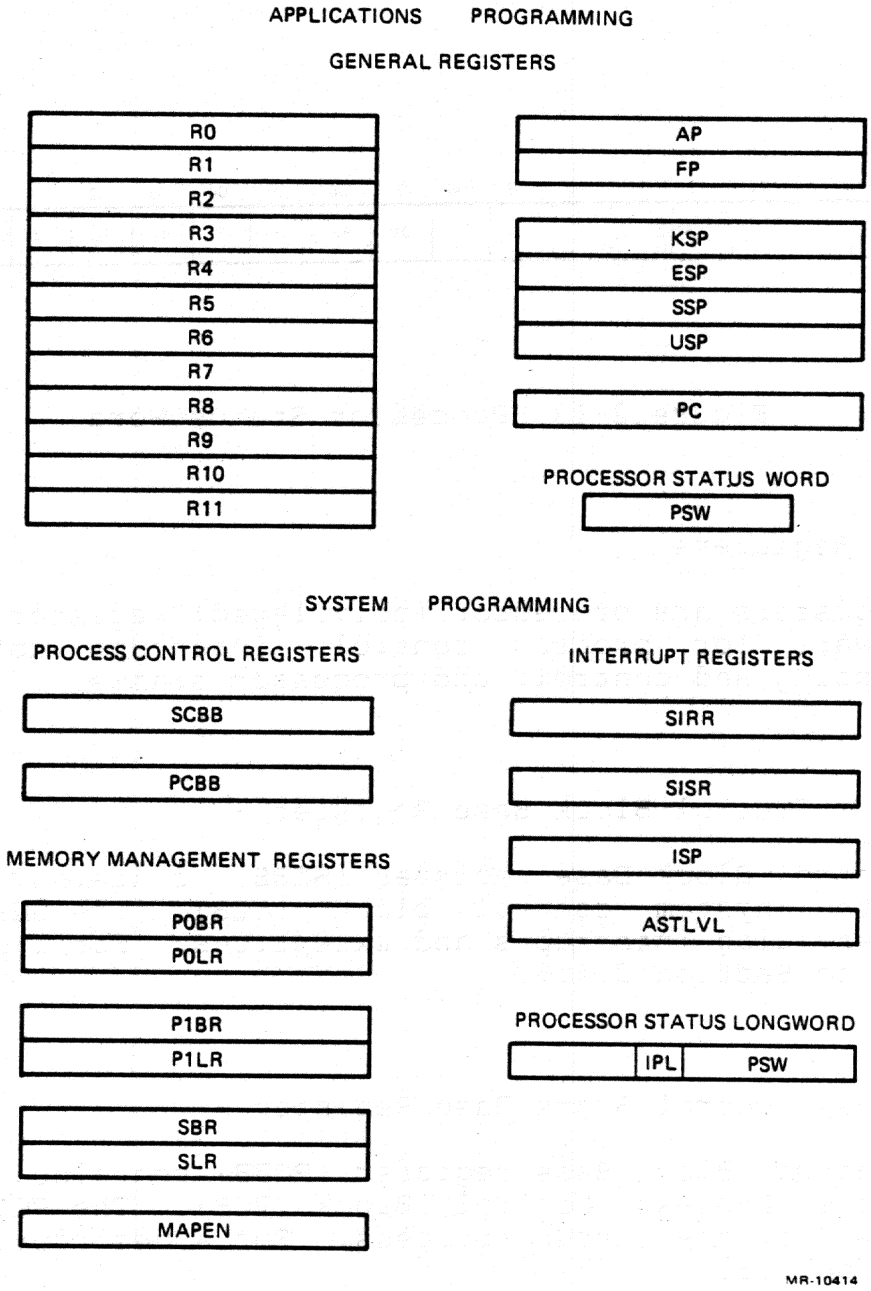


Figure 2-12 MicroVAX 78032 CPU Programming Model

2.3.1.2 Processor Status Word -

The Processor Status Word (PSW) contains the condition codes and trap enable flags for the MicroVAX 78032 CPU. The PSW is the non-privileged portion of the Processor Status Longword (PSL). The lower 16 bits of the PSL contain the PSW. The format of the PSW is shown in Figure 2-13 and the function of each bit is described in Table 2-1.

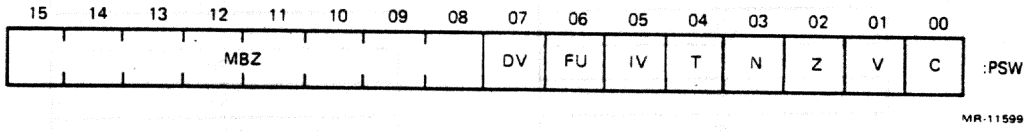


Figure 2-13 Processor Status Word

2.3.2 System Registers

The system registers are processor (privileged) registers for use by system software for process control, interrupt control, memory management mapping and control, and processor status.

2.3.2.1 System Control Block Base Register -

The System Control Block Base register (SCBB) contains the physical address of the system control block (SCB). The SCB contains the vectors for servicing interrupts and exceptions. For a description of the SCB refer to Section 2.5.8.

2.3.2.2 Process Control Block Base Register -

The Process Control Block Base register (PCBB) contains the physical address of the Process Control Block (PCB). The PCB contains the hardware context of the current process. For a description of the PCB refer to Section 2.6.

Table 2-1 Processor Status Word Bit Descriptions

Bits	Description
15:08	Must Be Zero (MBZ).
07:04	Trap Enable Flags - these bits are used to cause traps to occur under special circumstances. DV - Decimal overflow trap enable; used by emulation software in the emulation of decimal instructions. FU - Floating underflow fault enable; when set, this bit causes a floating underflow fault after an instruction that produced a floating result too small in magnitude to be represented. IV - Integer overflow trap enable; when set, this bit causes an integer overflow trap after an instruction that produced an integer result that could not be correctly represented in the space provided. T - Trace bit; when set, this bit causes a trace trap to occur after execution of the next instruction.
03:00	Condition Codes - these bits contain information on the result of the last processor arithmetic or logical operation. The bits are set as follows: N = 1 if the result was negative. Z = 1 if the result was zero. V = 1 if the operation resulted in an arithmetic overflow. C = 1 if the operand resulted in a carry out of or borrow into the MSB (most significant bit).

2.3.2.3 Interrupt Registers -

The Software Interrupt Summary (SISR), Software Interrupt Request (SIRR) and Interrupt Priority Level (IPL) registers are used to control the interrupt system of the processor. They keep track of interrupt requests, current interrupt priority level, and the interrupt stack pointer. The function of these registers is described in Section 2.5.3.

2.3.2.4 Memory Management Registers -

The Map Enable Register (MAPEN), System Base Register (SBR), System Length Register (SLR), P0 Base Register (P0BR), P0 Length Register (P0LR), P1 Base Register (P1BR), and P1 Length Register (P1LR) are used to enable the on chip virtual memory management unit and to access the page table entries (in memory) used to translate virtual addresses into physical addresses. The function of these registers is described in Section 2.4.

2.3.2.5 Processor Status Longword -

The Processor Status Longword (PSL) contains status information about the processor. The lower 16 bits of the PSL are the non-privileged Processor Status Word (PSW). The upper 16 bits of the PSL are privileged and accessed by system software when the processor is in the kernel mode. Table 2-2 describes the function of each bit of the PSL and Figure 2-14 shows the configuration of the PSL.

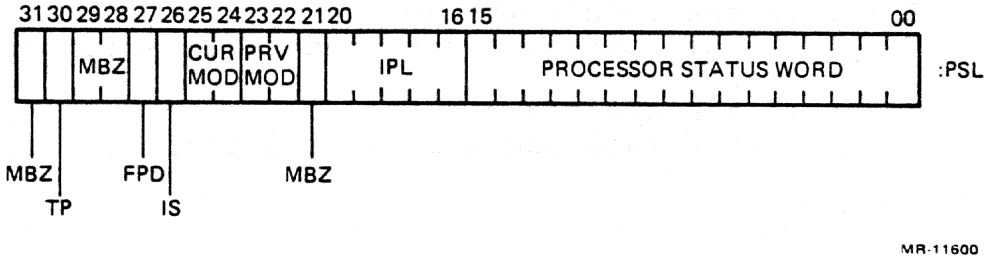


Figure 2-14 Processor Status Longword

Table 2-2 Processor Status Longword Bit Descriptions

Bit	Description
31	Must be zero.
30	Trace Pending (TP) - Forces a trace trap when set at the beginning of any instruction. Set by the processor if the T bit in the PSW is set at the beginning of an instruction.
29:28	Must be zero.
27	First Part Done (FPD) - Set when an exception or interrupt occurs during an instruction that can be suspended. If FPD is set when the processor returns from an exception or interrupt, it resumes the interrupted instruction where it left off, rather than restarting the instruction.
26	Interrupt Stack (IS) - Set when the processor is executing on the interrupt stack.
25:24	Current Access Mode (CUR MOD) - The access mode of the currently executing process: 0 = Kernel 1 = Executive 2 = Supervisor 3 = User
23:22	Previous Access Mode (PRV MOD) - Loaded from CUR MOD by exceptions and Change Mode instructions, cleared by interrupts, and restored by a Return From Exception or Interrupt (REI) instruction.
21	Must be zero.
20:16	Interrupt Priority Level (IPL) - Contains the current processor priority in the range 0 to 1F (hex). The processor will only accept interrupts on levels greater than the current IPL.
15:00	Processor Status Word (PSW) - Contains non-privileged processor status.

2.3.3 Processor Registers

The VAX architecture uses a number of processor (privileged) registers. Some of these registers are implemented in the MicroVAX 78032 CPU and some can be implemented in external logic and accessed by the MicroVAX 78032 CPU. These registers are explicitly accessed only by the Move to Processor Register (MTPR) and Move from Processor Register (MFPR) instructions. The processor registers are listed in Table 2-3.

Each of the processor registers listed in Table 2-3 falls into one of the following categories:

1 = implemented by the MicroVAX 78032 CPU as specified by the VAX Architecture
2 = implemented by the MicroVAX 78032 CPU uniquely
3 = passed to external logic via the external processor register protocol; if not implemented externally, read as zero, no-oped on write
4 = access not allowed (reserved operand fault)

Table 2-3 Internal Processor Registers

Number (Decimal)	Register Name	Mnemonic	Type	Scope	Init	Cat
0	Kernel Stack Pointer	KSP	rw	proc	--	1
1	Executive Stack Pointer	ESP	rw	proc	--	1
2	Supervisor Stack Pointer	SSP	rw	proc	--	1
3	User Stack Pointer	USP	rw	proc	--	1
4	Interrupt Stack Pointer	ISP	rw	cpu	--	1
5	reserved	--	--	--	--	4
6	reserved	--	--	--	--	4
7	reserved	--	--	--	--	4
8	P0 Base Register	P0BR	rw	proc	--	1
9	P0 Length Register	P0LR	rw	proc	--	1
10	P1 Base Register	P1BR	rw	proc	--	1
11	P1 Length Register	P1LR	rw	proc	--	1
12	System Base Register	SBR	rw	cpu	--	1
13	System Length Register	SLR	rw	cpu	--	1
14	reserved	--	--	--	--	4
15	reserved	--	--	--	--	4
16	Process Control Block Base	PCBB	rw	proc	--	1
17	System Control Block Base	SCBB	rw	cpu	--	1
18	Interrupt Priority Level	IPL	rw	cpu	yes	1
19	AST Level	ASTLVL	rw	proc	yes	1
20	Software Interrupt Request	SIRR	w	cpu	--	1
21	Software Interrupt Summary	SISR	rw	cpu	yes	1
22	Interprocessor Interrupt	IPIR	rw	cpu	--	4
23	CMI Error Register	CMIERR	r	cpu	--	4
24	Interval Clock Control	ICCS	rw	cpu	yes	2
25	Next Interval Count	NICR	w	cpu	--	3
26	Interval Count	ICR	r	cpu	--	3
27	Time Of Year	TODR	rw	cpu	--	3
28	Console Storage Receiver Status	CSRS	rw	cpu	--	3
29	Console Storage Receiver Data	CSRD	r	cpu	--	3
20	Console Storage Transmitter Status	CSTS	rw	cpu	--	3
31	Console Storage Transmitter Data	CSTD	w	cpu	--	3
32	Console Receiver Status	RXCS	rw	cpu	--	3
33	Console Receiver Data	RXDB	r	cpu	--	3
34	Console Transmitter Status	TXCS	rw	cpu	--	3
35	Console Transmitter Data	TXDB	w	cpu	--	3
36	Translation Buffer Disable	TBDR	rw	cpu	--	3
37	Cache Disable	CADR	rw	cpu	--	3
38	Machine Check Error Summary	MCESR	rw	cpu	--	3
39	Cache Error	CAER	rw	cpu	--	3

Table 2-3 Internal Processor Registers (Continued)

Number (Decimal)	Register Name	Mnemonic	Type	Scope	Init	Cat
40	Accelerator Control/Status	ACCS	rw	cpu	--	4
41	Console Saved ISP	SAVISP	r	cpu	--	2
42	Console Saved PC	SAVPC	r	cpu	--	2
43	Console Saved PSL	SAVPSL	r	cpu	--	2
44	WCS Address	WCSA	rw	cpu	--	4
45	WCS Data	WCSD	rw	cpu	--	4
46	reserved	--	--	--	--	4
47	reserved	--	--	--	--	4
48	SBI Fault/Status	SBIFS	rw	cpu	--	3
49	SBI Silo	SBIS	r	cpu	--	3
50	SBI Silo Comparator	SBISC	rw	cpu	--	3
51	SBI Maintenance	SBIMT	rw	cpu	--	3
52	SBI Error Register	SBIER	rw	cpu	--	3
53	SBI Timeout Address	SBITA	r	cpu	--	3
54	SBI Quadword Clear	SBIQC	w	cpu	--	3
55	IO Bus Reset	IORESET	w	cpu	--	3
56	Memory Management Enable	MAPEN	rw	cpu	yes	1
57	Trans. Buf. Invalidate All	TBIA	w	cpu	--	1
58	Trans. Buf. Invalidate Single	TBIS	w	cpu	--	1
59	Translation Buffer Data	TBDATA	rw	cpu	--	3
60	Microprogram Break	MBRK	rw	cpu	--	3
61	Performance Monitor Enable	PMR	rw	proc	--	3
62	System Identification	SID	r	cpu	--	1
63	Translation Buffer Check	TBCHK	w	cpu	--	1
64:127	reserved	--	--	--	--	4

Legend:

r = read
w = write
rw = read/write
cpu = process specific, loaded by LDPCTX
proc = system-wide, not affected by LDPCTX
Init = initialized at power-up or chip reset by the restart process

2.3.3.1 MicroVAX 78032 CPU Specific Registers -

The implementation specific processor registers are: Interval Clock Control and Status (ICCS), Console Saved Interrupt Stack Pointer (SAVISP), Console Saved PC (SAVPC), and Console Saved PSL (SAVPSL). These are described in the following paragraphs.

2.3.3.1.1 Interval Clock Control And Status Register (ICCS) -

The ICCS register controls the interval timer ($\overline{\text{INTTIM}}$) interrupt. It contains a single bit, bit <6>, to enable or disable the interval timer interrupt. Bit <6> is read/write. When set, interval timer interrupts are enabled at IPL16; when clear, interval timer interrupts are disabled. Bits <31:7,5:0> read as zero and are ignored on writes. Bit <6> is cleared by RESET. Figure 2-15 shows the ICCS register.

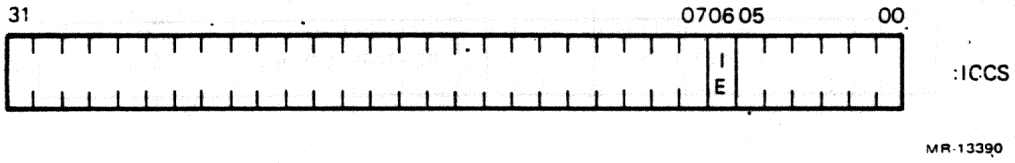


Figure 2-15 Interval Clock and Control Status Register

2.3.3.1.2 Console Saved Registers (SAVISP, SAVPC, SAVPSL) -

The console saved registers (SAVISP, SAVPC, SAVPSL) are limited life processor registers used to record the value of the interrupt stack pointer, PC, and PSL, respectively, at the time a chip restart occurs.

The SAVISP and SAVPC registers are used to save the interrupt stack pointer and the PC, respectively. The SAVPSL register is used to save the contents of the PSL, MAPEN, and the restart code. Figure 2-16 shows these registers. Refer to Section 2.8 for a description of how these registers are used.

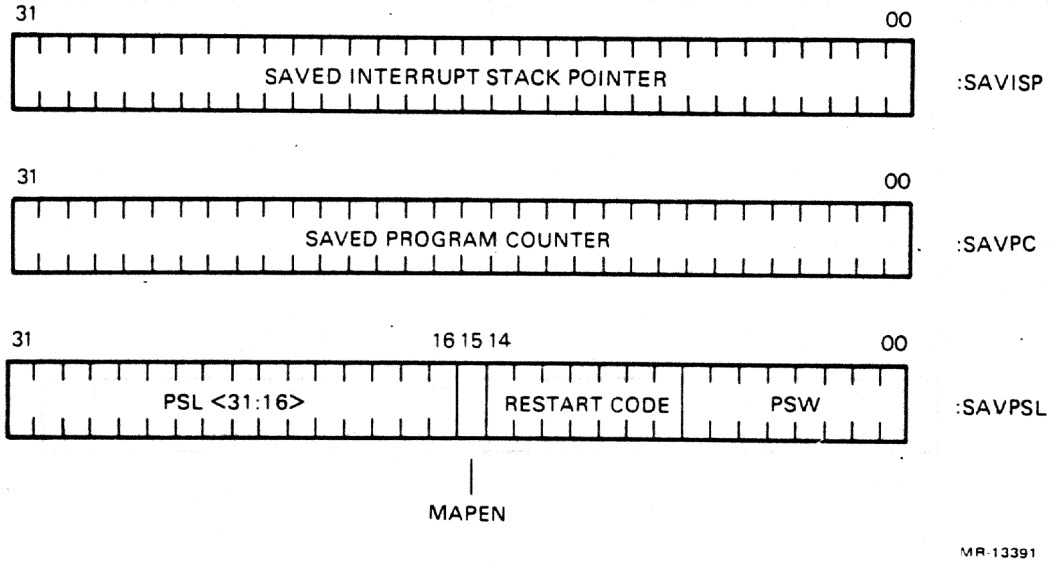


Figure 2-16 Console Saved Registers

2.4 MEMORY MANAGEMENT

Memory management consists of the hardware and software which control the allocation and use of physical memory. Typically, in a multiprogramming system, several processes may reside in physical memory at the same time. The MicroVAX 78032 CPU uses memory protection and multiple address spaces to ensure that one process will not affect other processes or the operating system.

To further improve software reliability, four hierarchical access modes provide memory access control. They are, from most to least privileged: kernel, executive, supervisor, and user. Protection is specified at the individual page level, where a page may be inaccessible, read-only, or read/write for each of the four access modes. Any location accessible to one mode is also accessible to all more privileged modes. For each access mode any location that can be written can also be read.

When memory management is enabled, the CPU generates virtual addresses when a program is executed. However, before these addresses can be used to access instructions and data, they must be translated into physical addresses. Memory management software must maintain tables of mapping information (page tables) that keep track of where each 512 byte virtual page is located in physical memory. The CPU utilizes this mapping information when it translates virtual addresses to physical addresses.

Memory management is the scheme that provides both the memory protection and memory mapping mechanisms of the MicroVAX 78032 CPU. Memory management meets several goals:

1. Provide a large address space for instructions and data.

2. Allow data structures up to one gigabyte.

3. Provide convenient and efficient sharing of instructions and data.

4. Contribute to software reliability.

A virtual memory system provides a large address space, yet allows programs to run on hardware with limited memory configurations. Programs execute in an environment termed a process. The virtual memory system for the MicroVAX 78032 CPU provides each process with a 4 billion byte virtual address space.

Memory management divides virtual address space into two equal size spaces, the system address space and the per-process address space. The system address space is the same for all processes. It is used for the operating system, which may be written as callable procedures. Thus all system code can be available to all other system and user code via a CALL instruction. Each process has its own separate process address space. However, several processes may have access to

the same page, thus providing controlled sharing.

2.4.1 Virtual Address Space

A virtual address is a 32 bit unsigned integer specifying a byte location in the address space. To the programmer memory is a linear array of 4,294,967,296 bytes. The virtual address space is broken into 512 byte units termed pages. Each page may be relocated and protected.

Memory management provides the mechanism to map the active part of the virtual address space to the available physical address space. Memory management also provides page protection between processes. The operating system controls the virtual-to-physical address mapping tables, and swaps the inactive but used parts of the virtual address space onto the external storage media.

The virtual address space is divided into two parts. The half with the lower addresses, known as "per-process space," is distinct for each process running on the system. The half with the higher addresses, known as "system space," is shared by all processes. Figure 2-17 shows the division of virtual address space.

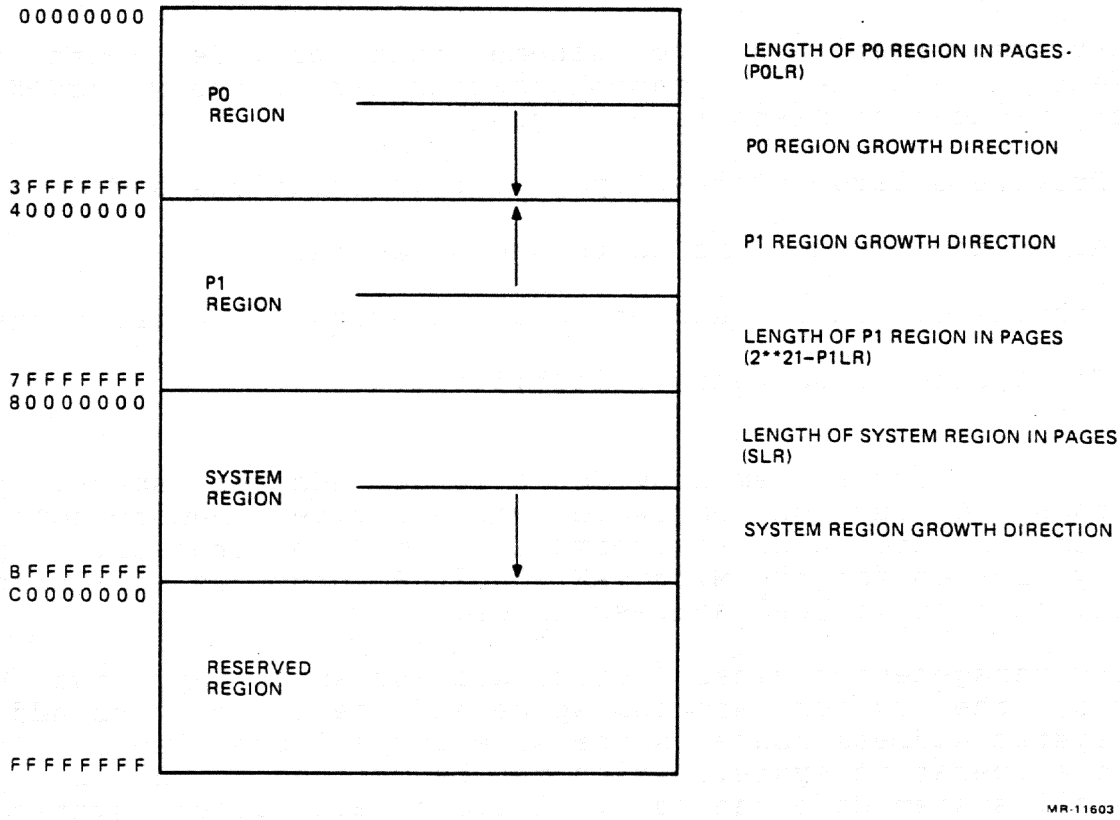


Figure 2-17 Virtual Address Space

2.4.1.1 Process Space -

Addresses 00000000-7FFFFFFF (hex) of virtual address space are called "per-process space". The per-process space is divided into two equal parts, the program region (P0 region) and the control region (P1 region). Each process has a separate address translation map for per-process space, so the per-process spaces of all processes can be completely disjointed. The address map for per-process space is context switched (changed) when the process running on the system is changed.

2.4.1.2 System Space -

Addresses 80000000-FFFFFFFF (hex) of virtual address space are called "system space". All processes use the same address translation map for system space, so system space is shared among all processes. The address map for system space is not context switched.

2.4.1.3 Virtual Address Format -

The MicroVAX 78032 CPU generates a 32-bit virtual address for each instruction and operand in memory. As the process executes, the processor translates each virtual address to a physical address. The format of a virtual address is shown in Figure 2-18 and described in Table 2-4.

When bit 31 is one, the address is in the system space. When bit 31 is zero, the address is in the per-process space.

Within the per-process space, bit 30 distinguishes between the program and control regions. When bit 30 is one, the control region is referenced, and when it is zero, the program region is referenced.

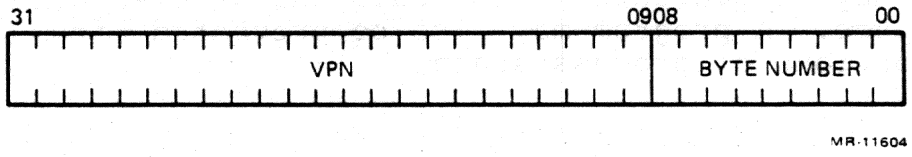


Figure 2-18 Virtual Address

Table 2-4 Virtual Address Bit Description

Field	Bits	Description
VPN	<31:09>	Virtual Page Number - This field specifies the virtual page to be referenced. Virtual address space contains 8,388,608 pages of 512 bytes each. Bits <31:30> of the VPN are used to select the region of virtual address space being referenced. Value of Bits <31:30> / Region Referenced 0 / P0 1 / P1 2 / System 3 / Reserved
BYTE NUMBER	<08:00>	Byte Number - This field specifies the byte number within the page.

2.4.1.4 Page Protection -

Independent of its location in virtual address space, a page (512 bytes) may be protected according to its use. Even though all of system space is shared, in that a program may generate any address, the program may be prevented from modifying or even accessing portions of system space. A program may also be prevented from accessing or modifying portions of process space.

2.4.2 Memory Management Control

The action of translating a virtual address into a physical address is controlled by the setting of the Memory Mapping Enable (MME) bit in the Map Enable (MAPEN) register. The format of the Map Enable Register is shown in Figure 2-19 and described in Table 2-5.

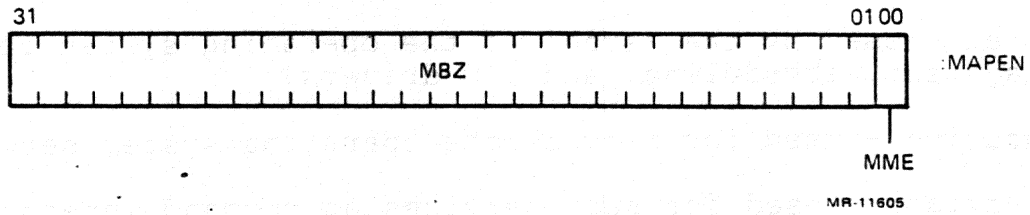


Figure 2-19 Map Enable Register

Table 2-5 Map Enable Register Bit Description

Field	Bit	Description
MBZ	<31:01>	Must Be Zero
MME	<00>	Memory Management Enable - used to enable and disable memory management. MME = 1 Enabled MME = 0 Disabled

2.4.3 Access Control

Access control is the function of validating whether a particular type of memory access is to be allowed to a particular page. Every page has associated with it a protection code that specifies for each mode whether or not read or write references are allowed. Also, each address is checked to make certain that it is in the P0, P1, or system region of virtual address space.

2.4.3.1 Processor Modes -

There are four hierarchical modes used for protection by the MicroVAX 78032 CPU. The modes in the order of most to least privileged are:

0 Kernel - used by the kernel of the operating system for page management, scheduling, and I/O drivers.

1 Executive - used for many of the operating system service calls.

2 Supervisor - used for such services as command interpretation.

3 User - used for user level code, utilities, compilers, debuggers, etc.

The mode at which the processor is currently running is stored in the current mode field of the Processor Status Longword (PSL).

2.4.3.2 Protection Code -

Associated with each page in virtual address space is a protection code that is located in the page table entry for that page. The protection code allows a choice of protection for each processor mode, within the following limits:

1. Access for each processor mode can be read/write, read only, or no access.

2. If a processor mode has read access then all more privileged modes also have read access.

3. If a processor mode has write access then all more privileged modes also have write access.

The protection codes are listed in Table 2-6.

Table 2-6 Protection Codes

code			current mode				
decimal	binary	mnemonic	K	E	S	U	comment
0	0000	NA	-	-	-	-	no access
1	0001		unpredictable				reserved
2	0010	KW	RW	-	-	-	
3	0011	KR	R	-	-	-	
4	0100	UW	RW	RW	RW	RW	all access
5	0101	EW	RW	RW	-	-	
6	0110	ERKW	RW	R	-	-	
7	0111	ER	R	R	-	-	
8	1000	SW	RW	RW	RW	-	
9	1001	SREW	RW	RW	R	-	
10	1010	SRKW	RW	R	R	-	
11	1011	SR	R	R	R	-	
12	1100	URSW	RW	RW	RW	R	
13	1101	UREW	RW	RW	R	R	
14	1110	URKW	RW	R	R	R	
15	1111	UR	R	R	R	R	

Legend:

- = no access
R = read only
RW = read/write

K = Kernel
E = Executive
S = Supervisor
U = User

2.4.3.3 Length Violation -

Every valid virtual address lies within bounds determined by the addressing region (P0, P1, or System) and its associated length register (P0LR, P1LR, or SLR). Virtual addresses that are outside these bounds will cause a length violation.

2.4.3.4 Access Control Violation -

An access control violation fault occurs if an illegal access is attempted, as determined by the current PSL mode and the page's protection field, or if the address causes a length violation.

2.4.3.5 Access Across A Page Boundary -

If an access is made across a page boundary, the order in which the pages are accessed is unpredictable. However, for a given page an access control violation always takes precedence over a translation not valid.

2.4.4 Address Translation

The action of translating a virtual address to a physical address by memory management is controlled by the setting of the Memory Management Enable (MME) bit in the MAPEN register. When MME is a 0, memory mapping disabled, bits<29:00> of the virtual address become the physical address and there is no page protection. This means that all accesses are allowed in all modes. When MME is a 1, memory mapping enabled, address translation and access control are on and the processor uses the following to determine whether an intended access is allowed:

1. The virtual address, which is used to index a page table,
2. The intended access type (read or write), and
3. The current privilege level from the Processor Status Longword, or kernel level for page table mapping references.

If the access is allowed and the address can be mapped, the result is the physical address corresponding to the specified virtual address.

The intended access is READ if the operation to be performed is a read. The intended access is WRITE if the operation to be performed is a write. If the operation to be performed is a modify (that is, read followed by write) the intended access is specified as a WRITE.

If an operand is an address operand, then no reference is made. This means the page does not need to be accessible or even exist.

2.4.4.1 Page Table Entry (PTE) -

The MicroVAX 78032 CPU uses a Page Table Entry (PTE) to translate virtual addresses to physical addresses. The page table entry is shown in Figure 2-20 and described in Table 2-7.

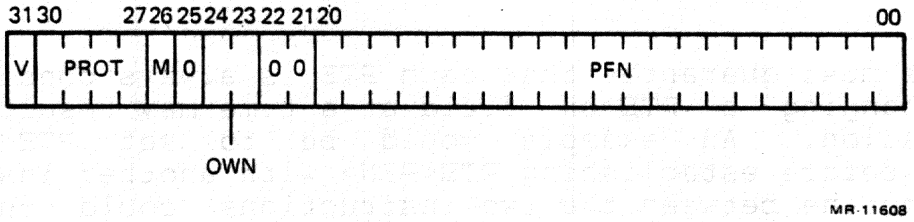


Figure 2-20 Page Table Entry

Table 2-7 Page Table Entry Bit Description

Field	Bit	Description
V	31	Valid bit - Governs the validity of the modify (M) bit and the page frame number (PFN) field. V = 1 for valid; V = 0 for not valid.
PROT	30:27	Protection field - Describes the protection for the page. This field is always valid and is used by the hardware even when V = 0.
M	26	Modify bit - This bit is set (= 1) if the page has already been recorded as modified. M = 0 if the page has not been recorded as modified. Used only if V = 1.
0	25	Must be zero.
OWN	24:23	Owner bits - reserved.
0	22:21	Must be zero.
PFN	20:00	Page Frame Number - The upper 21 bits of the physical address of the base of the page. Used only if V = 1.

2.4.4.1.1 Protection Check Before Valid Check -

The page table entry is defined as having a valid bit that only controls the validity of the modify bit and page frame number field. The protection field is defined as always being valid and is checked first.

2.4.4.1.2 Changes To Page Table Entries -

The operating system changes PTEs as part of its memory management functions. For example, MicroVMS sets and clears the valid bit and changes the PFN field as pages are swapped in and out of physical memory.

The software must guarantee that each PTE is always consistent within itself. Changing a PTE one field at a time may result in incorrect system operation. An example would be to set PTE<V> with one instruction before establishing PTE<PFN> with another instruction. An interrupt routine between the two instructions could use a virtual address that would be mapped by memory management using the inconsistent PTE. Software can solve this problem by building a new PTE in a register and then moving the new PTE to the page table with a single instruction such as Move Long (MOVL).

Multiprocessing makes the problem more complicated. Another processor, be it another CPU or an I/O processor, can reference the same page tables that the first CPU is changing. The second processor must always read consistent PTEs. In order to guarantee this, first note that PTEs are longwords, longword-aligned. Then two requirements must be met:

1. Whenever the software modifies a PTE in more than one byte, it must use a longword, longword-aligned, write-destination instruction, such as MOVL.

2. The hardware must guarantee that a longword, longword-aligned write is an atomic operation. That is, a second processor cannot read (or write over) any of the first processor's partial results.

2.4.4.2 System Space Address Translation -

A virtual address with bits <31:30> = 2 is defined as an address in system virtual address space.

System virtual address space is mapped by the System Page Table (SPT), which is defined by the System Base Register (SBR) and the System Length Register (SLR), Figure 2-21. The SBR contains the base physical address of the the System Page Table. The SLR contains the

size of the SPT in longwords, that is, the number of Page Table Entries. The Page Table Entry addressed by the System Base Register maps the first page of system virtual address space, that is, virtual byte address 80000000 (hex).

Figure 2-22 illustrates the translation of a system virtual address to a physical address.

The algorithm to generate a physical address from a system region virtual address is:

```
SYS_PA = (SBR+4*SVA<29:9>)<20:00>'SVA<08:00>
```

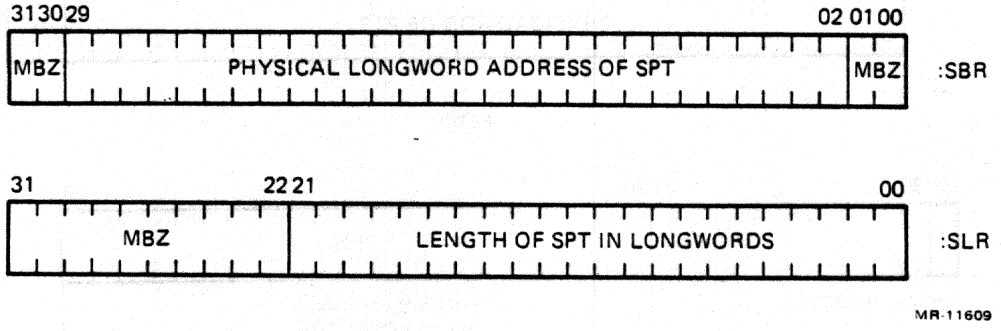


Figure 2-21 System Mapping Registers

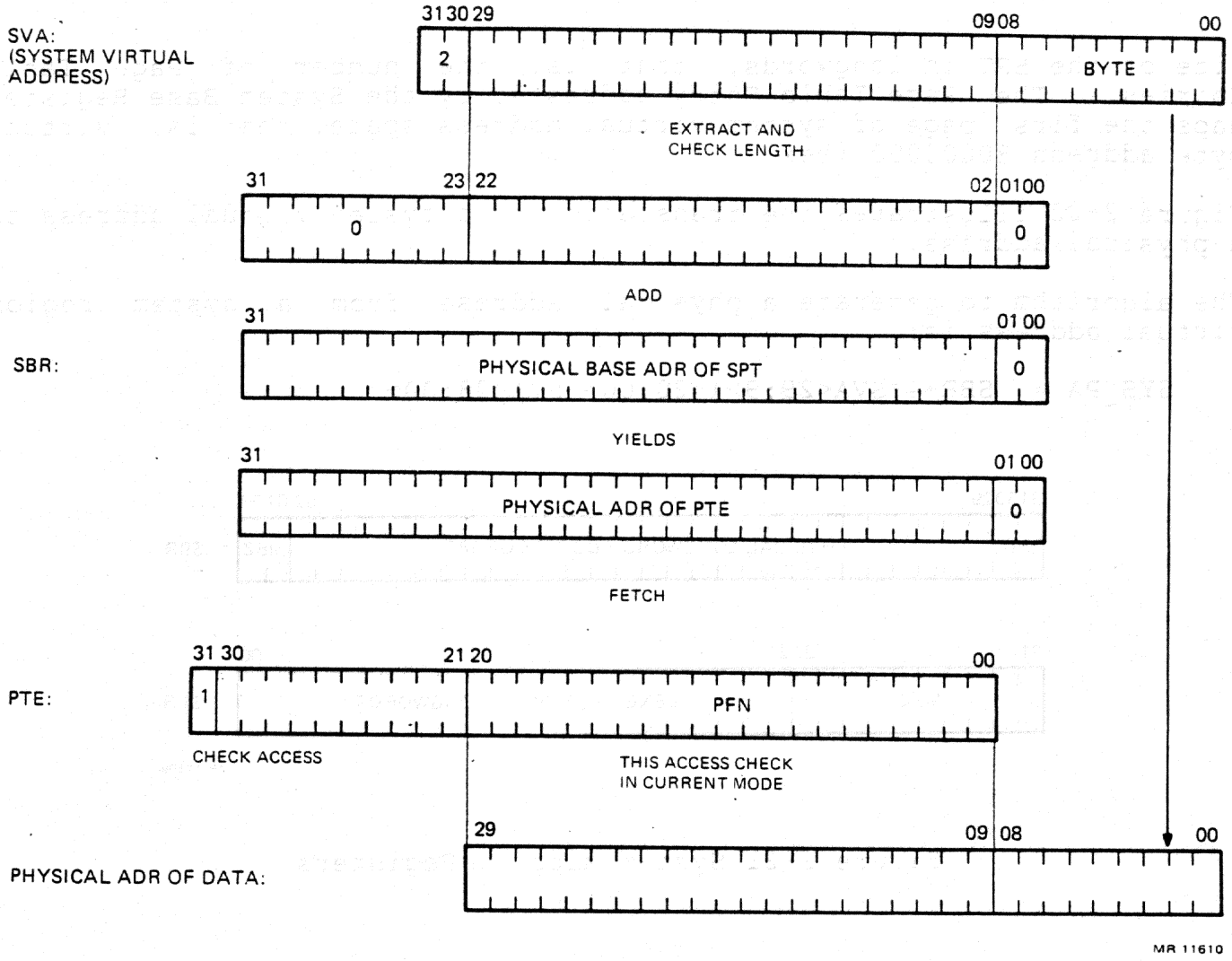


Figure 2-22 System Virtual to Physical Address Translation

2.4.4.3 Process Space Address Translation -

A virtual address with bit <31> = 0 is an address in the process virtual address space. Process space is divided into two equal sized, separately mapped regions. If virtual address bit <30> = 0, the address is in region P0. If virtual address bit <30> = 1, the address is in region P1.

2.4.4.3.1 P0 Region Address Translation -

The P0 region of virtual address space is mapped by the P0 Page Table (P0PT), which is defined by the P0 Base Register (P0BR) and the P0 Length Register (P0LR), Figure 2-23. The P0BR contains the system virtual address of the P0 Page Table. The P0LR contains the size of the P0PT in longwords, that is, the number of Page Table Entries. The Page Table Entry addressed by the P0 Base Register maps the first page of the P0 region of the virtual address space, that is, virtual byte address 0.

Figure 2-24 illustrates the translation of a P0 virtual address to a physical address.

The algorithm to generate a physical address from a P0 region virtual address is:

```
PVA_PTE  = P0BR+4*PVA<29:09>
PTE_PA   = (SBR+4*PVA_PTE<29:09>)<20:00>'PVA_PTE<08:00>
PROC_PA  = (PTE_PA)<20:00>'PVA<08:00>
```

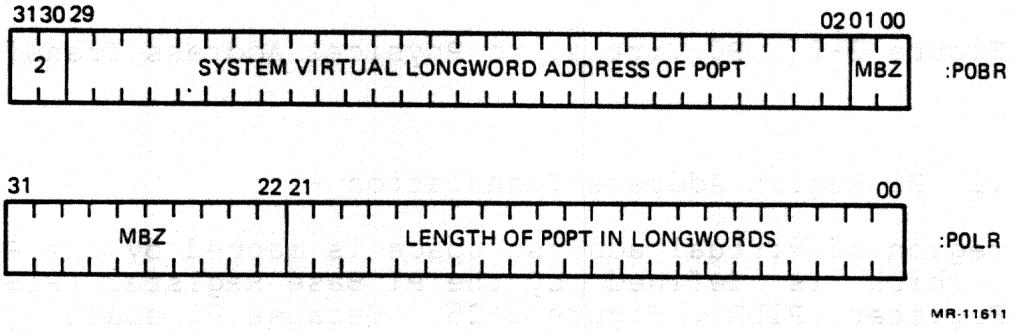


Figure 2-23 P0 Region Mapping Registers

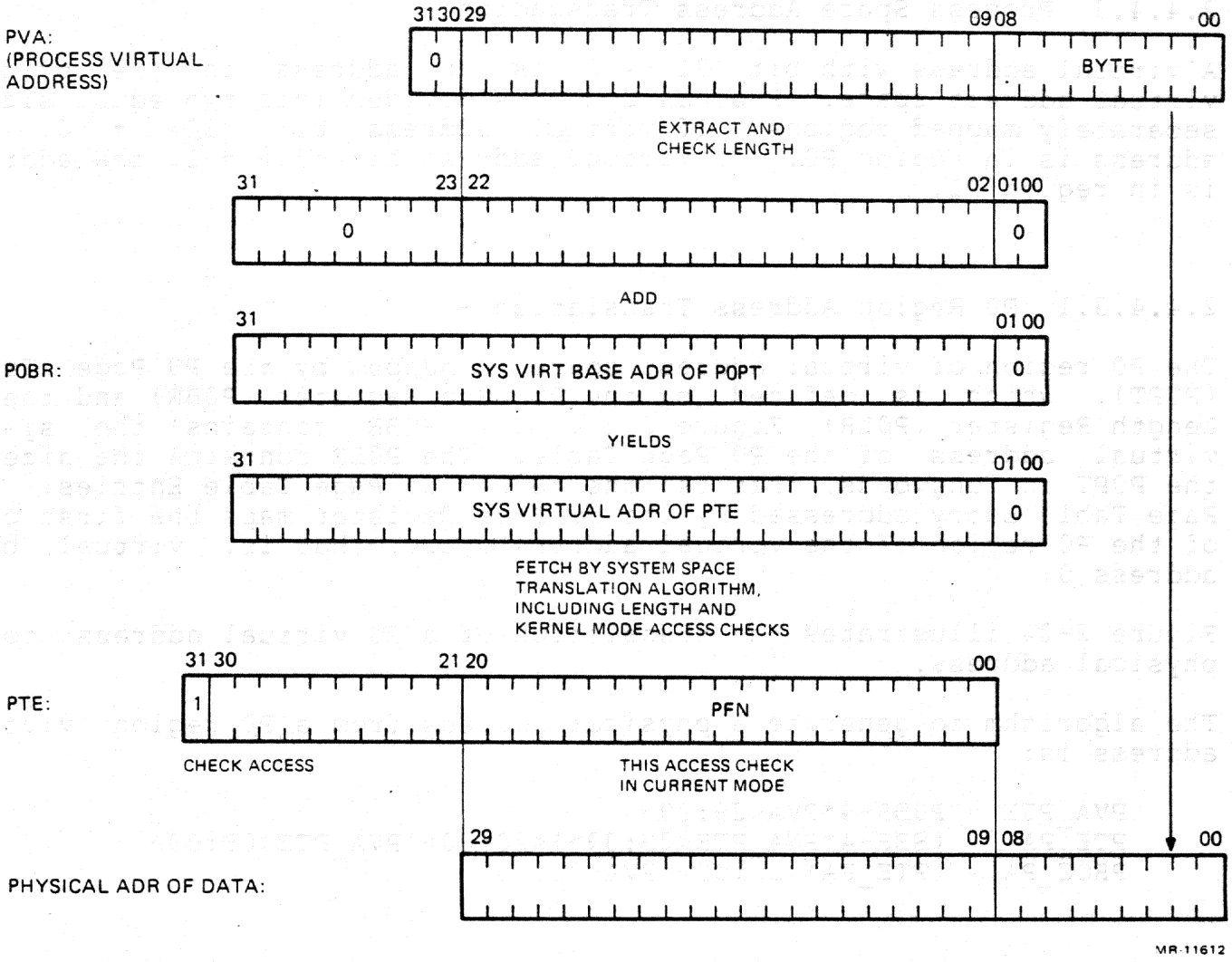


Figure 2-24 P0 Virtual to Physical Address Translation

2.4.4.3.2 P1 Region Address Translation -

The P1 region of virtual address space is mapped by the P1 Page Table (P1PT), which is defined by the P1 Base Register (P1BR) and the P1 Length Register (P1LR), Figure 2-25. Because P1 space grows towards smaller addresses, and because a consistent hardware interpretation of the base and length registers is desirable, P1BR and P1LR describe the portion of P1 space that is NOT accessible. Note that P1LR contains the number of nonexistent PTEs. P1BR contains the system virtual address of what would be the PTE for the first page of P1, that is, virtual byte address 40000000 (hex). The address in P1BR is not necessarily a valid system virtual address, but all the addresses of PTEs must be valid system virtual addresses.

Figure 2-26 illustrates the P1 virtual address to physical address translation.

The algorithm to generate a physical address from a P1 region virtual address is:

```
PVA_PTE = P1BR+4*PVA<29:09>
PTE_PA  = (SBR+4*PVA_PTE<29:09>)<20:00>'PVA_PTE<08:00>
PROC_PA = (PTE_PA)<20:00>'PVA<08:00>
```

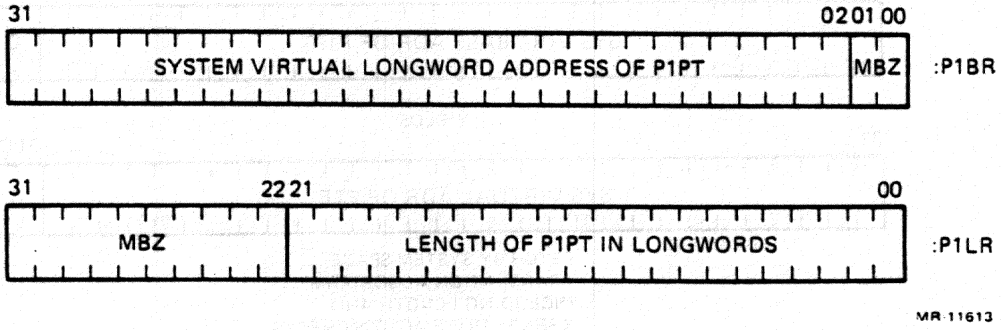


Figure 2-25 P1 Region Mapping Registers

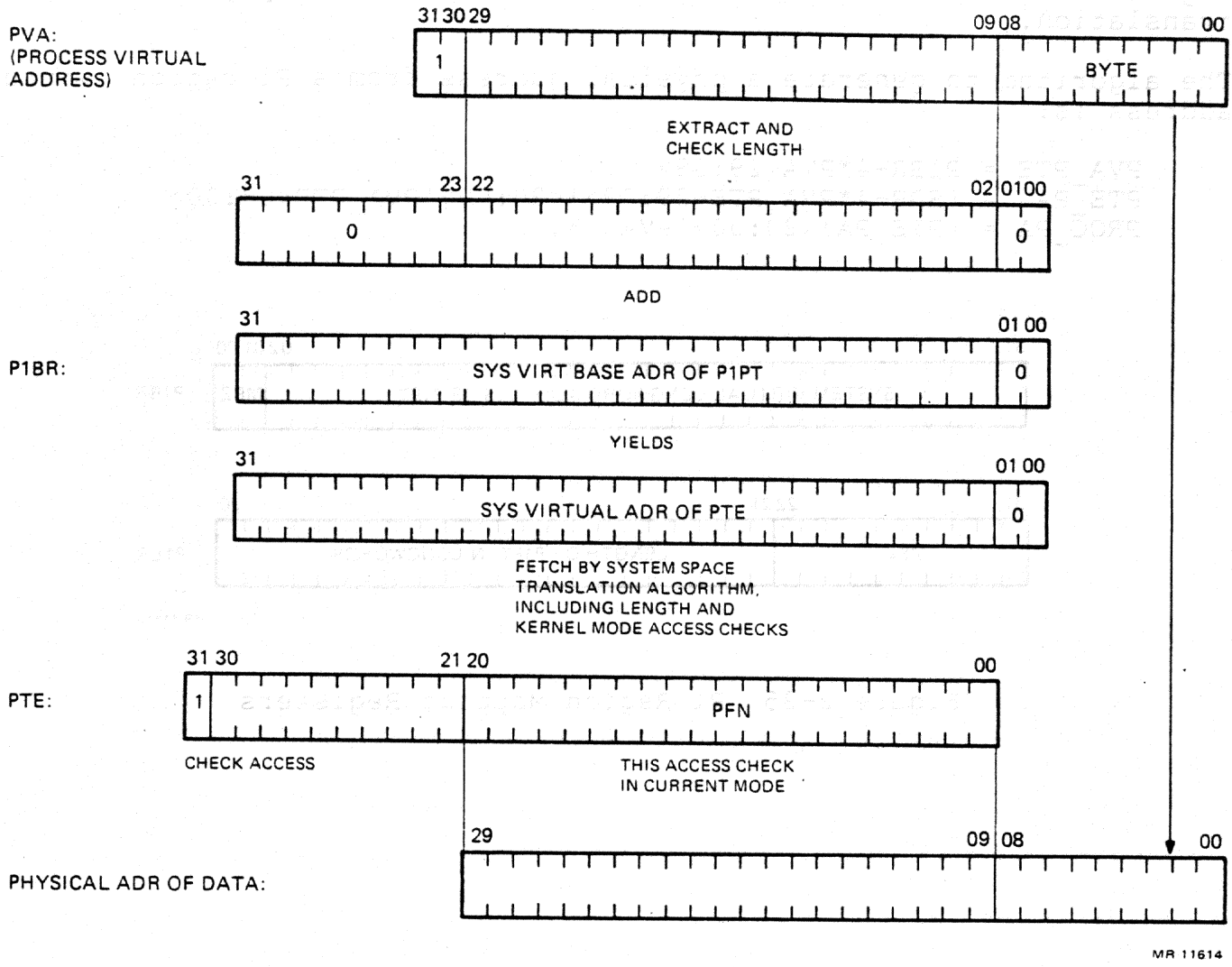


Figure 2-26 P1 Virtual to Physical Address Translation

2.4.5 Translation Buffer

In order to save actual memory references when repeatedly referencing pages, the MicroVAX 78032 CPU has a fully associative, eight entry translation buffer that contains page table entries (PTE) for successful virtual address translations and page status. This buffer helps to save memory references when repeatedly referencing the same pages. Control of the translation buffer is done through two registers: the Translation Buffer Invalidate Single (TBIS) register and the Translation Buffer Invalidate All (TBIA) register.

When the process context is loaded using the Load Process Context (LDPCTX) instruction, the translation buffer is automatically updated (that is, the process virtual address translations are invalidated).

However, when software changes any part of a valid PTE for the system or current process region, it must invalidate any translation buffer entry by moving a virtual address in the corresponding page to the TBIS register. Section 2.4.5.1 gives a description of the TBIS register.

When software changes a system page table entry that maps any part of the current process page table, all translation buffer entries for the affected process pages must be invalidated. This can be done by moving an address in each of the affected pages into the TBIS register, or by invalidating the entire translation buffer by moving a 0 into the TBIA register. Section 2.4.5.2 gives a description of the TBIA register.

The translation buffer does not store invalid PTEs. Therefore, software is not required to invalidate translation buffer entries when making changes for PTEs that are already invalid.

When the location or the size of the system map is changed (SBR, SLR) the entire translation buffer must be cleared.

Whenever memory management is disabled (MME = 0) the contents of the translation buffer are unpredictable. Therefore, before enabling memory management at processor initialization time, or any other time, the entire translation buffer must be cleared. This is done by moving a 0 into the TBIA register.

2.4.5.1 Translation Buffer Invalidate Single Register -

The TBIS register is used to invalidate single PTE entries in the translation buffer. This is done by system software writing a virtual address into the TBIS register, Figure 2-27. The MicroVAX 78032 CPU will invalidate the translation buffer entry that maps the page in virtual memory accessed by the virtual address written into the TBIS register.

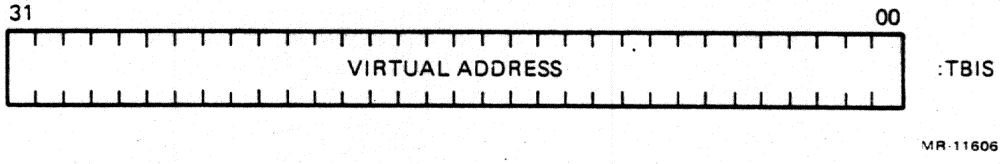


Figure 2-27 Translation Buffer Invalidate Single Register

2.4.5.2 Translation Buffer Invalidate All Register -

The TBIA register is used to clear the entire translation buffer by invalidating all the PTE's in the translation buffer. This is done by software moving a 0 into the 0 into the TBIA register, Figure 2-28. When a 0 is written into the TBIA register the MicroVAX 78032 CPU will invalidate all the PTE's in the translation buffer.

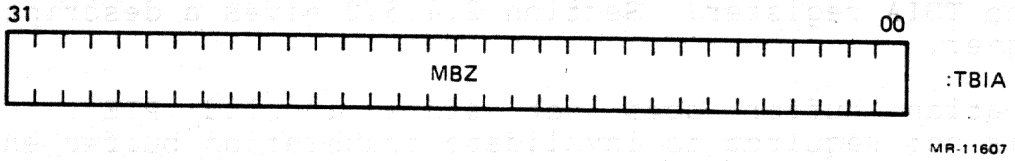


Figure 2-28 Translation Buffer Invalidate All Register

2.4.6 Memory Management Faults

Two types of faults are associated with memory mapping and protection: translation not valid and access control violation. A translation not valid fault is taken when a read or write reference is attempted through an invalid PTE (PTE<31> = 0). An access control violation fault is taken when the protection field of the PTE indicates that the intended page reference in the specified access mode would be illegal. Note that these two faults have separate vectors in the system control block. If both access control violation (ACV) and translation not valid (TNV) faults occur, the access control violation fault takes precedence. An access control violation fault is also taken if the virtual address referenced is beyond the end of the associated page table. Such a "length violation" is essentially the same as referencing a PTE that specifies "No Access" in its protection field. To eliminate the need to recompute the length check, a length violation indication is stored in the fault parameter block. For a description of the fault parameter block refer to Section 2.5.4.2.

2.5 EXCEPTIONS AND INTERRUPTS

At certain times during the operation of a system, events within the system require the execution of particular pieces of software outside the explicit flow of control. The processor transfers control by forcing a change in the flow of control from that explicitly indicated in the currently executing process.

Some of the events are relevant primarily to the currently executing process, and normally invoke software in the context of the current process. The notification of such events is termed an exception.

Other events are primarily relevant to other processes, or to the system as a whole, and are therefore serviced in a system-wide context. The notification process for these events is termed an interrupt, and the system-wide context is described as "executing on the interrupt stack" (IS). Further, some interrupts are of such urgency that they require high-priority service, while others must be synchronized with independent events. To meet these needs, the processor has priority logic that grants interrupt service to the highest priority event at any point in time. The priority associated with an interrupt is termed its interrupt priority level (IPL).

2.5.1 Processor Interrupt Priority Levels (IPL)

The VAX architecture has 31 interrupt priority levels (IPL), divided into 15 software levels (numbered, in hex, 01 to 0F), and 16 hardware levels (10 to 1F, hex). User applications, system calls, and system services all run at process level, which may be thought of as IPL 0. Higher numbered interrupt levels have higher priority, that is, any requests at an interrupt level higher than the processor's current IPL will interrupt immediately but requests at a lower or equal level are deferred.

The interrupt levels implemented by the MicroVAX 78032 CPU are listed in Table 2-8.

Table 2-8 Interrupt Priority Levels

IPL levels (hex)	interrupt condition
1F	unused
1E	$\overline{\text{PWRFL}}$ asserted
18 - 1D	unused
17	$\overline{\text{IRQ<3>}}$ asserted
16	$\overline{\text{INTTIM}}$ asserted
16	$\overline{\text{IRQ<2>}}$ asserted
15	$\overline{\text{IRQ<1>}}$ asserted
14	$\overline{\text{IRQ<0>}}$ asserted
10 - 13	unused
01 - 0F	software interrupt request

2.5.2 Processor Status

When an exception or an interrupt is serviced, the processor status must be preserved so that the interrupted process may continue normally. This is done by automatically saving the Program Counter (PC) and the Processor Status Longword (PSL). These are later restored with the Return from Exception or Interrupt instruction (REI). Any other status required to correctly resume an interruptable instruction is stored in the general registers. Process context such as mapping information is not saved or restored on each interrupt or exception. Instead, it is saved and restored only when process context switching is performed.

2.5.3 Interrupts

The processor arbitrates interrupt requests according to priority. Only when the priority of an interrupt request is greater than the current IPL (bits <20:16> of the Processor Status Longword) will the processor raise the IPL and service the interrupt request. The interrupt service routine is entered at the IPL of the interrupt request, or at IPL17 if the vector supplied by the interrupting device has bit <00> = 1.

Interrupt requests can come from devices, controllers, or the processor itself. Software executing in kernel mode can raise and lower the priority of the processor by executing MTPR src,#IPL, where src contains the new priority desired.

The processor services interrupt requests between instructions. The processor also services interrupt requests at well defined points during the execution of long, iterative instructions such as the string instructions. For these instructions, in order to avoid saving additional instruction state in memory, interrupts are initiated when the instruction state can be completely contained in the registers, PSL, and PC.

The following events cause interrupts:

1. Interrupt from a peripheral device received on $\overline{\text{IRQ<3:0>}}$ (IPL14 to 17 hex)

2. $\overline{\text{HALT}}$ (non-maskable)

3. Power fail (IPL1E hex)

4. Interval timer (IPL16 hex)

5. Software interrupt invoked by MTPR src,#SIRR (IPL01 to 0F hex)

6. AST delivery when REI restores a PSL with mode greater than or equal to ASTLVL (IPL 02 hex)

Each device has a separate interrupt vector location in the system control block (SCB). Thus interrupt service routines do not need to poll devices in order to determine which device interrupted. The vector address for each device is determined by hardware.

In order to reduce interrupt overhead, no memory mapping information is changed when an interrupt occurs. Thus the instructions, data, and contents of the interrupt vector for an interrupt service routine must be in the system address space or present in every process at the same address.

2.5.3.1 Urgent Interrupts -- Levels 18-1F (Hex) -

The VAX architecture has 8 priority levels for use by urgent conditions including serious errors (e.g., machine check) and power fail. Interrupts on these levels are initiated by the processor upon detection of certain conditions. Some of these conditions are not interrupts, but are exceptions that are run at IPL1F (hex).

2.5.3.2 Device Interrupts -- Levels 10-17 (Hex) -

The VAX architecture has 8 priority levels for use by peripheral devices. Of these 8 priority levels the MicroVAX 78032 CPU implements four, levels 14 through 17 (hex). The processor receives device interrupt requests via $\overline{\text{IRQ<3:0>}}$ and $\overline{\text{INTTIM}}$.

2.5.3.3 Software Generated Interrupts -- Levels 01-0F (Hex) -

The processor has 15 interrupt levels for use by software. These levels are 01 through 0F (hex). These interrupts are used by system software to generate software controlled interrupts.

2.5.3.4 Interrupt Control -

The hardware interrupt system is controlled by the $\overline{\text{IRQ<3:0>}}$, $\overline{\text{HALT}}$, $\overline{\text{PWRFL}}$, and $\overline{\text{INTTIM}}$ inputs to the processor along with three registers. Asserting any of the input pins results in an interrupt being generated at the hardware level given in Table 2-8. The three registers are used to control the software interrupt system.

2.5.3.4.1 Software Interrupt Summary Register -

The Software Interrupt Summary Register (SISR), Figure 2-29, is a privileged register that records pending software interrupts. It contains 1's in the bit positions corresponding to levels on which software interrupts are pending. All such levels, of course, must be lower than the current processor IPL, or the processor would have taken the requested interrupt.

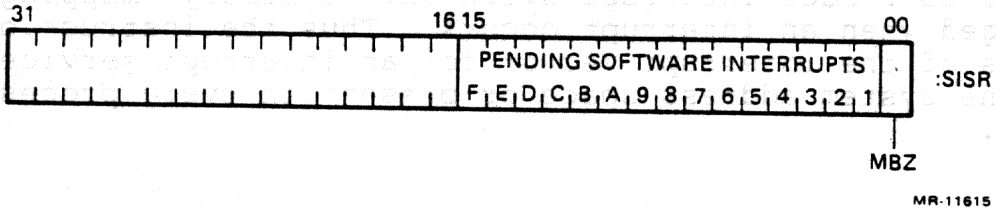


Figure 2-29 Software Interrupt Summary Register

2.5.3.4.2 Software Interrupt Request Register -

The software interrupt request register (SIRR), Figure 2-30, is a write-only four bit privileged register used for making software interrupt requests.

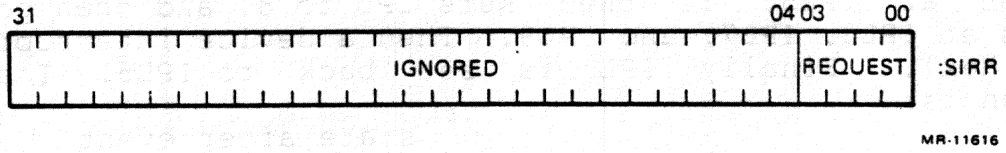


Figure 2-30 Software Interrupt Request Register

Executing MTPR src,#SIRR requests an interrupt at the level specified by src<3:0>. Once a software interrupt request is made, it will be cleared by the hardware when the interrupt is taken. If src<3:0> is greater than the current IPL, the interrupt occurs before execution of the following instruction. If src<3:0> is less than or equal to the current IPL, the interrupt will be deferred until the IPL is lowered to less than src<3:0> and there is no higher level interrupt pending.

2.5.3.5 Interrupt Priority Level Register -

Writing to the IPL, Figure 2-31, with the MTPR instruction will load the processor priority field in the Program Status Longword (PSL), that is, PSL<20:16> is loaded from IPL<4:0>. Reading from IPL with the MFPR instruction will read the processor priority field from the PSL.

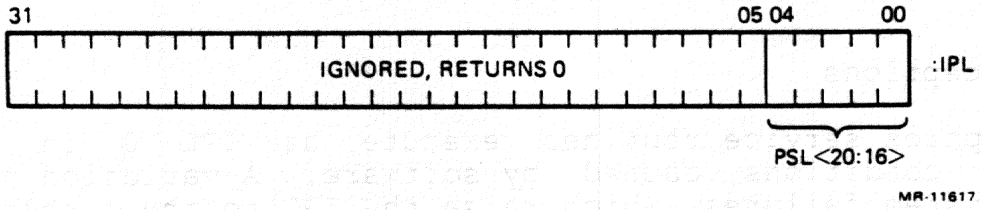


Figure 2-31 Interrupt Priority Level Register

Interrupt service routines must follow the discipline of not lowering IPL below their initial level. If they do, an interrupt at an intermediate level could cause the stack nesting to be improper. Actually, a service routine could lower the IPL if it ensures that no intermediate levels could interrupt. However, this would result in unreliable code.

2.5.3.6 Interrupt Example -

As an example, assume the processor is running in response to an interrupt at IPL5, it then sets IPL to 8, and then posts software requests at IPL3, IPL7, and IPL9. Then a device interrupt arrives at IPL16 (hex). Finally IPL is set back to IPL5. The sequence of execution is:

	state after event		
event	IPL (hex)	SISR (hex)	IPL in PSL on stack
(initial)	5	0	0
MTPR #8,#IPL	8	0	0
MTPR #3,#SIRR	8	8	0
MTPR #7,#SIRR	8	88	0
MTPR #9,#SIRR interrupts to	9	88	8,0
device interrupts to	16	88	9,8,0
device service routine REI	9	88	8,0
IPL9 service routine REI	8	88	0
MTPR #5,#IPL changes IPL to 5 and the request for 7 is granted immediately	7	8	5,0
IPL7 service routine REI	5	8	0
initial IPL5 service routine REI back to IPL0 and the request for 3 is granted immediately	3	0	0
IPL3 service routine REI	0	0	--

2.5.4 Exceptions

Most exception service routines execute at IPL 0 in response to exception conditions caused by software. A variation from this are serious system failures, which raise the IPL to the highest level (1F, hex) to minimize processor interruption until the problem is corrected. Exception service routines are usually coded to avoid exceptions; however, nested exceptions can occur.

A trap is an exception condition that occurs at the end of the instruction that caused the exception. Therefore the PC saved on the stack is the address of the next instruction that would normally have been executed. Software can enable and disable some of the trap conditions with a single instruction.

A fault is an exception condition that occurs during an instruction, and leaves the registers and memory in a consistent state such that elimination of the fault condition and restarting the instruction will give correct results. Note that faults do not always leave everything as it was prior to the faulted instruction; they only restore enough to allow restarting. Thus, the state of a process that faults may not be the same as that of a process that was interrupted at the same point.

An abort is an exception condition that occurs during an instruction, leaving the value of registers and memory unpredictable, such that the instruction cannot necessarily be correctly restarted, completed, simulated, or undone. After an instruction aborts, the PC addresses the opcode of the aborted instruction. The following are unpredictable:

- destination operands (including implied operands, such as the top of the stack in an JSB instruction)
- registers modified by operand specifier evaluation (including specifiers for implied operands)
- the PTE<M> bit in PTEs that map destination operands, if the operands could have been written but were not written, and PTE<M> was clear before the instruction
- condition codes
- PSL<FPD>
- PSL<TP>

Except where otherwise noted in the description of the abort, the rest of the PSL, other registers, and memory are unaffected.

The MicroVAX 78032 CPU recognizes six types of exceptions, as summarized in Table 2-9.

Table 2-9 Summary of Exceptions

exception class	instances
arithmetic traps/faults	integer overflow trap
	integer divide by zero trap
	subscript range trap
	floating overflow fault
	floating divide by zero fault
	floating underflow fault
memory management exceptions	access control violation fault
	translation not valid fault
operand reference exceptions	reserved addressing mode fault
	reserved operand fault or abort
instruction execution exceptions	reserved/privileged instruction fault
	emulated instruction fault
	extended function fault
	breakpoint fault
tracing exception	trace fault
system failure exceptions	memory read error abort
	memory write error abort
	kernel stack not valid abort
	interrupt stack not valid halt
	machine check abort

2.5.4.1 Arithmetic Traps/Faults -

The various exceptions that occur as the result of an arithmetic or conversion operation are mutually exclusive and are assigned the same vector in the SCB. Each indicates that an exception had occurred during the last instruction and that the instruction has been completed (trap) or backed up (fault). A code unique to each exception type is then pushed on the stack as a longword. The stack after an arithmetic exception is shown in Figure 2-32. Table 2-10 lists the type codes.

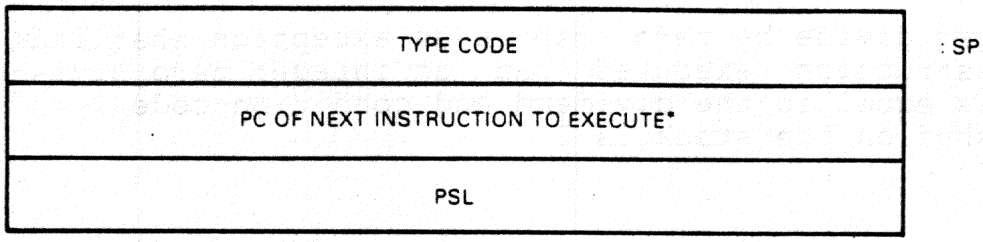


Figure 2-32 Stack After Arithmetic Exception

Table 2-10 Arithmetic Exception Type Codes

type code (hex)	exception type
	TRAPS
1	integer overflow
2	integer divide by zero
7	subscript range
	FAULTS
8	floating overflow
9	floating divide by zero
A	floating underflow

2.5.4.1.1 Integer Overflow Trap -

An integer overflow trap is an exception that indicates that the last instruction executed had an integer overflow setting the V condition code and that integer overflow was enabled (IV set). The result stored is the low-order part of the correct result. N and Z are set according to the stored result. The type code pushed on the stack is 1. Note that the instructions RET, REI, REMQUE, REMQHI, REMQTI, and BISPSW do not cause overflow even if they set V. Also note that the EMODx floating point instructions can cause integer overflow.

2.5.4.1.2 Integer Divide By Zero Trap -

An integer divide by zero trap is an exception that indicates that the last instruction executed had an integer zero divisor. The result stored is equal to the dividend and condition code V is set. The type code pushed on the stack is 2.

2.5.4.1.3 Subscript Range Trap -

A subscript range trap is an exception that indicates that the last instruction was an INDEX instruction with a subscript operand that failed the range check. The value of the subscript operand is lower than the low operand or greater than the high operand. The result is stored in indexout, and the condition codes are set as if the subscript were within range. The type code pushed on the stack is 7.

2.5.4.1.4 Floating Overflow Fault -

A floating overflow fault is an exception that indicates that the last instruction executed resulted in an exponent greater than the largest representable exponent for the data type after normalization and rounding. The destination was unaffected and the saved condition codes are unpredictable. The saved PC points to the instruction causing the fault. In the case of a POLY instruction, the instruction is suspended with FPD set. The type code pushed on the stack is 8.

2.5.4.1.5 Floating Divide By Zero Fault -

A floating divide by zero fault is an exception that indicates that the last instruction executed had a floating zero divisor. The quotient operand was unaffected and the saved condition codes are unpredictable. The saved PC points to the instruction causing the fault. The type code pushed on the stack is 9.

2.5.4.1.6 Floating Underflow Fault -

A floating underflow fault is an exception that indicates that the last instruction executed resulted in an exponent less than the smallest representable exponent for the data type after normalization and rounding and that floating underflow was enabled (FU set). The destination operand is unaffected and the saved condition codes are unpredictable. The saved PC points to the instruction causing the fault. In the case of a POLY instruction, the instruction is suspended with FPD set. The type code pushed on the stack is 10.

2.5.4.2 Memory Management Exceptions -

The two exceptions that occur as a result of memory management operations use separate vectors in the SCB but push the same information onto the stack. The information pushed on the stack is called the fault parameter block and is shown in Figure 2-33.

The same parameters are stored in the fault parameter block for both types of memory management fault. The first parameter pushed on the kernel stack after the PSL and PC is a virtual address that is located in the same page as the virtual address that caused the fault. Note that a process space reference can result in a system space virtual reference for the PTE. If the reference to the per-process PTE faults, the virtual address that is saved is the process virtual address and a 1 is stored in bit 1 of the fault parameter word. The second parameter pushed on the kernel stack is the fault parameter word and contains information related to the type of memory management violation, PTE reference, and type of memory access (read, write, etc.). Table 2-11 gives a description of the bits used in the fault parameter word.

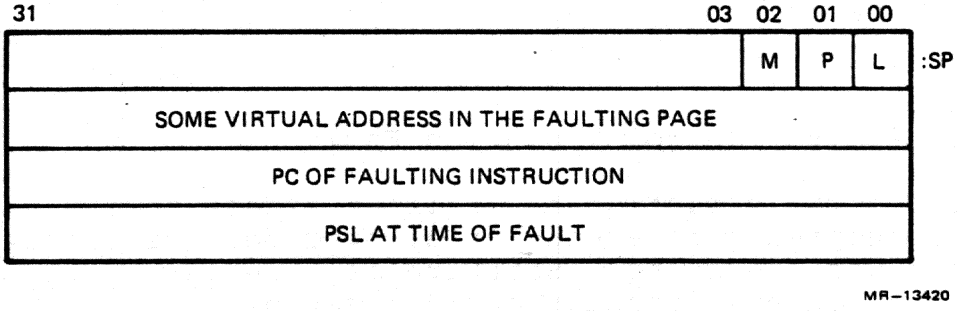


Figure 2-33 Fault Parameter Block

Table 2-11 Fault Parameter Word Bit Descriptions

Bit	Field	Description
<31:03>		Unused.
<2>	M	Write or Modify Intent - this bit is set to 1 to indicate that the intended access was a write or modify. This bit is 0 if the programs intended access was a read.
<1>	P	PTE Reference - this bit is set to 1 to indicate that the fault occurred during the reference to the process page table associated with the virtual address. This bit can be set on either length or protection faults.
<0>	L	Length Violation - this bit is set to 1 to indicate that an Access Control Violation was the result of a length violation rather than a protection violation. This bit is always 0 for a Translation Not Valid Fault.

2.5.4.2.1 Access Control Violation Fault -

An access control violation fault is an exception indicating that the process attempted a reference not allowed for the access mode at which the process was operating. Software may restart the process after changing the address translation information.

2.5.4.2.2 Translation Not Valid Fault -

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 was not set. Note that 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.

2.5.4.3 Operand Reference Exceptions -

2.5.4.3.1 Reserved Addressing Mode Fault -

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.

2.5.4.3.2 Reserved Operand Exception -

A reserved operand exception indicates that an operand accessed has a format reserved for future use by DIGITAL. No parameters are pushed. 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. Note that only the changes made by instruction fetch and because of operand specifier evaluation may be restored. Therefore, some instructions are not restartable. These exceptions are labeled as 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.

2.5.4.4 Instruction Execution Exceptions -

2.5.4.4.1 Reserved/Privileged Instruction Fault -

A reserved/privileged instruction fault occurs when the processor encounters an opcode that is not specifically defined, or that requires higher privileges than the current mode. No parameters are pushed. Opcode FFFF (hex) will always fault.

2.5.4.4.2 Emulated Instruction Fault -

An emulated instruction fault occurs when an opcode that has microcode assistance for instruction emulation is encountered by the processor. Section 4.12 describes microcode assistance for emulated instructions.

2.5.4.4.3 Extended Function Fault -

An extended instruction fault occurs when an opcode reserved to the user is executed. All opcodes reserved to the user start with FC (hex), which is the XFC instruction. No parameters are pushed on the stack.

2.5.4.4.4 Breakpoint Fault -

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

To proceed from a breakpoint, a debugger or tracing program typically restores the original contents of the location containing the BPT, sets T in the PSL saved by the BPT fault, and resumes. When the breakpointed instruction completes, a trace exception will occur. At this point, the tracing program can again re-insert the BPT instruction, restore T to its original state (usually clear), and resume. Note that if both tracing and breakpointing are in progress (i.e., if PSL<T> was set at the time of the BPT), then on the trace exception both the BPT restoration and a normal trace exception should be processed by the trace handler.

2.5.4.5 Tracing -

A trace trap is an exception that occurs between instructions when trace is enabled. Tracing is used for tracing programs, for performance evaluation, or debugging purposes. It is designed so that one and only one trace exception 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. If a trace fault and a memory management fault occur simultaneously, the order in which the exceptions are taken is unpredictable. The trace fault for an instruction takes precedence over all other exceptions.

In order 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). Instead of the PSL<T> bit being defined to produce a trap after any other traps or aborts at the end of an instruction, the trap effect is implemented by copying PSL<T> to a second bit (PSL<TP>) that is actually used to generate the exception. PSL<TP> generates a fault before any other processing at the start of the next instruction.

2.5.4.6 System Failure Exceptions -

2.5.4.6.1 Kernel Stack Not Valid Abort -

Kernel stack not valid abort is an exception that indicates that the kernel stack was not valid while the processor was pushing information onto the kernel stack 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 PSL and PC. Bits <1:0> of the exception vector should be 1, if they are 0, 2, or 3, the operation of the processor is UNDEFINED. Software may abort the process without aborting the system. However, because of the lost information, the process cannot be continued. 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.

2.5.4.6.2 Interrupt Stack Not Valid Halt -

An interrupt stack not valid halt is an exception that indicates that the interrupt stack was not valid while the processor was pushing information onto the interrupt stack during the initiation of an exception or interrupt. No further interrupt requests are acknowledged. The processor leaves the PC, the PSL, the ISP, and the reason for the halt in privileged registers (SAVISP, SAVPC, SAVPSL) and initiates the restart process, see Section 2.8.

2.5.4.6.3 Machine Check And Memory Read/Write Error Abort -

A machine check or memory read/write error abort indicates that the processor detected an internal error in itself or a memory read/write bus error ($\overline{\text{ERR}}$ asserted). Bits <1:0> of the exception vector should be equal to 1; if they are equal to 0, 2, or 3, the operation of the processor is UNDEFINED.

The parameters pushed on the stack are shown in Figure 2-34.

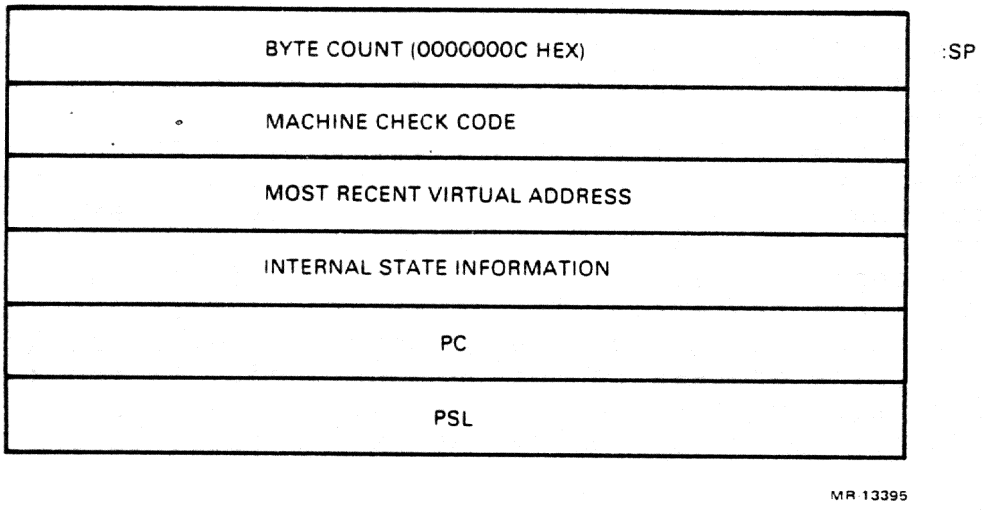


Figure 2-34 Machine Check Stack Parameters

The parameters are:

machine check code (hex):
- 1 = impossible microcode state (FSD)
- 2 = impossible microcode state (SSD)
- 3 = undefined FPU error code 0
- 4 = undefined FPU error code 7
- 5 = undefined memory management status (TB miss flows)
- 6 = undefined memory management status (M = 0 flows)
- 7 = process PTE address in P0 space
- 8 = process PTE address in P1 space
- 9 = undefined interrupt ID code
- 80 = read bus error, VAP is virtual address
- 81 = read bus error, VAP is physical address
- 82 = write bus error, VAP is virtual address
- 83 = write bus error, VAP is physical address

most recent virtual address:
- <31:0> = current contents of VAP register (possibly incremented by 4)

internal state information:
- <28:24> = current contents of ATDL register
- <23:20> = current contents of STATE<3:0>
- <19:16> = current contents of ALU cond codes
- <14> = current contents of VAX CANT RESTART bit
- <7:0> = delta PC at the time of the exception

PC:
- <31:0> = PC at the start of the current instruction

PSL:
- <31:0> = current contents of PSL

2.5.5 Contrast Between Exceptions And Interrupts

Exceptions and interrupts are very similar. When either is initiated, both the processor status longword (PSL) and the program counter (PC) are pushed onto the stack. However there are seven important differences:

1. An exception condition is caused by the execution of the current instruction while an interrupt is caused by some activity in the computing system that may be independent of the current instruction.

2. An exception condition is usually serviced in the context of the process that produced the exception condition, while an interrupt is serviced independently from the currently running process.

3. The IPL of the processor is usually not changed when the processor initiates an exception, while the IPL is always raised when an interrupt is initiated.

4. Exception service routines usually execute on a per-process stack while interrupt service routines normally execute on a per-CPU stack.

5. Enabled exceptions are always initiated immediately no matter what the processor IPL is, while interrupts are held off until the processor IPL drops below the IPL of the requesting interrupt.

6. Most exceptions can not be disabled. However, if an exception causing event occurs while that exception is disabled, no exception is initiated for that event even when enabled subsequently. This includes overflow which is the only exception whose occurrence is indicated by a condition code (V). If an interrupt condition occurs while it is disabled, or the processor is at the same or higher IPL, the condition will eventually initiate an interrupt when the proper enabling conditions are met if the condition is still present.

7. The previous mode field in the PSL is always set to kernel on an interrupt, but on an exception it indicates the mode at the time of the exception.

2.5.6 Serialization Of Exceptions And Interrupts

The sequence in which recognition of simultaneously occurring interrupts and exceptions takes place is:

1. Machine check exception.
2. Arithmetic exceptions.
3. Console Halt
4. Interrupts at a higher priority (IPL) than the current processor priority.
5. Trace fault (only one per instruction).
6. Start instruction execution or restart suspended instruction.

2.5.7 Initiate Exception Or Interrupt

The following operation describes the action taken by the MicroVAX 78032 CPU when initiating an exception or interrupt. For a description of the notation used to describe the operation refer to Section 4.1.3.

Operation:

```
!The notation PSL<xxx>_SP is used to refer
!to the SP appropriate to the mode xxx specified
!in the PSL.

{disable interrupts};
tmpl <- SCB[vector];                        !get correct vector
if tmpl<1:0> EQLU 3 then {UNDEFINED};
if tmpl<1:0> EQLU 0 AND {machine check or
        kernel stack not valid} then {UNDEFINED};
if tmpl<1:0> NEQU 0 AND {CHMx} then {UNDEFINED};
if tmpl<1:0> EQLU 2 then {UNDEFINED};
if PSL<IS> EQLU 0 then                      !switch stacks
        begin
        PSL<CUR_MOD>_SP <- SP;              !save old SP
        if tmpl<1:0> EQLU 1 then
                SP <- ISP;
                else
                SP <- new_mode_SP;          !kernel_SP unless CHMx
        end;
tmp2 <- PSL;
PSL<CM,TP,FPD,DV,FU,IV,T,N,Z,V,C> <- 0;     !cleanout PSL
if {interrupt} then
        PSL<PRV_MOD> <- 0;                  !kernel mode
```

```
            else
            PSL<PRV_MOD> <- PSL<CUR_MOD>;
    PSL<CUR_MOD> <- new_mode;                   !kernel mode unless CHMx
    -(SP) <- tmp2;                              !on a fault or abort, the
                                                !saved condition codes are
                                                !unpredictable as backed up
    -(SP) <- PC;                                !if necessary
    {push parameters if any};
                                                !if kernel stack not valid
                                                !exception occurs while
                                                !pushing tmp2, PC, or other
                                                !parameters then
                                                !PSL <- tmp2 before
                                                !initiating exception
    if {interrupt} then
            PSL<IPL> <- new_IPL                 !set new IPL
            else
            if tmp1<1:0> EQLU 1 then
                    PSL<IPL> <- 31;             !1F (hex)
    if tmp1<1:0> EQLU 1 then PSL<IS> <- 1;      !otherwise keep old IS
    PC <- tmp1<31:2> ' 0<1:0>;                  !longword aligned
    {enable interrupts};
    if {PSL<IPL> LEQU 15} AND {PSL<IPL> GEQU 1} then
            SISR<PSL<IPL>> <- 0;                !clear SISR bit for
                                                !software interrupt
                                                !being initiated
```

Condition Codes (if vector<1:0> code is 0 or 1):

```
N <- 0;
Z <- 0;
V <- 0;
C <- 0;
```

Exceptions:

interrupt stack not valid halt
kernel stack not valid abort

Description:

The handling is determined by the contents of a longword vector in the system control block which is indexed by the exception or interrupt being processed. If the processor is not executing on the interrupt stack, then the current stack pointer is saved and the new stack pointer is fetched. The old PSL is pushed onto the new stack. The PC is backed up (unless this is an interrupt between instructions or a trap) and is pushed onto the new stack. The PSL is set to a known state. IPL is changed if this is an interrupt or if it is an exception with vector<1:0> code 1. Any parameters are pushed. Except for interrupts, the previous mode in the new PSL is set to the old value of the current mode. Finally, the PC is changed to point to the longword indicated by the vector<31:2>.

Notes:

1. Interrupts are disabled during this sequence.

2. If the vector<1:0> code is invalid, the behavior is undefined.

3. On a fault or interrupt, the saved condition codes are unpredictable; they are only saved to the extent necessary to ensure correct completion of the instruction when resumed.

4. After an abort, the following are unpredictable:

   a. Destination operands, including implied operands such as the top of the stack during a JSR instruction.

   b. Registers modified by operand specifier evaluation, including registers modified by referencing implied operands.

   c. Condition codes.

   d. PSL<FPD>.

   e. PSL<TP>.

   f. The page table entry Modify bit for pages mapping write or modify type operands. The Modify bit will be set if the instruction modified the page. If the instruction did not modify the page, the Modify bit is unpredictable.

   After an abort, the PC pushed on the stack addresses the instruction which aborted, unless the instruction modified the PC in a way that produces unpredictable results. The other registers, other bits of the PSL, and the rest of memory are unaffected by an abort.

5. After an abort or fault or interrupt that pushes a PSL with FPD set, the general registers except PC, SP, and FP are unpredictable unless the instruction description specifies a setting. If FP is the destination in this case, then it is also unpredictable. On a kernel stack not valid abort, both SP and FP are unpredictable.

6. If the processor gets an Access Control Violation or a Translation Not Valid condition while attempting to push information on the kernel stack, a kernel stack not valid abort is initiated. The additional information, if any, associated with the original exception is lost. However, the PSL and PC are pushed on the interrupt stack with the same values as would have been pushed on the kernel stack, and the IPL is changed to 1F (hex). If vector<1:0> for the kernel stack not valid abort is 0, the operation of the processor is undefined.

7. If the processor gets an Access Control Violation or a Translation Not Valid condition while attempting to push information on the interrupt stack, the processor is halted and only the state of the ISP, PC, and PSL is ensured to be correct. The PSL and PC have the values that would have been pushed on the interrupt stack.

8. The value of PSL<TP> that is saved on the stack is as follows:

fault	clear
trace	clear
interrupt	clear (if FPD set) from PSL<TP> (if after traps, before trace)
abort	unpredictable
trap	from PSL<TP>
CHMx	from PSL<TP>
BPT,XFC	clear
reserved instruction	clear

9. The value of PC that is saved on the stack points to the following:

fault	instruction faulting
trace	next instruction to execute i.e., instruction at the beginning of which the trace fault was taken
interrupt	instruction interrupted or next instruction to execute
abort	instruction aborting or detecting kernel stack not valid (not ensured on machine check)
trap	next instruction to execute
CHMx	next instruction to execute
BPT, XFC	BPT, XFC instruction
reserved instruction	reserved instruction

2.5.8 System Control Block (SCB)

The System Control Block is a table containing the vectors by which exceptions and interrupts are dispatched to the appropriate service routines.

2.5.8.1 System Control Block Base (SCBB) -

The SCBB is a privileged register containing the physical address of the System Control Block, which must be page-aligned, Figure 2-35.

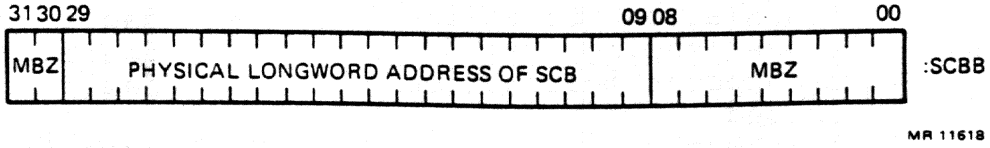


Figure 2-35 System Control Block Base Register

2.5.8.2 Vectors -

A vector is a longword in the SCB that is used by the processor when an exception or interrupt occurs to determine how to service the event. Table 2-12 lists the vectors contained in the SCB.

Separate vectors are defined for each interrupting device and each class of exceptions. Bits <1:0> of the vector are interpreted 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, the IPL is raised to 1F (hex).

2 This code results in the MicroVAX 78032 CPU entering the restart process, see Section 2.8.

3 This code results in the MicroVAX 78032 CPU entering the restart process, see Section 2.8.

Table 2-12 System Control Block Vectors

Vector (hex)	Name	Type
00	unused	-
04	machine check	abort
08	kernel stack not valid	abort
0C	power fail	interrupt
10	reserved/privileged instruction	fault
14	extended instruction	fault
18	reserved operand	fault/abort
1C	reserved addressing mode	fault
20	access control violation	fault
24	translation not valid	fault
28	trace pending (TP)	fault
2C	breakpoint instruction	fault
30	unused	-
34	arithmetic	trap/fault
38-3C	unused	-
40	CHMK	trap
44	CHME	trap
48	CHMS	trap
4C	CHMU	trap
50-80	unused	-
84	software level 1	interrupt
88	software level 2	interrupt
8C	software level 3	interrupt
90-BC	software levels 4-15	interrupt
C0	interval timer	interrupt
C4	unused	-
C8	emulation start	fault
CC	emulation continue	fault
D0-FC	unused	-
100-1FC	adapter vectors*	interrupt
200-3FC	device vectors*	interrupt

*Used by the MicroVAX 78032 CPU to directly vector interrupts from the external bus. The vector is determined from bits <9:2> of the value supplied by external hardware. If bit <0> of this value is 0, then the new IPL is forced to 17 (hex). Only device vectors in the range of 100 to 3FC (hex) should be used. Except by devices emulating console storage and terminal devices.

2.6 PROCESS STRUCTURE

A process is the basic entity that may be scheduled by the MicroVAX 78032 CPU. It consists of an address space, a hardware context, and a software context. The hardware context is defined by a data structure called the process control block (PCB), which contains images of 14 general registers, the processor status longword (PSL), the program counter (PC), the four per-process stack pointers, the process virtual memory defined by the base and length registers (P0BR, P0LR, P1BR, and P1LR), and several minor control fields. When a process is not executing, its hardware context is stored in the process control block. In order for a process to execute, the majority of the PCB must be moved into processor registers: while a process is being executed, some of its hardware context is being updated in the processor registers.

Saving the contents of the privileged registers in the PCB of the currently executing process and then loading a new set of context in the privileged registers from another PCB is termed context switching. Context switching occurs as one process after another is scheduled for execution.

2.6.1 Process Context

The process control block for the currently executing process is pointed to by the contents of the process control block base (PCBB) register, an internal privileged register, which contains the physical address of the PCB. Figure 2-36 shows the PCBB register.

The process control block contains all of the the switchable process context collected into a compact form for ease of movement to and from the privileged internal registers. Although in any normal operating system there is additional software context for each process, the following description is limited to that portion of the the PCB known to the hardware. Figure 2-37 shows the PCB and Table 2-13 describes its contents.

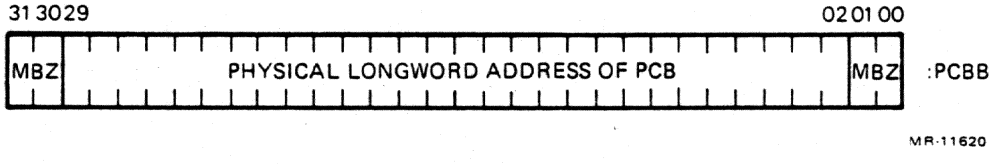
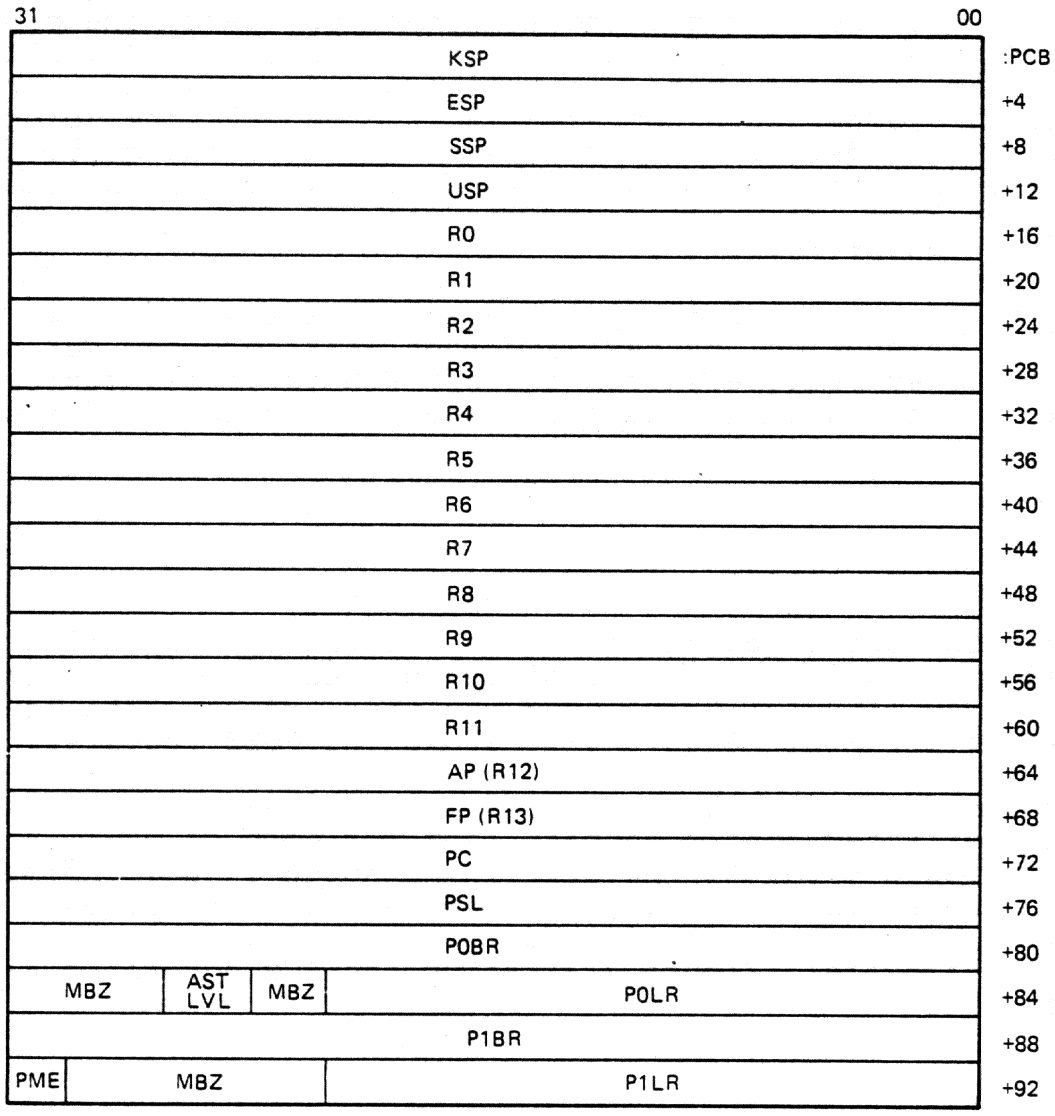


Figure 2-36 Process Control Block Base (PCBB) Register



NOTE: THE PME FIELD IS UNUSED.

MR-11619

Figure 2-37 Process Control Block (PCB)

Table 2-13 Description of Process Control Block

Longword	Bits	Mnemonic	Description
0	<31:0>	KSP	Kernel Stack Pointer. Contains the stack pointer to be used when the current access mode field in the PSL is 0 and IS = 0.
1	<31:0>	ESP	Executive Stack Pointer. Contains the stack pointer to be used when the current access mode field in the PSL is 1.
2	<31:0>	SSP	Supervisor Stack Pointer. Contains the stack pointer to be used when the current access mode field in the PSL is 2.
3	<31:0>	USP	User Stack Pointer. Contains the stack pointer to be used when the current access mode field in the PSL is 3.
4-17	<31:0>	R0-R11, AP,FP	General registers R0 through R11, AP, FP.
18	<31:0>	PC	Program Counter.
19	<31:0>	PSL	Program Status Longword.
20	<31:0>	P0BR	Base register for page table describing process virtual addresses from 0 to 2**30-1.
21	<21:0>	P0LR	Length register for page table located by P0BR. Describes effective length of page table.
21	<23:22>	MBZ	Must be zero.

Table 2-13 Description of Process Control Block (Continued)

Longword	Bits	Mnemonic	Description
21	<26:24>	ASTLVL	Contains access mode number (established by software) of the most privileged access mode for which an Asynchronous System Trap (AST) is pending. Controls the triggering of the AST delivery interrupt during REI instructions. ASTLVL — Meaning 0 — AST pending for access mode 0 (kernel) 1 — AST pending for access mode 1 (executive) 2 — AST pending for access mode 2 (supervisor) 3 — AST pending for access mode 3 (user) 4 — No pending AST 5-7 — Reserved to DIGITAL
21	<31:27>	MBZ	Must be zero.
22	<31:0>	P1BR	Base register for page table describing process virtual addresses from 2**30 to 2**31-1.
23	<21:0>	P1LR	Length register for page table located by P1BR. Describes effective length of page table.
23	<30:22>	MBZ	Must be zero.
23	<31>	PME	Unused.

A process must be executing in kernel mode to alter its P0BR, P1BR, P0LR, P1LR, or ASTLVL. It must first store the desired new value in the memory image of the PCB then move the value to the appropriate privileged register. This protocol results from the fact that these are read-only fields (for the context switch instructions) in the PCB.

The ASTLVL field of the PCB is contained in a processor privileged register when the process is running. Figure 2-38 shows the format of the AST Level Register.

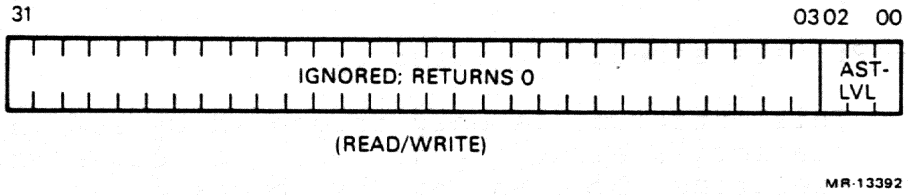


Figure 2-38 AST Level Register

2.6.2 Asynchronous System Traps (AST)

Asynchronous system traps are a technique for notifying a process of events that are not synchronized with its execution and initiating processing for asynchronous events with the least possible delay. This delay in delivery of the AST may be due to either process non-residence or an access mode mismatch. The efficient handling of AST's requires some hardware assistance to detect changes in access mode (current access mode in PSL). A process in any of the four execution access modes (kernel, executive, supervisor, and user) may receive AST's; however, an AST for a less privileged access mode must not be permitted to interrupt execution in a more protected access mode. Since outward access mode transitions occur only in the REI instruction, comparison of the current access mode field is made with a privileged register (ASTLVL) containing the most privileged access mode number for which an AST is pending. If the new access mode is greater than or equal to the pending ASTLVL, an IPL 2 software interrupt is triggered to cause delivery of the pending AST.

2.6.3 Process Structure Interrupts

Two of the 15 software interrupt priority levels are reserved for process structure software.

They are:

(IPL 2) - AST delivery interrupt.

This interrupt is triggered by a REI that detects PSL<CUR_MOD> GEQU ASTLVL and indicates that a pending AST may now be delivered for the currently executing process.

(IPL 3) - Process scheduling interrupt.

This interrupt is only triggered by software to allow the software running at IPL3 to cause the currently executing process to be blocked and the highest priority executable process to be scheduled.

2.7 STACKS

Stacks, also called pushdown lists or last-in/first-out queues, are an important feature of the MicroVAX 78032 CPU's architecture. They are used for:

- Saving the general registers, including PC, at entry to a subroutine, for restoration at exit.
- Saving PC, PSL, and general registers at the time of interrupts and exceptions, and during context switches.
- Creating storage space for temporary use, or for nesting of recursive routines.

At any time, the processor is either in a process context and in one of four access modes (kernel, executive, supervisor, or user and the interrupt stack bit (IS) of the PSL = 0) or in the system-wide interrupt service context (IS = 1) that operates with kernel privileges. There is a stack pointer (SP) associated with each of these five states. Any time the processor changes states, the SP (R14) is stored in the process context stack pointer for the old state and loaded from the one for the new state. The process context stack pointers (KSP=kernel, ESP=executive, SSP=supervisor, and USP=user) are stored in the hardware PCB with a copy of the current stack pointer in the SP (R14) register. The stack pointer values in the PCB are accessed whenever a stack pointer is switched.

2.7.1 Stack Residency

The user, supervisor, and executive stacks do not need to be resident in physical memory. The kernel can bring in or allocate process stack pages as address translation not valid faults occur. However, the kernel stack for the current process and the interrupt stack (which is process independent) must be resident and accessible.

Translation not valid and access control violation faults occurring on references to either the kernel or interrupt stack are serious failures from which recovery is impossible. If either of these faults occur on a reference to the kernel stack, the processor aborts the current sequence and initiates a kernel stack not valid abort. If either of these faults occur on a reference to the interrupt stack, the processor halts.

2.7.2 Stack Alignment

Except on CALLx instructions, the hardware makes no attempt to align the stacks. For best performance the stack should be aligned on a longword boundary and allocated in longword increments.

2.7.3 Stack Status Bits

The interrupt stack bit (IS) and current mode bits in the processor status longword (PSL) specify which of the five stack pointers is currently in use as given in Table 2-14.

The processor does not allow the current mode to be non-zero when IS = 1. This is achieved by clearing the mode bits of the PSL when taking an interrupt or exception, and by causing a reserved operand fault if return from exception or interrupt (REI) attempts to load a PSL in which both IS and the current mode are non-zero.

Table 2-14 Stack Pointer Selection

IS	Mode	Register
1	0*	ISP
0	0	KSP
0	1	ESP
0	2	SSP
0	3	USP

*Hardware will only allow a mode of 0 when the ISP is selected.

The stack to be used when servicing an interrupt or exception is selected by the IS bit of the PSL and the contents of bits <1:0> of the vector for the service routine as shown in Figure 2-39.

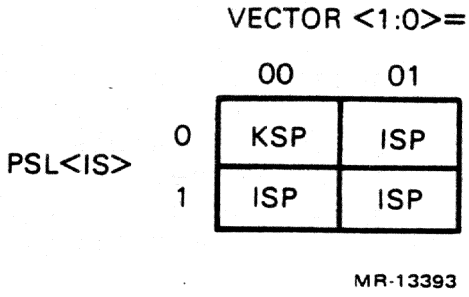


Figure 2-39 Stack Selection

2.7.4 Accessing Stack Registers

The MicroVAX 78032 CPU implements 5 privileged registers to allow access to each stack pointer. These registers always access the specified stack pointer, even for the current mode. Because the per process stack pointers are stored in the PCB, the MTPR and MFPR instructions access the hardware PCB. This means, the Process Control Block Base Register (PCBB) must contain a valid address. Table 2-15 lists the stack pointers and their related privileged register.

Table 2-15 Stack Pointer Registers

Stack Pointer	Mnemonic	Register
Kernel Stack Pointer	KSP	0
Executive Stack Pointer	ESP	1
Supervisor Stack Pointer	SSP	2
User Stack Pointer	USP	3
Interrupt Stack Pointer	ISP	4

2.8 RESTART PROCESS

The Restart process of the MicroVAX 78032 CPU is initiated when one of the following happens.

1. The $\overline{\text{RESET}}$ pin is asserted.

2. The $\overline{\text{HALT}}$ pin is asserted.

3. A HALT instruction is executed with the processor in kernel mode.

4. The hardware or kernel software environment becomes severely corrupted.

The restart process saves the current values of the PC, PSL, interrupt stack pointer, MAPEN<0>, and the restart code, in internal processor registers: SAVISP, SAVPC, and SAVPSL. See Section 2.3.3.2 for a description of these registers.

NOTE

SAVISP, SAVPC, and SAVPSL are limited life internal processor registers and must be saved in memory before re-enabling the memory management unit or using any emulated instructions.

The restart process sets the state of the processor as follows:

```
proc reg SAVISP =   saved interrupt stack pointer
proc reg SAVPC  =   saved PC
proc reg SAVPSL =   saved PSL<31:16,7:0> in SAVPSL<31:16,7:0>
                    saved MAPEN<0>       in SAVPSL<15>
                    saved restart code   in SAVPSL<14:8>
SP              =   stack pointer at time of restart
                    (NOT stack pointer specified by PSL<26:24>)
PSL             =   041F0000 (hex)
PC              =   20040000 (hex)
MAPEN           =   0
ICCS            =   0        (reset only)
SISR            =   0        (reset only)
ASTLVL          =   4        (reset only)
all else        =   undefined
```

After setting the state of the processor the restart process will start executing user code at physical address 20040000 (hex). The code there can execute MFPR's to read the saved PC, PSL and interrupt stack pointer, establish a new interrupt stack and start the console routine. Since memory management is disabled, and the current mode of the processor is kernel, the console routine has full privileges to

examine and or modify the internal state of the processor. The console routine may require an area of known good memory for scratch variables. The console routine entry and exit protocols are described in the following sections.

The reason for entering the restart process is given by the restart code that is saved in SAVPSL<14:8>. Table 2-16 gives a list of the restart codes.

Table 2-16 Restart Codes

code	condition
2	$\overline{\text{HALT}}$ asserted
3	initial power on, $\overline{\text{RESET}}$ asserted
4	interrupt stack not valid during exception
5	machine check during machine check or kernel stack not valid exception
6	HALT instruction executed in kernel mode
7	SCB vector bits<1:0> = 11
8	SCB vector bits<1:0> = 10
A	CHMx executed while on interrupt stack
10	ACV or TNV during machine check exception
11	ACV or TNV during kernel stack not valid exception

2.8.1 Console Entry Protocol

The console is entered with the state of the processor saved in internal registers. Because these are limited life registers the state of the processor is valid for a limited time only. The user must be careful not to enable memory management or use any emulated instructions until these registers are moved to memory. Therefore, the protocol for console entry is as follows:

1. Save the limited life internal registers SAVPC, SAVPSL, and SAVISP.

2. Save the current stack pointer. This is done to complete a stack swap if necessary, that is, store the SP in the KSP, ESP, SSP, or USP of the current process control block.

   A stack swap needs to be performed if the processor was not running on the interrupt stack when the restart process was entered. This is because the MicroVAX 78032 CPU stores the non-interrupt stack pointers in the PCB. If the IS bit of the saved PSL is clear (=0), the console routine must complete the

stack swap by storing the saved value of SP into the appropriate location of the PCB, if one exists. The location in the PCB that the SP is to be stored in is determined by the CUR MOD field of the saved PSL. If the IS bit of the saved PSL is set (=1), a stack swap does not have to be performed.

3. Set up the console stack pointer.

4. Begin the console routine.

A typical console entry routine and description can be found in Appendix C of this user's guide.

2.8.2 Console Exit Protocol

Exiting from the console depends on two things:

1. If memory management is to be enabled, the environment to which the console is exiting must have a validly mapped interrupt stack with at least two spare longwords at the bottom. The user's console code will have to verify this by simulating the memory management process, thereby proving that the interrupt stack is resident and points to valid physical memory. If the interrupt stack is not valid, the exit sequence must be aborted.

2. The REI which restores the PC and PSL must be in the same physical longword that contains the instruction that sets the MME bit in the MAPEN register.

The protocol for exiting from the console is as follows:

1. Push the saved PC and PSL onto the (mapped) bottom of the SAVED interrupt stack.

2. Enable memory mapping, if appropriate.

3. Exit via an REI.

A typical console exit routine with memory management simulation can be found in Appendix C of this user's guide.

CHAPTER 3

INSTRUCTION FORMAT AND ADDRESSING MODES

3.1 INSTRUCTION FORMAT

The VAX instruction set has a variable length instruction format which may be as short as one byte and as long as needed depending on the type of instruction. The general format of a VAX instruction is shown in Figure 3-1. Each instruction is made up of an opcode followed by zero to six operand specifiers. The number and type of operand specifiers is dependent on the opcode. All operand specifiers are of the same format, an address mode plus additional information used to locate the operand. This additional information contains up to two register designators and addresses, data, or displacements. The operand usage is determined implicitly from the opcode and is called the operand type. It includes both the access type and the data type.

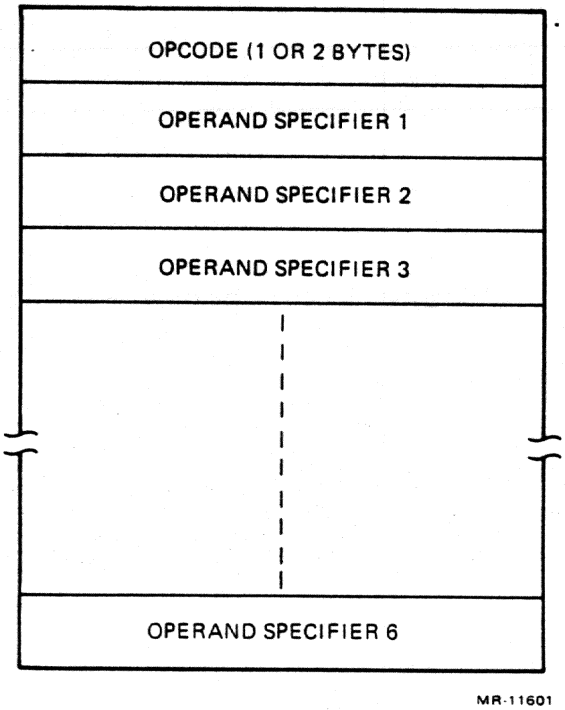


Figure 3-1 MicroVAX Instruction Format

3.1.1 Assembler Radix Notation

The radix of the assembler is decimal. To express a hexadecimal number in assembler notation, it is required to precede the number by ^X. For example, the assembler interprets the 3456 in "MOVW #3456,-(SP)" as a decimal number. If it is to be interpreted as a hexadecimal number, it would be written "MOVW #^X3456,-(SP)".

3.1.2 Operating Code

Each VAX instruction contains an operating code (opcode) which specifies the desired operation to be performed. The opcode may be one or two bytes long, depending on the contents of the byte at address A. The opcode is two bytes long if the value of the byte at address A is FD (hex). Figure 3-2 shows the opcode format.

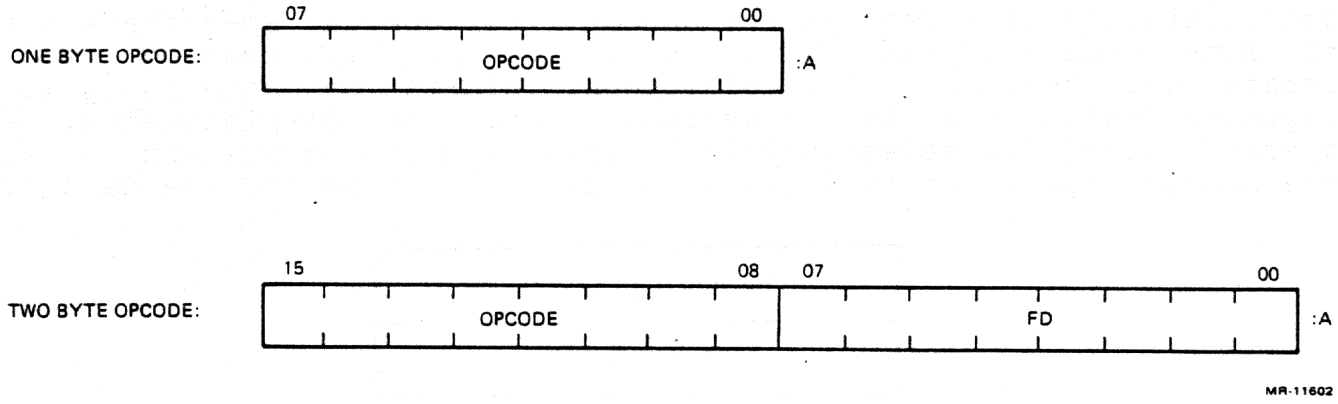


Figure 3-2 Opcode Formats

3.1.3 Operand Type

The operand type specifies how the operand associated with an instruction is used. Information provided by the opcode includes the data type of each operand and how it is accessed.

An operand may be accessed in one of 6 ways.

1. Read - The specified operand is read-only.

2. Write - The specified operand is write-only.

3. Modify - The specified operand is read, may or may not be modified, and is written.

4. Address - Address calculation occurs until the actual address of the operand is obtained. In this mode, the data type indicates the operand size to be used in address calculation. The specified operand is not accessed directly, although the instruction may use the address to access that operand.

5. Variable bit field base address - If just R[n] is specified, the field is in general register R[n] or in R[n+1]'Rn, that is R[n+1] concatenated with R[n]. Otherwise address calculation occurs until the actual address of the operand is obtained. This address specifies the base to which the field position (offset) is applied.

6. Branch - No operand is accessed. The operand specifier is the branch displacement. In this specifier, the data type indicates the size of the branch displacement.

3.2 ADDRESSING MODES

Addressing modes can be divided into two basic categories, general mode addressing and branch addressing. A summary of the addressing modes is given in Table 3-1 and Table 3-2. A description of each mode follows.

Table 3-1 Summary of General Register Addressing Modes

Mode (hex)	Name	Assembler Notation	Access r	m	w	a	v	PC	SP	Indexable
0-3	literal	S^#literal	y	f	f	f	f	-	-	f
4	index	i[Rx]	y	y	y	y	y	f	y	f
5	register	Rn	y	y	y	f	y	u	uq	f
6	register deferred	(Rn)	y	y	y	y	y	u	y	y
7	autodecrement	-(Rn)	y	y	y	y	y	u	y	ux
8	autoincrement	(Rn)+	y	y	y	y	y	p	y	ux
9	autoincrement deferred	@(Rn)+	y	y	y	y	y	p	y	ux
A	byte displacement	B^d(Rn)	y	y	y	y	y	p	y	y
B	byte displacement deferred	@B^d(Rn)	y	y	y	y	y	p	y	y
C	word displacement	W^d(Rn)	y	y	y	y	y	p	y	y
D	word displacement deferred	@W^d(Rn)	y	y	y	y	y	p	y	y
E	longword displacement	L^d(Rn)	y	y	y	y	y	p	y	y
F	longword displacement deferred	@L^d(Rn)	y	y	y	y	y	p	y	y

Addressing Legend

Access:

r	=	read
m	=	modify
w	=	write
a	=	address
v	=	field

Assembler Notation Syntax:

i	=	any indexable address mode
d	=	displacement
Rn	=	general register, n = 0 to 15
Rx	=	general register, x = 0 to 14
B	=	byte
W	=	word
L	=	longword

Results:

y	=	yes, always valid address mode
f	=	reserved address mode fault
-	=	logically impossible
p	=	program counter addressing
u	=	unpredictable
uq	=	unpredictable for quad, D/G_floating, or field if pos + size > 32
ux	=	unpredictable if index reg = base reg

Table 3-2 Summary of Program Counter Addressing Modes

Mode	Name	Assembler Notation	Access r	m	w	a	v	Indexable
8	Immediate	I^#constant	y	u	u	y	y	y
9	absolute	@#address	y	y	y	y	y	y
A	byte relative	B^address	y	y	y	y	y	y
B	byte relative deferred	@B^address	y	y	y	y	y	y
C	word relative	W^address	y	y	y	y	y	y
D	word relative deferred	W^address	y	y	y	y	y	y
E	longword relative	L^address	y	y	y	y	y	y
F	longword relative deferred	L^address	y	y	y	y	y	y

Addressing Legend

Access:

r = read
m = modify
w = write
a = address
v = field

Assembler Notation Syntax:

i = any indexable address mode
d = displacement
Rn = general register, n = 0 to 15
Rx = general register, x = 0 to 14
B = byte
W = word
L = longword

Results:

y = yes, always valid address mode
f = reserved address mode fault
- = logically impossible
p = program counter addressing
u = unpredictable
uq = unpredictable for quad, D/G_floating, or field if pos + size > 32
ux = unpredictable if index reg = base reg

3.2.1 General Mode Addressing

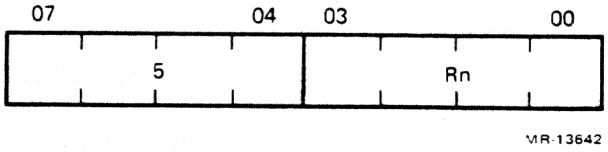
3.2.1.1 General Register Address Modes -

The general register address modes use one or more of the general registers, depending on the instruction and data type, to contain the operand(s) or information required to locate the operand(s) to be used by the specified instruction.

Register Mode (Figure 3-3)

Assembler Syntax: Rn

Mode Specifier: 5



The operand is the contents of register n (or R[n+1] concatenated with Rn for quadword, D_floating, and certain field operations):

```
Operand = Rn                    !if one register, or
          R[n+1]'Rn             !if two registers
```

Figure 3-3 Register Mode Operand Specifier Format

Description:

With register mode, any of the general registers may be used as simple accumulators and the operand is contained in the selected register. Since they are hardware registers within the processor, they provide speed advantages when used for operating on frequently accessed variables.

Special Comments:

This mode can be used with operand specifiers using read, write or modify access but cannot be used with the address access type; otherwise, an illegal addressing mode fault results. The program counter (PC) cannot be used in this mode. If the PC is read, the

value is unpredictable; if the PC is written, the next instruction executed or the next operand specified is unpredictable. Similarly, if PC is used in register mode for a write-access operand which takes two adjacent registers, the contents of R0 are unpredictable.

The stack pointer (SP) cannot be used in this mode for an operand which takes two adjacent registers since that would imply a direct reference to the PC and the results are unpredictable.

Example: REGISTER MODE, MOVE WORD INSTRUCTION

Instruction Format: MOVW R1,R2 Instruction moves a 16-bit word of data from R1 to R2.

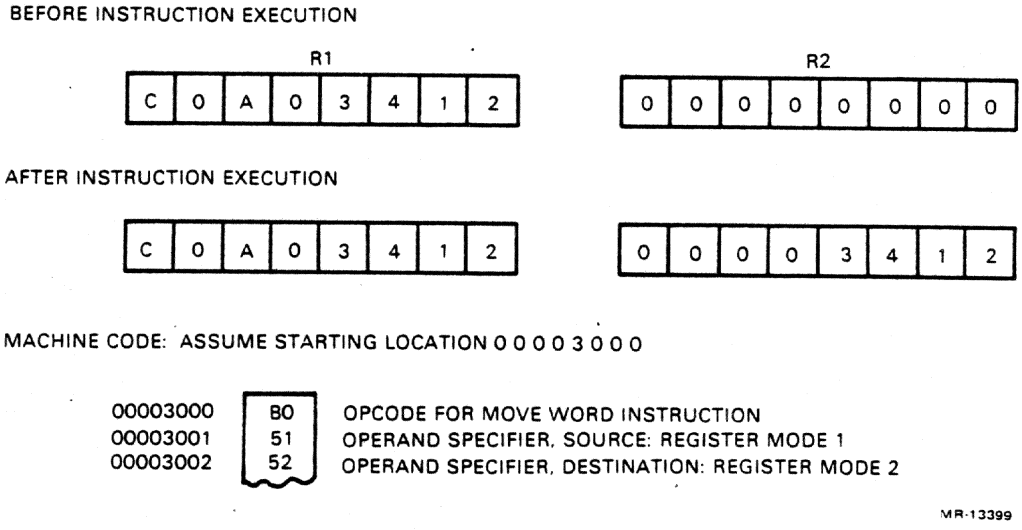


Figure 3-4 MOVW R1,R2 Move Word

This example, Figure 3-4, shows a Move Word instruction using register mode. The content of R1 is the operand. The Move Word instruction causes the least significant half of R1 to be transferred to the least significant half of register R2. The upper half of R2 is unaffected.

Register Deferred Mode (Figure 3-5)

Assembler Syntax: (Rn)

Mode Specifier: 6

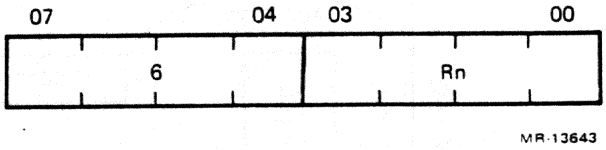


Figure 3-5 Register Deferred Mode Operand Specifier Format

Description:

The register deferred mode provides one level of indirect addressing over register mode; that is, the general register contains the address of the operand rather than the operand itself. The deferred modes are useful when dealing with an operand whose address is calculated.

Special Comments:

The PC may not be used in register deferred mode. If it is the address of the operand is unpredictable and the next instruction executed or the next operand is unpredictable.

Example: REGISTER DEFERRED MODE, CLEAR QUAD INSTRUCTION

Instruction Format: CLRQ (R4)

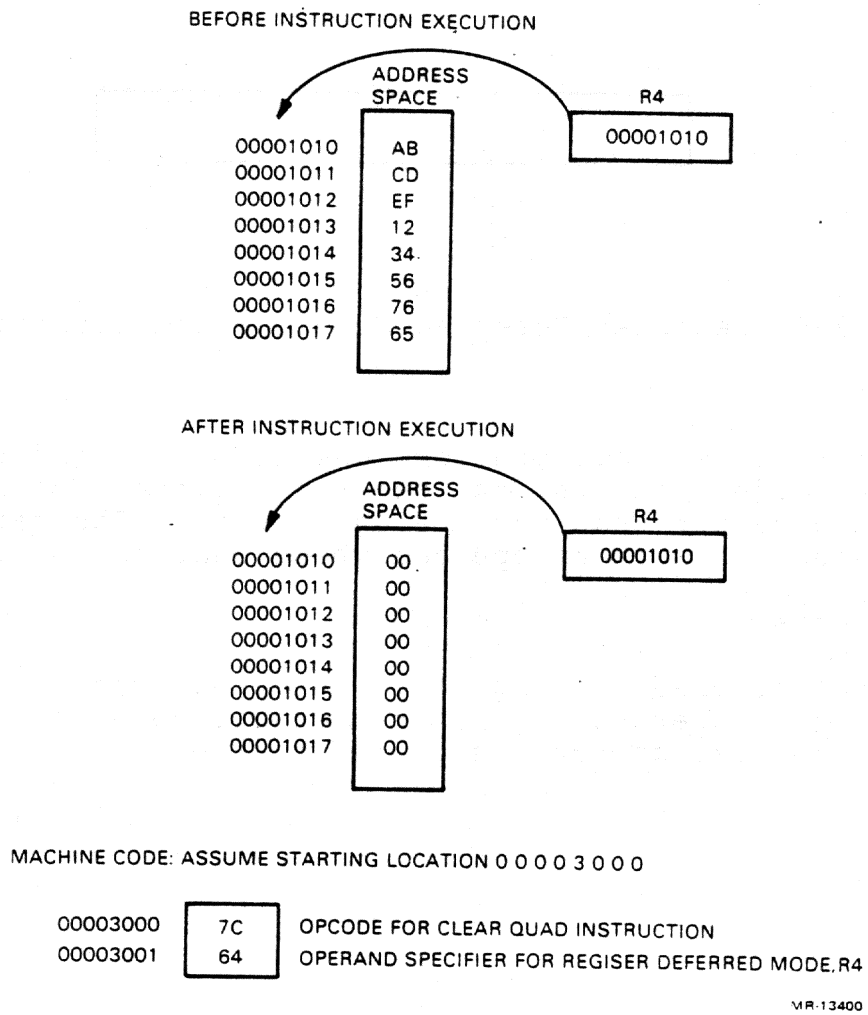


Figure 3-6 CLRQ (R4) Clear Quadword

This example, Figure 3-6, shows a Clear Quad instruction using Register Deferred Mode. R4 contains the address of the operand. The instruction specifies that the byte at this address plus the following seven bytes are to be cleared.

Autoincrement Mode (Figure 3-7)

Assembler Syntax: (Rn)+

Mode Specifier: 8

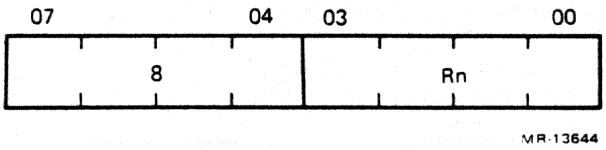


Figure 3-7 Autoincrement Mode Operand Specifier Format

Description:

In autoincrement mode addressing, Rn contains the address of the operand. After the operand address is determined, the size of the operand (which is determined by the instruction) in bytes (1 for byte, 2 for word, 4 for longword or F_floating, and 8 for quadword, D_floating, or G_floating) is added to the contents of Rn and the contents of Rn are replaced by the result. This mode provides for automatic stepping of a pointer through sequential elements of a table of operands. Contents of registers are incremented to address the next sequential location. The autoincrement mode is especially useful for array processing and stacks. It will access an element of a table and then step the pointer to address the next operand in the table. Although most useful for table handling, this mode is general and may be used for a variety of purposes.

Special Comments:

If the PC is used as the general register, this addressing mode is designated immediate mode and has special syntax (refer to immediate mode).

Example: AUTOINCREMENT MODE, MOVE LONG INSTRUCTION

Instruction Format: MOVL (R1)+,R2

This instruction will move a longword of data (32 bits) to R2

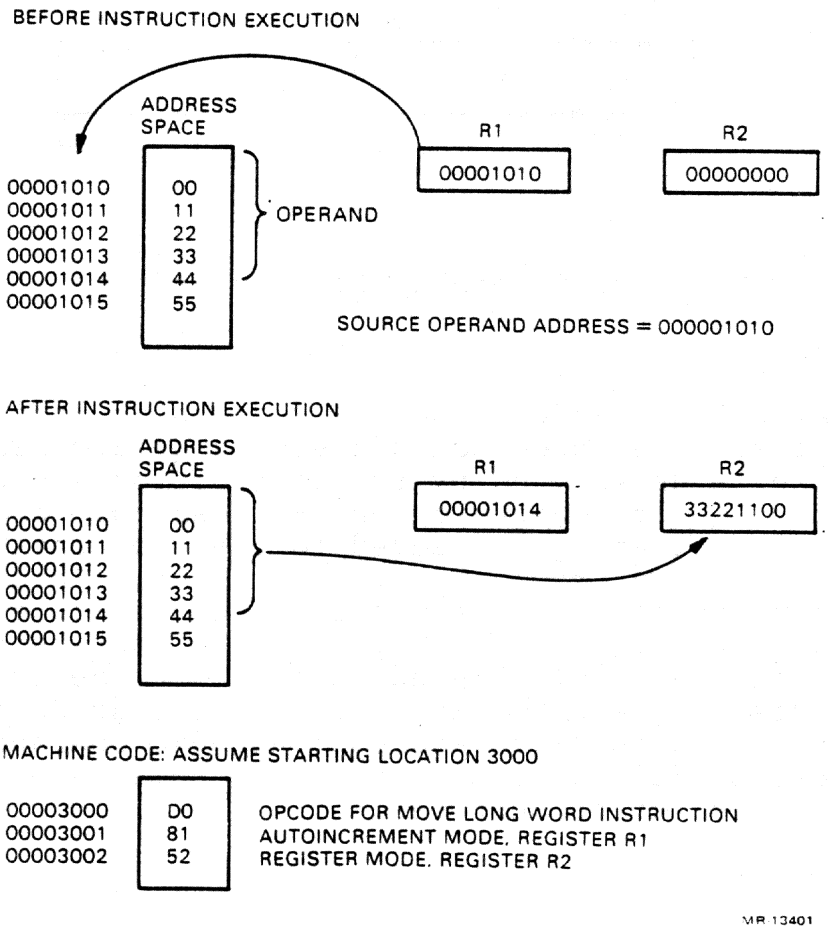


Figure 3-8 MOVL(R1)+,R2 Move Longword

This example, Figure 3-8, shows a Move Long instruction using autoincrement mode. The content of R1 is the effective address of the source operand. Since the operand is a 32-bit longword, four bytes are transferred to R2. R1 is then incremented by four since the instruction specifies a longword data type.

Autoincrement Deferred (Figure 3-9)

Assembler Syntax: @(Rn)+

Mode Specifier: 9

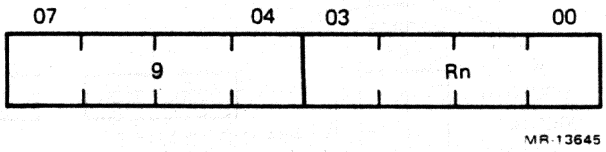


Figure 3-9 Autoincrement Deferred Operand Specifier Format

Description:

In autoincrement deferred addressing, Rn contains a longword address which is a pointer to the operand address. After the operand address has been determined, four is added to the contents of Rn and the contents of Rn are replaced with the result. The quantity four is used since there are four bytes in an address.

Special Comments:

If the PC is used as the general register, this addressing mode is designated absolute mode (refer to absolute mode).

Example: AUTOINCREMENT DEFERRED MODE, MOVE WORD INSTRUCTION

Instruction Format: MOVW @(R1)+,R2

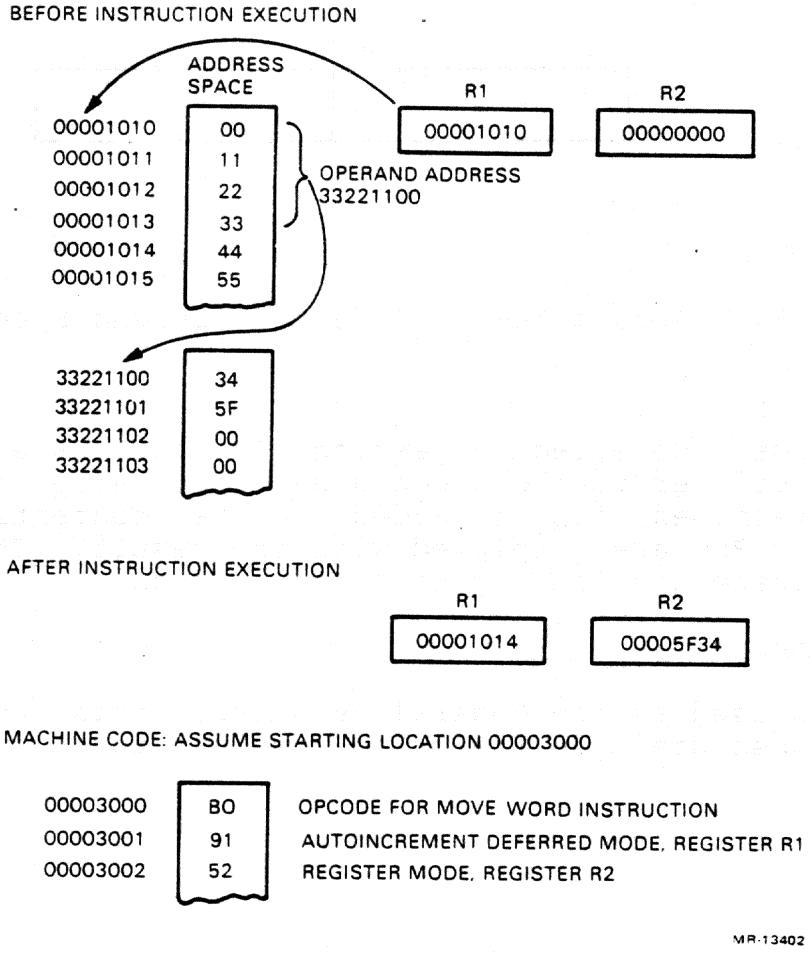


Figure 3-10 MOVW @(R1)+,R2 Move Word

This example, Figure 3-10, shows a Move Word instruction using autoincrement deferred mode. The contents of R1 is a pointer to the operand address. Since a word length instruction is specified, the byte at the effective address and the byte at the effective address plus one are loaded into the low-order half of register R2; the upper half of R2 is not altered. R1 is then incremented by four since it points to a 32-bit address.

Autodecrement Mode (Figure 3-11)

Assembler Syntax: -(Rn)

Mode Specifier: 7

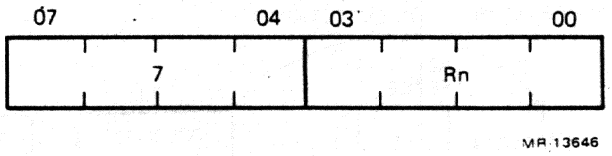


Figure 3-11 Autodecrement Mode Operand Specifier Format

Description:

In autodecrement mode the contents of Rn are decremented and then used as the address of the operand. The size of the operand, in bytes (1 for byte, 2 for word, 4 for longword or F_floating, and 8 for quadword, D_floating or G_floating) is subtracted from the contents of Rn and the contents of Rn are replaced by the result. The updated Rn contains the address of the operand.

Special Comments:

The PC may not be used in autodecrement mode. If it is, the address of the operand is unpredictable and the next instruction executed or the next operand is unpredictable.

Example: AUTODECREMENT MODE, MOVE LONG INSTRUCTION

Instruction Format: MOVL -(R3),R4

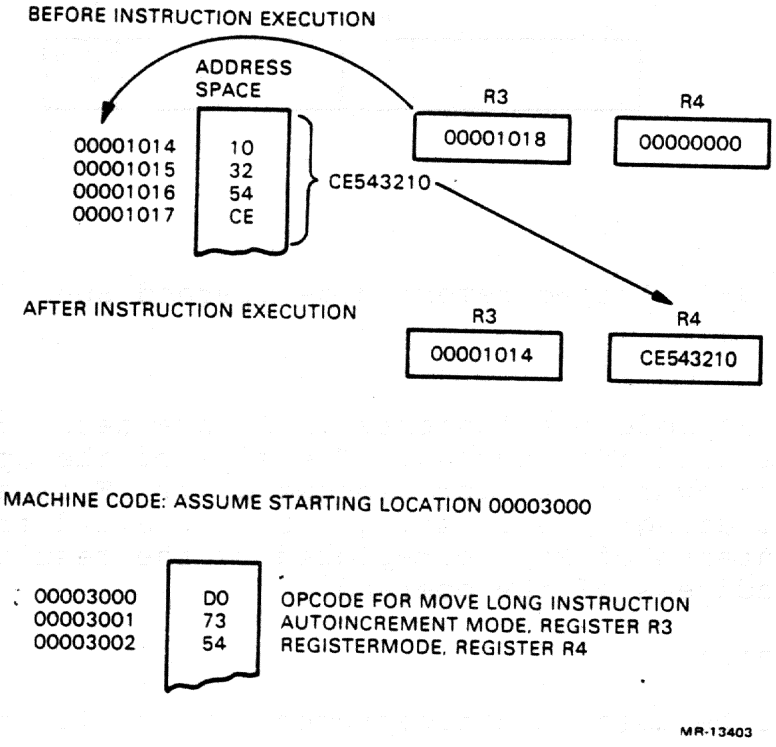


Figure 3-12 MOVL -(R3),R4 Move Longword

This example, Figure 3-12, shows a Move Long instruction using autodecrement mode. The contents or R3 are decremented according to the data type specified in the opcode (four in this example because a longword is used). The updated contents of R3 are then used as the address of the operand. The instruction causes the operand to be fetched and loaded into R4.

Literal Mode (Figure 3-13)

Assembler Syntax: S^# literal

Mode Specifier: 0,1,2, or 3
(depending on literal value specified)

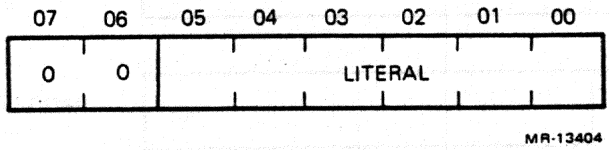


Figure 3-13 Literal Mode Operand Specifier Format

Description:

Literal mode addressing provides an efficient means of specifying integer constants in the range from 0 to 63 (decimal). This is called short literal. Literal values above 63 can be obtained by immediate mode (autoincrement mode using the PC) although immediate mode is longer. For predefined values, the assembler will choose between short literal and immediate modes. For short literal operands, the format is shown in Figure 3-14. Bits 7 and 6, however, are always set to zero.

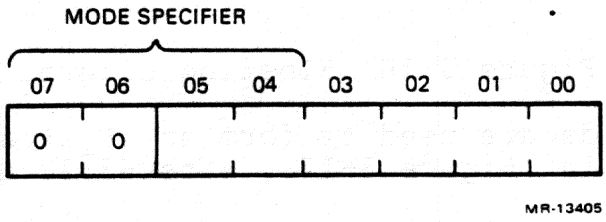


Figure 3-14 Short Literal Format

The following example, Figure 3-15 shows some short literals; the literals are 14, 30, 46, and 62.

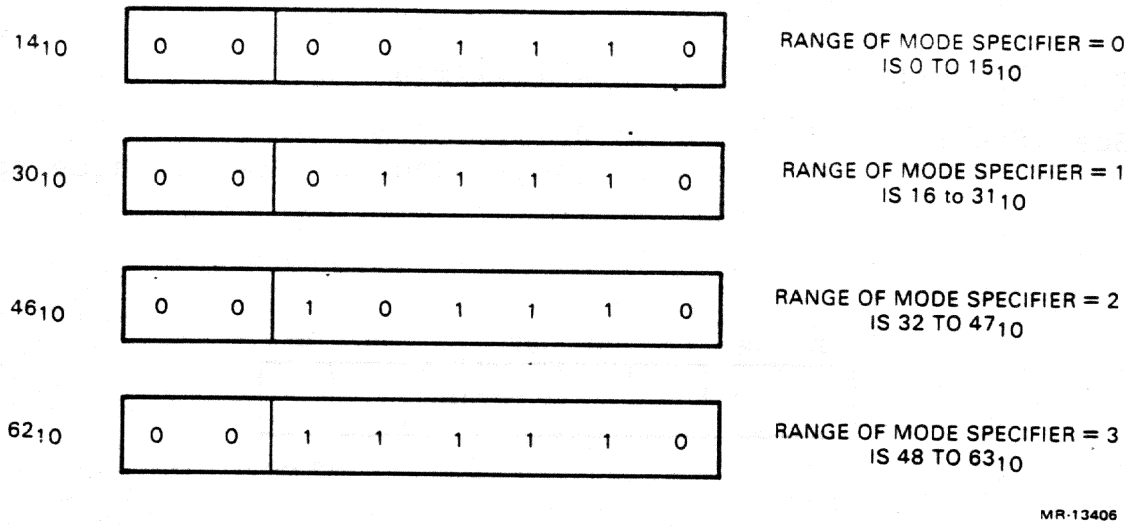


Figure 3-15 Examples of Short Literals

Floating point literals as well as short literals can be expressed. For operands of data type F_floating, D_floating, and G_floating, the 6-bit literal field is composed of two 3-bit fields, Figure 3-16, where EXP is exponent and FRA is fraction.

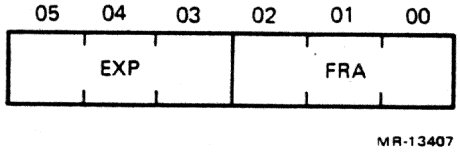


Figure 3-16 Floating Literal

The EXP and FRA fields are used to form an F_floating or D_floating operand as shown in Figure 3-17. Bits 63:32 are not present in an F_floating operand.

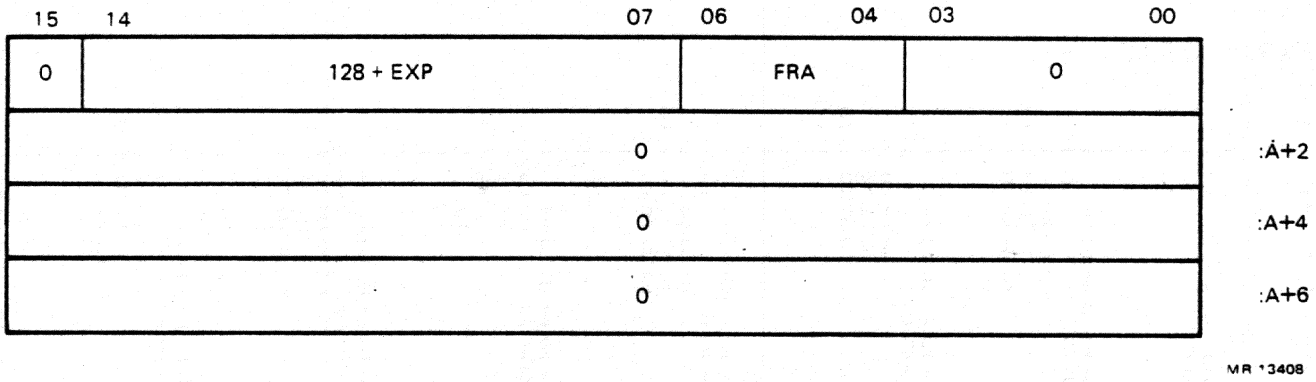


Figure 3-17 F_floating and D_floating Operand

The EXP and FRA fields are used to form a G_floating operand as shown in Figure 3-18.

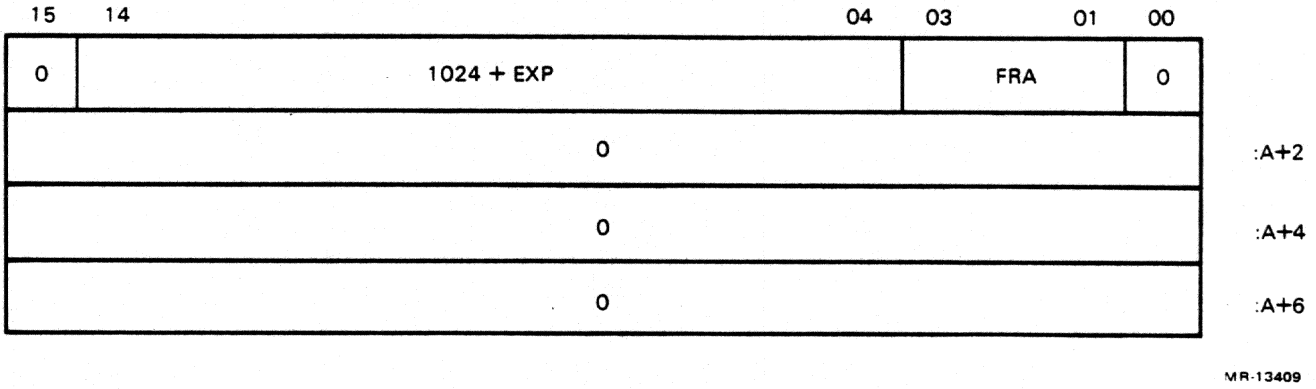


Figure 3-18 G_floating Operand

Table 3-3 gives the numbers that can be represented by floating literals.

Table 3-3 Floating Literals

EXP v	FRAC --> 0	1	2	3	4	5	6	7
0	1/2	9/16	5/8	11/16	3/4	13/16	7/8	15/16
1	1	1 1/8	1 1/4	1 3/8	1 1/2	1 5/8	1 3/4	1 7/8
2	2	2 1/4	2 1/2	2 3/4	3	3 1/4	3 1/2	3 3/4
3	4	4 1/2	5	5 1/2	6	6 1/2	7	7 1/2
4	8	9	10	11	12	13	14	15
5	16	18	20	22	24	26	28	30
6	32	36	40	44	48	52	56	60
7	64	72	80	88	96	104	112	120

Example: LITERAL MODE, MOVE LONG INSTRUCTION

Instruction Format: MOVL S^#9,R4

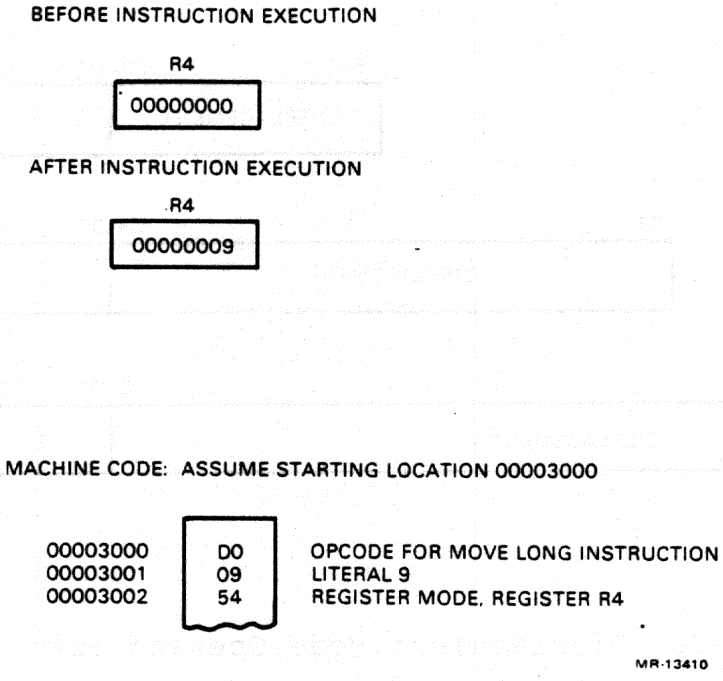


Figure 3-19 MOVL S^#9,R4 Move Longword

This example, Figure 3-19, shows a Move Long instruction using literal mode. The literal 9 is transferred to register R4 as a result of the instruction.

Displacement Mode (Figure 3-20)

Assembler Syntax:

D(Rn)
B^D(Rn) - Byte displacement
W^D(Rn) - Word displacement
L^D(Rn) - Longword displacement

Mode Specifier:

A - (byte displacement)
C - (word displacement)
E - (longword displacement)

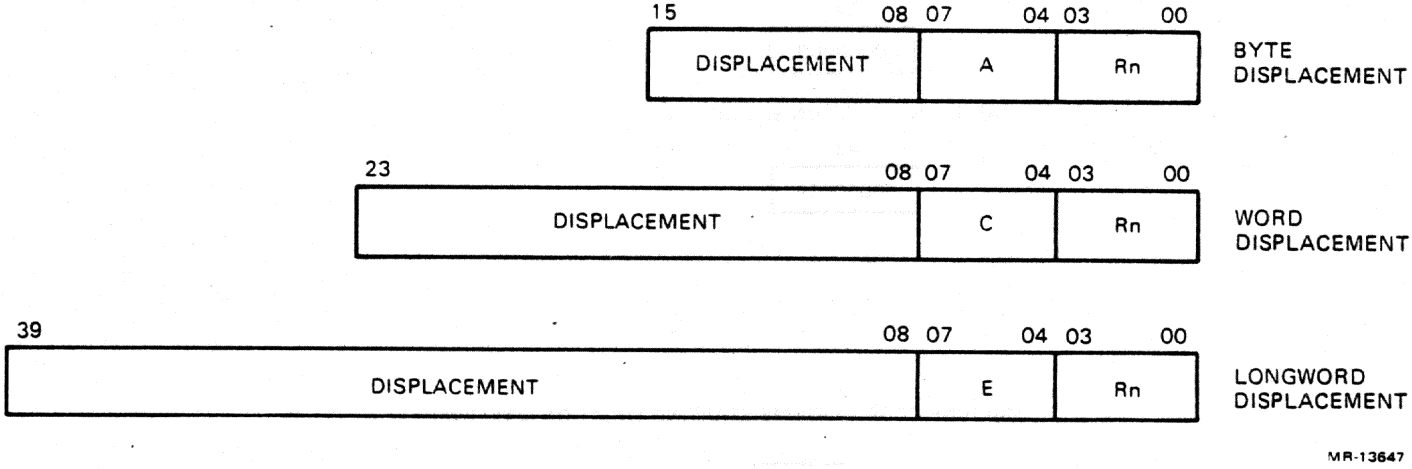


Figure 3-20 Displacement Mode Operand Specifier Format

Description:

In displacement mode addressing, the displacement (after being sign-extended to 32 bits if it is a byte or word) is added to the contents of register Rn and the result is the operand address.

The MicroVAX architecture provides for an 8-bit, 16-bit, or 32-bit offset. Since most program references occur within small discrete portions of the address space, a 32-bit offset is not always necessary and the 8- and 16-bit offsets will result in the saving of space (fewer bits are required).

If the PC is used as the general register, this mode is called relative mode (refer to relative mode).

Example: DISPLACEMENT MODE, MOVE BYTE INSTRUCTION

Instruction Format: MOVB B^5(R4),B^3(R3)

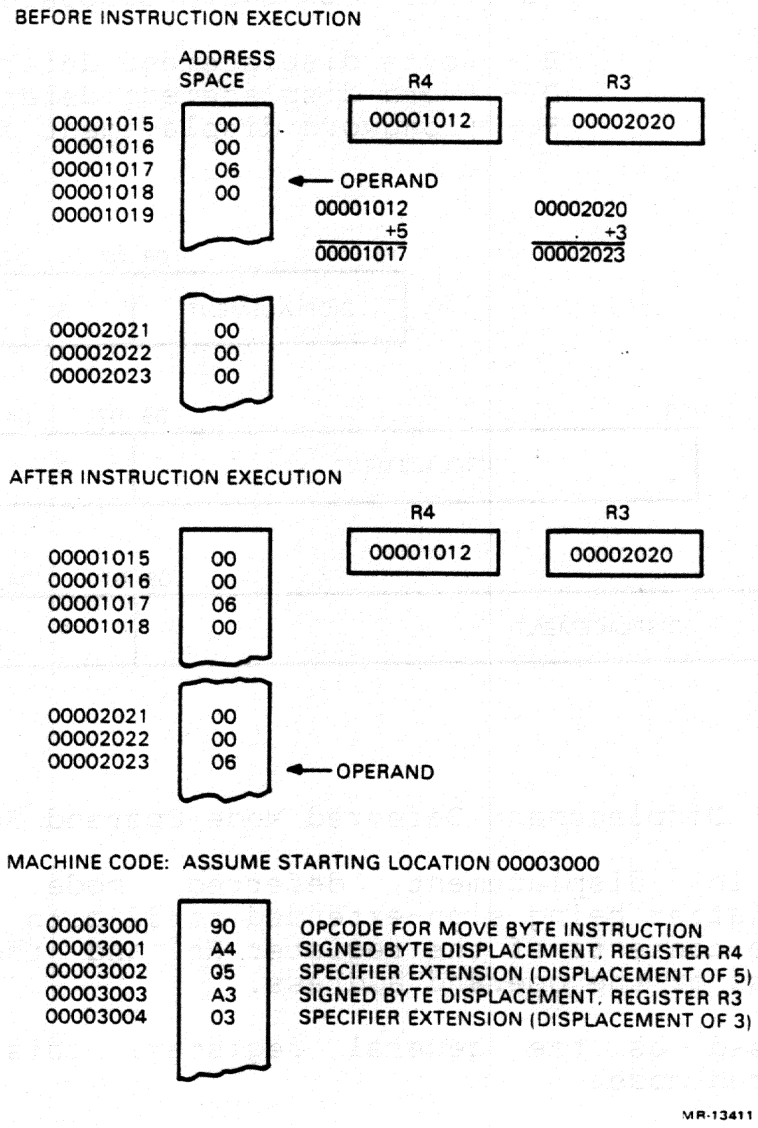


Figure 3-21 MOVB B^5(R4),B^3(R3) Move Byte

This example, Figure 3-21, shows a Move Byte instruction using displacement mode. A displacement of 5 is added to the contents of R4 to form the address of the byte operand. The operand is moved to the address formed by adding the displacement of 3 to the contents of R3.

Displacement Deferred Mode (Figure 3-22)

Assembler Syntax:
@D(Rn)
@B^D(Rn) - Byte displacement deferred
@W^D(Rn) - Word displacement deferred
@L^D(Rn) - Longword displacement deferred

Mode Specifier:
B - (byte displacement deferred)
D - (word displacement deferred)
F - (longword displacement deferred)

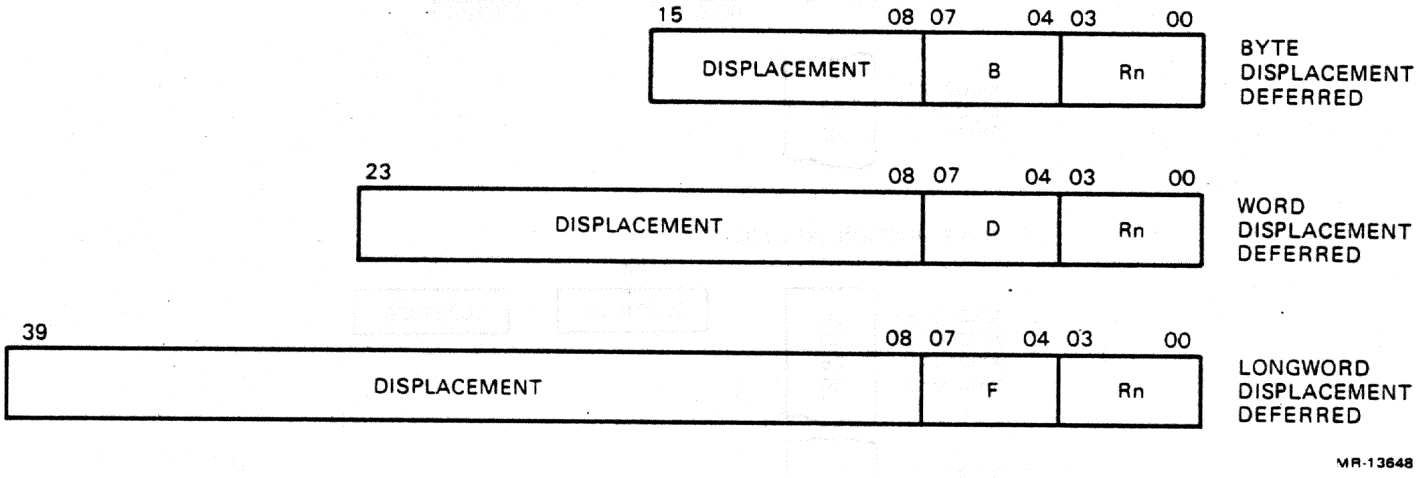


Figure 3-22 Displacement Deferred Mode Operand Specifier Format

Description: In displacement deferred mode addressing, the displacement (after being sign-extended to 32 bits if a byte or word) is added to the contents of the register Rn and the result is the longword address of the operand address.

If the PC is used as the general register, this mode is called relative deferred mode.

Example: DISPLACEMENT DEFERRED MODE, INCREMENT WORD INSTRUCTION

Instruction Format: INCW @B^5(R4)

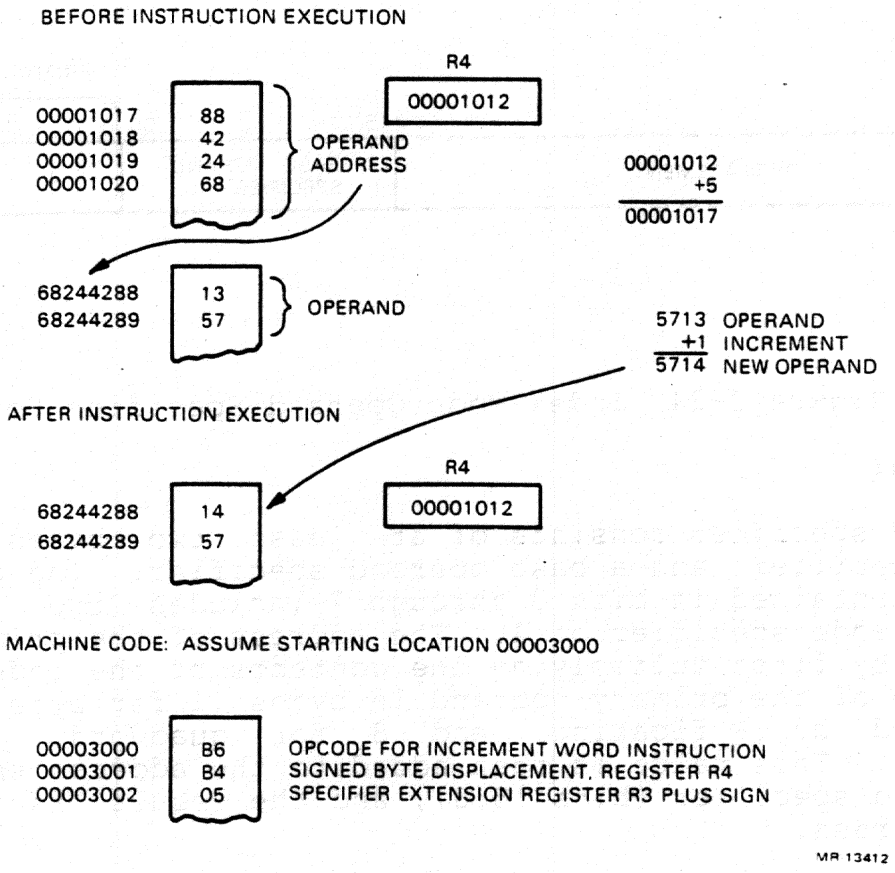


Figure 3-23 INCW @B^5(R4) Increment Word

This example, Figure 3-23, shows an Increment Word instruction using displacement deferred mode. The quantity 5 is added to the contents of R4 and the result is the longword address of the address of the operand. The operand of 5713 is incremented to 5714.

Index Mode (Figure 3-24)

Assembler Syntax: i[Rx]

Mode Specifier: 4

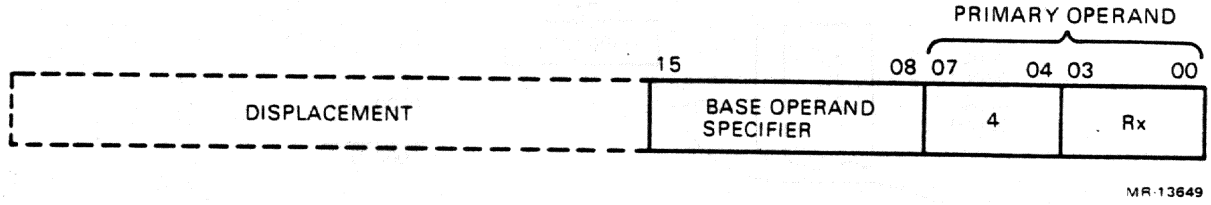


Figure 3-24 Index Mode Operand Specifier Format

Description:

The operand specifier consists of at least two bytes - a primary operand specifier and a base operand specifier. The primary operand specifier contained in bits 0 through 7 includes the index register (Rx) and a mode specifier of 4. The address of the primary operand is determined by first multiplying the contents of the index register Rx by the size of the primary operand in bytes (1 for byte, 2 for word, 4 for longword or F_floating, and 8 for quadword, D_floating, or G_floating). This value is then added to the address specified by the base operand specifier (bits 15:8), and the result is taken as the operand address.

Index mode addressing permits very general and efficient accessing of arrays. The base address of the array is determined by the operand address calculation of the base operand specifier. The contents of the index registers are taken as a logical index into the array. The logical index is converted into a real (byte) offset by multiplying the contents of the index register by the size of the primary operand in bytes.

Specifying register, literal, or index mode for the base operand specifier will result in an illegal addressing mode fault. If the use of some particular specifier is illegal (causes a fault or unpredictable behavior), then that specifier is also illegal as a base operand specifier in index mode under the same conditions.

Special Comments:

The following restrictions are placed on index register Rx:

1. The PC cannot be used as an index register. If it is, a reserved addressing mode fault occurs.

2. If the base operand specifier is for an addressing mode which results in register modification (autoincrement, autoincrement deferred, or autodecrement), the same register cannot be the index register. If it is, the primary operand address is unpredictable.

Table 3-4 lists the various forms of index mode addressing available. The names of the addressing modes resulting from index mode addressing are formed by adding indexed to the addressing mode of the base operand specifier. The general register is designated Rn and the index register is Rx.

Table 3-4 Index Mode Addressing

Mode	Assembler Notation
Register Deferred Indexed	(Rn)[Rx]
Autoincrement Indexed	(Rn)+[Rx]
Autoincrement Deferred Indexed	@(Rn)+[Rx]
Absolute Indexed	@#address[Rx]
Autodecrement Indexed	-(Rn)[Rx]
Byte, Word, Longword Displacement Indexed	B^D(Rn)[Rx] W^D(Rn)[Rx] L^D(Rn)[Rx]
Byte, Word, Longword Displacement Deferred Indexed	@B^D(Rn)[Rx] @W^D(Rn)[Rx] @L^D(Rn)[Rx]

It is important to note that the operand address (the address containing the operand) is first evaluated and then the index specified by the index register is added to the operand address to find the indexed address. To illustrate this, an example of each type of indexed addressing follows.

Example #1: REGISTER DEFERRED INDEXED MODE, INCREMENT WORD INSTRUCTION

Instruction Format: INCW (R2)[R5]

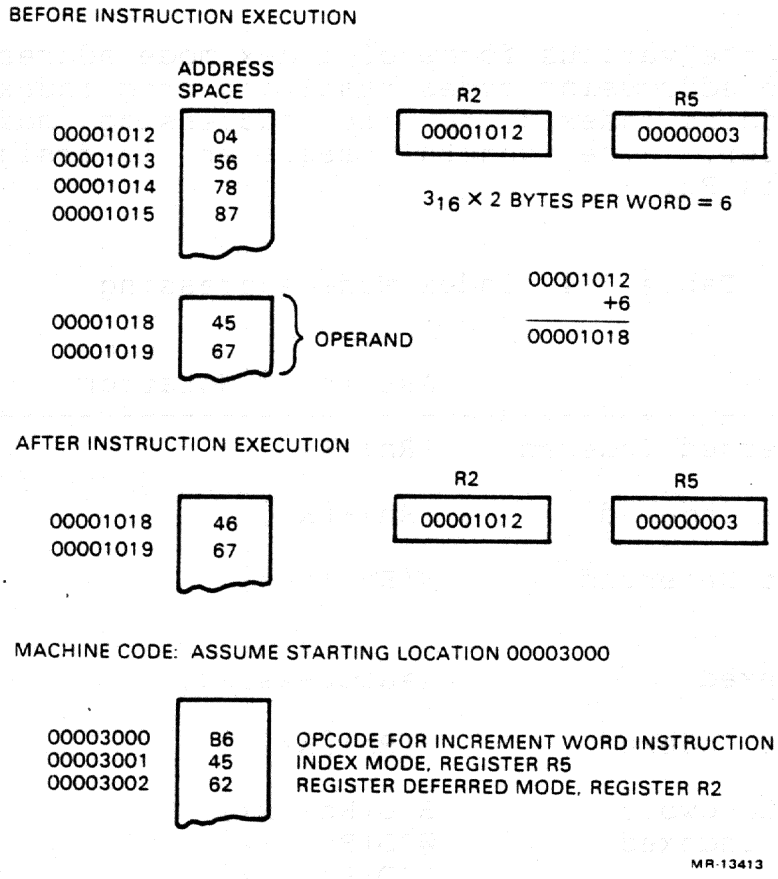


Figure 3-25 INCW (R2)[R5] Increment Word

This example, Figure 3-25, shows an Increment Word instruction using register deferred index addressing. The base operand address is evaluated. This location is indexed by 6 since the value (3) in the index register is multiplied by the word data size of two.

Example #2: AUTOINCREMENT INDEXED MODE, CLEAR LONGWORD INSTRUCTION

Instruction Format: CLRL (R4)+[R5]

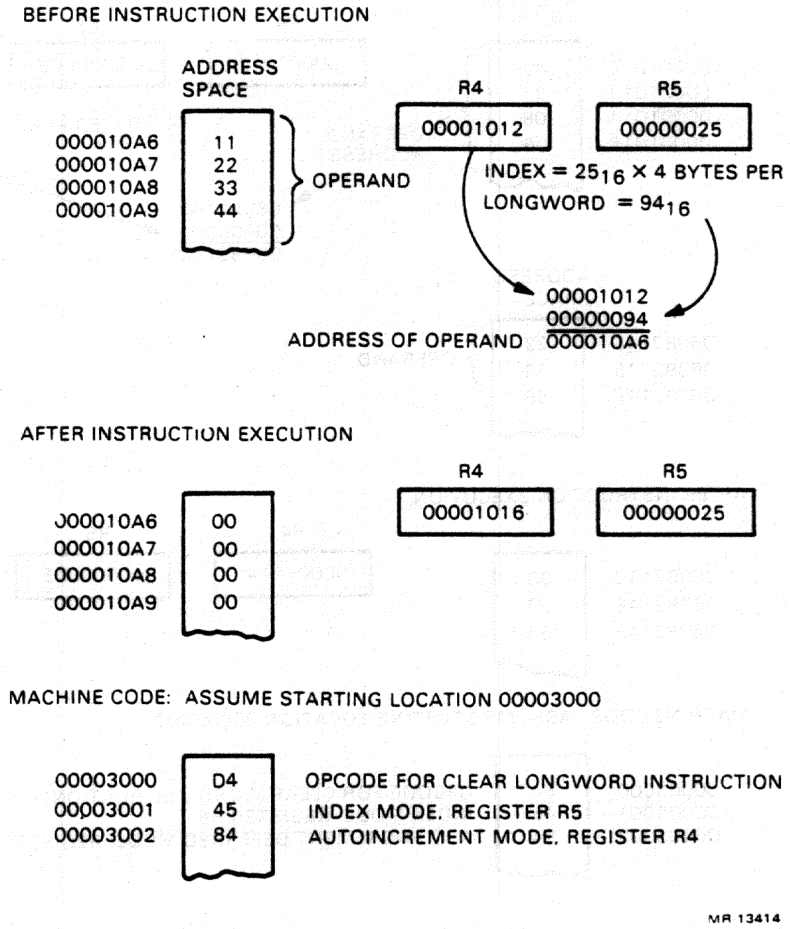


Figure 3-26 CLRL (R4)+[R5] Clear Longword

This example, Figure 3-26, shows a Clear Long instruction using the autoincrement indexed addressing mode. The base operand address is in R4. This value is indexed by the quantity in R5 multiplied by the data size, in bytes. This location, plus the next three, are cleared since a longword instruction is specified.

Example #3: AUTOINCREMENT DEFERRED INDEX MODE, CLEAR WORD INSTRUCTION

Instruction Format: CLRW @(R4)+[R5]

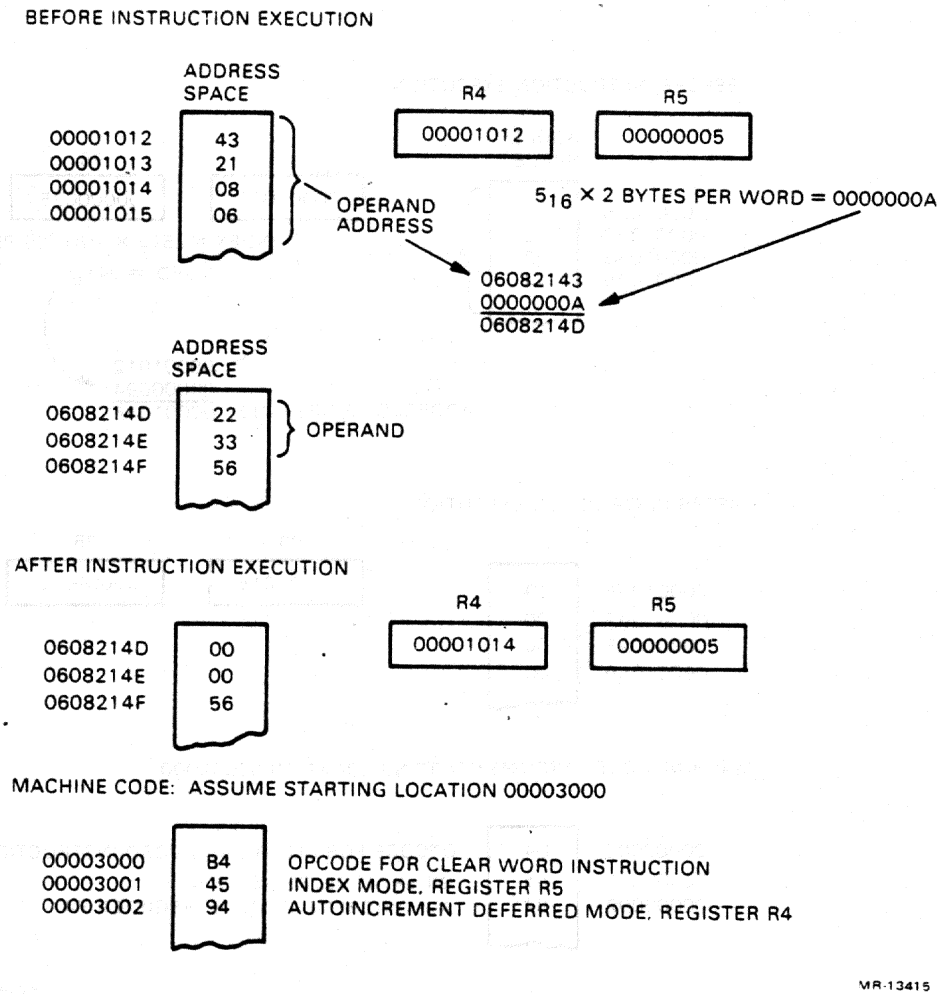


Figure 3-27 CLRW @(R4)+[R5] Clear Word

This example, Figure 3-27, shows a Clear Word instruction using the autoincrement deferred indexing mode. R4 contains the address of the operand address. The index value A is obtained by multiplying the contents of the index register, [R5], by the data size, in bytes, which is 2. The calculated word address is cleared.

Example #4: AUTODECREMENT INDEXED MODE, CLEAR WORD INSTRUCTION

Instruction Format: CLRW -(R2)[R4]

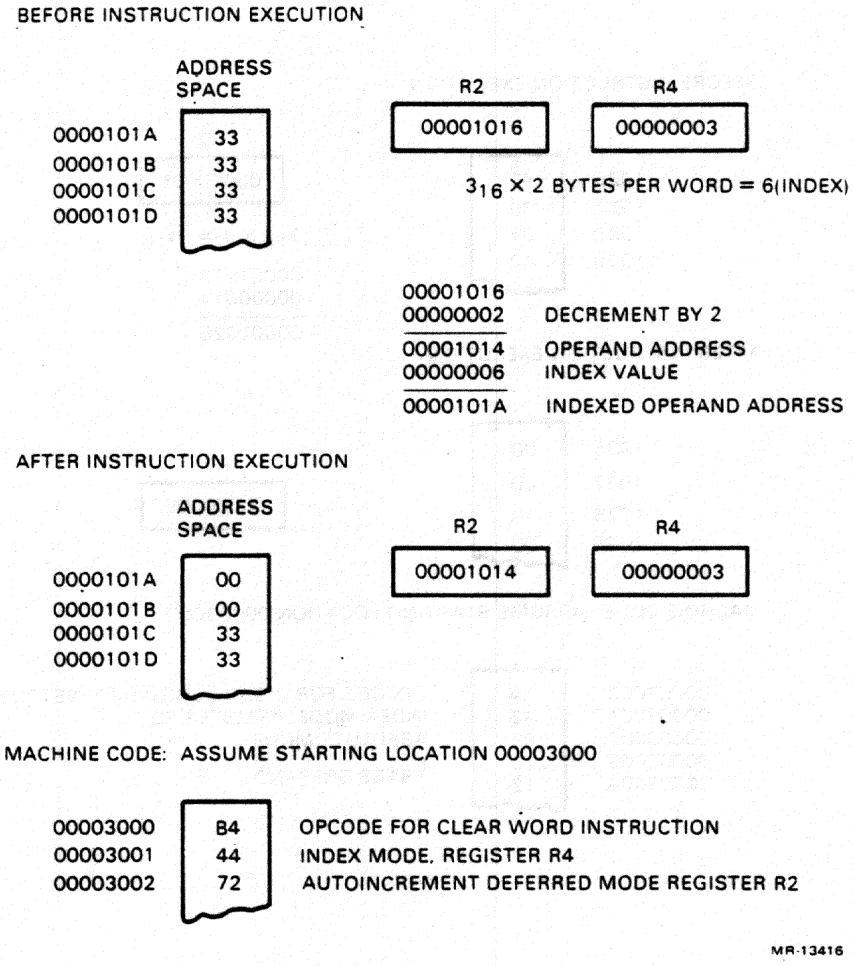


Figure 3-28 CLRW -(R2)[R4] Clear Word

This example, Figure 3-28, shows a Clear Word instruction using autodecrement indexed mode. The contents of R2 are decremented by two, the data size in bytes. The index register, R4, is multiplied by the data size and the result is added to the contents of R2 to form the operand address. Since a clear word instruction is specified, two bytes are cleared.

Example #5: ABSOLUTE INDEXED MODE, CLEAR LONGWORD INSTRUCTION

Instruction Format: CLRL @#^X1012[R2]

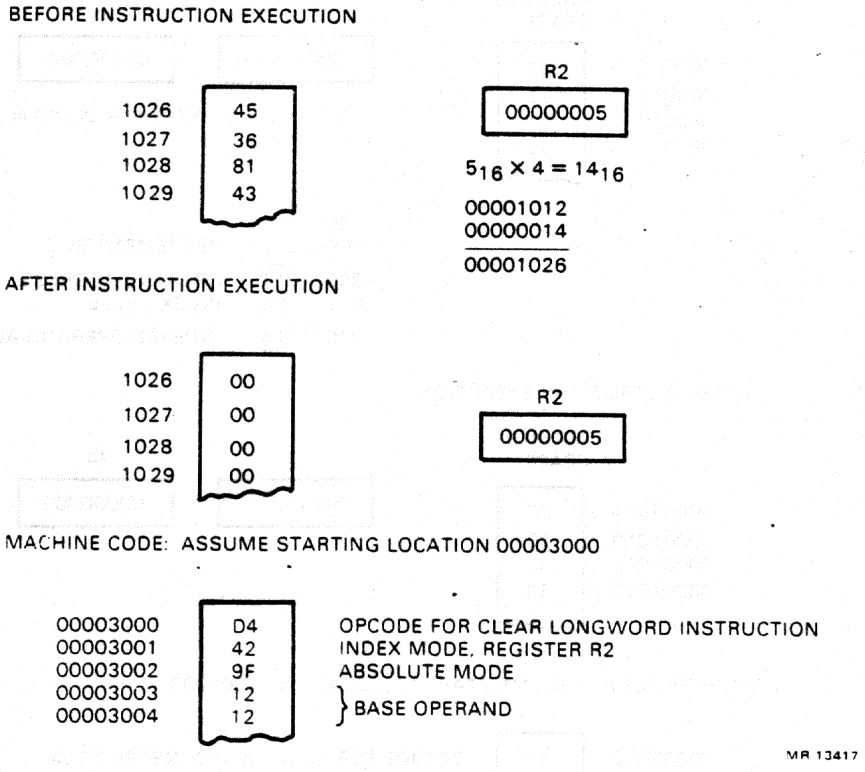


Figure 3-29 CLRL @#^X1012[R2] Clear Longword

This example, Figure 3-29, shows a Clear Longword instruction using absolute indexed mode. The base of 00001012 is indexed by R2 which contains 5. Since a longword data type is specified, 5x4 = 14(16), which becomes the index value. This value is added to 00001012 yielding 00001026. This is the operand address, and four bytes are cleared since a longword data type has been specified.

Example #6: DISPLACEMENT INDEXED MODE, CLEAR QUADWORD INSTRUCTION

Instruction Format: CLRQ 2(R1)[R3]

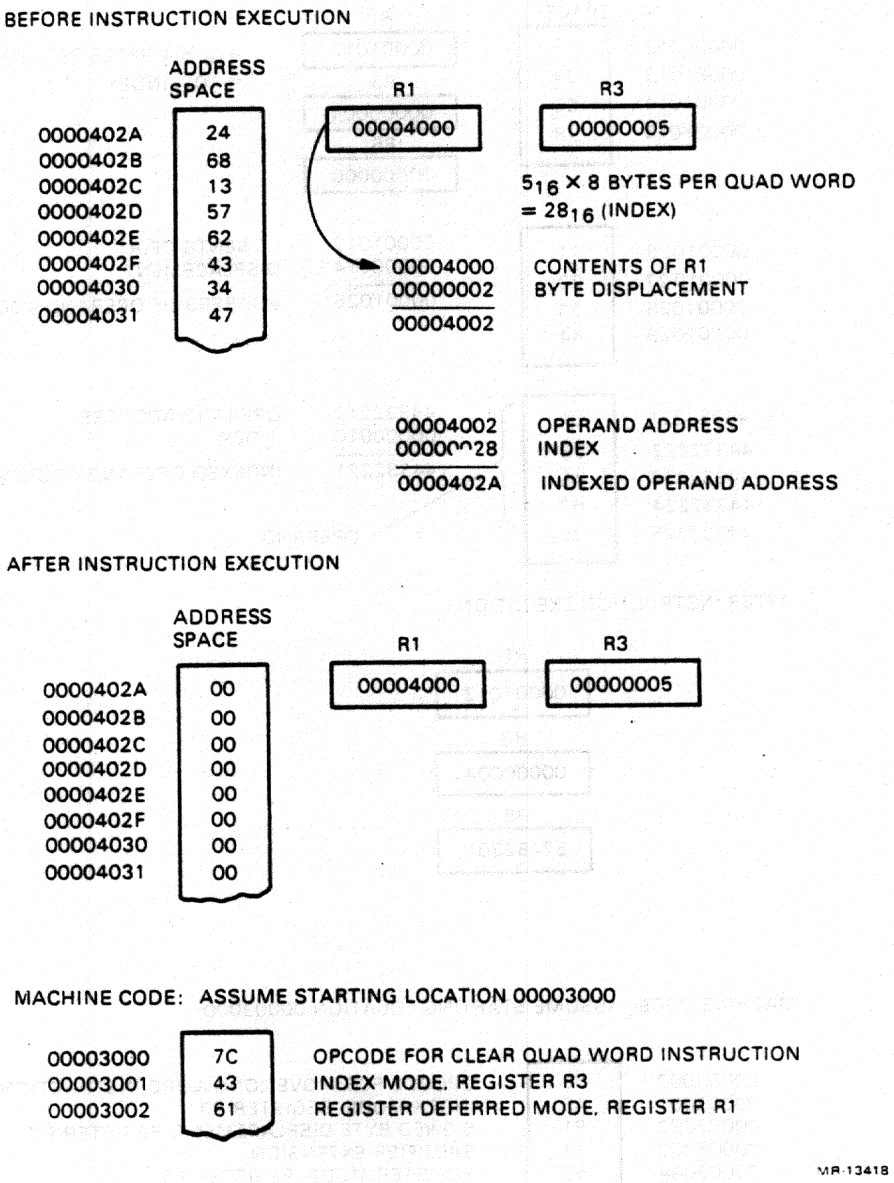
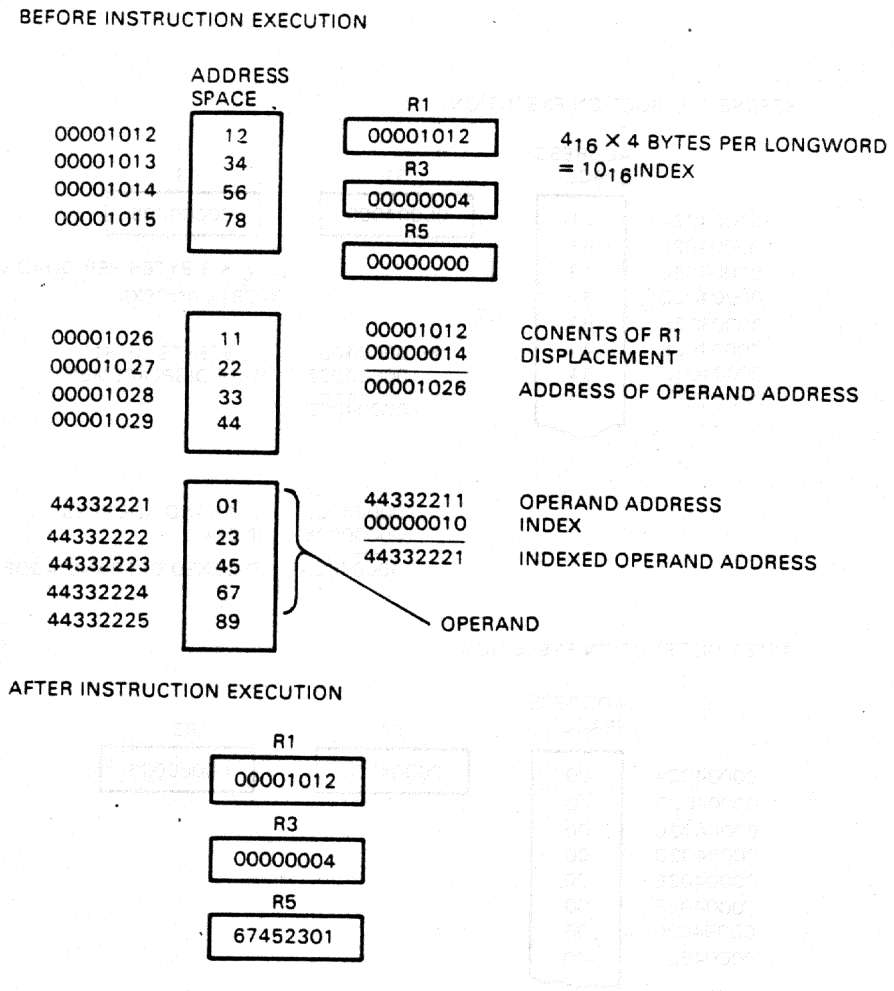


Figure 3-30 CLRQ 2(R1)[R3] Clear Quadword

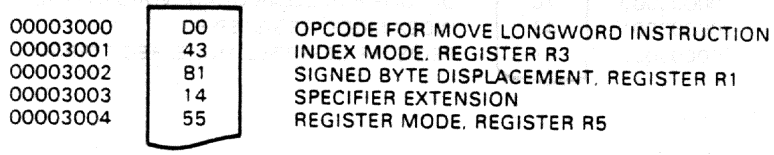
This example, Figure 3-30, shows a Clear Quadword instruction using displacement index mode. The byte displacement of two is added to the content of R1. The index which is calculated as 28 is added to this address. This location and the next seven locations (since a quadword instruction is specified) are cleared.

Example #7: DISPLACEMENT DEFERRED INDEX MODE, MOVE LONG INSTRUCTION

Instruction Format: MOVL @^X14(R1)[R3],R5



MACHINE CODE: ASSUME STARTING LOCATION 00003000



MR-13419

Figure 3-31 MOVL @^X14(R1)[R3],R5 Move Longword

This example, Figure 3-31, shows a Move Long instruction using displacement deferred indexed addressing. The displacement of 14 is added to the contents of R1 yielding 00001026. The contents of this location are the operand address, 44332211. This quantity is added to the index yielding the indexed operand address of 44332221. The contents of this address are moved into R5.

3.2.1.2. Program Counter Addressing -

Register 15 is the program counter (PC). It can also be used as a register in addressing modes. The processor increments the program counter as the opcode, operand specifier, and immediate data or addresses (of the instruction) are evaluated. The amount that the PC is incremented is determined by the opcode, number of operand specifiers, and so on.

The PC can be used with all of the addressing modes except register of index mode, since in these two modes the results will be unpredictable. Table 3-5 lists the addressing modes that use the PC as a general register.

Table 3-5 Program Counter Addressing Modes

Mode	Name	Assembler	Function
8	Immediate	I^#Operand	Constant operand follows address mode
9	Absolute	@#Location	Absolute address follows address mode
A	Byte Relative	B^G(PC)	Displacement is added to current value of PC to obtain operand address
C	Word Relative	W^G(PC)	
E	Longword Relative	L^G(PC)	
B	Byte Relative Deferred	@B^G(PC)	Displacement is added to current of PC to give the address of the operand address
D	Word Relative Deferred	@W^G(PC)	
F	Longword Relative Deferred	@L^G(PC)	

Immediate Mode (Figure 3-32)

Assembler Syntax: I^#operand

Mode Specifier: 8

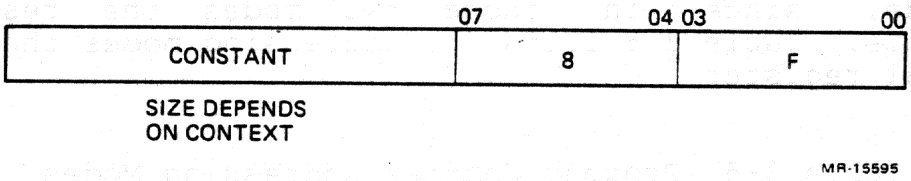


Figure 3-32 Immediate Mode Operand Specifier Format

Description:

This mode is autoincrement mode when the PC is used as the general register. The contents of the location following the addressing mode are immediate data.

Example: IMMEDIATE MODE, MOVE LONG INSTRUCTION

Instruction Format: MOVL I^#6,R4

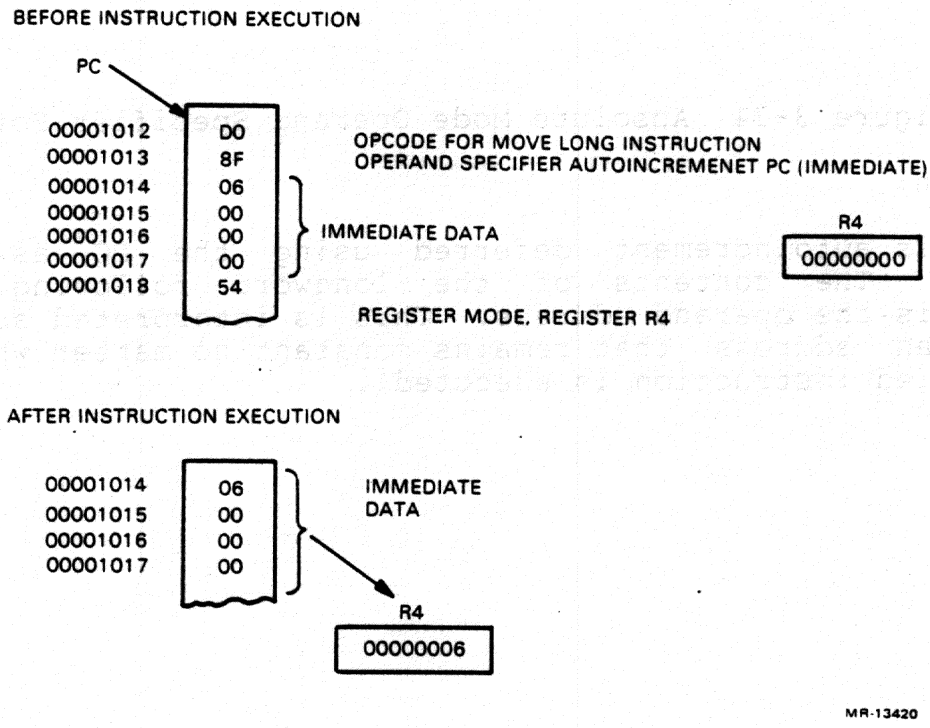


Figure 3-33 MOVL I^#6,R4 Move Longword

This example, Figure 3-33, shows a Move Long instruction using immediate mode. The immediate data (00000006) following the opcode and operand specifier are moved to R4.

Absolute Mode (Figure 3-34)

Assembler Syntax: @#location

Mode Specifier: 9

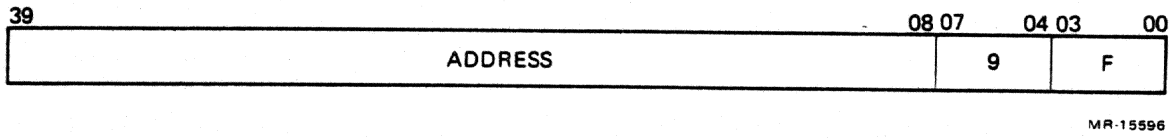


Figure 3-34 Absolute Mode Operand Specifier Format

Description:

This mode is autoincrement deferred using the PC as the general register. The contents of the longword following the operand specifier is the operand address. This is interpreted as an absolute address (an address that remains constant no matter where in memory the assembled instruction is executed).

Example: ABSOLUTE MODE, CLEAR LONG INSTRUCTION

Instruction Format: CLRL @#^X674533

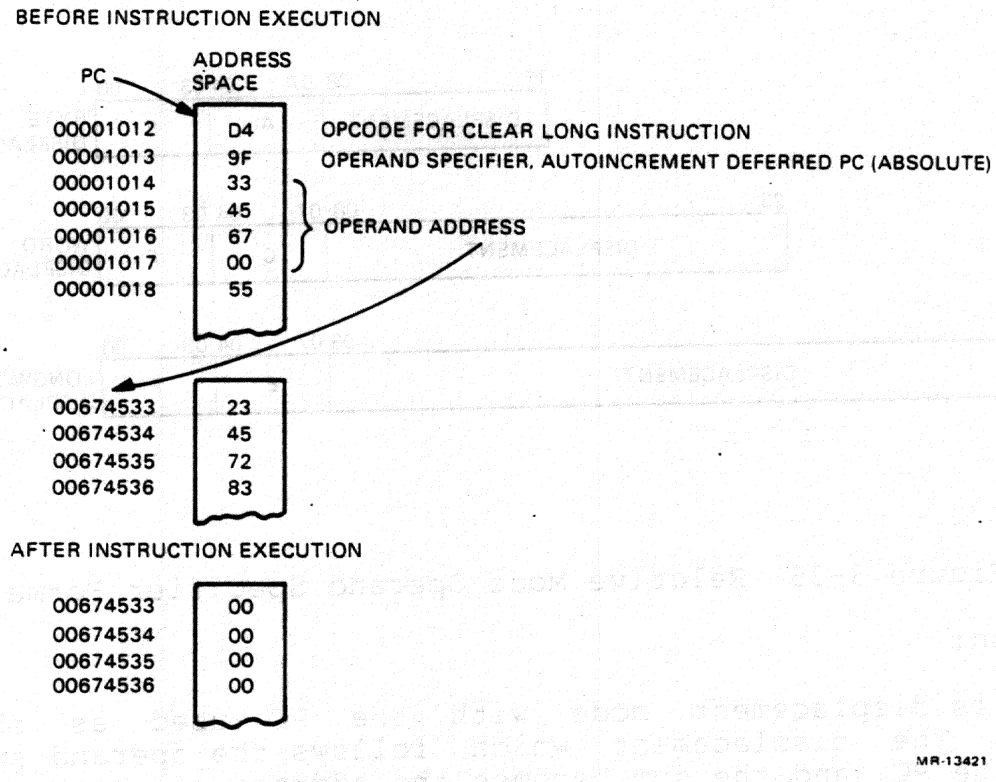


Figure 3-35 CLRL @#^674533 Clear Longword

This example, Figure 3-35, shows a Clear Longword instruction using the absolute addressing mode. This instruction causes the location(s) following the operand specifier to be taken as the address of the operand, and is 00674533 in this case. The longword operand associated with this address is cleared.

Relative Mode (Figure 3-36)

Assembler Syntax:

B^D - Byte displacement
W^D - Word displacement
L^D - Longword displacement

Mode Specifier:

A - (byte)
C - (word)
E - (longword)

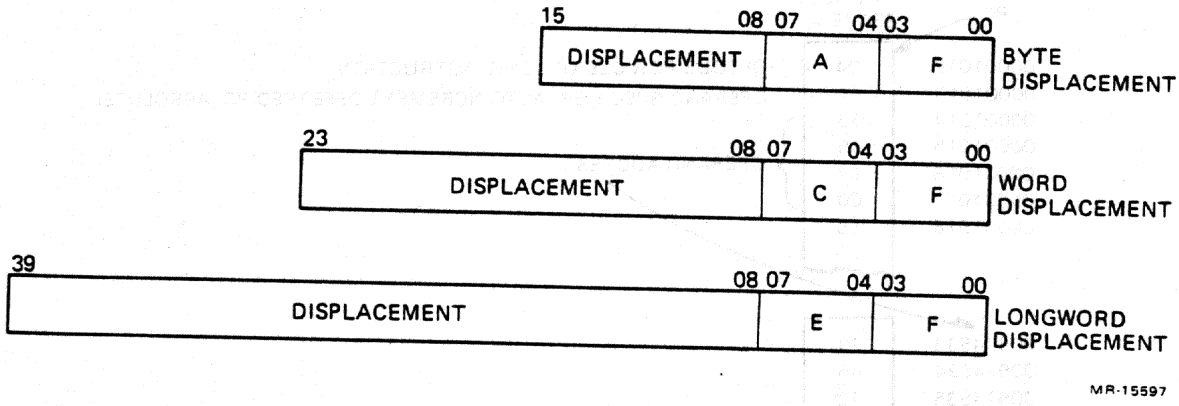


Figure 3-36 Relative Mode Operand Specifier Format

Description:

This mode is displacement mode with the PC used as the general register. The displacement which follows the operand specifier is added to the PC, and the sum becomes the address of the operand. This mode is useful for writing position independent code, since the location referenced is always fixed relative to the PC.

Example: RELATIVE MODE, MOVE LONGWORD INSTRUCTION

Instruction Format: MOVL ^X2016,R4

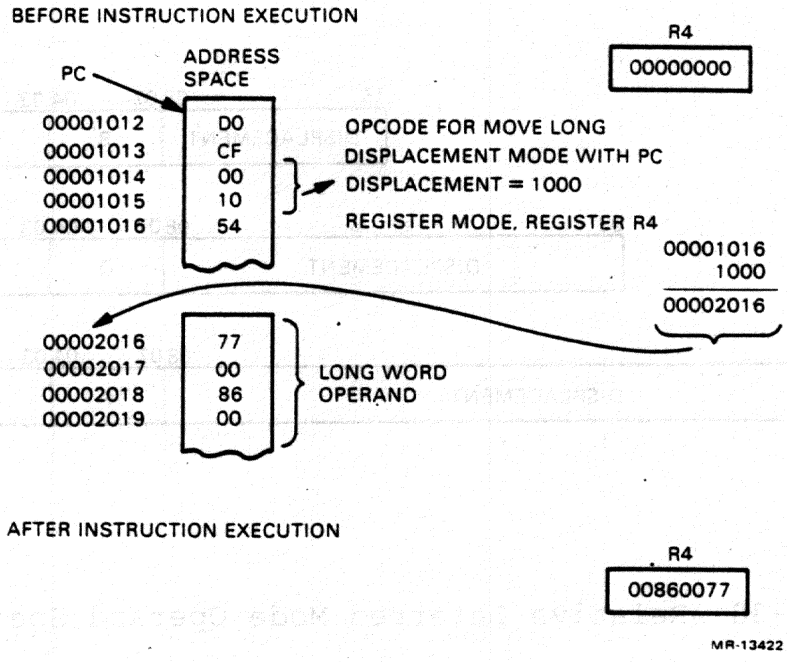


Figure 3-37 MOVL ^X2016,R4 Move Longword

This example, Figure 3-37, shows a Move Long instruction using relative mode. The word following the operand specifier is added to the PC to obtain the address of the PC.

In this example, the PC is pointing to location 00001016 after the first operand specifier is evaluated. The word following the first opcode and first operand specifier is 00001000 and is added to the PC. The result is 00002016. This value represents the address of the longword operand (00860077). The operand is then moved to register R4. The PC contains 00001017 after instruction execution.

Relative Deferred Mode (Figure 3-38)

Assembler Syntax:

@B^D - Byte displacement deferred
@W^D - Word displacement deferred
@L^D - Longword displacement deferred

Mode Specifier:

B - (byte)
D - (word)
F - (longword)

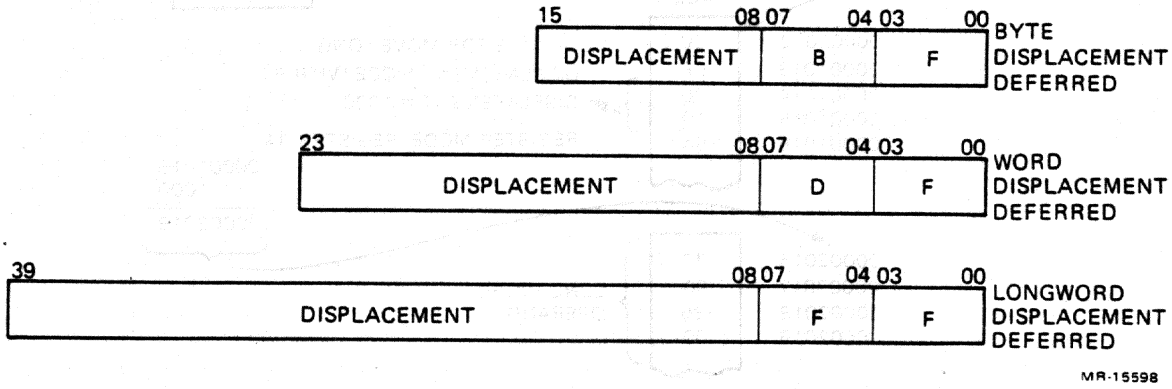


Figure 3-38 Relative Deferred Mode Operand Specifier Format

Description:

This mode is similar to relative mode, except that the displacement which follows the addressing mode is added to the PC and the sum is the longword address of the address of the operand. This addressing mode is useful when processing tables of addresses.

Example: RELATIVE DEFERRED MODE, MOVE LONG INSTRUCTION

Instruction Format: MOVL @^X2050,R2

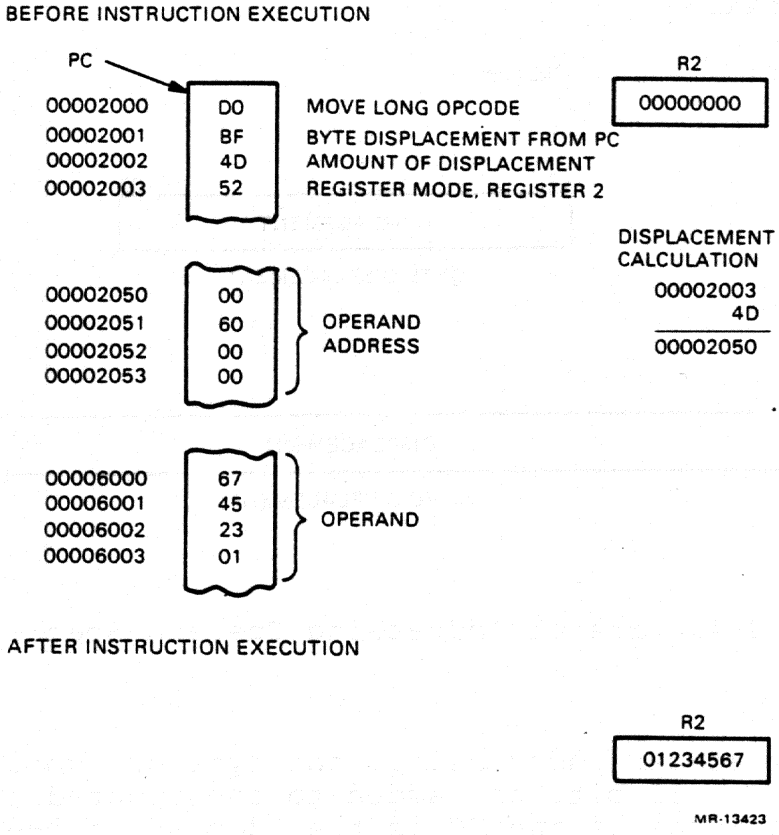


Figure 3-39 MOVL @^X2050,R2 Move Longword

This example, Figure 3-39, shows a Move Long instruction where 00002050 represents the address of the operand. A byte displacement would be selected by the assembler since the displacement is within 128 (decimal) addressable bytes. When the displacement is evaluated, the program counter is pointing to 00002003. The displacement of 4D is added to the current value of the PC to give the address of 00002050. The contents of this address are then used as the address of the operand (00006000) and the operand is moved to R2.

3.2.2 Branch Addressing

In branch displacement addressing the byte or word displacement is sign-extended to 32 bits and added to the updated content of the PC. The updated contents of the PC are the address of the first byte beyond the operand specifier.

Branch Addressing (Figure 3-40)

Assembler Syntax: A

Mode Specifier: None

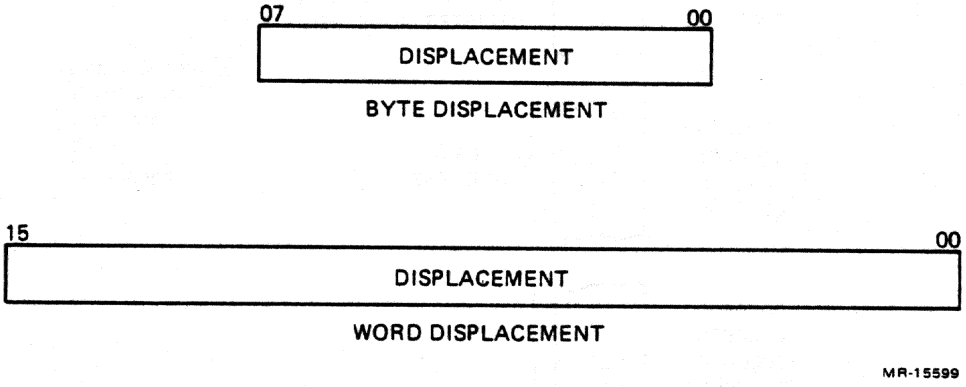


Figure 3-40 Branch Addressing Operand Specifier Format

Description:

In branch displacement addressing, the byte or word displacement is sign-extended to 32 bits and added to the updated contents of the PC. The updated contents of the PC is the address of the first byte after the operand specifier.

The assembler notation for byte and word branch displacement addressing is A, where A is the branch address. Note the branch address and not the displacement is used.

Branch instruction are most frequently used after instructions like compare (CMP) and are used to cause different actions depending on the results of the compare.

Example #1: UNSIGNED BRANCH

This example causes a branch to location NOT if C is not a digit (i.e., C is treated as an unsigned number outside the range 0 through 9).

```
CMPB C,#^A/0/       ;Compare C and ASCII representation
                    ;of digit 0.

BLSSU NOT           ;Branch to location NOT if less than an
                    ;unsigned 0.

CMPB C,#^A/9/       ;Compare C and ASCII representation
                    ;of digit 9.

BGTRU NOT           ;Branch to location NOT if greater than
                    ;an unsigned 9.
```

Example #2: BRANCH ON BIT

```
BBS #2,B,X          ;Branches to X if bit <2> in B is set (=1)

BBSC #2,B,X         ;Branches to X if bit <2> in B is set (=1)
                    ;and bit is then cleared.

BLBS B,X            ;Branches to X if bit <0> of B is set (=1)
```

CHAPTER 4

INSTRUCTION SET

4.1 INTRODUCTION

This chapter describes the instructions used by the MicroVAX 78032 CPU. The MicroVAX 78032 CPU implements a subset of the VAX instruction set. The instruction set is divided into the following major sections:

- Integer arithmetic and logical
- Address
- Variable length bit field
- Control
- Procedure call
- Miscellaneous
- Queue
- Character string
- Operating system support
- Floating point
- Emulated instructions with microcode assist

A concise list of instructions and opcode assignments appears in Appendix B.

4.1.1 Instruction Descriptions

Within each major section, instructions which are closely related are combined into groups and described together. The instruction group description is composed of the following:

1. The group name.

2. The format of each instruction in the group. This gives the name and type of each instruction operand specifier and the order in which it appears in memory. Operand specifiers from left to right appear in increasing memory addresses.

3. The operation of the instruction.

4. The effect on condition codes.

5. Exceptions specific to the instruction. Exceptions which are generally possible for all instructions (e.g., illegal or reserved addressing mode, T-bit, memory management violations, etc.) are not listed.

6. The opcodes, mnemonics, and names of each instruction in the group. The opcodes are given in hex.

7. A description in English of the instruction.

8. Optional notes on the instruction and programming examples.

4.1.2 Operand Specifier Notation

Operand specifiers are described in the following way:

<name>.<access type><data type>

where:

1. Name is a suggestive name for the operand in the context of the instruction. The name is often abbreviated.

2. Access type is a letter denoting the operand specifier access type:

   a - Calculate the effective address of the specified operand. Address is returned in a longword which is the actual instruction operand. Context of address calculation is given by <data type>; i.e. size to be used in autoincrement, autodecrement, and indexing.

b - No operand reference. Operand specifier is a branch displacement. Size of branch displacement is given by <data type>.

m - Operand is read, potentially modified and written. Note that this is NOT an indivisible memory operation. Also note that if the operand is not actually modified, it may not be written back. However, modify type operands are always checked for both read and write accessibility.

r - Operand is read only.

v - Calculate the effective address of the specified operand. If the effective address is in memory, the address is returned in a longword which is the actual instruction operand. Context of address calculation is given by <data type>. If the effective address is Rn, the operand is in Rn or R[n+1]'Rn.

w - Operand is write only.

3. Data type is a letter denoting the data type of the operand:

b - byte

w - word

l - longword

q - quadword

f - F_floating

d - D_floating

g - G_floating

x - first data type specified by instruction

y - second data type specified by instruction

4.1.3 Operation Description Notation

The operation of each instruction is given as a sequence of control and assignment statements. Table 4-1 describes the symbols used when describing an operation.

Table 4-1 Instruction Operation Symbols

Symbol	Description
+	addition
-	subtraction, unary minus
*	multiplication
/	division (quotient only)
**	exponentiation
'	concatenation
<-	is replaced by
=	is defined as
Rn or R[n]	contents of register Rn
PC, SP, FP, or AP	the contents of register R15, R14, R13, or R12 respectively
PSW	the contents of the processor status word
PSL	the contents of the processor status long word
(x)	contents of memory location whose address is x
(x)+	contents of memory location whose address is x; x incremented by the size of operand referenced at x
-(x)	x decremented by size of operand to be referenced at x; contents of memory location whose address is x
<x:y>	a modifier which delimits an extent from bit position x to bit position y inclusive
<x1,x2,..,xn>	a modifier which enumerates bits x1,x2,...,xn
{ }	arithmetic parentheses used to indicate precedence
AND	logical AND
OR	logical OR
XOR	logical XOR
NOT	logical (ones) complement
LSS	less than signed
LSSU	less than unsigned
LEQ	less than or equal signed
LEQU	less than or equal unsigned
EQL	equal signed
EQLU	equal unsigned
NEQ	not equal signed
NEQU	not equal unsigned
GEQ	greater than or equal signed
GEQU	greater than or equal unsigned
GTR	greater than signed
GTRU	greater than unsigned
SEXT(x)	x is sign extended to size of operand needed
ZEXT(x)	x is zero extended to size of operand needed
REM(x,y)	remainder of x divided by y, such that x/y and REM(x,y) have the same sign
MINU(x,y)	minimum unsigned of x and y
MAXU(x,y)	maximum unsigned of x and y

The following conventions are used when describing the operation of an instruction.

1. Other than that caused by ( )+, or -( ), and the advancement of PC, only operands or portions of operands appearing on the left side of assignment statements are affected.

2. No operator precedence is assumed, other than that replacement (<-) has the lowest precedence. Precedence is indicated explicitly by { }.

3. All arithmetic, logical, and relational operators are defined in the context of their operands. For example "+" applied to floating operands means a floating add, while "+" applied to byte operands is an integer byte add. Similarly, "LSS" is a floating comparison when applied to floating operands, while "LSS" is an integer byte comparison when applied to byte operands.

4. Instruction operands are evaluated according to the operand specifier conventions. The order in which operands appear in the instruction description has no effect on the order of evaluation.

5. Condition codes are in general affected on the value of actual stored results, not on "true" results (which might be generated internally to greater precision). Thus, for example, two positive integers can be added together and the sum stored, because of overflow, as a negative value. The condition codes will indicate a negative value even though the "true" result is clearly positive.

4.2 INTEGER ARITHMETIC AND LOGICAL INSTRUCTIONS

ADAWI Add Aligned Word Interlocked

Format:

```
opcode add.rw, sum.mw
```

Operation:

```
tmp <- add;
{set interlock};
sum <- sum + tmp;
{release interlock};
```

Condition Codes:

```
N <- sum LSS 0;
Z <- sum EQL 0;
V <- {integer overflow};
C <- {carry from most significant bit};
```

Exceptions:

```
reserved operand fault
integer overflow
```

Opcodes:

58	ADAWI	Add Aligned Word Interlocked

Description:

The addend operand is added to the sum operand and the sum operand is replaced by the result. The operation is interlocked against similar operations on other processors in a multiprocessor system. The destination must be aligned on a word boundary, i.e., bit 0 of the address of the sum operand must be zero. If it is not, a reserved operand fault is taken.

Notes:

1. Integer overflow occurs if the input operands to the add have the same sign and the result has the opposite sign. On overflow, the sum operand is replaced by the low order bits of the true result.

2. If the addend and the sum operands overlap, the result and the condition codes are unpredictable.

ADD Add

Format:

```
opcode add.rx, sum.mx              2 operand

opcode add1.rx, add2.rx, sum.wx 3 operand
```

Operation:

```
sum <- sum + add;          !2 operand

sum <- add1 + add2;        !3 operand
```

Condition Codes:

```
N <- sum LSS 0;
Z <- sum EQL 0;
V <- {integer overflow};
C <- {carry from most significant bit};
```

Exceptions:

integer overflow

Opcodes:

80	ADDB2	Add Byte 2 Operand
81	ADDB3	Add Byte 3 Operand
A0	ADDW2	Add Word 2 Operand
A1	ADDW3	Add Word 3 Operand
C0	ADDL2	Add Long 2 Operand
C1	ADDL3	Add Long 3 Operand

Description:

In 2 operand format, the addend operand is added to the sum operand and the sum operand is replaced by the result. In 3 operand format, the addend 1 operand is added to the addend 2 operand and the sum operand is replaced by the result.

Notes:

Integer overflow occurs if the input operands to the add have the same sign and the result has the opposite sign. On overflow, the sum operand is replaced by the low order bits of the true result.

ADWC Add With Carry

Format:

```
opcode add.rl, sum.ml
```

Operation:

```
sum <- sum + add + C;
```

Condition Codes:

```
N <- sum LSS 0;
Z <- sum EQL 0;
V <- {integer overflow};
C <- {carry from most significant bit};
```

Exceptions:

integer overflow

Opcodes:

D8	ADWC	Add With Carry

Description:

The contents of the condition code C bit and the addend operand are added to the sum operand and the sum operand is replaced by the result.

Notes:

1. On overflow, the sum operand is replaced by the low order bits of the true result.

2. The 2 additions in the operation are performed simultaneously.

ASH Arithmetic Shift

Format:

```
opcode cnt.rb, src.rx, dst.wx
```

Operation:

```
dst <- src shifted cnt bits;
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- {integer overflow};
C <- 0;
```

Exceptions:

integer overflow

Opcodes:

78	ASHL	Arithmetic Shift Long
79	ASHQ	Arithmetic Shift Quad

Description:

The source operand is arithmetically shifted by the number of bits specified by the count operand and the destination operand is replaced by the result. The source operand is unaffected. A positive count operand shifts to the left bringing 0's into the least significant bit. A negative count operand shifts to the right bringing in copies of the most significant (sign) bit into the most significant bit. A 0 count operand replaces the destination operand with the unshifted source operand.

Notes:

1. Integer overflow occurs on a left shift if any bit shifted into the sign bit position differs from the sign bit of the source operand.

2. If cnt GTR 32 (ASHL) or cnt GTR 64 (ASHQ) the destination operand is replaced by 0.

3. If cnt LEQ -31 (ASHL) or cnt LEQ -63 (ASHQ) all the bits of the destination operand are copies of the sign bit of the source operand.

BIC Bit Clear

Format:

```
opcode mask.rx, dst.mx              2 operand
opcode mask.rx, src.rx, dst.wx      3 operand
```

Operation:

```
dst <- dst AND {NOT mask};          !2 operand
dst <- src AND {NOT mask};          !3 operand
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- 0;
C <- C;
```

Exceptions:

none

Opcodes:

8A	BICB2	Bit Clear Byte
8B	BICB3	Bit Clear Byte
AA	BICW2	Bit Clear Word
AB	BICW3	Bit Clear Word
CA	BICL2	Bit Clear Long
CB	BICL3	Bit Clear Long

Description:

In 2 operand format, the destination operand is ANDed with the ones complement of the mask operand and the destination operand is replaced by the result. In 3 operand format, the source operand is ANDed with the ones complement of the mask operand and the destination operand is replaced by the result.

BIS Bit Set

Format:

```
opcode mask.rx, dst.mx            2 operand

opcode mask.rx, src.rx, dst.wx    3 operand
```

Operation:

```
dst <- dst OR mask;       !2 operand

dst <- src OR mask;       !3 operand
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- 0;
C <- C;
```

Exceptions:

none

Opcodes:

88	BISB2	Bit Set Byte 2 Operand
89	BISB3	Bit Set Byte 3 Operand
A8	BISW2	Bit Set Word 2 Operand
A9	BISW3	Bit Set Word 3 Operand
C8	BISL2	Bit Set Long 2 Operand
C9	BISL3	Bit Set Long 3 Operand

Description:

In 2 operand format, the mask operand is ORed with the destination operand and the destination operand is replaced by the result. In 3 operand format, the mask operand is ORed with the source operand and the destination operand is replaced by the result.

BIT Bit Test

Format:

```
opcode mask.rx, src.rx
```

Operation:

```
tmp <- src AND mask;
```

Condition Codes:

```
N <- tmp LSS 0;
Z <- tmp EQL 0;
V <- 0;
C <- C;
```

Exceptions:

none

Opcodes:

93	BITB	Bit Test Byte
B3	BITW	Bit Test Word
D3	BITL	Bit Test Long

Description:

The mask operand is ANDed with the source operand. Both operands are unaffected. The only action is to affect condition codes.

CLR Clear

Format:

```
opcode dst.wx
```

Operation:

```
dst <- 0;
```

Condition Codes:

```
N <- 0;
Z <- 1;
V <- 0;
C <- C;
```

Exceptions:

none

Opcodes:

94	CLRB	Clear Byte
B4	CLRW	Clear Word
D4	CLRL	Clear Long
	CLRF	Clear F_floating
7C	CLRQ	Clear Quad
	CLRD	Clear D_floating
	CLRG	Clear G_floating

Description:

The destination operand is replaced by 0.

Notes:

CLRx dst is equivalent to MOVx S^#0,dst, but is 1 byte shorter.

CMP Compare

Format:

```
opcode src1.rx, src2.rx
```

Operation:

```
src1 - src2;
```

Condition Codes:

```
N <- src1 LSS src2;
Z <- src1 EQL src2;
V <- 0;
C <- src1 LSSU src2;
```

Exceptions:

none

Opcodes:

91	CMPB	Compare Byte
B1	CMPW	Compare Word
D1	CMPL	Compare Long

Description:

The source 1 operand is compared with the source 2 operand. The only action is to affect the condition codes.

CVT Convert

Format:

```
opcode src.rx, dst.wy
```

Operation:

```
dst <- conversion of src;
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- {integer overflow};
C <- 0;
```

Exceptions:

integer overflow

Opcodes:

99	CVTBW	Convert Byte to Word
98	CVTBL	Convert Byte to Long
33	CVTWB	Convert Word to Byte
32	CVTWL	Convert Word to Long
F6	CVTLB	Convert Long to Byte
F7	CVTLW	Convert Long to Word

Description:

The source operand is converted to the data type of the destination operand and the destination operand is replaced by the result. Conversion of a shorter data type to a longer is done by sign extension; conversion of longer to a shorter is done by truncation of the higher numbered (most significant) bits.

Notes:

Integer overflow occurs if any truncated bits of the source operand are not equal to the sign bit of the destination operand.

DEC Decrement

Format:

```
opcode dif.mx
```

Operation:

```
dif <- dif - 1;
```

Condition Codes:

```
N <- dif LSS 0;
Z <- dif EQL 0;
V <- {integer overflow};
C <- {borrow into most significant bit};
```

Exceptions:

integer overflow

Opcodes:

97	DECB	Decrement Byte
B7	DECW	Decrement Word
D7	DECL	Decrement Long

Description:

One is subtracted from the difference operand and the difference operand is replaced by the result.

Notes:

1. Integer overflow occurs if the largest negative integer is decremented. On overflow, the difference operand is replaced by the largest positive integer.

2. DECx dif is equivalent to SUBx2 S^#1,dif, but is 1 byte shorter.

DIV Divide

Format:

```
opcode divr.rx, quo.mx                    2 operand

opcode divr.rx, divd.rx, quo.wx           3 operand
```

Operation:

```
quo <- quo / divr;        !2 operand

quo <- divd / divr;       !3 operand
```

Condition Codes:

```
N <- quo LSS 0;
Z <- quo EQL 0;
V <- {integer overflow} OR {divr EQL 0};
C <- 0;
```

Exceptions:

integer overflow
divide by zero

Opcodes:

86	DIVB2	Divide Byte 2 Operand
87	DIVB3	Divide Byte 3 Operand
A6	DIVW2	Divide Word 2 Operand
A7	DIVW3	Divide Word 3 Operand
C6	DIVL2	Divide Long 2 Operand
C7	DIVL3	Divide Long 3 Operand

Description:

In 2 operand format, the quotient operand is divided by the divisor operand and the quotient operand is replaced by the result. In 3 operand format, the dividend operand is divided by the divisor operand and the quotient operand is replaced by the result.

Notes:

1. Integer overflow occurs if and only if the largest negative integer is divided by -1. On overflow, operands are affected as in Note 2.

2. If the divisor operand is 0, then in 2 operand format the quotient operand is not affected; in 3 operand format the quotient operand is replaced by the dividend operand.

EDIV Extended Divide

Format:

```
opcode divr.rl, divd.rq, quo.wl, rem.wl
```

Operation:

```
quo <- divd / divr;
rem <- REM(divd, divr);
```

Condition Codes:

```
N <- quo LSS 0;
Z <- quo EQL 0;
V <- {integer overflow} OR {divr EQL 0};
C <- 0;
```

Exceptions:

```
integer overflow
divide by zero
```

Opcodes:

7B	EDIV	Extended Divide

Description:

The dividend operand is divided by the divisor operand; the quotient operand is replaced by the quotient and the remainder operand is replaced by the remainder.

Notes:

1. The division is performed such that the remainder operand (unless it is 0) has the same sign as the dividend operand.

2. On overflow or if the divisor operand is 0, then the quotient operand is replaced by bits 31:0 of the dividend operand, and the remainder operand is replaced by 0.

EMUL Extended Multiply

Format:

```
opcode mulr.rl, muld.rl, add.rl, prod.wq
```

Operation:

```
prod <- {muld * mulr} + SEXT(add);
```

Condition Codes:

```
N <- prod LSS 0;
Z <- prod EQL 0;
V <- 0;
C <- 0;
```

Exceptions:

none

Opcodes:

7A	EMUL	Extended Multiply

Description:

The multiplicand operand is multiplied by the multiplier operand giving a quadword result. The addend operand is sign-extended to a quadword and added to the result. The product operand is replaced by the final result.

INC Increment

Format:

```
opcode sum.mx
```

Operation:

```
sum <- sum + 1;
```

Condition Codes:

```
N <- sum LSS 0;
Z <- sum EQL 0;
V <- {integer overflow};
C <- {carry from most significant bit};
```

Exceptions:

integer overflow

Opcodes:

96	INCB	Increment Byte
B6	INCW	Increment Word
D6	INCL	Increment Long

Description:

One is added to the sum operand and the sum operand is replaced by the result.

Notes:

1. Arithmetic overflow occurs if the largest positive integer is incremented. On overflow, the sum operand is replaced by the largest negative integer.

2. INCx sum is equivalent to ADDx2 S^#1,sum, but is 1 byte shorter.

MCOM Move Complemented

Format:

```
opcode src.rx, dst.wx
```

Operation:

```
dst <- NOT src;
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- 0;
C <- C;
```

Exceptions:

none

Opcodes:

92	MCOMB	Move Complemented Byte
B2	MCOMW	Move Complemented Word
D2	MCOML	Move Complemented Long

Description:

The destination operand is replaced by the ones complement of the source operand.

MNEG Move Negated

Format:

```
opcode src.rx, dst.wx
```

Operation:

```
dst <- -src;
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- {integer overflow};
C <- dst NEQ 0;
```

Exceptions:

integer overflow

Opcodes:

8E	MNEGB	Move Negated Byte
AE	MNEGW	Move Negated Word
CE	MNEGL	Move Negated Long

Description:

The destination operand is replaced by the negative of the source operand.

Notes:

1. Integer overflow occurs if the source operand is the largest negative integer (which has no positive counterpart). On overflow, the destination operand is replaced by the source operand.

2. MNEGx src.rx,dst.wx is equivalent to SUBx3 src.rx,S^#0,dst.rx, but is one byte shorter.

MOV Move

Format:

```
opcode src.rx, dst.wx
```

Operation:

```
dst <- src;
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- 0;
C <- C;
```

Exceptions:

none

Opcodes:

90	MOVB	Move Byte
B0	MOVW	Move Word
D0	MOVL	Move Long
7D	MOVQ	Move Quad

Description:

The destination operand is replaced by the source operand.

MOVZ Move Zero-Extended

Format:

```
opcode src.rx, dst.wy
```

Operation:

```
dst <- ZEXT(src);
```

Condition Codes:

```
N <- 0;
Z <- dst EQL 0;
V <- 0;
C <- C;
```

Exceptions:

none

Opcodes:

9B	MOVZBW	Move Zero-Extended Byte to Word
9A	MOVZBL	Move Zero-Extended Byte to Long
3C	MOVZWL	Move Zero-Extended Word to Long

Description:

For MOVZBW, bits <7:0> of the destination operand are replaced by the source operand; bits <15:8> are replaced by zero. For MOVZBL, bits <7:0> of the destination operand are replaced by the source operand; bits <31:8> are replaced by 0. For MOVZWL, bits <15:0> of the destination operand are replaced by the source operand; bits <31:16> are replaced by 0.

MUL Multiply

Format:

```
opcode mulr.rx, prod.mx                    2 operand

opcode mulr.rx, muld.rx, prod.wx           3 operand
```

Operation:

```
prod <- prod * mulr;     !2 operand

prod <- muld * mulr;     !3 operand
```

Condition Codes:

```
N <- prod LSS 0;
Z <- prod EQL 0;
V <- {integer overflow};
C <- 0;
```

Exceptions:

integer overflow

Opcodes:

84	MULB2	Multiply Byte 2 Operand
85	MULB3	Multiply Byte 3 Operand
A4	MULW2	Multiply Word 2 Operand
A5	MULW3	Multiply Word 3 Operand
C4	MULL2	Multiply Long 2 Operand
C5	MULL3	Multiply Long 3 Operand

Description:

In 2 operand format, the product operand is multiplied by the multiplier operand and the product operand is replaced by the low half of the double length result. In 3 operand format, the multiplicand operand is multiplied by the multiplier operand and the product operand is replaced by the low half of the double length result.

Notes:

Integer overflow occurs if the high half of the double length result is not equal to the sign extension of the low half.

PUSHL Push Long

Format:

```
opcode src.rl
```

Operation:

```
-(SP) <- src;
```

Condition Codes:

```
N <- src LSS 0;
Z <- src EQL 0;
V <- 0;
C <- C;
```

Exceptions:

none

Opcodes:

DD	PUSHL	Push Long

Description:

The longword source operand is pushed on the stack.

Notes:

PUSHL is equivalent to MOVL src,-(SP), but is 1 byte shorter.

ROTL Rotate Long

Format:

```
opcode cnt.rb, src.rl, dst.wl
```

Operation:

```
dst <- src rotated cnt bits;
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- 0;
C <- C;
```

Exceptions:

none

Opcodes:

9C	ROTL	Rotate Long

Description:

The source operand is rotated logically by the number of bits specified by the count operand and the destination operand is replaced by the result. The source operand is unaffected. A positive count operand rotates to the left. A negative count operand rotates to the right. A 0 count operand replaces the destination operand with the source operand.

SBWC Subtract With Carry

Format:

```
pcode sub.rl, dif.ml
```

Operation:

```
dif <- dif - sub - C;
```

Condition Codes:

```
N <- dif LSS 0;
Z <- dif EQL 0;
V <- {integer overflow};
C <- {borrow into most significant bit};
```

Exceptions:

integer overflow

Opcodes:

D9	SBWC	Subtract With Carry

Description:

The subtrahend operand and the contents of the condition code C bit are subtracted from the difference operand and the difference operand is replaced by the result.

Notes:

1. On overflow, the difference operand is replaced by the low order bits of the true result.

2. The 2 subtractions in the operation are performed simultaneously.

SUB Subtract

Format:

```
opcode sub.rx, dif.mx              2 operand

opcode sub.rx, min.rx, dif.wx      3 operand
```

Operation:

```
dif <- dif - sub;          !2 operand

dif <- min - sub;          !3 operand
```

Condition Codes:

```
N <- dif LSS 0;
Z <- dif EQL 0;
V <- {integer overflow};
C <- {borrow into most significant bit};
```

Exceptions:

integer overflow

Opcodes:

82	SUBB2	Subtract Byte 2 Operand
83	SUBB3	Subtract Byte 3 Operand
A2	SUBW2	Subtract Word 2 Operand
A3	SUBW3	Subtract Word 3 Operand
C2	SUBL2	Subtract Long 2 Operand
C3	SUBL3	Subtract Long 3 Operand

Description:

In 2 operand format, the subtrahend operand is subtracted from the difference operand and the difference operand is replaced by the result. In 3 operand format, the subtrahend operand is subtracted from the minuend operand and the difference operand is replaced by the result.

Notes:

Integer overflow occurs if the input operands to the subtract are of different signs and the sign of the result is the sign of the subtrahend. On overflow, the difference operand is replaced by the low order bits of the true result.

TST Test

Format:

```
opcode src.rx
```

Operation:

```
src - 0;
```

Condition Codes:

```
N <- src LSS 0;
Z <- src EQL 0;
V <- 0;
C <- 0;
```

Exceptions:

none

Opcodes:

95	TSTB	Test Byte
B5	TSTW	Test Word
D5	TSTL	Test Long

Description:

The condition codes are affected according to the value of the source operand.

Notes:

TSTx src is equivalent to CMPx src,S^#0, but is 1 byte shorter.

XOR Exclusive OR

Format:

```
opcode mask.rx, dst.mx              2 operand

opcode mask.rx, src.rx, dst.wx      3 operand
```

Operation:

```
dst <- dst XOR mask;        !2 operand

dst <- src XOR mask;        !3 operand
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- 0;
C <- C;
```

Exceptions:

none

Opcodes:

8C	XORB2	Exclusive OR Byte 2 Operand
8D	XORB3	Exclusive OR Byte 3 Operand
AC	XORW2	Exclusive OR Word 2 Operand
AD	XORW3	Exclusive OR Word 3 Operand
CC	XORL2	Exclusive OR Long 2 Operand
CD	XORL3	Exclusive OR Long 3 Operand

Description:

In 2 operand format, the mask operand is XORed with the destination operand and the destination operand is replaced by the result. In 3 operand format, the mask operand is XORed with the source operand and the destination operand is replaced by the result.

4.3 ADDRESS INSTRUCTIONS

MOVA Move Address

Format:

```
opcode src.ax, dst.wl
```

Operation:

```
dst <- src;
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- 0;
C <- C;
```

Exceptions:

none

Opcodes:

9E	MOVAB	Move Address Byte
3E	MOVAW	Move Address Word
DE	MOVAL	Move Address Long
	MOVAF	Move Address F_floating
7E	MOVAQ	Move Address Quad
	MOVAD	Move Address D_floating
	MOVAG	Move Address G_floating

Description:

The destination operand is replaced by the source operand. The context in which the source operand is evaluated is given by the data type of the instruction. The operand whose address replaces the destination operand is not referenced.

PUSHA Push Address

Format:

```
opcode src.ax
```

Operation:

```
-(SP) <- src;
```

Condition Codes:

```
N <- src LSS 0;
Z <- src EQL 0;
V <- 0;
C <- C;
```

Exceptions:

none

Opcodes:

9F	PUSHAB	Push Address Byte
3F	PUSHAW	Push Address Word
DF	PUSHAL	Push Address Long
	PUSHAF	Push Address F_floating
7F	PUSHAQ	Push Address Quad
	PUSHAD	Push Address D_floating
	PUSHAG	Push Address G_floating

Description:

The source operand is pushed on the stack. The context in which the source operand is evaluated is given by the data type of the instruction. The operand whose address is pushed is not referenced.

Notes:

PUSHAx src is equivalent to MOVAx src,-(SP), but is 1 byte shorter.

4.4 VARIABLE LENGTH BIT FIELD INSTRUCTIONS

A variable length bit field is specified by 3 operands:

1. A longword position operand.

2. A byte field size operand which must be in the range 0 through 32 or a reserved operand fault occurs.

3. A base address (relative to which the position is used to locate the bit field). The address is obtained from an operand of address access type. However, unlike other instances of operand specifiers of address access type, register mode may be designated in the operand specifier. In this case the field is contained in the register n designated by the operand specifier (or register n+1 concatenated with register n). If the field is contained in a register and size is not zero, the position operand must have a value in the range 0 through 31 or a reserved operand fault occurs.

In order to simplify the description of the variable bit field instructions, a macro FIELD(pos, size, address) is introduced with the following expansion (if size NEQ 0):

```
FIELD(pos, size, address)

=(address + SEXT(pos<31:3>))<{size - 1} + pos<2:0>:pos<2:0>>

        !if address not specified by register mode

= {R[n+1]'Rn}<{size - 1} + pos:pos>


        !if address specified by register mode and pos + size
        !GTRU 32

= Rn<{size - 1} + pos:pos>

        !if address specified by register mode and pos + size
        !LEQU 32
```

The number of bytes referenced by the contents ( ) operator above is:

```
1 + {{{size - 1} + pos<2:0>} / 8}
```

Zero bytes are referenced if the field size is 0.

CMP Compare Field

Format:

```
opcode pos.rl, size.rb, base.vb, src.rl
```

Operation:

```
tmp <- if size NEQU 0 then SEXT(FIELD (pos, size, base))
                      else 0;          !CMPV
tmp <- if size NEQU 0 then ZEXT(FIELD (pos, size, base))
                      else 0;          !CMPZV

tmp - src;
```

Condition Codes:

```
N <- tmp LSS src;
Z <- tmp EQL src;
V <- 0;
C <- tmp LSSU src;
```

Exceptions:

reserved operand

Opcodes:

EC	CMPV	Compare Field
ED	CMPZV	Compare Zero-Extended Field

Description:

The field specified by the position, size, and base operands is compared with the source operand. For CMPV, the source operand is compared with the sign extended field. For CMPZV, the source operand is compared with the zero extended field. The only action is to affect the condition codes.

Notes:

1. A reserved operand fault occurs if:

   a. size GTRU 32.

   b. pos GTRU 31, size NEQ 0, and the field is contained in the registers.

2. On a reserved operand fault, the condition codes are unpredictable.

EXT Extract Field

Format:

```
opcode pos.rl, size.rb, base.vb, dst.wl
```

Operation:

```
dst <- if size NEQU 0 then SEXT(FIELD(pos, size, base))
                      else 0;          !EXTV

dst <- if size NEQU 0 then ZEXT(FIELD(pos, size, base))
                      else 0;          !EXTZV
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- 0;
C <- C;
```

Exceptions:

reserved operand

Opcodes:

EE	EXTV	Extract Field
EF	EXTZV	Extract Zero-Extended Field

Description:

For EXTV, the destination operand is replaced by the sign extended field specified by the position, size, and base operands. For EXTZV, the destination operand is replaced by the zero extended field specified by the position, size and base operands. If the size operand is 0, the only action is to replace the destination operand with 0 and affect the condition codes.

Notes:

1. A reserved operand fault occurs if:

   a. size GTRU 32.

   b. pos GTRU 31, size NEQ 0, and the field is contained in the registers.

2. On a reserved operand fault, the destination operand is unaffected and the condition codes are unpredictable.

FF Find First

Format:

```
opcode startpos.rl, size.rb, base.vb, findpos.wl
```

Operation:

```
state = if {FFS} then 1 else 0;
if size NEQU 0 then
        begin
        tmp1 <- FIELD(startpos, size, base);
        tmp2 <- 0;
        while {tmp1<tmp2> NEQ state} AND
                {tmp2 LEQU {size - 1}} do
                tmp2 <- tmp2 + 1;
                findpos <- startpos + tmp2;
        end
else
        findpos <- startpos;
```

Condition Codes:

```
N <- 0;
Z <- {bit not found};
V <- 0;
C <- 0;
```

Exceptions:

reserved operand

Opcodes:

EB	FFC	Find First Clear
EA	FFS	Find First Set

Description:

A field specified by the start position, size, and base operands is extracted. The field is tested for a bit in the state indicated by the instruction, starting at bit 0 and extending to the highest bit in the field. If a bit in the indicated state is found, the find position operand is replaced by the position of the bit, and the Z condition code bit is cleared. If no bit in the indicated state is found, the find position operand is replaced by the position (relative to the base) of a bit one position to the left of the specified field, and the Z condition code bit is set. If the size operand is 0, the find position operand is replaced by the start position operand, and the Z condition code bit is set.

Notes:

1. A reserved operand fault occurs if:

   a. size GTRU 32.

   b. startpos GTRU 31, size NEQ 0, and the field is contained in the registers.

2. On a reserved operand fault, the find position operand is unaffected and the condition codes are unpredictable.

INSV Insert Field

Format:

```
opcode src.rl, pos.rl, size.rb, base.vb
```

Operation:

```
if size NEQU 0 then FIELD(pos, size, base) <- src<{size-1}:0>;
```

Condition Codes:

```
N <- N;
Z <- Z;
V <- V;
C <- C;
```

Exceptions:

reserved operand

Opcodes:

F0	INSV	Insert Field

Description:

The field specified by the position, size, and base operands is replaced by bits size-1:0 of the source operand. If the size operand is 0, the instruction has no effect.

Notes:

1. A reserved operand fault occurs if:

   a. size GTRU 32.

   b. pos GTRU 31, size NEQ 0, and the field is contained in the registers.

2. On a reserved operand fault, the field is unaffected and the condition codes are unpredictable.

4.5 CONTROL INSTRUCTIONS

ACB Add Compare and Branch

Format:

```
opcode limit.rx, add.rx, index.mx, displ.bw
```

Operation:

```
index <- index + add;
if {{add GEQ 0} AND {index LEQ limit}} OR
        {{add LSS 0} AND {index GEQ limit}} then
        PC <- PC + SEXT(displ);
```

Condition Codes:

```
N <- index LSS 0;
Z <- index EQL 0;
V <- {integer overflow};
C <- C;
```

Exceptions:

integer overflow

Opcodes:

9D	ACBB	Add Compare and Branch Byte
3D	ACBW	Add Compare and Branch Word
F1	ACBL	Add Compare and Branch Long

Description:

The addend operand is added to the index operand and the index operand is replaced by the result. The index operand is compared with the limit operand. If the addend operand is positive (or 0) and the comparison is less than or equal, or if the addend is negative and the comparison is greater than or equal, the sign-extended branch displacement is added to PC and PC is replaced by the result.

Notes:

1. ACB efficiently implements the general FOR or DO loops in high level languages, since the sense of the comparison between index and limit is dependent on the sign of the addend.

2. On integer overflow, the index operand is replaced by the low order bits of the true result. Comparison and branch determination proceed normally on the updated index operand.

AOBLEQ Add One and Branch Less Than or Equal

Format:

```
opcode limit.rl, index.ml, displ.bb
```

Operation:

```
index <- index + 1;
if index LEQ limit then PC <-
        PC + SEXT(displ);
```

Condition Codes:

```
N <- index LSS 0;
Z <- index EQL 0;
V <- {integer overflow};
C <- C;
```

Exceptions:

integer overflow

Opcodes:

F3	AOBLEQ	Add One and Branch Less Than or Equal

Description:

One is added to the index operand and the index operand is replaced by the result. The index operand is compared with the limit operand. If it is less than or equal, the sign-extended branch displacement is added to PC and PC is replaced by the result.

Notes:

Integer overflow occurs if the index operand before addition is the largest positive integer. On overflow, the index operand is replaced by the largest negative integer, and the branch is taken.

AOBLSS Add One and Branch Less Than

Format:

```
opcode limit.rl, index.ml, displ.bb
```

Operation:

```
index <- index + 1;
if index LSS limit then PC <-
        PC + SEXT(displ);
```

Condition Codes:

```
N <- index LSS 0;
Z <- index EQL 0;
V <- {integer overflow};
C <- C;
```

Exceptions:

integer overflow

Opcodes:

F2	AOBLSS	Add One and Branch Less Than

Description:

One is added to the index operand and the index operand is replaced by the result. The index operand is compared with the limit operand. If it is less than, the sign-extended branch displacement is added to the PC and PC is replaced by the result.

Notes:

Integer overflow occurs if the index operand before addition is the largest positive integer. On overflow, the index operand is replaced by the largest negative integer, and, unless the limit operand is the largest negative integer, the branch is taken.

B Branch on (condition)

Format:

```
opcode displ.bb
```

Operation:

```
if condition then PC <- PC + SEXT(displ);
```

Condition Codes:

```
N <- N;
Z <- Z;
V <- V;
C <- C;
```

Exceptions:

none

Opcodes:	Condition		
14	{N OR Z} EQL 0	BGTR	Branch on Greater Than (signed)
15	{N OR Z} EQL 1	BLEQ	Branch on Less Than or Equal (signed)
12	Z EQL 0	BNEQ, BNEQU	Branch on Not Equal (signed) Branch on Not Equal Unsigned
13	Z EQL 1	BEQL, BEQLU	Branch on Equal (signed) Branch on Equal Unsigned
18	N EQL 0	BGEQ	Branch on Greater Than or Equal (signed)
19	N EQL 1	BLSS	Branch on Less Than (signed)
1A	{C OR Z} EQL 0	BGTRU	Branch on Greater Than Unsigned
1B	{C OR Z} EQL 1	BLEQU	Branch Less Than or Equal Unsigned
1C	V EQL 0	BVC	Branch on Overflow Clear
1D	V EQL 1	BVS	Branch on Overflow Set
1E	C EQL 0	BGEQU, BCC	Branch on Greater Than or Equal Unsigned Branch on Carry Clear
1F	C EQL 1	BLSSU, BCS	Branch on Less Than Unsigned Branch on Carry Set

Description:

The condition codes are tested and if the condition indicated by the instruction is met, the sign-extended branch displacement is added to the PC and PC is replaced by the result.

Notes:

The VAX conditional branch instructions permit considerable flexibility in branching but require care in choosing the correct branch instruction. The conditional branch instructions are best seen as 3 overlapping groups:

1. Overflow and Carry Group

```
BVS     V EQL 1
BVC     V EQL 0
BCS     C EQL 1
BCC     C EQL 0
```

These instructions are typically used to check for overflow (when overflow traps are not enabled), for multiprecision arithmetic, and for other special purposes.

2. Unsigned Group

```
BLSSU   C EQL 1
BLEQU   {C OR Z} EQL 1
BEQLU   Z EQL 1
BNEQU   Z EQL 0
BGEQU   C EQL 0
BGTRU   {C OR Z} EQL 0
```

These instructions typically follow integer and field instructions where the operands are treated as unsigned integers, address instructions, and character string instructions.

3. Signed Group

```
BLSS    N EQL 1
BLEQ    {N OR Z} EQL 1
BEQL    Z EQL 1
BNEQ    Z EQL 0
BGEQ    N EQL 0
BGTR    {N OR Z} EQL 0
```

These instructions typically follow integer and field instructions where the operands are being treated as signed integers, floating point instructions, and decimal string instructions.

BB Branch on Bit

Format:

```
opcode pos.rl, base.vb, displ.bb
```

Operation:

```
teststate = if {BBS} then 1 else 0;
if FIELD(pos, 1, base) EQL teststate then
        PC <- PC + SEXT(displ);
```

Condition Codes:

```
N <- N;
Z <- Z;
V <- V;
C <- C;
```

Exceptions:

reserved operand

Opcodes:

E0	BBS	Branch on Bit Set
E1	BBC	Branch on Bit Clear

Description:

The single bit field specified by the position and base operands is tested. If it is in the test state indicated by the instruction, the sign-extended branch displacement is added to PC and PC is replaced by the result.

Notes:

1. A reserved operand fault occurs if pos GTRU 31 and the bit is contained in a register.

2. On a reserved operand fault, the condition codes are unpredictable.

BB Branch on Bit (and modify without interlock)

Format:

```
opcode pos.rl, base.vb, displ.bb
```

Operation:

```
teststate = if {BBSS or BBSC} then 1 else 0;
newstate = if {BBSS or BBCS} then 1 else 0;
tmp <- FIELD(pos, 1, base);
FIELD(pos, 1, base) <- newstate;
if tmp EQL teststate then
        PC <- PC + SEXT(displ);
```

Condition Codes:

```
N <- N;
Z <- Z;
V <- V;
C <- C;
```

Exceptions:

reserved operand

Opcodes:

E2	BBSS	Branch on Bit Set and Set
E3	BBCS	Branch on Bit Clear and Set
E4	BBSC	Branch on Bit Set and Clear
E5	BBCC	Branch on Bit Clear and Clear

Description:

The single bit field specified by the position and base operands is tested. If it is in the test state indicated by the instruction, the sign-extended branch displacement is added to PC and PC is replaced by the result. Regardless of whether the branch is taken or not, the tested bit is put in the new state as indicated by the instruction.

Notes:

1. A reserved operand fault occurs if pos GTRU 31 and the bit is contained in a register.

2. On a reserved operand fault, the field is unaffected and the condition codes are unpredictable.

3. The modification of the bit is not an interlocked operation. See BBSSI and BBCCI for interlocking instructions.

BB Branch on Bit Interlocked

Format:

```
opcode pos.rl, base.vb, displ.bb
```

Operation:

```
teststate = if {BBSSI} then 1 else 0;
newstate = teststate;
{set interlock};
tmp <- FIELD(pos, 1, base);
FIELD(pos, 1, base) <- newstate;
{release interlock};
if tmp EQL teststate then
        PC <- PC + SEXT(displ);
```

Condition Codes:

```
N <- N;
Z <- Z;
V <- V;
C <- C;
```

Exceptions:

reserved operand

Opcodes:

E6	BBSSI Branch on Bit Set and Set Interlocked
E7	BBCCI Branch on Bit Clear and Clear Interlocked

Description:

The single bit field specified by the position and base operands is tested. If it is in the test state indicated by the instruction, the sign-extended branch displacement is added to the PC and PC is replaced by the result. Regardless of whether the branch is effected or not, the tested bit is put in the new state as indicated by the instruction. If the bit is contained in memory, the reading of the state of the bit and the setting of it to the new state is an interlocked operation. No other processor or I/O device can do an interlocked access on the bit during the interlocked operation.

Notes:

1. A reserved operand fault occurs if pos GTRU 31 and the bit is contained in a register.

2. On a reserved operand fault, the field is unaffected and the condition codes are unpredictable.

3. Except for memory interlocking BBSSI is equivalent to BBSS and BBCCI is equivalent to BBCC.

4. This instruction is designed to support interlocks with other processors or devices.

   Example: To implement "busy waiting"

```
1$:     BBSSI   bit,base,1$
```

BLB Branch on Low Bit

Format:

```
opcode src.rl, displ.bb
```

Operation:

```
teststate = if {BLBS} then 1 else 0;
if src<0> EQL teststate then
        PC <- PC + SEXT(displ);
```

Condition Codes:

```
N <- N;
Z <- Z;
V <- V;
C <- C;
```

Exceptions:

none

Opcodes:

E8	BLBS	Branch on Low Bit Set
E9	BLBC	Branch on Low Bit Clear

Description:

The low bit (bit 0) of the source operand is tested and if it is equal to the test state indicated by the instruction, the sign-extended branch displacement is added to PC and PC is replaced by the result.

BR Branch

Format:

```
opcode displ.bx
```

Operation:

```
PC <- PC + SEXT(displ);
```

Condition Codes:

```
N <- N;
Z <- Z;
V <- V;
C <- C;
```

Exceptions:

none

Opcodes:

11	BRB	Branch With Byte Displacement
31	BRW	Branch With Word Displacement

Description:

The sign-extended branch displacement is added to PC and PC is replaced by the result.

BSB Branch To Subroutine

Format:

```
opcode displ.bx
```

Operation:

```
-(SP) <- PC;
PC <- PC + SEXT(displ);
```

Condition Codes:

```
N <- N;
Z <- Z;
V <- V;
C <- C;
```

Exceptions:

none

Opcodes:

10	BSBB	Branch to Subroutine With Byte Displacement
30	BSBW	Branch to Subroutine With Word Displacement

Description:

PC is pushed on the stack as a longword. The sign-extended branch displacement is added to PC and PC is replaced by the result.

CASE Case

Format:

```
opcode selector.rx, base.rx, limit.rx,
       displ[0].bw,..., displ[limit].bw
```

Operation:

```
tmp <- selector - base;
PC <- PC + if tmp LEQU limit then
        SEXT(displ[tmp]) else {2 + 2 * ZEXT(limit)};
```

Condition Codes:

```
N <- tmp LSS limit;
Z <- tmp EQL limit;
V <- 0;
C <- tmp LSSU limit;
```

Exceptions:

none

Opcodes:

8F	CASEB	Case Byte
AF	CASEW	Case Word
CF	CASEL	Case Long

Description:

The base operand is subtracted from the selector operand and a temporary is replaced by the result. The temporary is compared with the limit operand and if it is less than or equal unsigned, a branch displacement selected by the temporary value is added to PC and PC is replaced by the result. Otherwise, 2 times the sum of the limit operand and 1 is added to PC and PC is replaced by the result. This causes PC to be moved past the array of branch displacements. Regardless of the branch taken, the condition codes are affected by the comparison of the temporary operand with the limit operand.

Notes:

1. After operand evaluation, PC is pointing at displ[0], not the next instruction. The branch displacements are relative to the address of displ[0].

2. The selector and base operands can both be considered either as signed or unsigned integers.

JMP Jump

Format:

```
opcode dst.ab
```

Operation:

```
PC <- dst;
```

Condition Codes:

```
N <- N;
Z <- Z;
V <- V;
C <- C;
```

Exceptions:

none

Opcodes:

17	JMP	Jump

Description:

PC is replaced by the destination operand.

JSB Jump to Subroutine

Format:

```
opcode dst.ab
```

Operation:

```
-(SP) <- PC;
PC <- dst;
```

Condition Codes:

```
N <- N;
Z <- Z;
V <- V;
C <- C;
```

Exceptions:

none

Opcodes:

16	JSB	Jump to Subroutine

Description:

PC is pushed on the stack as a longword. PC is replaced by the destination operand.

Notes:

Since the operand specifier conventions cause the evaluation of the destination operand before saving PC, JSB can be used for coroutine calls with the stack used for linkage. The form of such a call is JSB @(SP)+.

RSB Return from Subroutine

Format:

opcode

Operation:

PC <- (SP)+;

Condition Codes:

N <- N;
Z <- Z;
V <- V;
C <- C;

Exceptions:

none

Opcodes:

05	RSB	Return From Subroutine

Description:

PC is replaced by a longword popped from the stack.

Notes:

1. RSB is used to return from subroutines called by the BSBB, BSBW and JSB instructions.

2. RSB is equivalent to JMP @(SP)+, but is 1 byte shorter.

SOBGEQ Subtract One and Branch Greater Than or Equal

Format:

```
opcode index.ml, displ.bb
```

Operation:

```
index <- index - 1;
if index GEQ 0 then PC <-
        PC + SEXT(displ);
```

Condition Codes:

```
N <- index LSS 0;
Z <- index EQL 0;
V <- {integer overflow};
C <- C;
```

Exceptions:

integer overflow

Opcodes:

F4	SOBGEQ	Subtract One and Branch Greater Than or Equal

Description:

One is subtracted from the index operand and the index operand is replaced by the result. If the index operand is greater than or equal to 0, the sign-extended branch displacement is added to PC and PC is replaced by the result.

Notes:

Integer overflow occurs if the index operand before subtraction is the largest negative integer. On overflow, the index operand is replaced by the largest positive integer, and the branch is taken.

SOBGTR Subtract One and Branch Greater Than

Format:

```
opcode index.ml, displ.bb
```

Operation:

```
index <- index - 1;
if index GTR 0 then PC <-
        PC + SEXT(displ);
```

Condition Codes:

```
N <- index LSS 0;
Z <- index EQL 0;
V <- {integer overflow};
C <- C;
```

Exceptions:

integer overflow

Opcodes:

F5	SOBGTR	Subtract One and Branch Greater Than

Description:

One is subtracted from the index operand and the index operand is replaced by the result. If the index operand is greater than 0, the sign-extended branch displacement is added to PC and PC is replaced by the result.

Notes:

Integer overflow occurs if the index operand before subtraction is the largest negative integer. On overflow, the index operand is replaced by the largest positive integer, and the branch is taken.

4.6 PROCEDURE CALL INSTRUCTIONS

Three instructions are used to implement a standard procedure calling interface. Two instructions implement the CALL to the procedure; the third implements the matching RETURN. The CALLG instruction calls a procedure with the argument list actuals in an arbitrary location. The CALLS instruction calls a procedure with the argument list actuals on the stack. Upon return after a CALLS this list is automatically removed from the stack. Both call instructions specify the address of the entry point of the procedure being called. The entry point is assumed to consist of a word termed the entry mask followed by the procedure's instructions. The procedure terminates by executing a RET instruction.

The entry mask, Figure 4-1, specifies the procedure's register use and overflow enables:

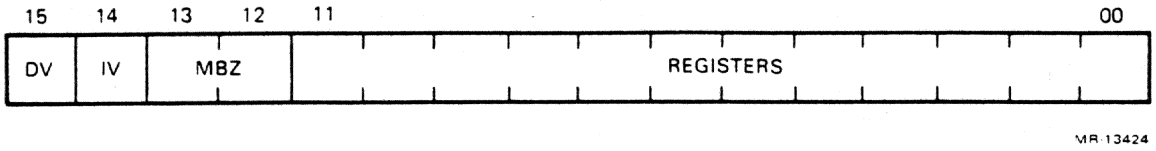


Figure 4-1 Entry Mask

On CALL the stack is aligned to a longword bouhdary and the trap enables in the PSW are set to a known state to ensure consistent behavior of the called procedure. Integer overflow enable and decimal overflow enable are affected according to bits 14 and 15 of the entry mask respectively. Floating underflow enable is cleared. The registers R11 through R0 specified by bits 11 through 0 respectively are saved on the stack and are restored by the RET instruction. In addition, PC, SP, FP, and AP are always preserved by the CALL instructions and restored by the RET instruction.

All external procedure CALLs generated by standard DIGITAL language processors, and all inter-module CALLs to major VAX software subsystems, comply with the procedure calling software standard. The procedure calling standard requires that all registers in the range R2 through R11 used in the procedure must appear in the mask. R0 and R1 are not preserved by any called procedure that complies with the procedure calling standard.

In order to preserve the state, the CALL instructions form a structure on the stack termed a call frame or stack frame. This contains the saved registers, the saved PSW, the register save mask, and several control bits. The frame also includes a longword which the CALL instructions clear; this is used to implement the VAX/VMS condition

handling facility. At the end of execution of the CALL instruction, FP contains the address of the stack frame. The RET instruction uses the contents of FP to find the stack frame and restore state. The condition handling facility assumes that FP always points to the stack frame. The stack frame has the format shown in Figure 4-2.

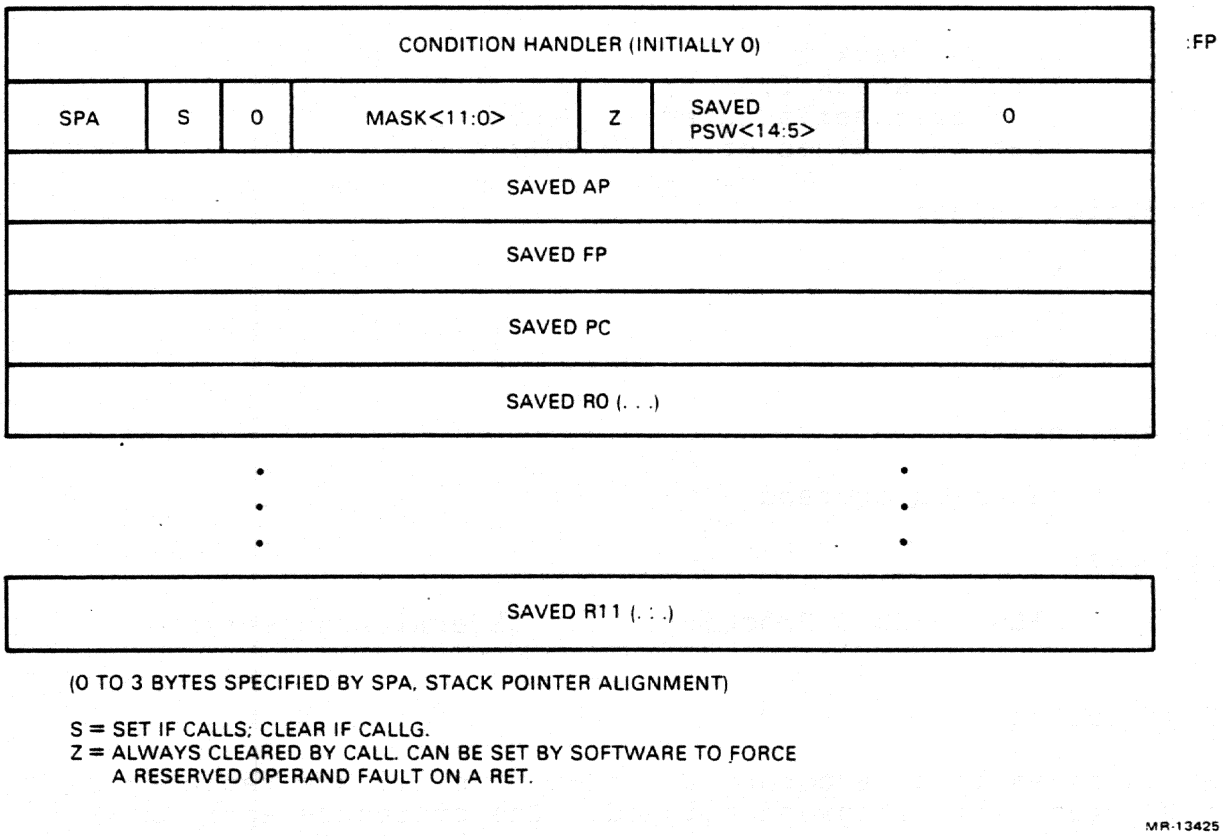


Figure 4-2 Stack Frame

Note that the saved condition codes and the saved trace enable (PSW<T>) are cleared.

The contents of the frame PSW<3:0> at the time RET is executed will become the condition codes resulting from the execution of the procedure. Similarly, the content of the frame PSW<4> at the time the RET is executed will become the PSW<T> bit.

CALLG Call Procedure With General Argument List

Format:

```
opcode arglist.ab, dst.ab
```

Operation:

```
{align stack};
{create stack frame};
{set arithmetic exception enables};
{set new values of AP,FP,PC};
```

Condition Codes:

```
N <- 0;
Z <- 0;
V <- 0;
C <- 0;
```

Exceptions:

```
reserved operand
```

Opcodes:

FA	CALLG	Call Procedure with General Argument List

Description:

SP is saved in a temporary and then bits 1:0 are replaced by 0 so that the stack is longword aligned. The procedure entry mask is scanned from bit 11 to 0 and the contents of registers whose number corresponds to set bits in the mask are pushed on the stack as longwords. PC, FP, and AP are pushed on the stack as longwords. The condition codes are cleared. A longword containing the saved two low bits of SP in bits 31:30, a 0 in bit 29 and bit 28, the low 12 bits of the procedure entry mask in bits 27:16, a 0 in bit 15 and PSW<14:0> in bits 14:0 with T cleared is pushed on the stack. A longword 0 is pushed on the stack. FP is replaced by SP. AP is replaced by the arglist operand. The trap enables in the PSW are set to a known state. Integer overflow and decimal overflow are affected according to bits 14 and 15 of the entry mask respectively; floating underflow is cleared. T-bit is unaffected. PC is replaced by the sum of destination operand plus 2, which transfers control to the called procedure at the byte beyond the entry mask. The appearance of the stack after a CALLG instruction is executed is shown in Figure 4-3.

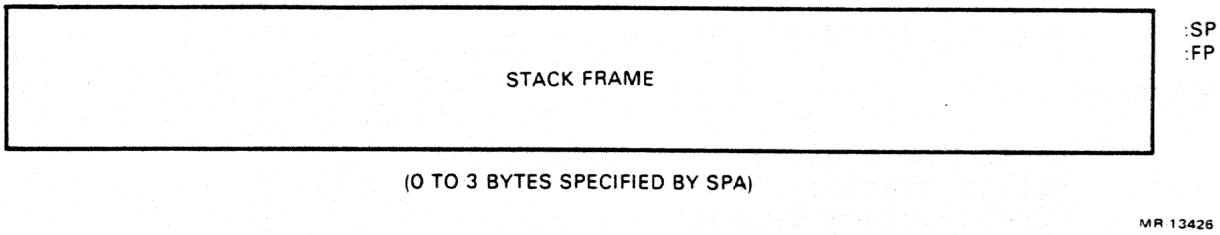


Figure 4-3 CALLG Stack Frame

Notes:

1. If bits 13:12 of the entry mask are not 0, a reserved operand fault occurs.

2. On a reserved operand fault, the condition codes are unpredictable.

3. The procedure calling standard and the condition handling facility require the following register saving conventions. R0 and R1 are always available for function return values and are never saved in the entry mask. All registers R2 through R11 which are modified in the called procedure must be preserved in the mask.

CALLS Call Procedure with Stack Argument List

Format:

```
opcode numarg.rl, dst.ab
```

Operation:

```
{push arg count};
{align stack};
{create stack frame};
{set arithmetic exception enables};
{set new values of AP,FP,PC};
```

Condition Codes:

```
N <- 0;
Z <- 0;
V <- 0;
C <- 0;
```

Exceptions:

reserved operand

Opcodes:

FB	CALLS	Call Procedure With Stack Argument List

Description:

The numarg operand is pushed on the stack as a longword (byte 0 contains the number of arguments, the high order 24 bits are used by DIGITAL software). SP is saved in a temporary and then bits 1:0 of SP are replaced by 0 so that the stack is longword aligned. The procedure entry mask is scanned from bit 11 to bit 0 and the contents of registers whose number corresponds to set bits in the mask are pushed on the stack. PC, FP, and AP are pushed on the stack as longwords. The condition codes are cleared. A longword containing the saved two low bits of SP in bits 31:30, a 1 in bit 29, a 0 in bit 28, the low 12 bits of the procedure entry mask in bits 27:16, a 0 in bit 15 and PSW<14:0> in bits 14:0 with T cleared is pushed on the stack. A longword 0 is pushed on the stack. FP is replaced by SP. AP is set to the value of the stack pointer after the numarg operand was pushed on the stack. The trap enables in the PSW are set to a known state. Integer overflow and decimal overflow are affected according to bits 14 and 15 of the entry mask; respectively, floating underflow is cleared. T-bit is unaffected. PC is replaced by the sum of destination operand plus 2, which transfers control to the called procedure at the byte beyond the entry mask. The appearance of the

stack after CALLS is executed is shown in Figure 4-4.

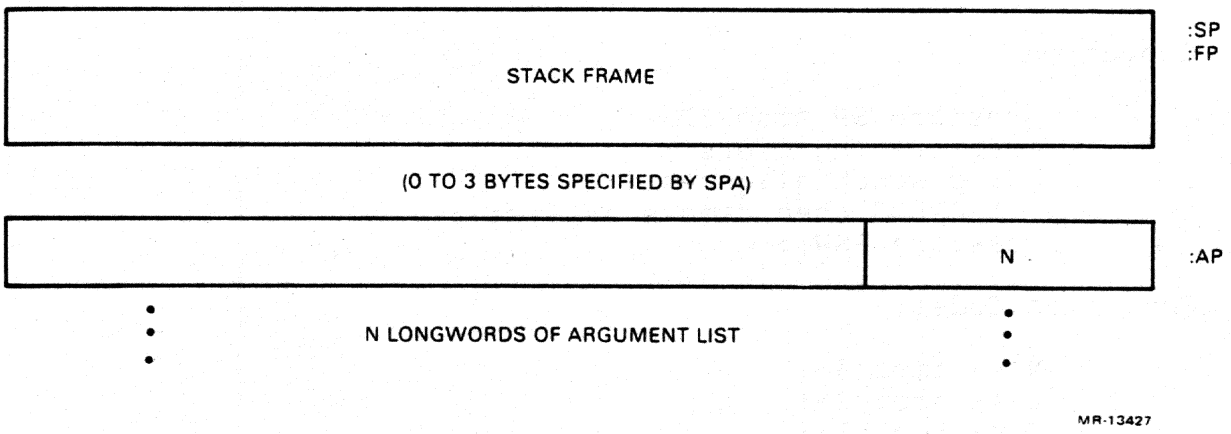


Figure 4-4 CALLS Stack Frame

Notes:

1. If bits 13:12 of the entry mask are not 0, a reserved operand fault occurs.

2. On a reserved operand fault, the condition codes are unpredictable.

3. Normal use is to push the arglist onto the stack in reverse order prior to the CALLS. On return, the arglist is removed from the stack automatically.

4. The procedure calling standard and the condition handling facility require the following register saving conventions. R0 and R1 are always available for function return values and are never saved in the entry mask. All registers R2 through R11 which are modified in the called procedure must be preserved in the entry mask.

RET Return from Procedure

Format:

```
opcode
```

Operation:

```
{restore SP from FP};
{restore registers};
{drop stack alignment};
{if CALLS then remove arglist};
{restore PSW};
```

Condition Codes:

```
N <- tmpl<3>;
Z <- tmpl<2>;
V <- tmpl<1>;
C <- tmpl<0>;
```

Exceptions:

reserved operand

Opcodes:

04	RET	Return from Procedure

Description:

SP is replaced by FP plus 4. A longword containing stack alignment bits in bits 31:30, a CALLS/CALLG flag in bit 29, the low 12 bits of the procedure entry mask in bits 27:16, and a saved PSW in bits 15:0 is popped from the stack and saved in a temporary. PC, FP, and AP are replaced by longwords popped from the stack. A register restore mask is formed from bits 27:16 of the temporary. Scanning from bit 0 to bit 11 of the restore mask, the contents of registers whose number is indicated by set bits in the mask are replaced by longwords popped from the stack. SP is incremented by 31:30 of the temporary. PSW is replaced by bits 15:0 of the temporary. If bit 29 in the temporary is 1 (indicating that the procedure was called by CALLS), a longword containing the number of arguments is popped from the stack. Four times the unsigned value of the low byte of this longword is added to SP and SP is replaced by the result.

Notes:

1. A reserved operand fault occurs if tmpl<15:8> NEQ 0.

2. On a reserved operand fault, the condition codes are unpredictable.

3. The value of tmpl<28> is ignored.

4. The procedure calling standard and condition handling facility assume that procedures which return a function value or a status code do so in R0 or R0 and R1.

5. If FP<1:0> is not zero, the results are unpredictable.

4.7 MISCELLANEOUS INSTRUCTIONS

BICPSW Bit Clear PSW

Format:

```
opcode mask.rw
```

Operation:

```
PSW <- PSW AND {NOT mask};
```

Condition Codes:

```
N <- N AND {NOT mask<3>};
Z <- Z AND {NOT mask<2>};
V <- V AND {NOT mask<1>};
C <- C AND {NOT mask<0>};
```

Exceptions:

reserved operand

Opcodes:

B9	BICPSW	Bit Clear PSW

Description:

PSW is ANDed with the ones complement of the mask operand and PSW is replaced by the result.

Notes:

A reserved operand fault occurs if mask <15:8> is not zero. On a reserved operand fault, the PSW is not affected.

BISPSW Bit Set PSW

Format:

opcode mask.rw

Operation:

```
PSW <- PSW OR mask;
```

Condition Codes:

```
N <- N OR mask<3>;
Z <- Z OR mask<2>;
V <- V OR mask<1>;
C <- C OR mask<0>;
```

Exceptions:

reserved operand

Opcodes:

B8	BISPSW	Bit Set PSW

Description:

PSW is ORed with the mask operand and PSW is replaced by the result.

Notes:

A reserved operand fault occurs if mask<15:8> is not zero. On a reserved operand fault, the PSW is not affected.

BPT Breakpoint Fault

Format:

```
opcode
```

Operation:

```
PSL<TP> <- 0;
{breakpoint fault};      !push current PSL on stack
```

Condition Codes:

```
N <- 0; !condition codes cleared after BPT fault
Z <- 0;
V <- 0;
C <- 0;
```

Exceptions:

none

Opcodes:

03	BPT	Breakpoint Fault

Description:

This instruction is used, together with the T-bit, to implement debugging facilities.

HALT Halt

Format:

```
opcode
```

Operation:

```
If PSL<current_mode> NEQU kernel then
        {privileged instruction fault}
        else
        {halt the processor};
```

Condition Codes:

```
N <- 0; !If privileged instruction fault
Z <- 0; !condition codes are cleared after
V <- 0; !the fault. PSL saved on stack
C <- 0; !contains condition codes prior to HALT.

N <- N; !If processor halt
Z <- Z;
V <- V;
C <- C;
```

Exceptions:

```
privileged instruction
```

Opcodes:

00	HALT	Halt

Description:

If the process is running in kernel mode, the processor is halted. Otherwise, a privileged instruction fault occurs.

Notes:

This opcode is 0 to trap many branches to data.

INDEX Compute Index

Format:

```
opcode  subscript.rl, low.rl, high.rl,
        size.rl, indexin.rl, indexout.wl
```

Operation:

```
indexout <- {indexin + subscript} *size;
if {subscript LSS low} or {subscript GTR high}
then {subscript range trap};
```

Condition Codes:

```
N <- indexout LSS 0;
Z <- indexout EQL 0;
V <- 0;
C <- 0;
```

Exceptions:

subscript range

Opcodes:

0A	INDEX	Compute Index

Description:

The indexin operand is added to the subscript operand and the sum multiplied by the size operand. The indexout operand is replaced by the result. If the subscript operand is less than the low operand or greater than the high operand, a subscript range trap is taken.

Notes:

1. No arithmetic exception other than subscript range can result from this instruction. Thus no indication is given if overflow occurs in either the add or multiply steps. If overflow occurs on the add step the sum is the low order 32 bits of the true result. If overflow occurs on the multiply step, the indexout operand is replaced by the low order 32 bits of the true product of the sum and the subscript operand. In the normal use of this instruction, overflow cannot occur without a subscript range trap occurring.

2. The index instruction is useful in index calculations for arrays of the fixed length data types (integer and floating) and for index calculations for arrays of bit fields, character strings, and decimal strings. The indexin operand permits cascading INDEX

instructions for multidimensional arrays. For one-dimensional bit field arrays it also permits introduction of the constant portion of an index calculation which is not readily absorbed by address arithmetic. The following example shows some of the uses of INDEX.

Example:

The COBOL statements:

```
01  A-ARRAY.
    02  A PIC X(25) OCCURS 15 TIMES INDEXED BY I.
01  B PIC X(25).

MOVE A(I) TO B.
```

are equivalent to:

```
INDEX   I(R11), #^X01, #^X0F, #^X19, #^X00, R0
MOVC3   #^X19, A-25(R11)[R0], B(R11)
```

The FORTRAN statements:

```
INTEGER*4       A(11:24), I
A(I) = 1
```

are equivalent to:

```
INDEX   I(R11), #11, #24, #1, #0, R0
MOVL    #1, A-44(R11)[R0]
```

The PASCAL statements:

```
var
   i : integer;
   a : array[11..24] of integer;

   a[i] := 1
```

are equivalent to:

```
INDEX   I,#11,#24,#1,#0,R0
MOVZBL  #1,A-44[R0]
```

MOVPSL Move from PSL

Format:

opcode dst.wl

Operation:

```
dst <- PSL;
```

Condition Codes:

```
N <- N;
Z <- Z;
V <- V;
C <- C;
```

Exceptions:

none

Opcodes:

DC	MOVPSL	Move from PSL

Description:

The destination operand is replaced by PSL.

NOP No Operation

Format:

opcode

Operation:

none

Condition Codes:

N <- N;
Z <- Z;
V <- V;
C <- C;

Exceptions:

none

Opcodes:

01	NOP	No Operation

Description:

No operation is performed.

POPR Pop Registers

Format:

```
opcode mask.rw
```

Operation:

```
for tmp <- 0 step 1 until 14 do
if mask<tmp> EQL 1 then R[tmp] <- (SP)+;
```

Condition Codes:

```
N <- N;
Z <- Z;
V <- V;
C <- C;
```

Exceptions:

none

Opcodes:

BA	POPR	Pop Registers

Description:

The contents of registers whose number corresponds to set bits in the mask operand are replaced by longwords popped from the stack. Rn is replaced if mask<n> is set. The mask is scanned from bit 0 to bit 14. Bit 15 is ignored.

PUSHR Push Registers

Format:

```
opcode mask.rw
```

Operation:

```
for tmp <- 14 step -1 until 0 do
if mask<tmp> EQL 1 then -(SP) <- R[tmp];
```

Condition Codes:

```
N <- N;
Z <- Z;
V <- V;
C <- C;
```

Exceptions:

none

Opcodes:

BB	PUSHR	Push Registers

Description:

The contents of registers whose number corresponds to set bits in the mask operand are pushed on the stack as longwords. Rn is pushed if mask<n> is set. The mask is scanned from bit 14 to bit 0. Bit 15 is ignored.

Notes:

The order of pushing is specified so that the contents of higher numbered registers are stored at higher memory addresses. This results in, say, a quadword datum stored in adjacent registers being stored by PUSHR in memory in the correct order.

XFC Extended Function Call

Format:

```
opcode
```

Operation:

```
{XFC fault};
```

Condition Codes:

```
N <- 0;
Z <- 0;
V <- 0;
C <- 0;
```

Exceptions:

none

Opcodes:

FC	XFC	Extended Function Call

Description:

This instruction provides for user defined extensions to the instruction set.

4.8 QUEUE INSTRUCTIONS

A queue is a circular, doubly linked list whose entries are specified by their addresses. Each queue entry links to two others via a pair of longwords. The first longword is the forward link: it specifies the location of the succeeding entry. The second longword is the backward link: it specifies the location of the preceding entry.

VAX supports two distinct types of links: absolute, and self-relative. An absolute link contains the absolute address of the entry that it points to. A self-relative link contains a displacement from the present queue entry. A queue is classified by the type of link it uses.

4.8.1 Absolute Queues

Absolute queues use absolute addresses as links. Queue entries are linked by a pair of longwords.

The first (lowest addressed) longword is the forward link: the address of the succeeding queue entry. The second (highest addressed) longword is the backward link: the address of the preceding queue entry. A queue is specified by a queue header which is identical to a pair of queue linkage longwords. The forward link of the header is the address of the entry termed the head of the queue. The backward link of the header is the address of the entry termed the tail of the queue. The forward link of the tail points to the header.

Two general operations can be performed on queues: insertion of entries and removal of entries. Generally entries can be inserted or removed only at the head or tail of a queue.

Figures 4-5 through 4-9 illustrate some queue operations. An empty queue is specified by its header at address H as shown in Figure 4-5.

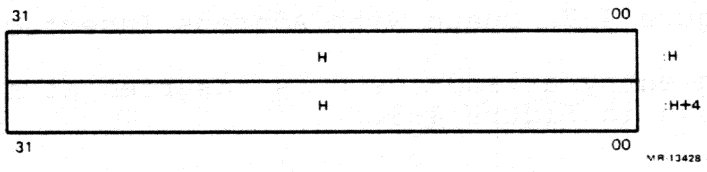


Figure 4-5 Empty Queue Header

If an entry at address B is inserted into an empty queue (at either the head or tail), the queue is as shown in Figure 4-6.

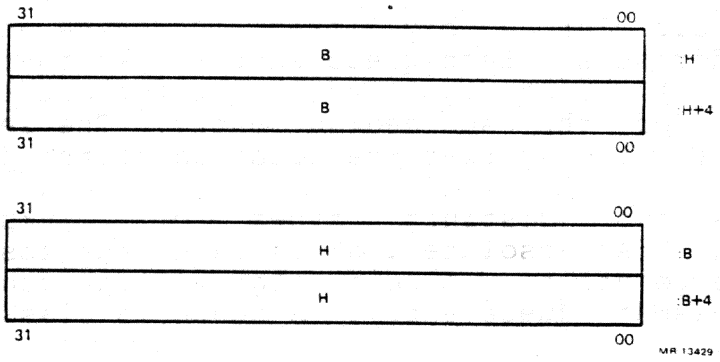


Figure 4-6 Queue With Address B Inserted

If an entry at address A is inserted at the head of the queue, the queue is as shown in Figure 4-7.

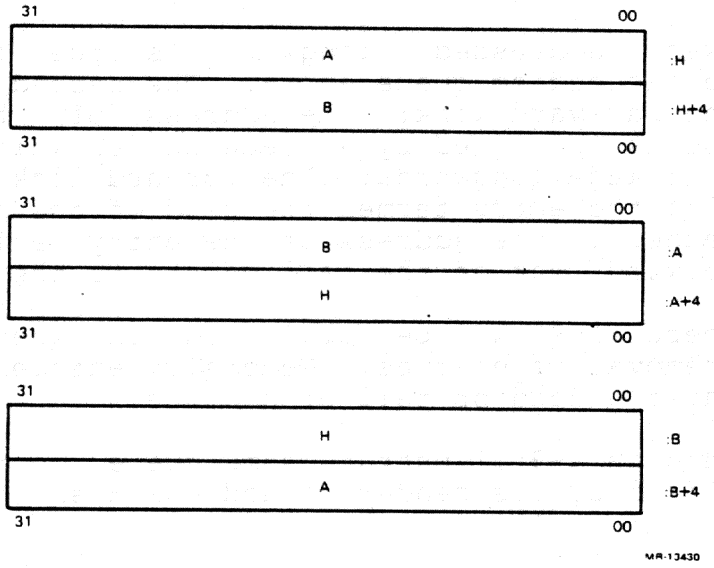


Figure 4-7 Queue With Address Inserted at Head

Finally, if an entry at address C is inserted at the tail, the queue appears as shown in Figure 4-8.

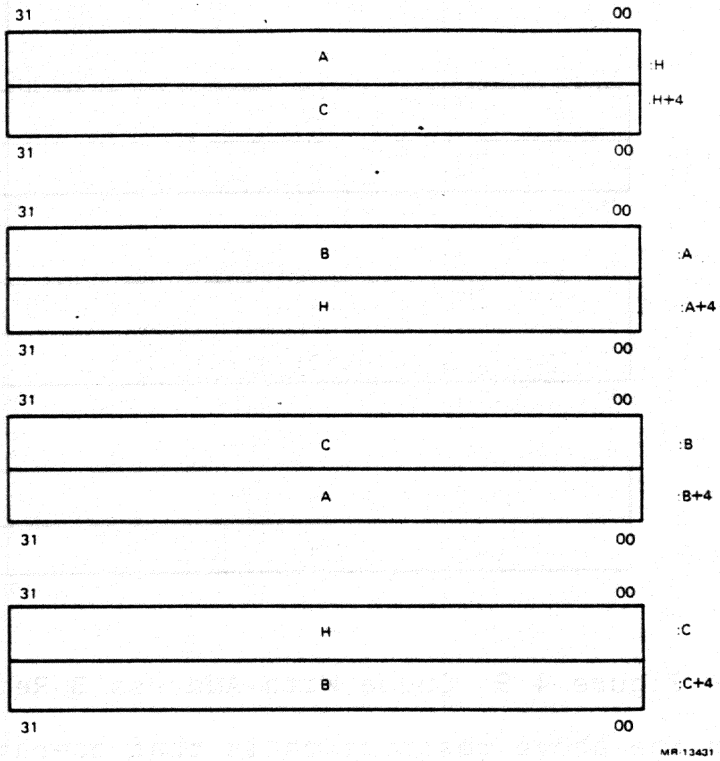


Figure 4-8 Queue With Address Inserted at Tail

Following the above steps in reverse order gives the effect of removal at the tail and removal at the head.

If more than 1 process can perform operations on a queue simultaneously, insertions and removals should only be done at the head or tail of the queue. If only 1 process (or 1 process at a time) can perform operations on a queue, insertions and removals can be made at other than the head or tail of the queue. In the example above with the queue containing entries A,B, and C, the entry at address B can be removed and the queue appears as shown in Figure 4-9.

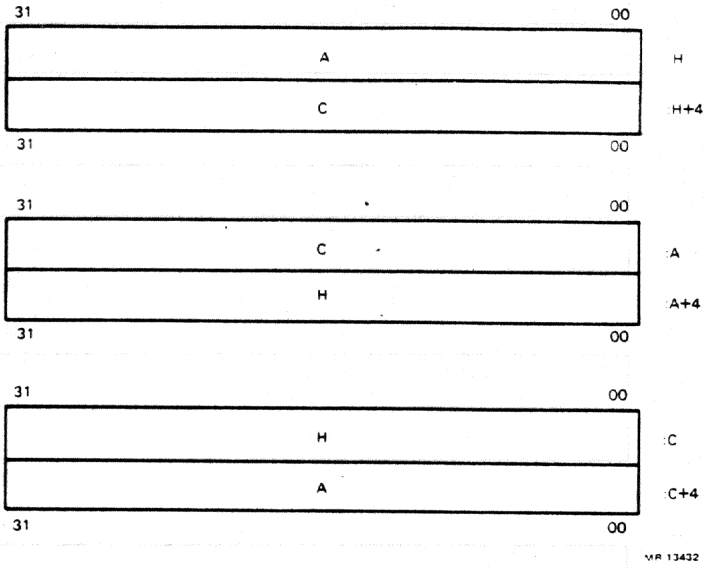


Figure 4-9 Queue With Address B Removed

The reason for the above restriction is that operations at the head or tail are always valid because the queue header is always present; operations elsewhere in the queue depend on specific entries being present and may become invalid if another process is simultaneously performing operations on the queue.

Two instructions are provided for manipulating absolute queues: INSQUE and REMQUE. INSQUE inserts an entry specified by an entry operand into the queue following the entry specified by the predecessor operand. REMQUE removes the entry specified by the entry operand. Queue entries can be on arbitrary byte boundaries. Both INSQUE and REMQUE are implemented as non-interruptible instructions.

4.8.2 Self-relative Queues

Self-relative queues use displacements from queue entries as links. Queue entries are linked by a pair of longwords. The first longword (lowest addressed) is the forward link: the displacement of the succeeding queue entry from the present entry. The second longword (highest addressed) is the backward link: the displacement of the preceding queue entry from the present entry. A queue is specified by a queue header, which also consists of two longword links.

Figures 4-10 through 4-13 show examples of queue operations. An empty queue is specified by its header at address H. Since the queue is empty, the self-relative links must be zero as shown in Figure 4-10.

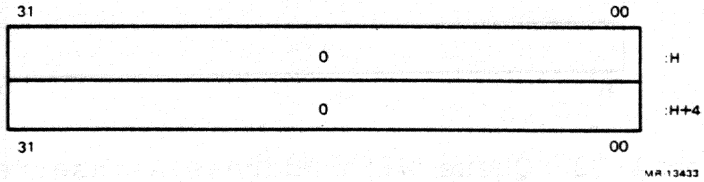


Figure 4-10 Empty Queue

If an entry at address B is inserted into an empty queue (at either the head or tail), the queue is as shown in Figure 4-11.

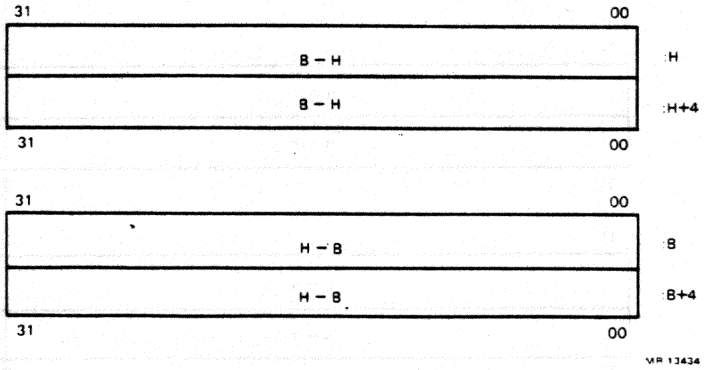


Figure 4-11 Queue With Address B Inserted

If an entry at address A is inserted at the head of the queue, the queue is as shown in Figure 4-12.

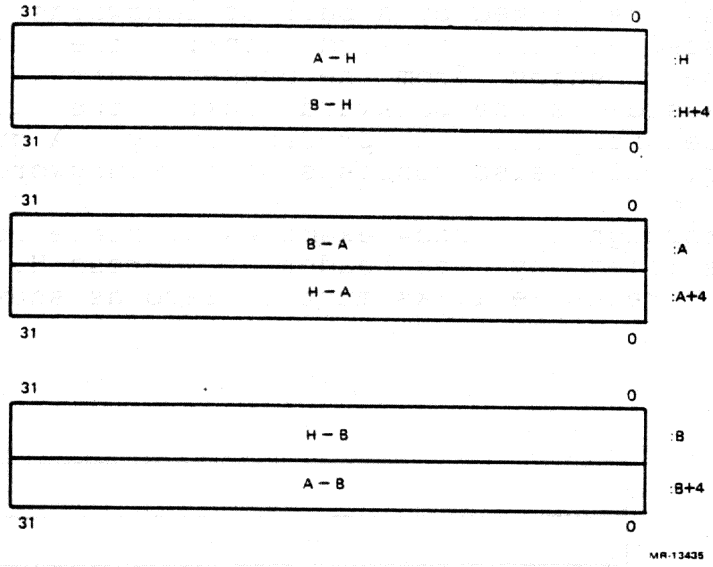


Figure 4-12 Queue With Address A Inserted at Head

Finally, if an entry at address C is inserted at the tail, the queue appears as shown in Figure 4-13.

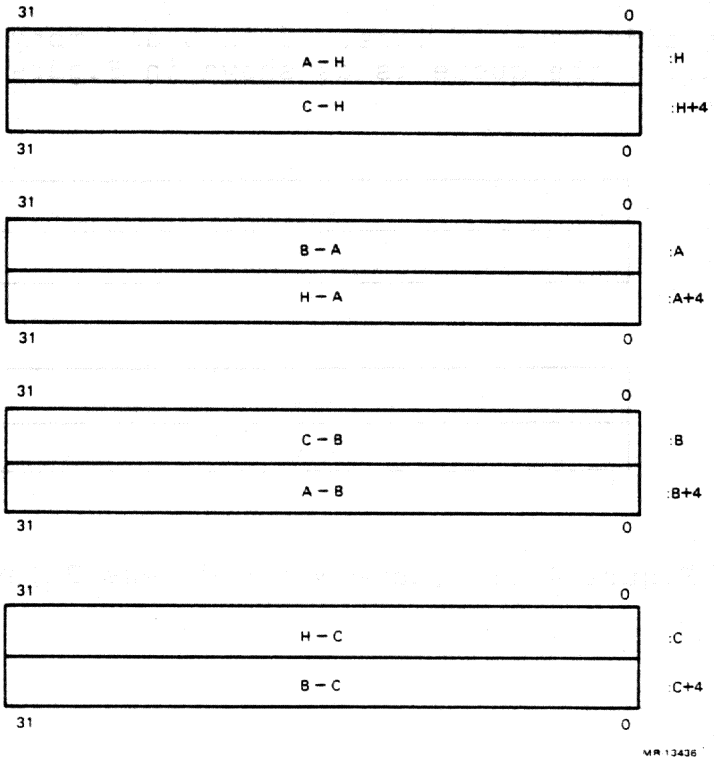


Figure 4-13 Queue With Address C Inserted at Tail

Following the above steps in reverse order gives the effect of removal at the tail and removal at the head.

Four operations can be performed on self-relative queues: insert at head, insert at tail, remove from head, and remove from tail. Furthermore, these operations are interlocked to allow cooperating processes in a multiprocessor system to access a shared list without additional synchronization. Queue entries must be quadword aligned. A hardware supported interlocked memory access mechanism is used to read the queue header. Bit 0 of the queue header is used as a secondary interlock and is set when the queue is being accessed. If an interlocked queue instruction encounters the secondary interlock set, it terminates after setting the condition codes to indicate failure to gain access to the queue. If the secondary interlock bit is not set, then the interlocked queue instruction sets it during its operation and clears it at instruction completion. This prevents other interlocked queue instructions from operating on the same queue.

INSQHI Insert Entry into Queue at Head, Interlocked

Format:

```
opcode  entry.ab, header.aq
```

Operation:

```
                                !must have write access to header
                                !header must be quadword aligned
                                !header cannot be equal to entry
tmpl <- (header){interlocked};  !acquire hardware interlock
                                !tmpl<2:1> must be zero

if tmpl<0> EQLU 1 then
begin
(header){interlocked} <- tmpl;!release hardware interlock
{set condition codes and terminate instruction};
end;
else
begin
(header){interlocked} <- tmpl v 1;!set secondary interlock
                                  !release hardware interlock
If {all memory accesses can be completed} then
                !check if following addresses can be written
                !without causing a memory management exception:
                !       entry
                !       header + tmpl
                !also, check for quadword alignment
        begin
        {insert entry into queue};
        {release secondary interlock};
        end;
        else
        begin
        {release secondary interlock};
        {backup instruction};
        {initiate fault};
        end;
end;
```

Condition Codes:

```
if {insertion succeeded} then
        begin
        N <- 0;
        Z <- (entry) EQL (entry+4);     !first entry in queue
        V <- 0;
        C <- 0;
        end;
else
        begin
        N <- 0;
        Z <- 0;
        V <- 0;
        C <- 1;          !secondary interlock failed
        end;
```

Exceptions:

reserved operand

Opcodes:

5C INSQHI Insert Entry into Queue at Head, Interlocked

Description:

The entry specified by the entry operand is inserted into the queue following the header. If the entry inserted was the first one in the queue, the condition code Z-bit is set; otherwise it is cleared. The insertion is a non-interruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process even in a multiprocessor environment. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state. If the instruction fails to acquire the secondary interlock, the instruction sets condition codes and terminates.

Notes:

1. Because the insertion is non-interruptible, processes running in kernel mode can share queues with interrupt service routines.

2. The INSQHI, INSQTI, REMQHI, and REMQTI instructions are implemented such that cooperating software processes in a multiprocessor may access a shared list without additional synchronization.

3. To set a software interlock realized with a queue, the following can be used:

```
INSERT: INSQHI   ...              ;was queue empty?
        BEQL     1$               ;yes
        BCS      INSERT           ;try inserting again
        CALL     WAIT(...)        ;no, wait

1$:
```

4. During access validation, any access which cannot be completed results in a memory management exception even though the queue insertion is not started.

5. A reserved operand fault occurs if entry or header is an address that is not quadword aligned (i.e. <2:0> NEQU 0) or if (header)<2:1> is not zero. A reserved operand fault also occurs if header equals entry. The queue is not altered.

INSQTI Insert Entry into Queue at Tail, Interlocked

Format:

```
opcode  entry.ab, header.aq
```

Operation:

```
                                !must have write access to header
                                !header must be quadword aligned
                                !header cannot be equal to entry
tmpl <- (header){interlocked};  !acquire hardware interlock
                                !tmpl<2:1> must be zero

if tmpl<0> EQLU 1 then
begin
(header){interlocked} <- tmpl;!release hardware interlock
{set condition codes and terminate instruction};
end;
else
begin
(header){interlocked} <- tmpl v 1;!set secondary interlock
                                  !release hardware interlock
If {all memory accesses can be completed} then
        !check if the following addresses can be written
        !without causing a memory management exception:
        !       entry
        !       header + (header + 4)
        !also, check for quadword alignment
        begin
        {insert entry into queue};
        {release secondary interlock};
        end;
else
        begin
        {release secondary interlock};
        {backup instruction};
        {initiate fault};
        end;
end;
```

Condition Codes:

```
        if {insertion succeeded} then
                begin
                N <- 0;
                Z <- (entry) EQL (entry+4);     !first entry in queue
                V <- 0;
                C <- 0;
                end;
        else
                begin
                N <- 0;
                Z <- 0;
                V <- 0;
                C <- 1;          !secondary interlock failed
                end;
```

Exceptions:

reserved operand

Opcodes:

5D INSQTI Insert Entry into Queue at Tail, Interlocked

Description:

The entry specified by the entry operand is inserted into the queue preceding the header. If the entry inserted was the first one in the queue, the condition code Z-bit is set; otherwise it is cleared. The insertion is a non-interruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process even in a multiprocessor environment. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state. If the instruction fails to acquire the secondary interlock, the instruction sets condition codes and terminates.

Notes:

1. Because the insertion is non-interruptible, processes running in kernel mode can share queues with interrupt service routines.

2. The INSQHI, INSQTI, REMQHI, and REMQTI instructions are implemented such that cooperating software processes in a multiprocessor may access a shared list without additional synchronization.

3. To set a software interlock realized with a queue, the following can be used:

```
INSERT: INSQTI   ...              ;was queue empty?
        BEQL     1$               ;yes
        BCS      INSERT           ;try inserting again
        CALL     WAIT(...)        ;no, wait

1$:
```

4. During access validation, any access which cannot be completed results in a memory management exception even though the queue insertion is not started.

5. A reserved operand fault occurs if entry, header, or (header+4) is an address that is not quadword aligned (i.e. <2:0> NEQU 0) or if (header)<2:1> is not zero. A reserved operand fault also occurs if header equals entry. The queue is not altered.

INSQUE Insert Entry in Queue

Format:

```
opcode  entry.ab, pred.ab
```

Operation:

```
If {all memory accesses can be completed} then
        begin

        (entry) <- (pred);          !forward link of entry
        (entry + 4) <- pred;        !backward link of entry
        ((pred) + 4) <- entry;      !backward link of successor
        (pred) <- entry;            !forward link of predecessor
        end;
else
        begin
        {backup instruction};
        {initiate fault};
        end;
```

Condition Codes:

```
N <- (entry) LSS (entry+4);
Z <- (entry) EQL (entry+4);     !first entry in queue
V <- 0;
C <- (entry) LSSU (entry+4);
```

Exceptions:

none

Opcodes:

0E	INSQUE	Insert Entry in Queue

Description:

The entry specified by the entry operand is inserted into the queue following the entry specified by the predecessor operand. If the entry inserted was the first one in the queue, the condition code Z-bit is set; otherwise it is cleared. The insertion is a non-interruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state.

Notes:

1. Three types of insertion can be performed by appropriate choice of predecessor operand:

   a. Insert at head

```
INSQUE  entry,h          ;h is queue head
```

   b. Insert at tail

```
INSQUE  entry,@h+4       ;h is queue head
(Note "@" in this case only)
```

   c. Insert after arbitrary predecessor

```
INSQUE  entry,p          ;p is predecessor
```

2. Because the insertion is non-interruptible, processes running in kernel mode can share queues with interrupt service routines.

3. The INSQUE and REMQUE instructions are implemented such that cooperating software processes in a single processor may access a shared list without additional synchronization if the insertions and removals are only at the head or tail of the queue.

4. To set a software interlock realized with a queue, the following can be used:

```
        INSQUE  ...             ;was queue empty?
        BEQL    1$              ;yes
        CALL    WAIT(...)       ;no, wait

1$:
```

5. During access validation, any access which cannot be completed results in a memory management exception even though the queue insertion is not started.

REMQHI Remove Entry from Queue at Head, Interlocked

Format:

```
opcode  header.aq, addr.wl
```

Operation:

```
                                !must have write access to header
                                !header must be quadword aligned
                                !header cannot equal address of addr
tmpl <- (header){interlocked};  !acquire hardware interlock
                                !tmpl<2:1> must be zero

if tmpl<0> EQLU 1 then
begin
(header){interlocked} <- tmpl;!release hardware interlock
{set condition codes and terminate instruction};
end;
else
begin
(header){interlocked} <- tmpl v 1;!set secondary interlock
                                  !release hardware interlock
If {all memory accesses can be completed} then
        !check if the following can be done without
        !causing a memory management exception:
        !write addr operand
        !read contents of header + tmpl {if tmpl NEQU 0}
        !write into header + tmpl + (header + tmpl)
        !                                {if tmpl NEQU 0}
        !also, check for quadword alignment
        begin
        {remove entry from queue};
        {release secondary interlock};
        end;
        else
        begin
        {release secondary interlock};
        {backup instruction};
        {initiate fault};
        end;
end;
```

Condition Codes:

```
if {removal succeeded} then
        begin
        N <- 0;
        Z <- (header) EQL 0;     !queue empty after removal
        V <- {queue empty before this instruction};
        C <- 0;
        end;
else
        begin
        N <- 0;
        Z <- 0;
        V <- 1;          !did not remove anything
        C <- 1;          !secondary interlock failed
        end;
```

Exceptions:

reserved operand

Opcodes:

5E REMQHI Remove Entry from Queue at Head, Interlocked

Description:

If the secondary interlock is clear, the queue entry following the header is removed from the queue and the address operand is replaced by the address of the entry removed. If the queue was empty prior to this instruction or if the secondary interlock failed, the condition code V bit is set; otherwise it is cleared. If the interlock succeeded and the queue is empty at the end of this instruction, the condition code Z-bit is set; otherwise it is cleared. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process even in a multiprocessor environment. The removal is a non-interruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state. If the instruction fails to acquire the secondary interlock, the instruction sets condition codes and terminates without altering the queue.

Notes:

1. Because the removal is non-interruptible, processes running in kernel mode can share queues with interrupt service routines.

2. The INSQHI, INSQTI, REMQHI, and REMQTI instructions are implemented such that cooperating software processes in a multiprocessor may access a shared list without additional synchronization.

3. To release a software interlock realized with a queue, the following can be used:

```
1$:     REMQHI  ...             ;removed last?
        BEQL    2$              ;yes
        BCS     1$              ;try removing again
        CALL    ACTIVATE(...)   ;Activate other waiters

2$:
```

4. To remove entries until the queue is empty, the following can be used:

```
1$:     REMQHI  ...             ;anything removed?
        BVS     2$              ;no
          .
        process removed entry
          .
        BR      1$              ;
          .
2$:     BCS     1$              ;try removing again
        queue empty
```

5. During access validation, any access which cannot be completed results in a memory management exception even though the queue removal is not started.

6. A reserved operand fault occurs if header or (header + (header)) is an address that is not quadword aligned (i.e. <2:0> NEQU 0) or if (header)<2:1> is not zero. A reserved operand fault also occurs if the header address operand equals the address of the addr operand. The queue is not altered.

REMQTI Remove Entry from Queue at Tail, Interlocked

Format:

```
opcode  header.aq, addr.wl
```

Operation:

```
                                !must have write access to header
                                !header must be quadword aligned
                                !header cannot equal address of addr
tmpl <- (header){interlocked};  !acquire hardware interlock
                                !tmpl<2:1> must be zero

if tmpl<0> EQLU 1 then
begin
(header){interlocked} <- tmpl;!release hardware interlock
{set condition codes and terminate instruction};
end;
else
begin
(header){interlocked} <- tmpl v 1;!set secondary interlock
                                  !release hardware interlock
If {all memory accesses can be completed} then
        !check if the following can be done without
        !causing a memory management exception:
        !write addr operand
        !read contents of header + (header + 4)
        !                          {if tmpl NEQU 0}
        !write into header + (header + 4)
        !  + (header + 4 + (header + 4)) {if tmpl NEQU 0}
        !also, check for quadword alignment
        begin
        {remove entry from queue};
        {release secondary interlock};
        end;
        else
        begin
        {release secondary interlock};
        {backup instruction};
        {initiate fault};
        end;
end;
```

Condition Codes:

```
if {removal succeeded} then
        begin
        N <- 0;
        Z <- (header + 4) EQL 0;!queue empty after removal
        V <- {queue empty before this instruction};
        C <- 0;
        end;
else
        begin
        N <- 0;
        Z <- 0;
        V <- 1;          !did not remove anything
        C <- 1;          !secondary interlock failed
        end;
```

Exceptions:

reserved operand

Opcodes:

5F REMQTI Remove Entry from Queue at Tail, Interlocked

Description:

If the secondary interlock is clear, the queue entry preceding the header is removed from the queue and the address operand is replaced by the address of the entry removed. If the queue was empty prior to this instruction or if the secondary interlock failed, the condition code V bit is set; otherwise it is cleared. If the interlock succeeded and the queue is empty at the end of this instruction, the condition code Z-bit is set; otherwise it is cleared. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process even in a multiprocessor environment. The removal is a non-interruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state. If the instruction fails to acquire the secondary interlock, the instruction sets condition codes and terminates without altering the queue.

Notes:

1. Because the removal is non-interruptible, processes running in kernel mode can share queues with interrupt service routines.

2. The INSQHI, INSQTI, REMQHI, and REMQTI instructions are implemented such that cooperating software processes in a multiprocessor may access a shared list without additional synchronization.

3. To release a software interlock realized with a queue, the following can be used:

```
1$:     REMQTI  ...             ;removed last?
        BEQL    2$              ;yes
        BCS     1$              ;try removing again
        CALL    ACTIVATE(...)   ;Activate other waiters

2$:
```

4. To remove entries until the queue is empty, the following can be used:

```
1$:     REMQTI  ...             ;anything removed?
        BVS     2$              ;no
          .
        process removed entry
          .
        BR      1$              ;
          .
2$:     BCS     1$              ;try removing again
        queue empty
```

5. During access validation, any access which cannot be completed results in a memory management exception even though the queue removal is not started.

6. A reserved operand fault occurs if header, (header + 4), or (header + (header + 4)+4) is an address that is not quadword aligned (i.e. <2:0> NEQU 0) or if (header)<2:1> is not zero. A reserved operand fault also occurs if the header address operand equals the address of the addr operand. The queue is not altered.

REMQUE Remove Entry From Queue

Format:

```
opcode  entry.ab,addr.wl
```

Operation:

```
if {all memory accesses can be completed} then
        begin

        ((entry+4)) <- (entry); !forward link of predecessor
        ((entry)+4) <- (entry +4);!backward link of successor
        addr <- entry;
        end;
else
        begin
        {backup instruction};
        {initiate fault};
        end;
```

Condition Codes:

```
N <- (entry) LSS (entry+4);
Z <- (entry) EQL (entry+4);      !queue empty
V <- entry EQL (entry+4);        !no entry to remove
C <- (entry) LSSU (entry+4);
```

Exceptions:

none

Opcodes:

0F	REMQUE	Remove Entry from Queue

Description:

The queue entry specified by the entry operand is removed from the queue. The address operand is replaced by the address of the entry removed. If there was no entry in the queue to be removed, the condition code V bit is set; otherwise it is cleared. If the queue is empty at the end of this instruction, the condition code Z-bit is set; otherwise it is cleared. The removal is a non-interruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state.

Notes:

1. Three types of removal can be performed by suitable choice of entry operand:

   1. Remove at head

```
REMQUE  @h,addr          ;h is queue header
```

   2. Remove at tail

```
REMQUE  @h+4,addr        ;h is queue header
```

   3. Remove arbitrary entry

```
REMQUE  entry,addr       ;
```

2. Because the removal is non-interruptible, processes running in kernel mode can share queues with interrupt service routines.

3. The INSQUE and REMQUE instructions are implemented such that cooperating software processes in a single processor may access a shared list without additional synchronization if the insertions and removals are only at the head or tail of the queue.

4. To release a software interlock realized with a queue, the following can be used:

```
        REMQUE  ...              ;queue empty?
        BEQL    1$               ;yes
        CALL    ACTIVATE(...)    ;Activate other waiters

1$:
```

5. To remove entries until the queue is empty, the following can be used:

```
1$:     REMQUE  ...              ;anything removed?
        BVS     EMPTY            ;no
          .
          .
          .
        BR      1$               ;
```

6. During access validation, any access which cannot be completed results in a memory management exception even though the queue removal is not started.

4.9 CHARACTER STRING INSTRUCTIONS

A character string is specified by 2 operands:

1. An unsigned word operand which specifies the length of the character string in bytes.

2. The address of the lowest addressed byte of the character string. This is specified by a byte operand of address access type.

Each of the character string instructions uses general registers R0 through R5 to contain a control block which maintains updated addresses and state during the execution of the instruction. At completion, these registers are available to software to use as string specification operands for a subsequent instruction on a contiguous character string. During the execution of the instructions, pending interrupt conditions are tested and if any is found, the control block is updated, a first part done bit is set in the PSL, and the instruction interrupted. After the interruption, the instruction resumes transparently. The format of the control block is shown in Figure 4-14.

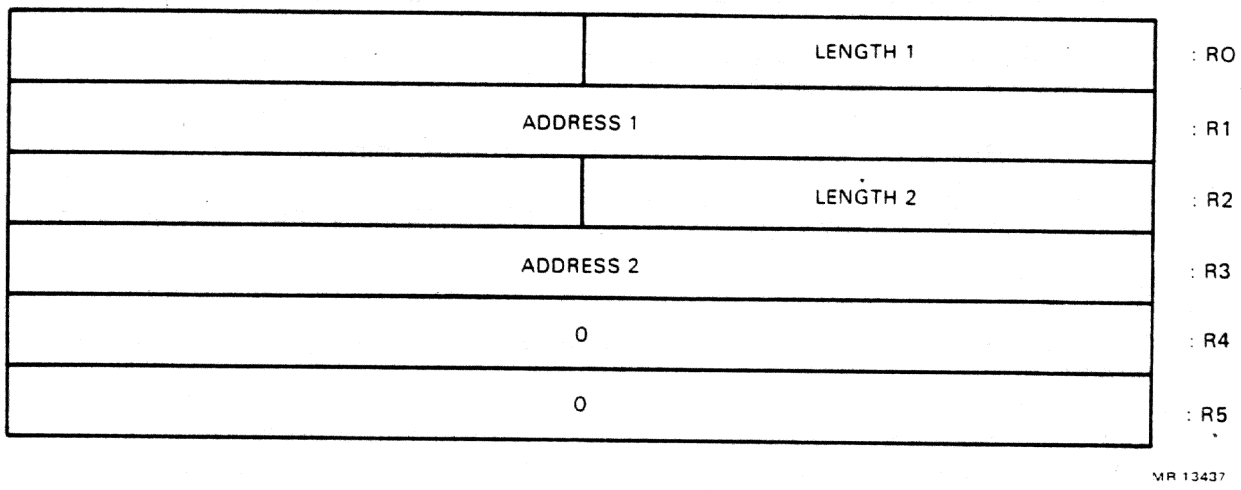


Figure 4-14 Character String Control Block

The fields LENGTH 1 and LENGTH 2 contain the number of bytes remaining to be processed in the first and second string operands respectively. The fields ADDRESS 1 and ADDRESS 2 contain the address of the next byte to be processed in the first and second string operands respectively.

Memory access faults will not occur when a zero length string is specified because no memory reference occurs.

MOVC Move Character

Format:

```
        opcode len.rw, srcaddr.ab, dstaddr.ab          3 operand

        opcode srclen.rw, srcaddr.ab, fill.rb,
               dstlen.rw, dstaddr.ab                   5 operand
```

Operation:

```
        tmp1 <- len;                                   !3 operand
        tmp2 <- srcaddr;
        tmp3 <- dstaddr;
        if tmp2 GTRU tmp3 then
                begin
                while tmp1 NEQU 0 do
                        begin
                        (tmp3) <- (tmp2);
                        tmp1 <- tmp1 - 1;
                        tmp2 <- tmp2 + 1;
                        tmp3 <- tmp3 + 1;
                        end;
                R1 <- tmp2;
                R3 <- tmp3;
                end
        else
                begin
                tmp4 <- tmp1;
                tmp2 <- tmp2 + ZEXT(tmp1);
                tmp3 <- tmp3 + ZEXT(tmp1);
                while tmp1 NEQU 0 do
                        begin
                        tmp1 <- tmp1 - 1;
                        tmp2 <- tmp2 - 1;
                        tmp3 <- tmp3 - 1;
                        (tmp3) <- (tmp2);
                        end;
                R1 <- tmp2 + ZEXT(tmp4);
                R3 <- tmp3 + ZEXT(tmp4);
                end;
        R0 <- 0;
        R2 <- 0;
        R4 <- 0;
        R5 <- 0;
```

```
tmp1 <- srclen;                                    !5 operand
tmp2 <- srcaddr;
tmp3 <- dstlen;
tmp4 <- dstaddr;
if tmp2 GTRU tmp4 then
        begin
        while {tmp1 NEQU 0} AND {tmp3 NEQU 0} do
                begin
                (tmp4) <- (tmp2);
                tmp1 <- tmp1 - 1;
                tmp2 <- tmp2 + 1;
                tmp3 <- tmp3 - 1;
                tmp4 <- tmp4 + 1;
                end;
        while tmp3 NEQU 0 do
                begin
                (tmp4) <- fill;
                tmp3 <- tmp3 - 1;
                tmp4 <- tmp4 + 1;
                end;
        R1 <- tmp2;
        R3 <- tmp4;
        end
else
        begin
        tmp5 <- MINU(tmp1, tmp3);
        tmp6 <- tmp3;
        tmp2 <- tmp2 + ZEXT(tmp5);
        tmp4 <- tmp4 + ZEXT(tmp6);
        while tmp3 GTRU tmp1 do
                begin
                tmp3 <- tmp3 - 1;
                tmp4 <- tmp4 - 1;
                (tmp4) <- fill;
                end;
        while tmp3 NEQU  0 do
                begin
                tmp1 <- tmp1 - 1;
                tmp2 <- tmp2 - 1;
                tmp3 <- tmp3 - 1;
                tmp4 <- tmp4 - 1;
                (tmp4) <- (tmp2);
                end;
        R1 <- tmp2 + ZEXT (tmp5);
        R3 <- tmp4 + ZEXT (tmp6);
        end;
R0 <- tmp1;
R2 <- 0;
R4 <- 0;
R5 <- 0;
```

Condition Codes:

```
N <- 0;                          !MOVC3
Z <- 1;
V <- 0;
C <- 0;

N <- srclen LSS dstlen; !MOVC5
Z <- srclen EQL dstlen;
V <- 0;
C <- srclen LSSU dstlen;
```

Exceptions:

none

Opcodes:

28	MOVC3	Move Character 3 Operand
2C	MOVC5	Move Character 5 Operand

Description:

In 3 operand format, the destination string specified by the length and destination address operands is replaced by the source string specified by the length and source address operands. In 5 operand format, the destination string specified by the destination length and destination address operands is replaced by the source string specified by the source length and source address operands. If the destination string is longer than the source string, the highest addressed bytes of the destination are replaced by the fill operand. If the destination string is shorter than the source string, the highest addressed bytes of the source string are not moved. The operation of the instruction is such that overlap of the source and destination strings does not affect the result.

Notes:

1. After execution of MOVC3:

   R0 = 0

   R1 = address of one byte beyond the source string

   R2 = 0

   R3 = address of one byte beyond the destination string.

   R4 = 0

   R5 = 0

2. After execution of MOVC5:

   R0 = number of unmoved bytes remaining in source string.
   R0 is non-zero only if source string is longer
   than destination string

   R1 = address of one byte beyond the last byte
   in source string that was moved

   R2 = 0

   R3 = address of one byte beyond the destination string .

   R4 = 0

   R5 = 0

3. MOVC3 is the preferred way to copy one block of memory to another.

4. MOVC5 with a 0 source length operand is the preferred way to fill a block of memory with the fill character.

4.10 OPERATING SYSTEM SUPPORT INSTRUCTIONS

CHM Change Mode

Purpose: Request services of more privileged software

Format:

```
opcode  code.rw
```

Operation:

```
tmp1 <- {mode selected by opcode (K=0, E=1, S=2, U=3)};
tmp2 <- MINU(tmp1, PSL<CUR_MOD>);       !maximize privilege
tmp3 <- SEXT (code);
if {PSL<IS> EQLU 1} then HALT;          !illegal from I stack

PSL<CUR_MOD> SP <- SP;                  !save old stack pointer
tmp4 <- tmp2_SP;                        !get new stack pointer
PROBEW (from tmp4-1 through
        tmp4-12 with mode=tmp2);        !check new stack access

    if {access control violation} then
        {initiate access violation fault};
    if {translation not valid} then
        {initiate translation not valid fault};

{initiate CHMx exception with new_mode=tmp2
        and parameter=tmp3
        using 40+tmp1*4 (hex) as SCB offset
        using tmp4 as the new SP
        and not storing SP again};
```

Condition Codes:

```
N <- 0;
Z <- 0;
V <- 0;
C <- 0;
```

Exceptions:

halt

Opcodes:

BC	CHMK	Change Mode to Kernel
BD	CHME	Change Mode to Executive
BE	CHMS	Change Mode to Supervisor
BF	CHMU	Change Mode to User

Description:

Change Mode instructions allow processes to change their access mode in a controlled manner. The instruction only increases privilege (i.e., decreases the access mode).

A change in mode also results in a change of stack pointers; the old pointer is saved, the new pointer is loaded. The PSL, PC, and code passed by the instruction are pushed onto the stack of the new mode. The saved PC addresses the instruction following the CHMx instruction. The code is sign extended. After execution, the new stack's appearance is as shown in Figure 4-15.

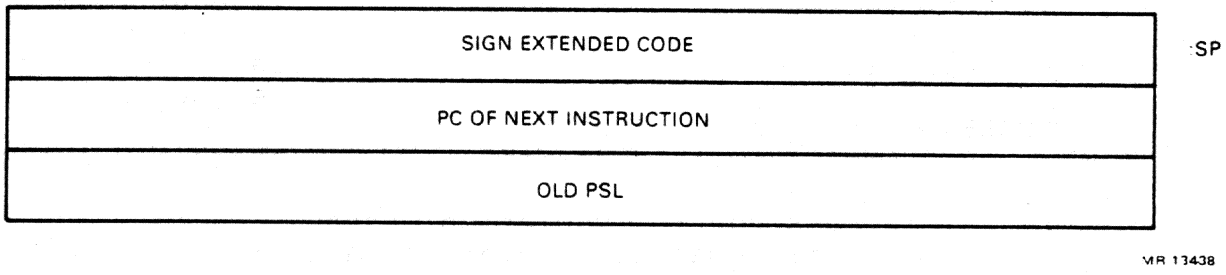


Figure 4-15 Stack After Change Mode Instruction

The destination mode selected by the opcode is used to obtain a location from the System Control Block. This location addresses the CHMx dispatcher for the specified mode. If the vector<1:0> code NEQU 0 then the operation is UNDEFINED.

Notes:

1. As usual for faults, any Access Violation or Translation Not Valid fault saves PC, PSL, and leaves SP as it was at the beginning of the instruction except for any pushes onto the kernel stack.

2. By software convention, negative codes are reserved to CSS and customers.

REI Return from Exception or Interrupt

Format:

Opcode

Operation:

```
tmp1 <- (SP)+;    !Pick up saved PC
tmp2 <- (SP)+;    !and PSL

if {tmp2<CUR_MOD> LSSU PSL<CUR_MOD>} OR
   {tmp2<IS> EQLU 1 AND PSL<IS> EQLU 0} OR
   {tmp2<IS> EQLU 1 AND tmp2<CUR_MOD> NEQU 0} OR
   {tmp2<IS> EQLU 1 AND tmp2<IPL> EQLU 0)} OR
   {tmp2<IPL> GTRU 0 AND tmp2<CUR_MOD> NEQU 0} OR
   {tmp2<PRV_MOD> LSSU tmp2<CUR_MOD>} OR
   {tmp2<IPL> GTRU PSL<IPL>} OR
   {tmp2<PSL_MBZ> NEQU 0} then {reserved operand fault};

if PSL<IS> EQLU 1 then ISP <- SP          !save old stack pointer
                  else PSL<CUR_MOD>_SP <- SP;
if PSL<TP> EQLU 1 then tmp2<TP> <- 1;     !TP <- TP or stack TP
PC <- tmp1;
PSL <- tmp2;
if PSL<IS> EQLU 0 then
        begin
        SP <- PSL<CUR_MOD>_SP;            !switch stack
        if PSL<CUR_MOD> GEQU ASTLVL       !check for AST delivery
                then {request interrupt at IPL 2};
        end;
{check for software interrupts};
{clear instruction look-ahead}
```

Condition Codes:

```
N <- saved PSL<3>;
Z <- saved PSL<2>;
V <- saved PSL<1>;
C <- saved PSL<0>;
```

Exceptions:

reserved operand

Opcodes:

02 REI Return from Exception or Interrupt

Description:

A longword is popped from the current stack and held in a temporary PC. A second longword is popped from the current stack and held in a temporary PSL. Validity of the popped PSL is checked. The current stack pointer is saved and a new stack pointer is selected according to the new PSL CUR_MOD and IS fields. The level of the highest privilege AST is checked against the current mode to see whether a pending AST can be delivered. Execution resumes with the instruction being executed at the time of the exception or interrupt. Any instruction look ahead in the processor is reinitialized.

Notes:

1. The exception or interrupt service routine is responsible for restoring any registers saved and removing parameters from the stack.

2. As usual for faults, any Access Violation or Translation Not Valid conditions for the stack pops restore the stack pointer and fault.

LDPCTX Load Process Context

Purpose: restore register and memory management context

Format:

opcode

Operation:

```
if PSL<CUR_MOD> NEQU 0
          then {privileged instruction fault};
{invalidate per-process translation buffer entries};
!PCB is located by physical address in PCBB
R0 <- (PCB+16);
R1 <- (PCB+20);
R2 <- (PCB+24);
R3 <- (PCB+28);
R4 <- (PCB+32);
R5 <- (PCB+36);
R6 <- (PCB+40);
R7 <- (PCB+44);
R8 <- (PCB+48);
R9 <- (PCB+52);
R10 <- (PCB+56);
R11 <- (PCB+60);
AP <- (PCB+64);
FP <- (PCB+68);
tmp1 <- (PCB+80);
if {tmp1<31:30> NEQU 2} OR {tmp1<1:0> NEQU 0} then
                    {UNDEFINED};
P0BR <- tmp1;
if (PCB+84)<31:27> NEQU 0 then {UNDEFINED};
if (PCB+84)<23:22> NEQU 0 then {UNDEFINED};
P0LR <- (PCB+84)<21:0>;
if (PCB+84)<26:24> GEQU 5 then {UNDEFINED};
ASTLVL <- (PCB+84)<26:24>;
tmp1 <- (PCB+88);
tmp2 <- tmp1 + 2**23;
if {tmp2<31:30> NEQU 2} OR {tmp2<1:0> NEQU 0} then
                    {UNDEFINED};
P1BR <- tmp1;
if (PCB+92)<30:22> NEQU 0 then {UNDEFINED};
P1LR <- (PCB+92)<21:0>;
if (PCB+92)<30:22> NEQU 0 then {UNDEFINED};
if PSL<IS> EQLU 1 then
          begin
          ISP <- SP;
          {interrupts off};
          PSL<IS> <- 0;
          {interrupts on};
```

```
                    end;
      (SP) <- (PCB)                          !get new KSP
     -(SP) <- (PCB+76);                      !push PSL
     -(SP) <- (PCB+72);                      !push PC
```

Condition Codes:

```
N <- N;
Z <- Z;
V <- V;
C <- C;
```

Exceptions:

reserved operand
privileged instruction

Opcodes:

06	LDPCTX	Load Process Context

Description:

The Process Control Block is specified by the privileged register Process Control Block Base. The general registers are loaded from the PCB. The memory management registers describing the process address space are also loaded and the process entries in the translation buffer are cleared. Execution is switched to the kernel stack. The PC and PSL are moved from the PCB to the stack, suitable for use by a subsequent REI instruction.

Note:

1. Loading ASTLVL with LDPCTX does not affect SISR or request a software interrupt. Those effects of ASTLVL occur only during REI.

2. To guarantee correct operation, a LDPCTX must be followed by an REI instruction.

3. To guarantee correct operation, a LDPCTX on the kernel stack must be executed with interrupts disabled.

SVPCTX Save Process Context

Purpose: save register context

Format:

opcode

Operation:

```
if PSL<CUR_MOD> NEQU 0 then
          {privileged instruction fault};
!PCB is located by physical address in PCBB
(PCB+16) <- R0;
(PCB+20) <- R1;
(PCB+24) <- R2;
(PCB+28) <- R3;
(PCB+32) <- R4;
(PCB+36) <- R5;
(PCB+40) <- R6;
(PCB+44) <- R7;
(PCB+48) <- R8;
(PCB+52) <- R9;
(PCB+56) <- R10;
(PCB+60) <- R11;
(PCB+64) <- AP;
(PCB+68) <- FP;
(PCB+72) <- (SP)+;                    !pop PC
(PCB+76) <- (SP)+;                    !pop PSL
If PSL<IS> EQLU 0 then
          begin
          PSL<IPL> <- MAXU(1, PSL<IPL>);
          (PCB) <- SP;                !save KSP
          {interrupts off};
          PSL<IS> <- 1;
          SP <- ISP;
          {interrupts on};
          end;
```

Condition Codes:

```
N <- N;
Z <- Z;
V <- V;
C <- C;
```

Exceptions:

privileged instruction

Opcodes:

07 SVPCTX Save Process Context

Description:

The Process Control Block is specified by the privileged register Process Control Block Base. The general registers are saved into the PCB. The PC and PSL currently on the top of the current stack are popped and stored in the PCB. If a SVPCTX instruction is executed when IS is clear, then IS is set, the interrupt stack pointer is activated, and IPL is maximized with 1 because of the switch to the interrupt stack.

Notes:

1. The map, ASTLVL, and PME contents of the PCB are not saved because they are rarely changed. Thus, not writing them saves overhead.

2. Between the SVPCTX instruction that saves state for one process and the LDPCTX that loads the state of another, the internal stack pointers may not be referenced by MFPR or MTPR instructions. This implies that interrupt service routines invoked at a priority higher than the lowest one used for context switching must not reference the process stack pointers.

MTPR Move To Processor Register

Format:

```
opcode  src.rl, procreg.rl
```

Operation:

```
if PSL<CUR_MOD> NEQ 0 then {reserved
        instruction fault};
PRS[procreg] <- src;
```

Condition Codes:

```
N <- src LSS 0;    !if register is replaced
Z <- src EQL 0;
V <- 0;
C <- C;

N <- N;            !if register is not replaced
Z <- Z;
V <- V;
C <- C;
```

Exceptions:

```
reserved operand fault
reserved instruction fault
```

Opcode:

```
  DA    MTPR    Move To Processor Register
```

Description:

Loads the source operand specified by source into the processor register specified by procreg. The procreg operand is a longword which contains the processor register number. Execution may have register-specific side effects.

Notes:

1. A reserved instruction fault occurs if instruction execution is attempted in other than kernel mode.

2. A reserved operand fault occurs on a move to a read only register, or if the register does not exist. However, if a register is implemented only as a PCB location, a reserved operand fault does not occur.

MFPR Move From Processor Register

Format:

```
opcode procreg.rl, dst.wl
```

Operation:

```
if PSL<CUR_MOD> NEQ 0 then {reserved
        instruction fault};
dst <- PRS[procreg];
```

Condition Codes:

```
N <- dst LSS 0;    !if destination is replaced
Z <- dst EQL 0;
V <- 0;
C <- C;

N <- N;            !if destination is not replaced
Z <- Z;
V <- V;
C <- C;
```

Exceptions:

```
reserved operand fault
reserved instruction fault
```

Opcode:

```
DB    MFPR    Move From Processor Register
```

Description:

The destination operand is replaced by the contents of the processor register specified by procreg. The procreg operand is a longword which contains the processor register number. Execution may have register-specific side effects.

Notes:

1. A reserved instruction fault occurs if instruction execution is attempted in other than kernel mode.

2. A reserved operand fault occurs on a move from a write only register, or if the register does not exist. However, if a register is implemented only as a PCB location, a reserved operand fault does not occur.

PROBEx PROBE ACCESSIBILITY

Purpose:

verify that arguments can be accessed

Format:

```
opcode  mode.rb, len.rw, base.ab
```

Operation:

```
probe_mode <- MAXU (mode<1:0>, PSL<PRV_MOD>)
condition codes <- {accessibility of base} and
                   {accessibility of {base+ZEXT(len)-1}}
                   using probe_mode
```

Condition Codes:

```
N <- 0;
Z <- if {both accessible} then 0 else 1;
V <- 0;
C <- C;
```

Exceptions:

translation not valid

Opcodes:

0C	PROBER	Probe Read Accessibility
0D	PROBEW	Probe Write Accessibility

Description:

The PROBE instruction checks the read or write accessibility of the first and last byte specified by the base address and the zero extended length. Note that the bytes in between are not checked. System software must check all pages between the two end bytes if they will be accessed.

The protection is checked against the larger (and therefore less privileged) of the modes specified in bits <1:0> of the mode operand and the Previous Mode field of the PSL. Note that probing with a mode operand of 0 is equivalent to probing the mode specified in PSL<previous-mode>.

Example:

```
MOVL    4(AP),R0          ;Copy the address of first arg so that
                          ; it can't be changed.
PROBER  #0,#4,(R0)        ;Verify that the longword pointed to by
```

```
                                ; the first arg could be read by the
                                ;  previous access mode.
                                ;Note that the arg list itself must
                                ; already have been probed
BEQL    violation               ;Branch if either byte gives an access
                                ; violation.
MOVQ    8(AP),R0                ;Copy length and address of buffer args
                                ; so that they can't change.
PROBEW  #0,R0,(R1)              ;Verify that the buffer described by the
                                ; 2nd and 3rd args could be written by
                                ;  the previous access mode.
                                ;Note that the arg list must already
                                ; have been probed and that the 2nd arg
                                ;  must be known to be less than 512.
BEQL    violation               ;Branch if either byte gives an access
                                ; violation.
```

Flows:

The following flows describe the operation of PROBE on each of the virtual addresses it is checking. Note that probing an address returns only the accessibility of the page(s) and has no effect on their residency. However, probing a process address may cause a page fault in the system address space on the per-process page tables.

1. Look up the virtual address in the translation buffer. If found, use the associated protection field to determine the accessibility and EXIT.

2. Check for length violation for System or per-Process address as appropriate. If length violation then return No Access and EXIT.

3. If System virtual address, form physical address of PTE, fetch the PTE, use the protection field to determine the accessibility and EXIT.

4. For per-Process virtual address, must do a virtual memory reference for the PTE.

   a. Look up the virtual address of the PTE in the translation buffer, form the physical address of the PTE if found, fetch the PTE, use the protection field to determine the accessibility and EXIT.

   b. Check the System virtual address of the PTE for length violation. If length violation, then return No Access and EXIT.

   c. T1 <- Page Table Entry for the page containing the per-process PTE.

d. If the protection field of T1 indicates no access (not even readable by kernel), then return No Access and EXIT. A no access to a page of PTE's conserves storage space for a page full of no access PTE's.

e. If the valid bit in T1 is 0, then take a Translation Not Valid Fault and EXIT. This case allows for the demand paging of per-process page tables.

f. Finally, calculate the physical address of the per-process PTE from the PFN field of T1, fetch the PTE, use the protection field to determine the accessibility, and EXIT.

Notes:

1. If the Valid bit of the examined Page Table Entry is set, it is unpredictable whether the Modify bit of the examined Page Table Entry is set by a PROBEW. If the Valid bit is clear, the Modify bit is not changed.

2. Except for 1, above, the valid bit of the Page Table Entry, PTE<31>, mapping the probed address is ignored.

3. A length violation gives a status of "not-accessible."

4. On the probe of a process virtual address, if the valid bit of the system Page Table Entry is 0 then a Translation Not Valid Fault occurs. This allows for the demand paging of the process page tables.

5. On the probe of a process virtual address, if the protection field of the system Page Table Entry indicates No Access, then a status of "not-accessible" is given. Thus, a single No Access Page Table Entry in the system map is equivalent to 128 No Access Page Table Entries in the process map.

4.11 FLOATING POINT INSTRUCTIONS

These instructions are implemented in hardware only if the optional floating point unit (MicroVAX 78132 FPU) is present in the system. If the optional floating point unit is not present the MicroVAX 78032 CPU will execute a reserved operand fault for any of these instructions.

The MicroVAX architecture includes floating point instructions that operate on F_floating, D_floating, and G_floating data types (see Chapter 2).

4.11.1 Representation

Mathematically, a floating point number may be defined as having the form

(+ or -) (2**K)*f,

where K is an integer and f is a non-negative fraction. For a non-vanishing number, K and f are uniquely determined by imposing the condition

1/2 LEQ f LSS 1.

The fractional factor, f, of the number is then said to be binary normalized. For the number zero, f must be assigned the value 0, and the value of K is indeterminate.

The MicroVAX floating point data formats are derived from this mathematical representation for floating point numbers. Single precision, or F_floating, data is 32 bits long. Double precision, or D_floating, and extended range double precision, or G_floating, data is 64 bits long. Sign magnitude notation is used and the following paragraphs describe its use.

4.11.1.1 Non-zero Floating Point Numbers -

The most significant bit of the floating point data is the sign bit: 0 for positive, and 1 for negative.

The fractional factor f is assumed normalized, so that its most significant bit must be 1. This 1 is the "hidden" bit: it is not stored in the data word, but of course the hardware restores it before carrying out arithmetic operations. The F_floating, D_floating, and G_floating data types use 23, 55 and 52 bits, respectively, for f, which, with the hidden bit, imply effective significance of 24 bits, 56 bits, and 53 bits for arithmetic operations.

In the F_floating and D_floating data types, eight bits are reserved for the storage of the exponent K in excess 128 notation. Thus exponents from -128 to +127 could be represented, in biased form, by 0 to 255. For reasons given below, a biased EXP of 0 (true exponent of -128), is reserved for floating point zero. Thus, for the F_floating and D_floating data types, exponents are restricted to the range -127 to +127 inclusive, or in excess 128 notation, 1 to 255.

In the G_floating data type, eleven bits are reserved for the storage of the exponent in excess 1024 notation. A biased exponent of 0 is reserved for floating point zero. Thus, exponents are restricted to -1023 to +1023 inclusive (in excess notation, 1 to 2047).

4.11.1.2 Floating Point Zero -

Because of the hidden bit, the fractional factor is not available to distinguish between zero and non-zero numbers whose fractional factor is exactly 1/2. Therefore VAX reserves a sign-exponent field of 0 for this purpose. Any positive floating point number with biased exponent of 0 is treated as if it were an exact 0 by the floating point instruction set. In particular, a floating point operand, whose bits are all 0's, is treated as zero, and this is the format generated by all floating point instructions for which the result is zero.

4.11.1.3 Reserved Operands -

A reserved operand is defined to be any bit pattern with a sign bit of one and a biased exponent of zero. All floating point instructions generate a fault if a reserved operand is encountered. Since a reserved operand has a biased exponent of 0, it can be (internally) generated only if overflow occurs.

4.11.2 Accuracy

A floating point instruction is defined to be exact if its result, extended on the right by an infinite sequence of zeros, is identical to that of an infinite precision calculation involving the same operands. The prior accuracy of the operands is thus ignored. For all arithmetic operations, except DIV, a zero operand implies that the instruction is exact. The same statement holds for DIV if the zero operand is the dividend. But if it is the divisor, division is undefined and the instruction faults.

Note that an arithmetic result is exact if no non-zero bits are lost in chopping the infinite precision result to the data length to be stored. Chopping is defined to mean that the 24 (F_floating), 56 (D_floating), or 53 (G_floating) high order bits of the normalized

fractional factor of a result are stored; the rest of the bits are discarded. The first bit lost in chopping is referred to as the "rounding" bit. The value of a rounded result is related to the chopped result as follows:

1. If the rounding bit is one, the rounded result is the chopped result incremented by an LSB (least significant bit).

2. If the rounding bit is zero, the rounded and chopped results are identical.

The VAX architecture implements rounding so as to produce results identical to the results produced by the following algorithm. Add a 1 to the rounding bit, and propagate the carry, if it occurs. Note that a renormalization may be required after rounding takes place; if this happens, the new rounding bit will be zero, so it can happen only once. The following summarizes the relations among chopped, rounded and true (infinite precision) results:

1. If a stored result is exact

   rounded value = chopped value = true value.

2. If a stored result is not exact, its magnitude is:

   a. always less than that of the true result for chopping.

   b. always less than that of the true result for rounding if the rounding bit is zero.

   c. greater than that of the true result for rounding if the rounding bit is one.

4.11.3 Instruction Descriptions

ACB Add Compare and Branch

Format:

```
opcode limit.rx, add.rx, index.mx, displ.bw
```

Operation:

```
index <- index + add;
if {{add GEQ 0} AND {index LEQ limit}} OR
        {{add LSS 0} AND {index GEQ limit}} then
        PC <- PC + SEXT(displ);
```

Condition Codes:

```
N <- index LSS 0;
Z <- index EQL 0;
V <- 0;
C <- C;
```

Exceptions:

```
floating overflow
floating underflow
reserved operand
```

Opcodes:

4F	ACBF	Add Compare and Branch F_floating
6F	ACBD	Add Compare and Branch D_floating
4FFD	ACBG	Add Compare and Branch G_floating

Description:

The addend operand is added to the index operand and the index operand is replaced by the result. The index operand is compared with the limit operand. If the addend operand is positive (or 0) and the comparison is less than or equal or if the addend is negative and the comparison is greater than or equal, the sign-extended branch displacement is added to PC and PC is replaced by the result.

Notes:

1. ACB efficiently implements the general FOR or DO loops in high level languages since the sense of the comparison between index and limit is dependent on the sign of the addend.

2. On floating underflow, if FU is set, a fault occurs and the index operand is unaffected. If FU is clear, the index operand is replaced by 0 and comparison and branch determination proceed normally.

3. On floating overflow, the instruction takes a floating overflow fault and the index operand is unaffected.

4. On a reserved operand fault, the index operand is unaffected and the condition codes are unpredictable.

ADD Add

Format:

```
opcode add.rx, sum.mx                    2 operand

opcode add1.rx, add2.rx, sum.wx          3 operand
```

Operation:

```
sum <- sum + add;          !2 operand

sum <- add1 + add2;        !3 operand
```

Condition Codes:

```
N <- sum LSS 0;
Z <- sum EQL 0;
V <- 0;
C <- 0;
```

Exceptions:

```
floating overflow
floating underflow
reserved operand
```

Opcodes:

40	ADDF2	Add F_floating 2 Operand
41	ADDF3	Add F_floating 3 Operand
60	ADDD2	Add D_floating 2 Operand
61	ADDD3	Add D_floating 3 Operand
40FD	ADDG2	ADD G_floating 2 Operand
41FD	ADDG3	ADD G_floating 3 Operand

Description:

In 2 operand format, the addend operand is added to the sum operand and the sum operand is replaced by the rounded result. In 3 operand format, the addend 1 operand is added to the addend 2 operand and the sum operand is replaced by the rounded result.

Notes:

1. On a reserved operand fault, the sum operand is unaffected and the condition codes are unpredictable.

2. On floating underflow, if FU is set, a fault occurs and the sum operand is unaffected. If FU is clear, the sum operand is replaced by 0 and no exception occurs.

3. On floating overflow, the instruction faults; the sum operand is unaffected, and the condition codes are unpredictable.

CLR Clear

Format:

```
opcode dst.wx
```

Operation:

```
dst <- 0;
```

Condition Codes:

```
N <- 0;
Z <- 1;
V <- 0;
C <- C;
```

Exceptions:

none

Opcodes:

D4	CLRF	Clear F_floating
7C	CLRG,	Clear G_floating,
	CLRD	Clear D_floating

Description:

The destination operand is replaced by 0.

CMP Compare

Format:

```
opcode src1.rx, src2.rx
```

Operation:

```
src1 - src2;
```

Condition Codes:

```
N <- src1 LSS src2;
Z <- src1 EQL src2;
V <- 0;
C <- 0;
```

Exceptions:

reserved operand

Opcodes:

51	CMPF	Compare F_floating
71	CMPD	Compare D_floating
51FD	CMPG	Compare G_floating

Description:

The source 1 operand is compared with the source 2 operand. The only action is to affect the condition codes.

Notes:

On a reserved operand fault, the condition codes are unpredictable.

CVT Convert

Format:

```
opcode src.rx, dst.wy
```

Operation:

```
dst <- conversion of src;
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- {integer overflow};
C <- 0;
```

Exceptions:

```
integer overflow
floating overflow
floating underflow
reserved operand
```

Opcodes:

4C	CVTBF	Convert Byte to F_floating
4D	CVTWF	Convert Word to F_floating
4E	CVTLF	Convert Long to F_floating
6C	CVTBD	Convert Byte to D_floating
6D	CVTWD	Convert Word to D_floating
6E	CVTLD	Convert Long to D_floating
4CFD	CVTBG	Convert Byte to G_floating
4DFD	CVTWG	Convert Word to G_floating
4EFD	CVTLG	Convert Long to G_floating

48	CVTFB	Convert F_floating to Byte
49	CVTFW	Convert F_floating to Word
4A	CVTFL	Convert F_floating to Long
4B	CVTRFL	Convert Rounded F_floating to Long
68	CVTDB	Convert D_floating to Byte
69	CVTDW	Convert D_floating to Word
6A	CVTDL	Convert D_floating to Long
6B	CVTRDL	Convert Rounded D_floating to Long
48FD	CVTGB	Convert G_floating to Byte
49FD	CVTGW	Convert G_floating to Word
4AFD	CVTGL	Convert G_floating to Long
4BFD	CVTRGL	Convert Rounded G_floating to Long
56	CVTFD	Convert F_floating to D_floating
99FD	CVTFG	Convert F_floating to G_floating
76	CVTDF	Convert D_floating to F_floating
33FD	CVTGF	Convert G_floating to F_floating

Description:

The source operand is converted to the data type of the destination operand and the destination operand is replaced by the result. The form of the conversion is as follows:

CVTBF	exact
CVTBD	exact
CVTBG	exact
CVTWF	exact
CVTWD	exact
CVTWG	exact
CVTLF	rounded
CVTLD	exact
CVTLG	exact
CVTFB	truncated
CVTDB	truncated
CVTGB	truncated
CVTFW	truncated
CVTDW	truncated
CVTGW	truncated
CVTFL	truncated
CVTRFL	rounded
CVTDL	truncated
CVTRDL	rounded
CVTGL	truncated
CVTRGL	rounded

CVTFD	exact
CVTFG	exact
CVTDF	rounded
CVTGF	rounded

Notes:

1. Only CVTDF and CVTGF can result in a floating overflow fault; the destination operand is unaffected and the condition codes are unpredictable.

2. Only converts with a floating point source operand can result in a reserved operand fault. On a reserved operand fault, the destination operand is unaffected and the condition codes are unpredictable.

3. Only converts with an integer destination operand can result in integer overflow. On integer overflow, the destination operand is replaced by the low order bits of the true result.

4. Only CVTGF can result in floating underflow. If FU is set, a fault occurs and the destination operand is unaffected. If FU is clear, the destination operand is replaced by 0 and no fault occurs.

DIV Divide

Format:

```
opcode divr.rx, quo.mx              2 operand

opcode divr.rx, divd.rx, quo.wx 3 operand
```

Operation:

```
quo <- quo / divr;          !2 operand

quo <- divd / divr;         !3 operand
```

Condition Codes:

```
N <- quo LSS 0;
Z <- quo EQL 0;
V <- 0;
C <- 0;
```

Exceptions:

floating overflow
floating underflow
divide by zero
reserved operand

Opcodes:

46	DIVF2	Divide F_floating 2 Operand
47	DIVF3	Divide F_floating 3 Operand
66	DIVD2	Divide D_floating 2 Operand
67	DIVD3	Divide D_floating 3 Operand
46FD	DIVG2	Divide G_floating 2 Operand
47FD	DIVG3	Divide G_floating 3 Operand

Description:

In 2 operand format, the quotient operand is divided by the divisor operand and the quotient operand is replaced by the rounded result. In 3 operand format, the dividend operand is divided by the divisor operand and the quotient operand is replaced by the rounded result.

Notes:

1. On a reserved operand fault, the quotient operand is unaffected and the condition codes are unpredictable.

2. On floating underflow, if FU is set, a fault occurs and the quotient operand is unaffected. If FU is clear, the quotient operand is replaced by 0 and no exception occurs.

3. On floating overflow, the instruction faults; the quotient operand is unaffected, and the condition codes are unpredictable.

4. On divide by zero, the quotient operand and condition codes are affected as in 3. above.

EMOD Extended Multiply and Integerize

Format:

EMODF:

```
opcode mulr.rx, mulrx.rb, muld.rx, int.wl, fract.wx
```

EMODG:

```
opcode mulr.rx, mulrx.rw, muld.rx, int.wl, fract.wx
```

Operation:

```
int <- integer part of muld * {mulr'mulrx};
fract <- fractional part of muld * {mulr'mulrx};
```

Condition Codes:

```
N <- fract LSS 0;
Z <- fract EQL 0;
V <- {integer overflow};
C <- 0;
```

Exceptions:

integer overflow
floating underflow
reserved operand

Opcodes:

54	EMODF	Extended Multiply and Integerize F_floating
74	EMODD	Extended Multiply and Integerize D_floating
54FD	EMODG	Extended Multiply and Integerize G_floating

Description:

The multiplier extension operand is concatenated with the multiplier operand to gain 8 (EMODF and EMODD) or 11 (EMODG) additional low order fraction bits. The low order 5 bits of the 16-bit multiplier extension operand are ignored by the EMODG instruction. The multiplicand operand is multiplied by the extended multiplier operand. The multiplication is such that the result is equivalent to the exact product truncated (before normalization) to a fraction field of 32 bits in F_floating, 64 bits in D_floating and G_floating. Regarding the result as the sum of an integer and fraction of the same sign, the integer operand is replaced by the integer part of the result and the fraction operand is replaced by the rounded fractional part of the result.

Notes:

1. On a reserved operand fault, the integer operand and the fraction operand are unaffected. The condition codes are unpredictable.

2. On floating underflow, if FU is set, a fault occurs and the integer and fraction parts are unaffected. If FU is clear, the integer and fraction parts are replaced by 0 and no exception occurs.

3. On integer overflow, the integer operand is replaced by the low order bits of the true result.

4. Floating overflow is indicated by integer overflow; however integer overflow is possible in the absence of floating overflow.

5. The signs of the integer and fraction are the same unless integer overflow results.

6. Because the fraction part is rounded after separation of the integer part, it is possible that the value of the fraction operand is 1.

MNEG Move Negated

Format:

```
opcode src.rx, dst.wx
```

Operation:

```
dst <- -src;
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- 0;
C <- 0;
```

Exceptions:

reserved operand

Opcodes:

52	MNEGF	Move Negated F_floating
72	MNEGD	Move Negated D_floating
52FD	MNEGG	Move Negated G_floating

Description:

The destination operand is replaced by the negative of the source operand.

Notes:

On a reserved operand fault, the destination operand is unaffected and the condition codes are unpredictable.

MOV Move

Format:

```
opcode src.rx, dst.wx
```

Operation:

```
dst <- src;
```

Condition Codes:

```
N <- dst LSS 0;
Z <- dst EQL 0;
V <- 0;
C <- C;
```

Exceptions:

reserved operand

Opcodes:

50	MOVF	Move F_floating
70	MOVD	Move D_floating
50FD	MOVG	Move G_floating

Description:

The destination operand is replaced by the source operand.

Notes:

On a reserved operand fault, the destination operand is unaffected and the condition codes are unpredictable.

MUL Multiply

Format:

```
opcode mulr.rx, prod.mx                    2 operand

opcode mulr.rx, muld.rx, prod.wx           3 operand
```

Operation:

```
prod <- prod * mulr;     !2 operand

prod <- muld * mulr;     !3 operand
```

Condition Codes:

```
N <- prod LSS 0;
Z <- prod EQL 0;
V <- 0;
C <- 0;
```

Exceptions:

```
floating overflow
floating underflow
reserved operand
```

Opcodes:

44	MULF2	Multiply F_floating 2 Operand
45	MULF3	Multiply F_floating 3 Operand
64	MULD2	Multiply D_floating 2 Operand
65	MULD3	Multiply D_floating 3 Operand
44FD	MULG2	Multiply G_floating 2 Operand
45FD	MULG3	Multiply G_floating 3 Operand

Description:

In 2 operand format, the product operand is multiplied by the multiplier operand and the product operand is replaced by the rounded result. In 3 operand format, the multiplicand operand is multiplied by the multiplier operand and the product operand is replaced by the rounded result.

Notes:

1. On a reserved operand fault, the product operand is unaffected and the condition codes are unpredictable.

2. On floating underflow, if FU is set, a fault occurs and the product operand is unaffected. If FU is clear, the product operand is replaced by 0 and no exception occurs.

3. On floating overflow, the instruction faults; the product operand is unaffected, and the condition codes are unpredictable.

POLY Polynomial Evaluation

Format:

```
opcode arg.rx, degree.rw, tbladdr.ab
```

Operation:

```
tmp1 <- degree;
if tmp1 GTRU 31 then RESERVED OPERAND FAULT;
tmp2 <- tbladdr;
tmp3 <- {(tmp2)+};      !tmp3 accumulates the partial result
                        !tmp3 is of type x
while tmp1 GTRU 0 do
begin           !computation loop
tmp4 <- {arg * tmp3};   !tmp4 accumulates new partial result.
                        !tmp3 has old partial result.
        !Perform multiply, and retain the 31 (POLYF) or
        !63 (POLYD, POLYG) most significant bits of the fraction
        !by truncating the unnormalized product. (The most
        !significant bit of the 31 or 63 bits
        !in the product magnitude will be zero
        !if the product magnitude is LSS 1/2 and GEQ 1/4.)
        !Use the result in the following add operation.
tmp4 <- tmp4 + (tmp2);
        !Align fractions, perform add, and retain the
        !31 (POLYF), 63 (POLYD, POLYG) most significant bits of
        !the fraction by truncating the unnormalized result.
        !normalize, and round to type x.
        !Check for over/underflow only after the combined
        !multiply/add/normalize/round sequence.
if OVERFLOW then FLOATING OVERFLOW FAULT
if UNDERFLOW then
        begin
        if FU EQL 1 then FLOATING UNDERFLOW FAULT;
        tmp4 <- 0;      !force result to 0;
        end;
tmp1 <- tmp1 - 1;
tmp2 <- tmp2 + {size of data type};
tmp3 <- tmp4;   !update partial result in tmp3
end;
if POLYF then
        begin
        R0 <- tmp3;
        R1 <- 0;
        R2 <- 0;
        R3 <- tmp2;
        end;
if POLYD or POLYG then
        begin
```

```
R1'R0 <- tmp3;
R2 <- 0;
R3 <- tmp2;
R4 <- 0;
R5 <- 0;
end;
```

Condition Codes:

```
N <- R0 LSS 0;
Z <- R0 EQL 0;
V <- 0;
C <- 0;
```

Exceptions:

floating overflow
floating underflow
reserved operand

Opcodes:

55	POLYF	Polynomial Evaluation F_floating
75	POLYD	Polynomial Evaluation D_floating
55FD	POLYG	Polynomial Evaluation G_floating

Description:

The table address operand points to a table of polynomial coefficients. The coefficient of the highest order term of the polynomial is pointed to by the table address operand. The table is specified with lower order coefficients stored at increasing addresses. The data type of the coefficients is the same as the data type of the argument operand. The evaluation is carried out by Horner's method and the contents of R0 (R1'R0 for POLYD or POLYG) are replaced by the result. The result computed is:

```
if d = degree
and x = arg
result = C[0]*x**0 + x*(C[1] + x*(C[2] + ... x*C[d]))
```

The unsigned word degree operand specifies the highest numbered coefficient to participate in the evaluation.

Notes:

1. After execution:

```
POLYF
R0 = result
R1 = 0
R2 = 0
R3 = table address + degree*4 + 4

POLYD or POLYG
R0 = high order part of result
R1 = low order part of result
R2 = 0
R3 = table address + degree*8 + 8
R4 = 0
R5 = 0
```

2. On a floating fault:

   a. If PSL<FPD> = 0, the instruction faults and all relevant side effects are restored to their original state.

   b. If PSL<FPD> = 1, the instruction is suspended and state is saved in the general registers as follows:

```
POLYF
R0 = tmp3          !partial result after iteration prior to the
                   !one causing the overflow/underflow
R1 = arg
R2<7:0> = tmp1     !number of iterations remaining
R2<31:8> = implementation specific
R3 = tmp2          !points to table entry causing exception

POLYD and POLYG
R1'R0 = tmp3       !partial result after iteration prior to the
                   !one causing the overflow/underflow
R2<7:0> = tmp1     !number of iterations remaining
R2<31:8> = implementation specific
R3 = tmp2          !points to table entry causing exception
R5'R4 = arg
```

3. If the unsigned word degree operand is 0 and the argument is not a reserved operand, the result is C[0]. If the degree is 0, and either the argument or C[0] is a reserved operand, a reserved operand fault occurs.

4. If the unsigned word degree operand is greater than 31, a reserved operand fault occurs.

5. On a reserved operand fault:

   a. if PSL<FPD> = 0, the reserved operand is either the degree operand (greater than 31), or the argument operand, or some coefficient.

   b. if PSL<FPD> = 1, the reserved operand is a coefficient, and R3 is pointing at the value which caused the exception.

   c. The state of the saved condition codes and the other registers is unpredictable. If the reserved operand is changed and the contents of the condition codes and all registers are preserved, the fault is able to be continued.

6. On floating underflow after the rounding operation at any iteration of the computation loop, a fault occurs if FU is set. If FU is clear, the temporary result (tmp3) is replaced by zero and the operation continues. In this case the final result may be non zero if underflow occurred before the last iteration.

7. On floating overflow after the rounding operation at any iteration of the computation loop, the instruction terminates with a fault.

8. If the argument is zero, the result is C[0]. Additionally, if one of the coefficients in the table (other than C[0]) is a reserved operand, whether a reserved operand fault occurs is unpredictable.

Example:

To compute P(x) = C0 + C1*x + C2*x**2
where C0 = 1.0, C1 = .5, and C2 = .25

```
        POLYF   X,#2,PTABLE
        .
        .
        .
PTABLE: .FLOAT  0.25    ;C2
        .FLOAT  0.5     ;C1
        .FLOAT  1.0     ;C0
```

SUB Subtract

Format:

```
opcode sub.rx, dif.mx              2 operand

opcode sub.rx, min.rx, dif.wx      3 operand
```

Operation:

```
dif <- dif - sub;          !2 operand

dif <- min - sub;          !3 operand
```

Condition Codes:

```
N <- dif LSS 0;
Z <- dif EQL 0;
V <- 0;
C <- 0;
```

Exceptions:

```
floating overflow
floating underflow
reserved operand
```

Opcodes:

42	SUBF2	Subtract F_floating 2 Operand
43	SUBF3	Subtract F_floating 3 Operand
62	SUBD2	Subtract D_floating 2 Operand
63	SUBD3	Subtract D_floating 3 Operand
42FD	SUBG2	Subtract G_floating 2 Operand
43FD	SUBG3	Subtract G_floating 3 Operand

Description:

In 2 operand format, the subtrahend operand is subtracted from the difference operand and the difference is replaced by the rounded result. In 3 operand format, the subtrahend operand is subtracted from the minuend operand and the difference operand is replaced by the rounded result.

Notes:

1. On a reserved operand fault, the difference operand is unaffected and the condition codes are unpredictable.

2. On floating underflow, if FU is set, a fault occurs and the difference operand is unaffected. If FU is clear, the difference operand is replaced by 0 and no exception occurs.

3. On floating overflow, the instruction faults; the difference operand is unaffected, and the condition codes are unpredictable.

TST Test

Format:

```
opcode src.rx
```

Operation:

```
src - 0;
```

Condition Codes:

```
N <- src LSS 0;
Z <- src EQL 0;
V <- 0;
C <- 0;
```

Exceptions:

reserved operand

Opcodes:

53	TSTF	Test F_floating
73	TSTD	Test D_floating
53FD	TSTG	Test G_floating

Description:

The condition codes are affected according to the value of the source operand.

Notes:

On a reserved operand fault, the condition codes are unpredictable.

4.12 EMULATED INSTRUCTIONS WITH MICROCODE ASSIST

The MicroVAX 78032 CPU provides microcode assistance for emulation by system software of the following instructions:

1. Character string: MOVTC, MOVTUC, SKPC, LOCC, SCANC, SPANC, MATCHC, CMPC3, CMPC5.

2. Decimal string: MOVP, CMPP3, CMPP4, ADDP4, ADDP6, SUBP4, SUBP6, MULP, DIVP, ASHP, CVTPL, CVTLP, CVTPS, CVTSP, CVTTP, CVTPT.

3. Cyclic redundancy check: CRC.

4. Edit: EDITPC.

The processor processes the operand specifiers, creates an argument list, and takes an emulated instruction fault.

The emulation process consists of the following steps if the TP and FPD bits of the PSL are clear.

1. Evaluate the operand specifiers in order of instruction stream occurrence. Save the address for address and write access type; read the operand and save it for read access type.

2. Build the argument list on the exception stack. The exception stack, Figure 4-16, contains 10 longword parameters (in addition to the PC and PSL to be restored on returning from this exception).

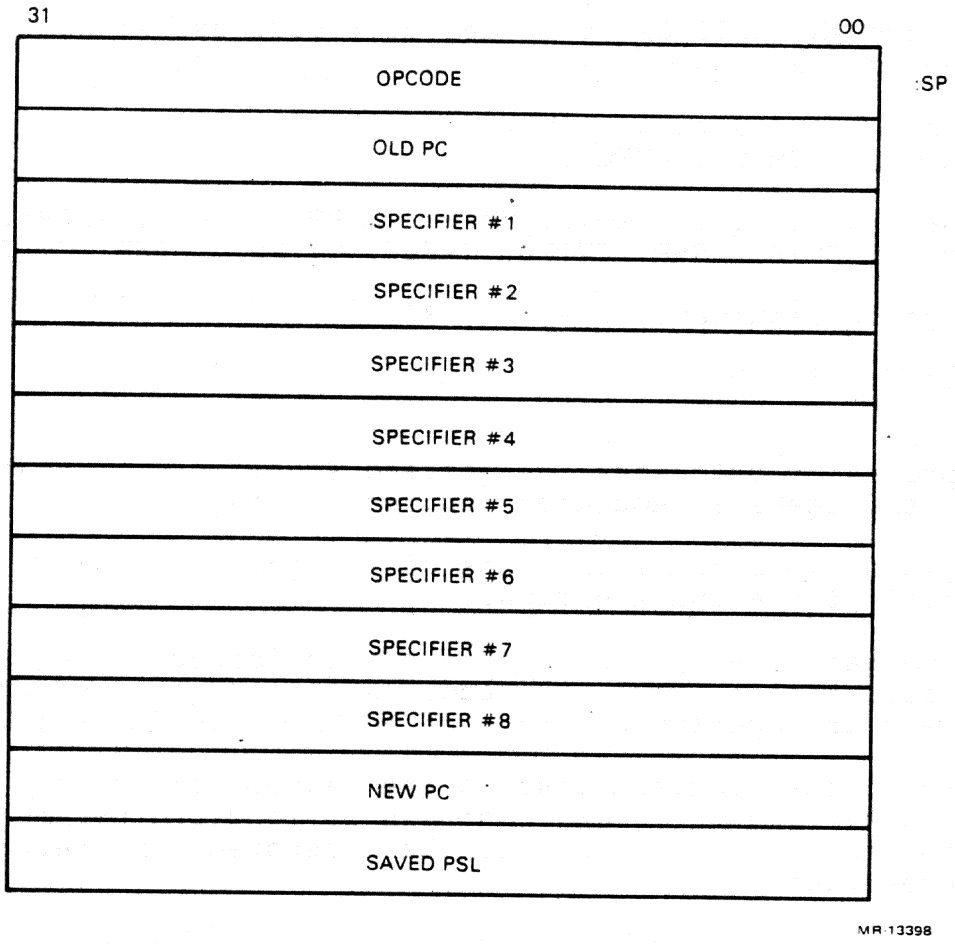


Figure 4-16 Emulated Instruction Argument List

The opcode parameter contains the opcode of the instruction to be emulated. The old PC points to the location of the instruction causing the exception. The specifier parameters contain the address of the operand or the operand itself. For a .rx specifier, the parameter is the operand value; for .wx and .ax specifiers, the parameter is the operand address. A register is denoted by a reserved system space address corresponding to the one's 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 instruction causing the exception.

3. Initiate an exception in current mode through the emulated instruction vector using C8 (hex) as the SCB offset. The FPD bit in the saved PSL is clear. The TP bit in the saved PSL is set if T was set. The T, TP, IV, DV, FU and condition code bits in the

new PSL are cleared. All other bits in the new PSL are unchanged from their previous state.

If FPD is set, a suspended emulated instruction fault is taken in current mode through the vector at SCB offset CC (hex). No parameters are pushed onto the stack. The TP bit in the saved PSL is set if T was set. All other bits in the saved PSL are unchanged. The saved PC points to the instruction causing the exception. The FPD, T, TP, IV, DV, FU and condition code bits in the new PSL are cleared. All other bits in the new PSL are unchanged from their previous state.

CHAPTER 5

BUS TRANSACTIONS

5.1 INTRODUCTION

This chapter describes the bus cycles used by the MicroVAX 78032 CPU. The processor will perform a bus cycle for one of the following reasons:

1. Reading or writing information from or to memory or a peripheral device.

2. Acknowledging an interrupt by reading the device interrupt vector.

3. Transferring information from or to an external processor or externally implemented processor register.

Figure 5-1 shows the bus connections used by the MicroVAX 78032 CPU.

A microcycle is the basic timing unit for a bus cycle. A microcycle is defined as four cycles of CLKO, T1 through T4, as shown in Figure 5-2. A bus cycle may be one or more microcycles.

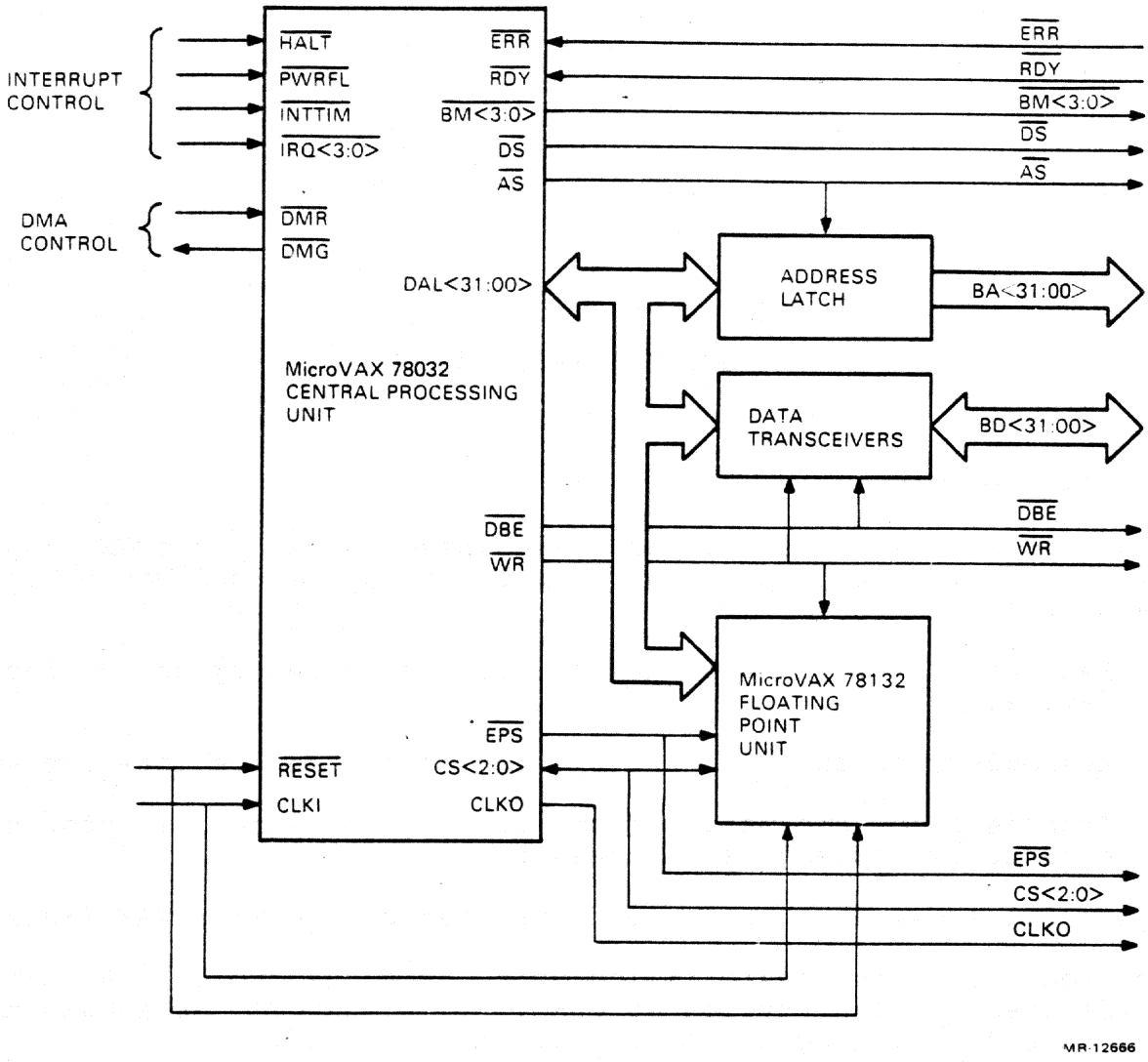


Figure 5-1 MicroVAX 78032 Bus Connections

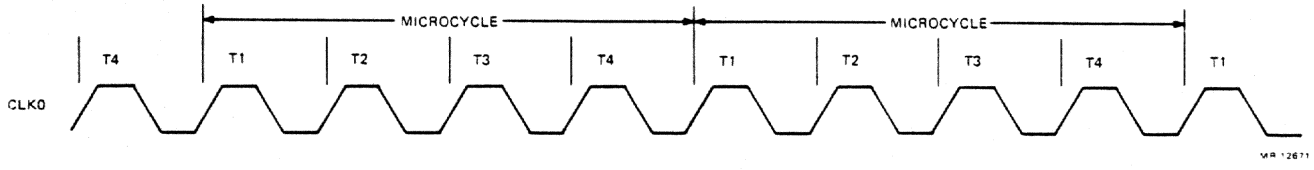


Figure 5-2 MicroVAX 78032 Microcycle

5.2 BUS CYCLES

The MicroVAX 78032 CPU uses read, write, DMA, and interrupt acknowledge bus cycles for the transfer of information between the processor, memory, and I/O devices. Each of these bus cycles is described in the following paragraphs.

5.2.1 CPU Read Cycle

The processor uses a CPU read cycle (Figure 5-3) to input information from memory or an I/O device. A CPU read cycle requires a minimum of 2 microcycles and may be extended for slower memory or I/O devices.

The first microcycle of a CPU read is used to output the address and control information. During the last microcycle of a CPU read, data is latched into the processor.

The sequence of events for a CPU read is as follows:

1. The physical (longword) address is driven onto DAL<29:02> by the processor.

2. $\overline{WR}$ is unasserted and CS<2:0> are asserted as required to indicate the type of read cycle being performed.

3. $\overline{BM}$<3:0> are asserted as required.

4. $\overline{AS}$ is asserted to indicate that the address is valid and can be latched for demultiplexing. $\overline{AS}$ also qualifies CS<2:0>, $\overline{BM}$<3:0>, and $\overline{WR}$.

5. $\overline{DS}$ is asserted to indicate that the bus is free to receive the requested information. $\overline{DBE}$ is also asserted at this time and can be used to control the DAL bus transceivers.

6. If the requested data can be placed on the bus and be valid during T3 of the next microcycle, external logic asserts $\overline{RDY}$, and the next microcycle is the last one for this bus cycle. If $\overline{RDY}$ is not asserted by the end of the current microcycle, the bus cycle will be extended by at least one microcycle.

   If a bus error occurs, external logic responds by asserting $\overline{ERR}$. $\overline{ERR}$ takes precedence over $\overline{RDY}$. If $\overline{ERR}$ is asserted during a data read, the processor ignores the data on DAL<31:00>, extends the bus cycle by one microcycle and initiates a machine check. If $\overline{ERR}$ is asserted during an instruction read (CS<2:0> = 100), the processor stops prefetching; when the instruction buffer is empty, the processor will attempt to fetch the next instruction byte with a data read cycle.

The assertion of either $\overline{RDY}$ or $\overline{ERR}$ results in the completion of the current bus cycle.

7. The requested data is latched into the processor and $\overline{DS}$ is deasserted.

8. $\overline{AS}$ and $\overline{DBE}$ are deasserted, ending the bus cycle.

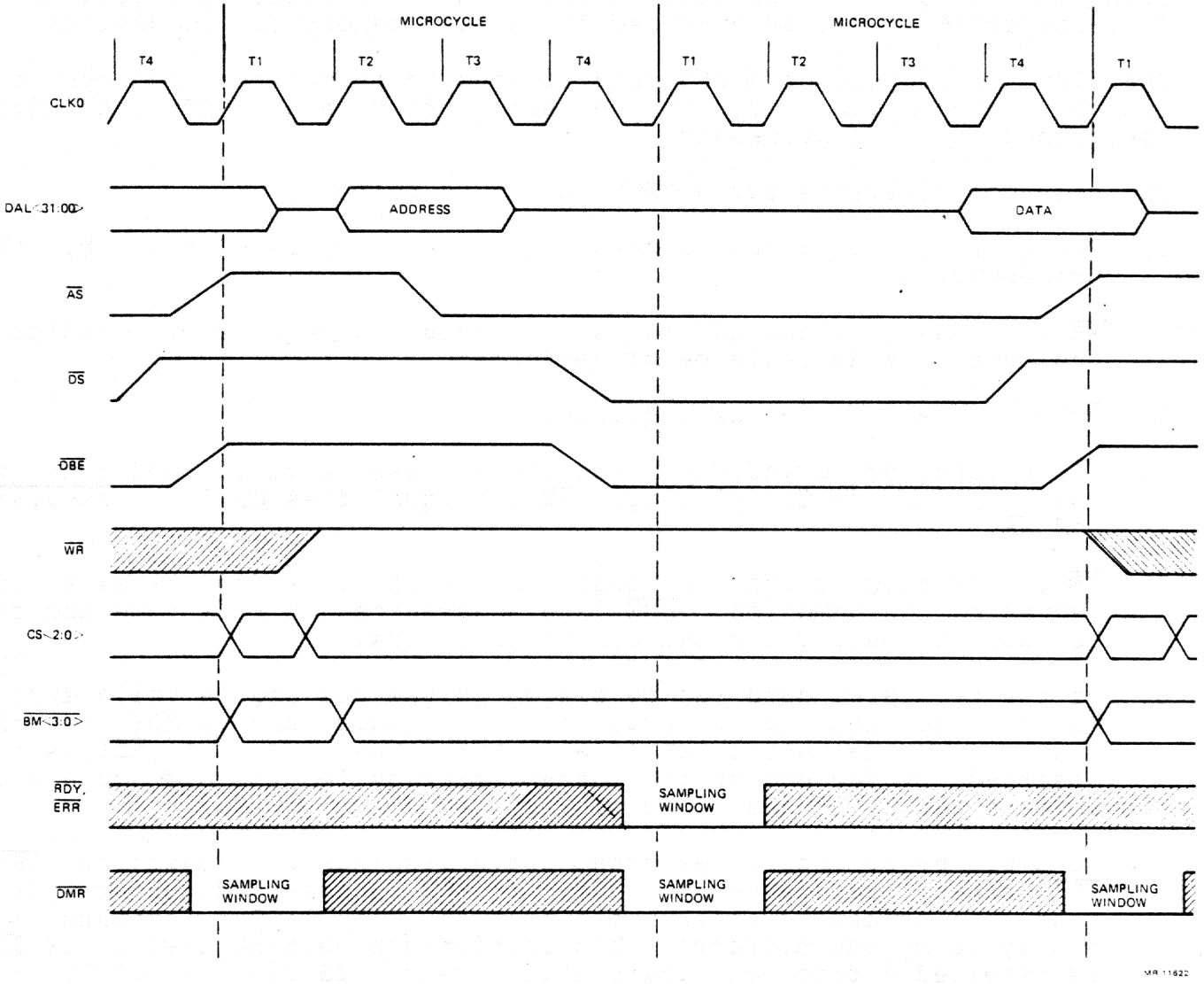


Figure 5-3 CPU Read Bus Cycle

5.2.2 CPU Write Cycle

The processor uses a CPU write cycle (Figure 5-4) to output information to memory or an I/O device. A CPU write cycle requires a minimum of 2 microcycles and may be extended for slower memory or I/O devices.

The first microcycle of a CPU write is used to output the address and control information. During the last microcycle of a CPU write, the data is valid and can be written.

The sequence of events for a CPU write is as follows:

1. The physical (longword) address is driven onto DAL<29:02> by the processor.

2. $\overline{WR}$ is asserted and CS<2:0> are asserted as required.

3. $\overline{BM<3:0>}$ are asserted as required indicating which byte(s) are to be written.

4. $\overline{AS}$ is asserted to indicate that the address is valid and can be latched for demultiplexing. $\overline{AS}$ also qualifies CS<2:0>, $\overline{BM<3:0>}$, and $\overline{WR}$.

5. $\overline{DBE}$ is asserted and can be used to control DAL bus transceivers.

6. The processor drives data onto the DAL bus and asserts $\overline{DS}$ to indicate that the data is valid.

7. If the data can be written during the next microcycle, external logic asserts $\overline{RDY}$ and the next microcycle is the last one for this bus cycle. If $\overline{RDY}$ is not asserted by the end of the current microcycle, the bus cycle will be extended by at least one microcycle.

   If a bus error occurs, external logic responds by asserting $\overline{ERR}$ and the processor initiates a machine check. $\overline{ERR}$ takes precedence over $\overline{RDY}$.

   The assertion of either $\overline{RDY}$ or $\overline{ERR}$ results in the completion of the current bus cycle.

8. $\overline{DS}$ is deasserted to indicate that the data is going to be removed from the DAL bus by the processor.

9. $\overline{AS}$ and $\overline{DBE}$ are deasserted, ending the bus cycle.

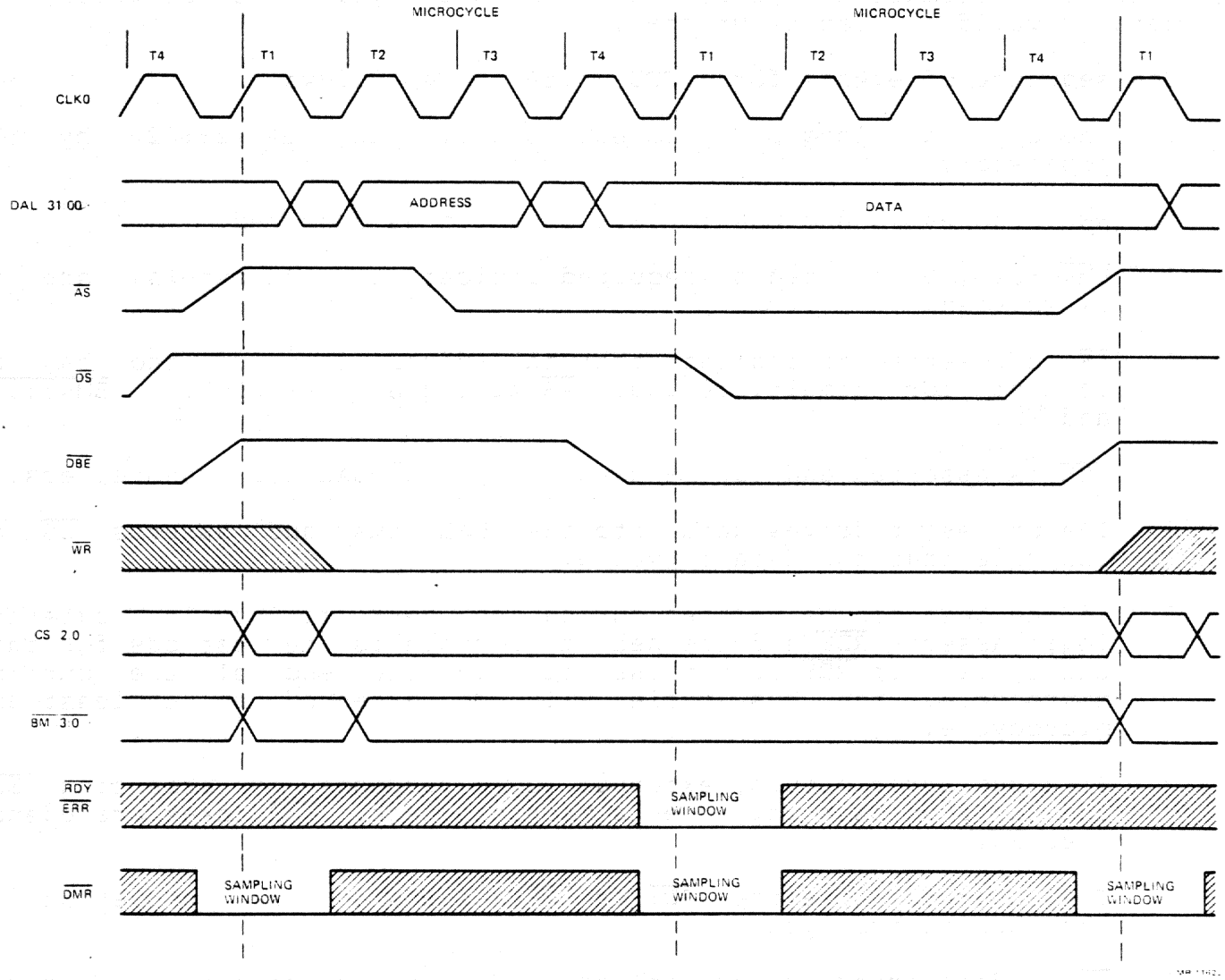


Figure 5-4 CPU Write Bus Cycle

5.2.3 Interrupt Acknowledge Cycle

An interrupt acknowledge cycle is used to acknowledge an interrupt request from an I/O device, and to read a vector. The structure of this cycle is the same as a CPU read cycle (Figure 5-3).

The first microcycle of an interrupt acknowledge cycle is used to output the IPL, in hex, that is being acknowledged. During the last microcycle the interrupt vector from the interrupting device is latched into the processor.

The sequence of events for an interrupt acknowledge cycle is as follows:

1. The processor places the IPL, in hex, of the interrupt being acknowledged on DAL<04:00>. DAL<29:05> are zero and DAL<31:30> = 10 (longword access).

2. $\overline{WR}$ is unasserted and CS<2:0> are asserted to indicate an interrupt acknowledge cycle.

3. $\overline{BM<3:0>}$ are all asserted.

4. $\overline{AS}$ is asserted to indicate that the IPL level on DAL<04:00> is valid. $\overline{AS}$ also qualifies CS<2:0>, $\overline{BM<3:0>}$, and $\overline{WR}$.

5. $\overline{DS}$ is asserted to indicate that the bus is free to receive incoming data. $\overline{DBE}$ is also asserted at this time and can be used to control the DAL bus transceivers.

6. External logic responds by placing the interrupt vector on DAL<09:02> and the normal processing/Qbus processing flag on DAL<00> and asserting $\overline{RDY}$. If $\overline{RDY}$ is not asserted by the end of the first microcycle, the bus cycle will be extended by at least one microcycle.

   DAL<15:10,01> MUST be driven to a valid high or low level in accordance with the set-up times specified in the detailed timing diagrams in Appendix A.

   If an error occurs, external logic asserts $\overline{ERR}$ and the processor cancels the cycle and ignores the data on the DAL bus.

7. The interrupt vector is latched into the processor and $\overline{DS}$ is deasserted.

8. $\overline{AS}$ and $\overline{DBE}$ are deasserted ending the bus cycle.

5.2.4 DMA Cycle

A DMA cycle (Figure 5-5) is used by the processor to relinquish control of the DAL bus and related control signals upon request from a DMA device or another processor.

The sequence of events for a DMA cycle is as follows:

1. The DMA device requests use of the bus by asserting $\overline{DMR}$.

2. The processor samples the $\overline{DMR}$ line during each microcycle, unless the current bus cycle is a read lock cycle, to see if there is a DMA request.

3. The processor three-states DAL<31:00>, $\overline{AS}$, $\overline{DS}$, $\overline{DBE}$, $\overline{WR}$, $\overline{BM<3:0>}$, and CS<2:0> and then asserts $\overline{DMG}$ to grant the DMA device use of the DAL bus.

4. When the requesting device is finished using the bus, it deasserts $\overline{AS}$, $\overline{DS}$, $\overline{DBE}$, and three-states DAL<31:00>. The requesting device deasserts $\overline{DMR}$ and the MicroVAX 78032 CPU takes control of the bus.

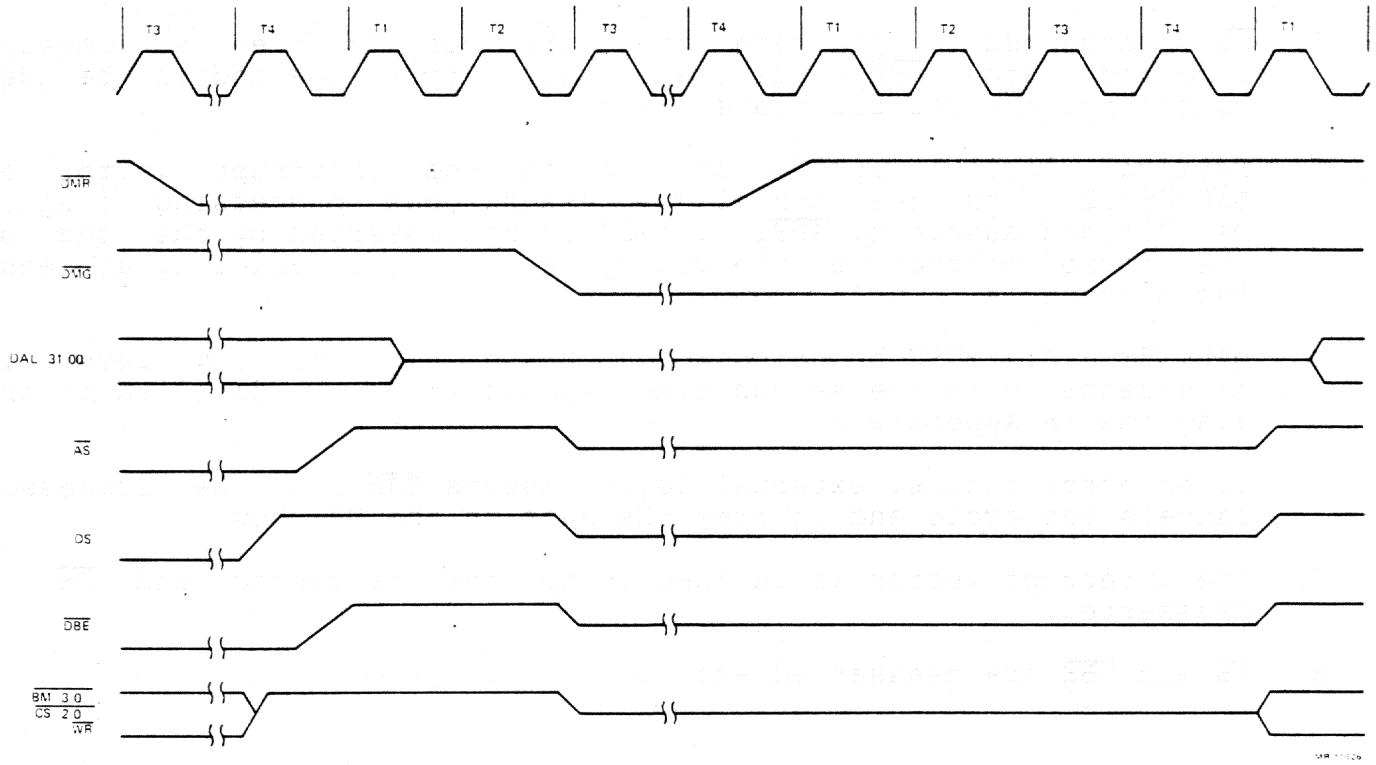


Figure 5-5 DMA Cycle

5.3 EXTERNAL PROCESSOR CYCLES

The processor uses the external processor cycles to communicate with an external processor or externally implemented processor registers. The external processor protocols are described in Section 5.5.

It is important to note that $\overline{AS}$, $\overline{DBE}$, $\overline{DS}$, and $\overline{BM<3:0>}$ are not used and are not asserted during external processor cycles. External processor cycles are always 32-bits. This means that DAL<31:00> must be driven to a valid level for all read cycles.

5.3.1 External Processor Read Cycle

The processor uses an external processor read cycle (Figure 5-6) to input information from an external processor or external processor registers. An external processor read cycle takes one microcycle and can not be extended.

The sequence of events for an external processor read is as follows:

1. CS<1:0> are asserted as required and CS<2> is precharged and sustained high.
2. $\overline{WR}$ is not asserted for a read cycle.
3. $\overline{EPS}$ is asserted to indicate an external processor bus cycle and to qualify CS<2:0> and $\overline{WR}$.
4. The external processor places the requested information on the DAL.
5. The requested information is latched into the processor and $\overline{EPS}$ is deasserted.
6. The external processor removes its information from the DAL, ending the bus cycle.

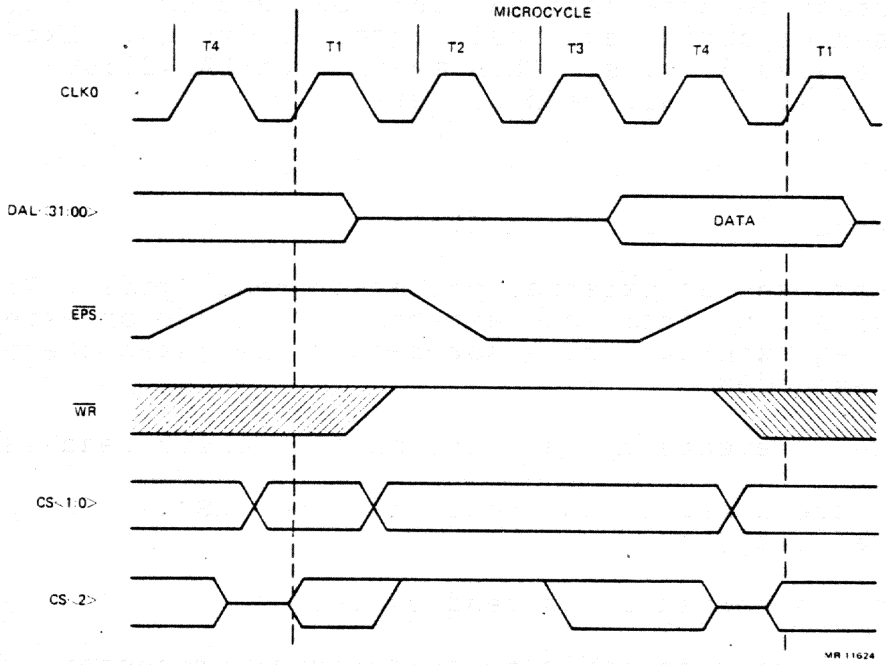


Figure 5-6 External Processor Read/Response Cycle

5.3.2 External Processor Response Cycle

The processor uses an external processor response cycle (Figure 5-6) to input information and a completion or confirmation signal from an external processor or external processor register. An external processor response cycle takes one microcycle and can not be extended.

The sequence of events for an external processor response is as follows:

1. CS<1:0> are asserted as required and CS<2> is precharged and sustained high.
2. $\overline{WR}$ is not asserted for a read cycle.
3. $\overline{EPS}$ is asserted to indicate an external processor bus cycle and to qualify CS<2:0> and $\overline{WR}$.
4. The external processor places the requested information on the DAL, and optionally drives CS<2> low with an open drain driver.
5. The requested information is latched into the processor and $\overline{EPS}$ is deasserted.
6. The external processor removes its information from the DAL and deasserts CS<2>, if asserted, ending the bus cycle.

5.3.3 External Processor Write Cycle

The processor uses an external processor write cycle (Figure 5-7) to output information to an external processor or external processor register. An external processor write cycle takes one microcycle and can not be extended.

The sequence of events for an external processor write is as follows:

1. CS<1:0> are asserted as required and CS<2> is precharged and sustained high.
2. $\overline{WR}$ is asserted for a write cycle.
3. $\overline{EPS}$ is asserted to indicate an external processor bus cycle and to qualify CS<2:0> and $\overline{WR}$.
4. The processor drives the information onto the DAL.
5. $\overline{EPS}$ is deasserted and the external processor reads the information, ending the bus cycle.

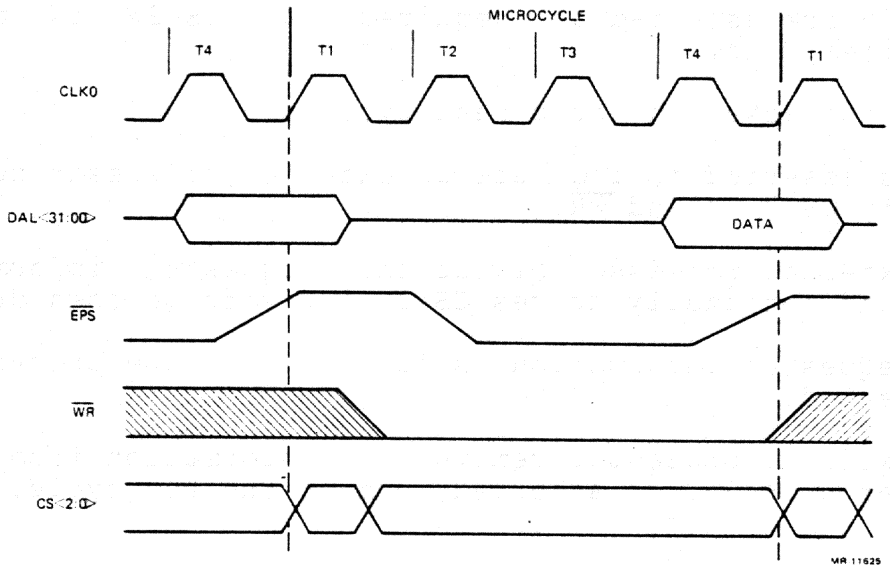


Figure 5-7 External Processor Write Cycle

5.4 MEMORY ACCESS PROTOCOL

The 28-bit address provided by the processor on DAL<29:02> is a LONGWORD address which uniquely identifies one of up to 268,435,456 32-bit memory locations. To facilitate byte accesses within the 32-bit memory locations four byte masks, $\overline{BM<3:0>}$, are used. There are no restrictions on data alignment, with the exception of the aligned operands of ADAWI and the interlocked queue instructions. With these exceptions, any data item, regardless of size, may be placed starting at any memory address.

Memory is viewed as four parallel eight-bit banks, each of which receives the longword address DAL<29:02>. Each bank reads or writes one byte of the data bus (DAL<31:00>), when its associated byte mask signal is asserted. This is illustrated in Figure 5-8.

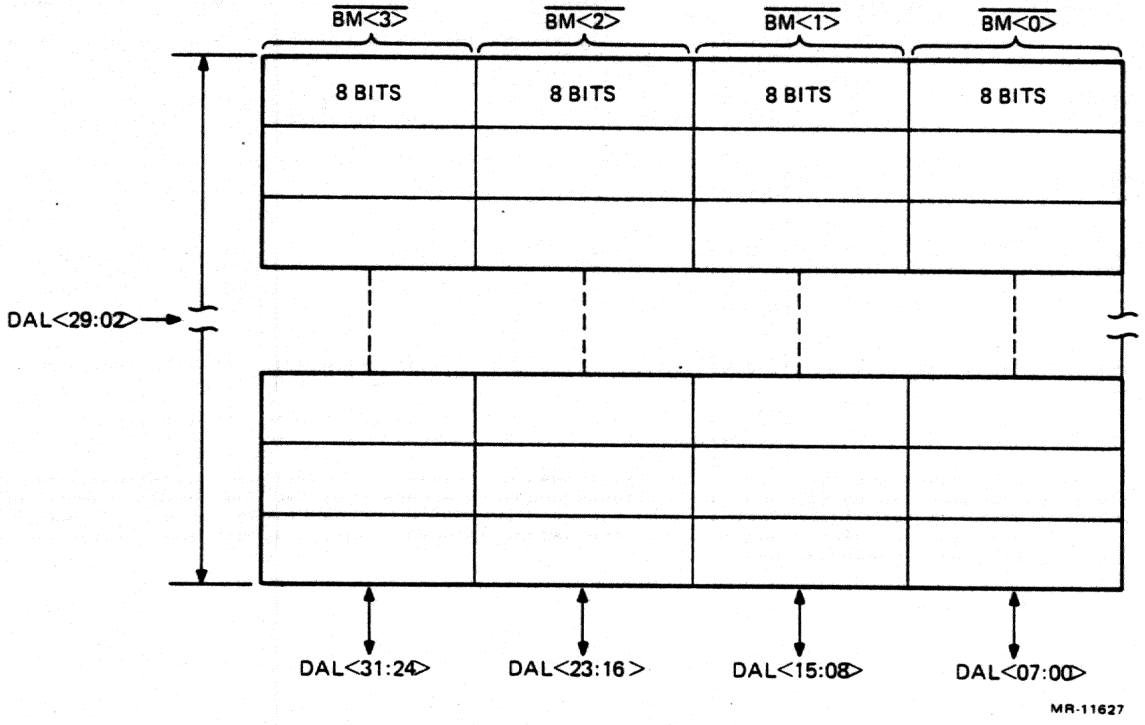


Figure 5-8 MicroVAX 78032 Memory Organization

Any CPU read or CPU write falls into one of the following categories: byte access, word access within a longword, word access across longwords, aligned longword access, unaligned longword access. (Quadword accesses are treated as two successive longword accesses, with no optimization.) Byte accesses, word accesses within a longword, and aligned longword accesses require one bus cycle. Word accesses which cross a longword boundary, and unaligned longword accesses, require two bus cycles. The exact signal usage is shown in Table 5-1.

It is important to note that accesses requiring more than one bus cycle are performed sequentially, with no computation in between. However, DMA grants may occur between the bus cycles of an unaligned reference.

Table 5-1 Memory Access Control

Access Type	Cycle	DAL<31:30>	DAL<29:02>	$\overline{BM<3>}$	$\overline{BM<2>}$	$\overline{BM<1>}$	$\overline{BM<0>}$
Byte	1	00	A<29:02>	if A<1:0>=11	if A<1:0>=10	if A<1:0>=01	if A<1:0>=00
Word within longword	1	01	A<29:02> [A<1:0> ne 11]	if A<1:0>=10	if A<1:0>=10 or A<1:0>=01	if A<1:0>=0X	if A<1:0>=00
Aligned longword	1	10	A<29:02> [A<1:0> = 00]	L	L	L	L
Word across longwords	1	01	A<29:02>	L	H	H	H
	2		A+4<29:02> [A<1:0> = 11]	H	H	H	L
Unaligned longword	1	10	A<29:02>	L	if A<1:0>=01 or A<1:0>=10	if A<1:0>=01	H
	2	10	A+4<29:02> [A<1:0> ne 00]	H	if A<1:0>=11	if A<1:0>=10 or A<1:0>=11	L

Note: Quadword accesses are performed using two longword accesses. An aligned quadword access uses two aligned longword accesses and an unaligned quadword access uses two unaligned longword accesses. DAL<31:30> = 11 for the first aligned longword access and the first cycle of each unaligned longword access and DAL<31:30> = 10 for the second aligned longword access and the second cycle of each unaligned longword access.

5.5 EXTERNAL PROCESSOR PROTOCOLS

The external processor protocols allow the MicroVAX 78032 CPU to communicate efficiently with one one or more external processors. There are two distinct external processor protocols: one for communicating with the optional MicroVAX 78132 Floating Point Unit, the second for communicating with external processor register logic.

5.5.1 FPU Protocol

The optional MicroVAX 78132 Floating Point Unit (FPU) functions under the control of the processor. When the CPU receives a floating point instruction it passes the opcode and operands to the FPU for processing. The CPU waits for the FPU to finish and then requests status information and any results. The FPU protocol is as follows:

1. Command Transfer - The processor performs an external processor write cycle to transmit a command to the FPU. During this cycle, CS<1:0> = 00 (FPU command), and the opcode of the floating point instruction is placed on DAL<08:00>.

2. Operand Transfer - The VAX opcode determines the number and data type of operands to be transferred from the processor to the FPU. The processor performs one or more external processor write cycles to transfer the operands. During these cycles $\overline{WR}$ is asserted, CS<1:0> = 01 (data transfer), and DAL<31:00> contain the data to be transferred.

3. Operand Processing - While the FPU is processing the operands, the processor polls for operation completion by executing external processor response enable cycles.

4. Status Transfer - When the FPU has finished processing the operands, it responds to the next external processor response enable cycle by placing status information on DAL<05:00> and driving CS<2> low. The processor responds to CS<2> being driven low by reading the status information on DAL<05:00>.

5. Result Transfer - After reading the status code, the processor may initiate one or more external processor read cycles to transfer the result operands(s), if any. During these cycles $\overline{WR}$ is deasserted, CS<1:0> = 01 (data transfer), and DAL<31:00> contain the data to be transferred. The VAX opcode determines the number and data type of the operand(s) to be transferred from the FPU to the processor.

5.5.2 Register Protocol

The external processor register protocol permits external logic to implement processor register functions that are a part of the VAX Architecture but are not implemented in the MicroVAX 78032 CPU. Refer to Table 1-3 for a list of the processor registers implemented by the MicroVAX 78032 CPU. The processor will use one of the following protocols when an MFPR or MTPR instruction is used to access a register not contained in the processor.

5.5.2.1 Read From Processor Register -

This sequence (Figure 5-9) is performed when an MFPR instruction is used to read data from one of the following processor registers: 25 through 39, 48 through 55, or 59 through 61. The read from processor register protocol is as follows:

1. The processor initiates the transaction with an external processor write cycle to specify the register number. During this cycle, CS<1:0> = 10 (non-FPU command), DAL<31> = 1 (read register), and DAL<05:00> contain the register number specified by the MFPR instruction.

2. The processor waits one cycle.

3. The processor executes an external processor response cycle to read the register data. If CS<2> is driven low by the external logic, the data on DAL<31:00> is the result of the MFPR instruction. Otherwise, the processor returns zero as the result.

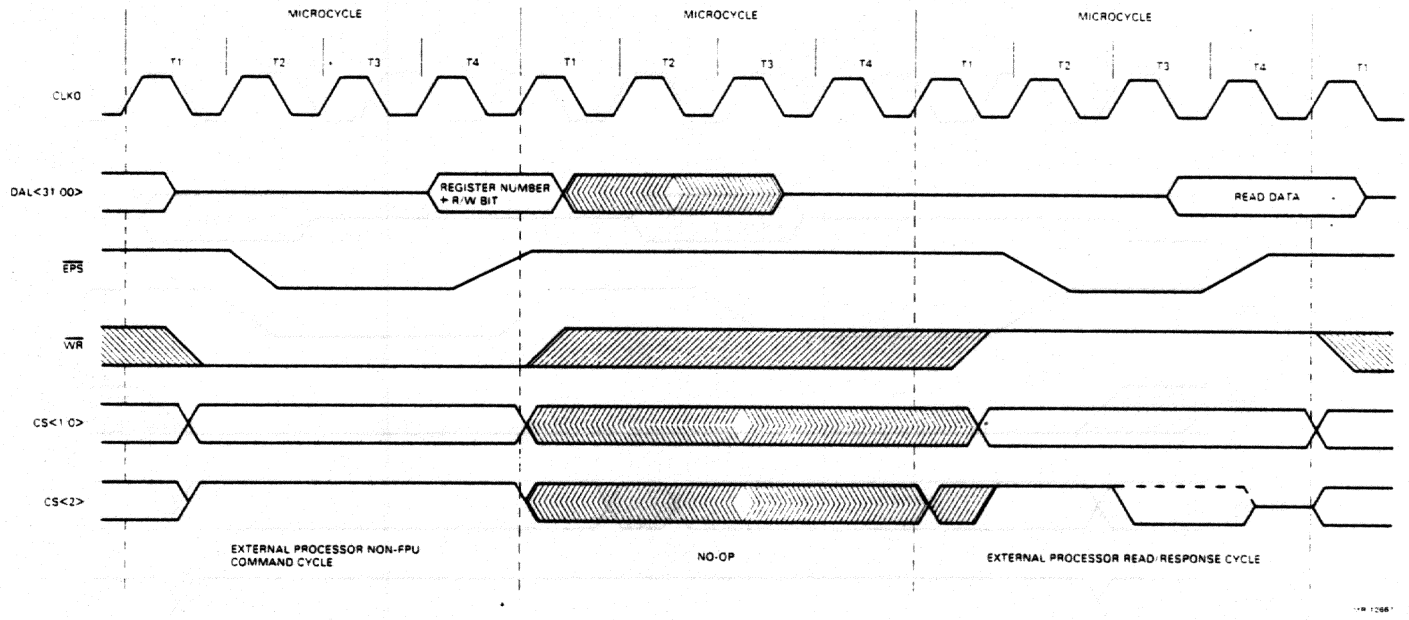


Figure 5-9 Read From Processor Register

5.5.2.2 Write To Processor Register -

This sequence (Figure 5-10) is performed when an MTPR instruction is used to read data from one of the following processor registers: 25 through 39, 48 through 55, or 59 through 61. The move to processor register protocol is as follows:

1. The processor initiates the transaction with an external processor write cycle to specify the register number. During this cycle, CS<1:0> = 10 (non-FPU command), DAL<31> = 0 (write register), and DAL<05:00> contain the register number specified by the MTPR instruction.

2. The processor executes an external processor write cycle to write the register data. During this cycle, CS<1:0> = 01 (write data), and DAL<31:00> contain the data specified in the MTPR instruction.

3. The next cycle is guaranteed not to be another external processor cycle.

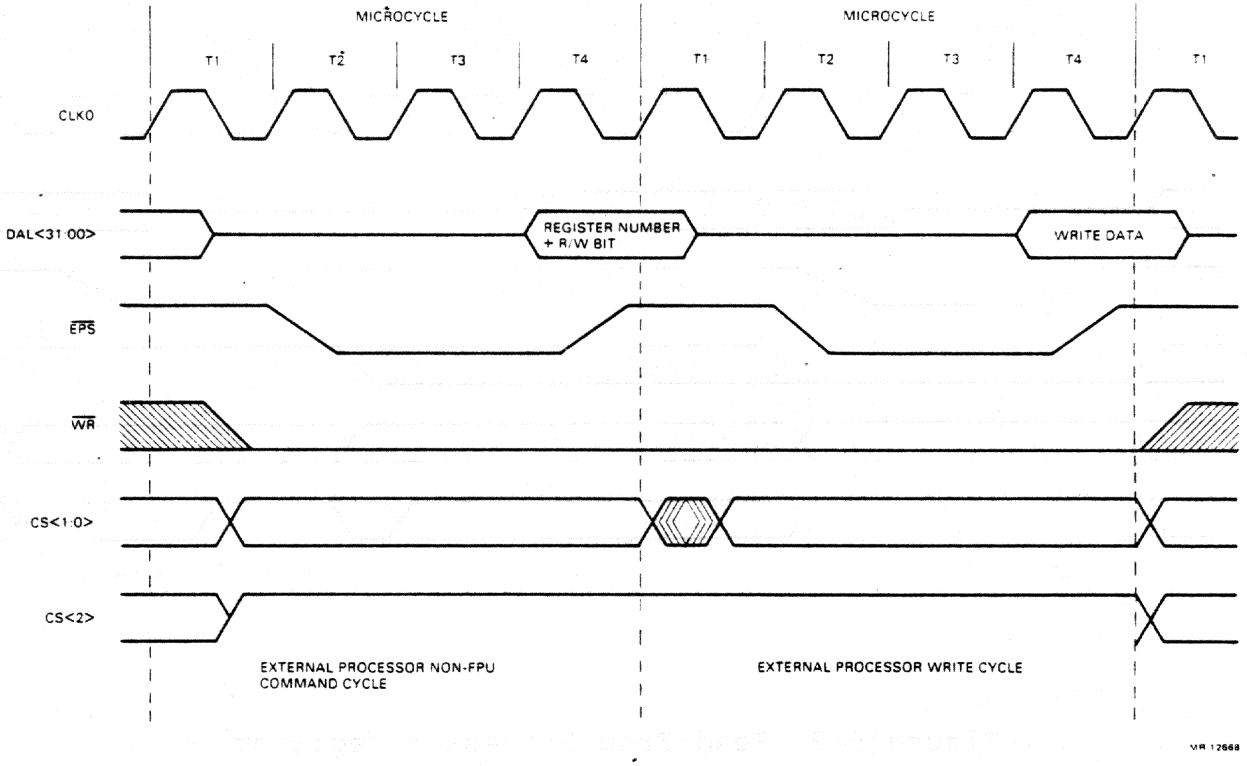


Figure 5-10 Write To Processor Register

CHAPTER 6

PIN DESCRIPTION

6.1 INTRODUCTION

This chapter describes the function performed by each pin of the MicroVAX 78032 CPU. The pins are divided into 8 groups:

- Data/Address bus
- Bus control
- System control
- Interrupt control
- DMA control
- Power supply
- Clocks
- Test

Figure 6-1 shows the pin assignments of the MicroVAX 78032 CPU.

NOTE

During the pin descriptions references will be made to the different bus cycles executed by the MicroVAX 78032 CPU. For a description of these bus cycles refer to Chapter 5, Bus Cycles.

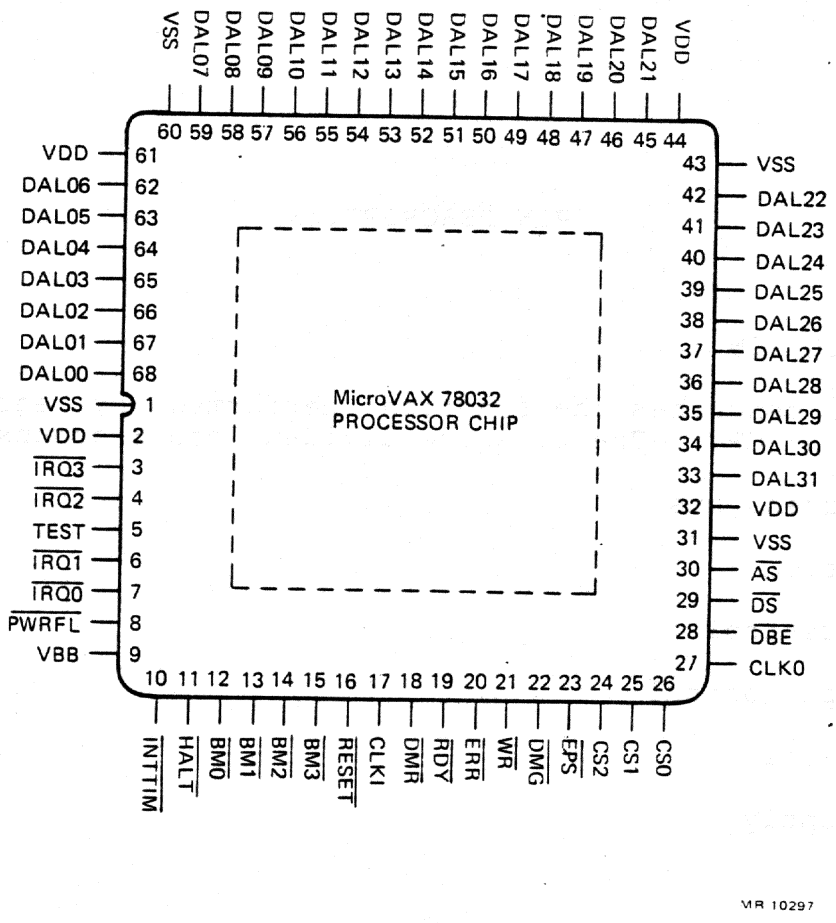


Figure 6-1 MicroVAX 78032 Pin Assignments

6.2 DATA/ADDRESS BUS

The Data/Address Bus (DAL<31:00>) is a 32-bit time-multiplexed bus used for the transfer of address, data, and interrupt information. The information carried on DAL<31:00> is determined by the type of bus cycle being executed. During the first part of a CPU read or CPU write cycle DAL<31:00> carries the following address information:

DAL<31:30> - indicates the length of the memory operand.

DAL<31>	DAL<30>	Operand Length
0	0	Byte
0	1	Word
1	0	Longword
1	1	Quadword

DAL<29:02> - contains the longword address of the memory operand. DAL<29> is used to distinguish a memory space address from an I/O space address.

DAL<29>	Address Space
0	Memory
1	I/O

DAL<01:00> - are undefined. $\overline{\text{BM<3:0>}}$ determine which byte(s) of the longword address are to be used. Refer to Section 5.4, Memory Access Protocol, for an explanation.

During the first part of an interrupt acknowledge cycle, DAL<04:00> carry the interrupt priority level (IPL), in hex, of the interrupt being acknowledged. During the second part of a CPU read or interrupt acknowledge cycle, DAL<31:00> receive incoming data or an interrupt vector. During the second part of a CPU write bus cycle, DAL<31:00> are used to transmit outgoing data.

In addition to the transfer of information between the CPU and memory, I/O devices, etc., the DAL bus is used to exchange information between external processors (i.e., FPU) and externally implemented processor registers. The information present on the DAL bus for these bus cycles is described in Section 5.5.

6.3 BUS CONTROL

There are 10 pins associated with bus control: $\overline{\text{AS}}$, $\overline{\text{DS}}$, $\overline{\text{BM<3:0>}}$, $\overline{\text{WR}}$, $\overline{\text{DBE}}$, $\overline{\text{RDY}}$, and $\overline{\text{ERR}}$. The function of each of these pins is described in the following paragraphs.

6.3.1 Address Strobe ($\overline{AS}$)

$\overline{AS}$ is used to provide timing and control information to external logic. This signal notifies external logic that a bus cycle is being executed. The assertion of $\overline{AS}$ marks the beginning of a bus cycle, and notifies external logic that the following signals are valid.

1. DAL<31:00> - valid address or interrupt priority level (IPL)
2. $\overline{WR}$ - direction of transfer
3. CS<2:0> - type of bus cycle
4. $\overline{BM<3:0>}$ - bytes of DAL bus that contain valid data during second part of bus cycle

External logic should latch and/or decode these signals as required. $\overline{AS}$ is negated to indicate the end of a bus cycle.

6.3.2 Data Strobe ($\overline{DS}$)

$\overline{DS}$ is used to provide timing information for the transfer of data. During a CPU read or interrupt acknowledge cycle, the CPU asserts $\overline{DS}$ to indicate that DAL<31:00> is free to receive incoming data. When the CPU has received and latched the incoming data, it deasserts $\overline{DS}$. During a CPU write cycle, $\overline{DS}$ is asserted by the CPU to indicate that DAL<31:00> contain valid outgoing data. $\overline{DS}$ is deasserted by the CPU to notify external logic that the CPU is about to remove the data from DAL<31:00>.

6.3.3 Byte Masks ($\overline{BM<3:0>}$)

The four byte mask lines, $\overline{BM<3:0>}$, specify which byte or bytes of the DAL bus contain valid data during the second part of a CPU read or CPU write cycle. During a CPU read cycle, the byte masks indicate which bytes of the DAL bus are latched by the CPU. During a CPU write bus cycle, the byte masks indicate which byte(s) of the DAL bus contain valid data. $\overline{BM<3:0>}$ are valid when $\overline{AS}$ is asserted.

Byte Mask bit asserted	Data valid on
$\overline{BM<3>}$	DAL<31:24>
$\overline{BM<2>}$	DAL<23:16>
$\overline{BM<1>}$	DAL<15:08>
$\overline{BM<0>}$	DAL<07:00>

NOTE

During a CPU read or external processor read/response cycle, all bits of the selected byte(s) must be driven to a valid state, except for an interrupt acknowledge cycle when only bits<15:00> must be driven.

6.3.4 Write ($\overline{WR}$)

$\overline{WR}$ specifies whether data for the current cycle is to be transferred to or from the CPU. When $\overline{WR}$ is asserted, the CPU will drive data onto the DAL bus. When $\overline{WR}$ is not asserted, the CPU is ready to receive data from the DAL bus. $\overline{WR}$ can be used to control the direction input of external DAL bus transceivers. $\overline{WR}$ is valid when $\overline{AS}$ or $\overline{EPS}$ is asserted.

6.3.5 Data Buffer Enable ($\overline{DBE}$)

$\overline{DBE}$ can be used by external logic to enable external DAL bus transceivers. This signal in conjunction with $\overline{WR}$ provide the necessary control signals for external bus transceivers.

6.3.6 Ready ($\overline{RDY}$)

$\overline{RDY}$ is asserted by external logic to normally end the current CPU read, CPU write, or interrupt acknowledge cycle. During a CPU read or interrupt acknowledge cycle, the assertion of $\overline{RDY}$ indicates that external logic will place the requested data on the DAL bus as specified in Table A-2 and Figure A-2. During a CPU write cycle, the assertion of $\overline{RDY}$ indicates that the information placed on the DAL bus by the CPU will be received as specified in Table A-2 and Figure A-3 finishes the current bus cycle and proceeds. At the conclusion of the current bus cycle ($\overline{AS}$ deasserted), external logic deasserts $\overline{RDY}$.

6.3.7 Error ($\overline{ERR}$)

$\overline{ERR}$ is asserted by external logic to indicate that an error (i.e., bus timeout or parity error) occurred during the current CPU read, CPU write, or interrupt acknowledge cycle. The assertion of $\overline{ERR}$ has priority over $\overline{RDY}$ and results in the current bus cycle being extended. At the conclusion of the extended bus cycle ($\overline{AS}$ deasserted), external logic deasserts $\overline{ERR}$. For a description of how the MicroVAX 78032 CPU handles errors refer to Section 7.6.

6.3.8 External Processor Strobe ($\overline{EPS}$)

$\overline{EPS}$ provides timing and control information for external processor transactions. When $\overline{EPS}$ is asserted, an external processor bus cycle is beginning and:

1. DAL<31:00> is ready to receive or contains valid information
2. $\overline{WR}$ is valid
3. CS<2:0> are valid

$\overline{EPS}$ is deasserted at the end of the external processor cycle. For an explanation of external transactions refer to Section 5.3.

6.4 SYSTEM CONTROL

There are 5 pins associated with system control: $\overline{RESET}$, $\overline{HALT}$, and CS<2:0>. The function of each of these pins is described in the following paragraphs.

6.4.1 Reset ($\overline{RESET}$)

$\overline{RESET}$ is asserted by external logic to force the CPU to a known state. A description of the reset sequence is given in Section 7.3.

6.4.2 Halt ($\overline{HALT}$)

$\overline{HALT}$ is asserted by external logic to halt the execution of macroinstructions by the CPU. When HALT is asserted the CPU will:

1. Execute an external processor write cycle at the conclusion of the current macroinstruction. During this cycle, CS<1:0> = 10 (non-FPU command) and DAL<05:00> = 111111.
2. The CPU will enter the restart process with a restart code = 2 ($\overline{HALT}$ asserted), see Section 2.8.

$\overline{HALT}$ is an edge sensitive signal that is sampled every microcycle, is synchronized internally, and generates a non-maskable interrupt. $\overline{HALT}$ must be asserted an minimum of two microcycles to guarantee it is sampled. $\overline{HALT}$ must be deasserted a minimum of two microcycles before another halt request will be recognized.

6.4.3 Control Status (CS<2:0>)

The three control status lines are used in conjunction with $\overline{WR}$ and either $\overline{AS}$ or $\overline{EPS}$ to define the type of operation in progress for the current bus cycle. CS<2:0> are valid when $\overline{AS}$ or $\overline{EPS}$ is asserted.

During a read, write, or interrupt cycle ($\overline{AS}$ asserted), $\overline{WR}$ and CS<2:0> have the following meaning:

$\overline{WR}$	CS<2:0>	Bus Cycle Type
H	LLL	reserved
H	LLH	reserved
H	LHL	reserved
H	LHH	interrupt acknowledge
H	HLL	read (instruction)
H	HLH	read lock
H	HHL	read (data, modify intent)
H	HHH	read (data, no modify intent)
L	LLL	reserved
L	LLH	reserved
L	LHL	reserved
L	LHH	reserved
L	HLL	reserved
L	HLH	write unlock
L	HHL	reserved
L	HHH	write (data)

During an External Processor read, write, or response cycle ($\overline{EPS}$ asserted), CS<2> is always high. $\overline{WR}$ and CS<1:0> have the following meaning:

$\overline{WR}$	CS<1:0>	Bus Cycle Type
H	LL	reserved
H	LH	read data
H	HL	reserved
H	HH	response enable
L	LL	write command (FPU)
L	LH	write data
L	HL	write command (non-FPU)
L	HH	reserved

6.5 INTERRUPT CONTROL

There are 6 pins associated with interrupt control: $\overline{IRQ<3:0>}$, $\overline{PWRFL}$, and $\overline{INTTIM}$. The function of each of these pins is described in the following paragraphs.

6.5.1 Interrupt Request ($\overline{IRQ<3:0>}$)

These four lines are used by external logic to send interrupt requests to the CPU. If the interrupt request is at a higher interrupt priority level (IPL) than the current IPL of the CPU, an interrupt acknowledge bus cycle will be executed. Each line has the following interrupt priority level (IPL):

IRQ Line	Interrupt Priority Level (hex)
$\overline{IRQ<3>}$	IPL17
$\overline{IRQ<2>}$	IPL16
$\overline{IRQ<1>}$	IPL15
$\overline{IRQ<0>}$	IPL15

$\overline{IRQ<3:0>}$ are level sensitive, are sampled during every microcycle, and are synchronized internally. For a description of the interrupt handling process refer to Section 7.7.

6.5.2 Power Fail ($\overline{PWRFL}$)

$\overline{PWRFL}$ allows external logic to notify the CPU of a power fail condition. The assertion of $\overline{PWRFL}$ results in an interrupt at IPL1E that uses vector 0C (hex) in the system control block (SCB). A power fail interrupt is not acknowledged with an interrupt acknowledge bus cycle. $\overline{PWRFL}$ is edge sensitive, is sampled every microcycle, and is synchronized internally. $\overline{PWRFL}$ must be asserted an minimum of two microcycles to guarantee it is sampled. $\overline{PWRFL}$ must be deasserted a minimum of two microcycles before another halt request will be recognized. For a description of how $\overline{PWRFL}$ is used refer to Section 7.7.1.

6.5.3 Interval Timer ($\overline{INTTIM}$)

$\overline{INTTIM}$ allows external logic to notify the CPU of an interval timer rollover. The assertion of $\overline{INTTIM}$ results in an interrupt at IPL16 that uses vector C0 (hex) in the SCB. An interval timer interrupt is not acknowledged with an interrupt acknowledge bus cycle. $\overline{INTTIM}$ is edge sensitive, is sampled every microcycle, and is synchronized

internally. INTTIM must be asserted an minimum of two microcycles to guarantee it is sampled. INTTIM must be deasserted a minimum of two microcycles before another halt request will be recognized. For a description of how INTTIM is used refer to Section 7.7.2.

6.6 DMA CONTROL

There are 2 pins associated with DMA control: DMR and DMG. The function of each of these pins is described in the following paragraphs.

6.6.1 DMA Request (DMR)

DMR is asserted by external logic to notify the CPU that it would like to take control of the DAL bus and related control signals. DMR is level sensitive, is sampled every microcycle, and is internally synchronized.

6.6.2 DMA Grant (DMG)

DMG is asserted by the CPU in response to DMR. When DMG is asserted the CPU three-states DAL<31:00>, AS, WR, DS, DBE, BM<3:0>, and CS<2:0>. When external logic is finished using the bus, it deasserts AS, DS, DBE, and DMR and the CPU responds by deasserting DMG and beginning the next bus cycle.

6.7 SUPPLIES

There are 9 pins associated with power: 4 for +5 VDC (Vdd), 4 for ground (Vss), and 1 for the back bias generator (Vbb). The function of each of these pins is described in the following paragraphs.

6.7.1 Power (Vdd)

There are 4 pins called Vdd, which are used to input +5 VDC to the MicroVAX 78032 CPU. +5 VDC is supplied by external circuitry and must be maintained to within +/-5%.

6.7.2 Ground (Vss)

There are 4 pins called Vss, which provide a ground reference for the

MicroVAX 78032 CPU. These pins are connected to the ground reference for external logic.

6.7.3 Back Bias Generator (Vbb)

The Vbb pin is the output of the on chip back bias generator. This pin MUST NOT be connected.

6.8 CLOCKS

There a 2 pins associated with clock signals: CLKI and CLKO. The function of each of these pins is described in the following paragraphs.

6.8.1 Clock In (CLKI)

CLKI receives the output of a TTL oscillator, which provides basic timing to the CPU.

6.8.2 Clock Out (CLKO)

CLKO supplies a timing output at half the frequency of CLKI.

6.9 TEST (TEST)

This pin is used by chip manufacturing for internal testing of the MicroVAX 78032 CPU. For normal use this pin MUST be tied to ground.

6.10 PIN DESCRIPTION SUMMARY

Table 6-1 summarizes the function of the pins on the MicroVAX 78032 CPU.

Table 6-1 MicroVAX 78032 Pin Summary

Pin No.	Signal Name	I/O	Function
33-42 45-59 62-68	DAL<31:00>	I/O	(Time-multiplexed) During the first part CPU read cycles, CPU write cycles, interrupt acknowledge cycles, provides address information on DAL<29:02> and length information on DAL<31:30>. During During the second part of a read or interrupt acknowledge cycle, receives data driven by memory or I/O devices. During the second part of a write cycle, provides data from the MicroVAX 78032.
30	$\overline{AS}$	O	A strobe that indicates $\overline{WR}$, $\overline{BM<3:0>}$, CS<2:0>, and DAL<31:00> are valid.
29	$\overline{DS}$	O	A strobe that indicates that DAL<31:00>: -- Are free to receive data during a read cycle or interrupt cycle. It is deasserted to signal that the data has been received. -- Contain valid data during a write write cycle. It is deasserted to signal that the data is about to be removed.
12-15	$\overline{BM<3:0>}$	I/O	Specify which bytes of the DAL contain valid data. Used as an input for testing purposes only.
21	$\overline{WR}$	O	Specifies the direction of data transfer on the DAL.
28	$\overline{DBE}$	O	Asserted by the MicroVAX 78032 CPU to enable external DAL transceivers.

Table 6-1 MicroVAX 78032 Pin Summary (Continued)

Pin No.	Signal Name	I/O	Function
19	$\overline{\text{RDY}}$	I	Asserted by external logic to notify the MicroVAX 78032 CPU that: -- during a read cycle or interrupt cycle that requested data is present on the DAL -- during a write cycle that the data on the DAL has been received
18	$\overline{\text{ERR}}$	I	Asserted by external logic to indicate a bus error, e.g., non-existent memory or parity error.
16	$\overline{\text{RESET}}$	I	Asserted by external logic to initialize the MicroVAX 78032 CPU to a known initial state.
11	$\overline{\text{HALT}}$	I	A non-maskable interrupt used to halt the execution of macroinstructions.
24-26	CS<2:0>	I/O	Indicate the type of bus cycle,i.e., data access, instruction access, lock/unlock, read/modify/write, or interrupt acknowledge. CS<2> is used as an input during external processor response cycles.
3,4 6,7	$\overline{\text{IRQ<3:0>}}$	I	Four maskable interrupt request lines for device interrupts.
8	$\overline{\text{PWRFL}}$	I	A maskable interrupt used to signal a power fail condition.
9	$\overline{\text{INTTIM}}$	I	A maskable interrupt used to signal a system clock tick.
20	$\overline{\text{DMR}}$	I	Asserted by external logic to request a DMA cycle.
22	$\overline{\text{DMG}}$	O	Asserted by the MicroVAX 78032 CPU to acknowledge a DMA request.
23	$\overline{\text{EPS}}$	O	Asserted by the MicroVAX 78032 CPU to coordinate external processor transactions.

Table 6-1 MicroVAX 78032 Pin Summary (Continued)

Pin No.	Signal Name	I/O	Function
2,32, 44,61	VDD	I	+5 volt supply
1,31, 43,60	VSS	I	Ground reference
17	CLKI	I	A double frequency clock input that provides chip timing.
27	CLKO	O	Clock output at half the frequency of CLKI. Can be used as system clock.
9	VBB	O	Output of on-chip back bias generator. Must not be connected.
5	TEST	I	Reserved. Must be tied to ground.

CHAPTER 7

INTERFACING

7.1 INTRODUCTION

This chapter provides the user with some general guidelines and examples for interfacing to the MicroVAX 78032 CPU. Some of the areas covered are:

- Power
- Power-Up/Reset
- Memory Subsystem
- Bus Errors
- Interrupts

7.2 POWER

The MicroVAX 78032 CPU requires a single +5 V supply. There are 8 pins associated with the power supply, four VDD pins and four VSS pins. The VDD pins are connected to +5 V and the VSS pins are connected to ground. Decoupling and grounding with the MicroVAX 78032 CPU is very important. Decoupling the power supply is done by connecting a capacitor between each VDD pin and its associated VSS pin as shown in Figure 7-1. The recommended value of the decoupling capacitor is 10 uf Tantalum +1,-10%. The ground pins (VSS) should be tied to the common point ground for the power supply. The ground pins should be tied together at the chip.

NOTE

All VDD pins must be connected to the +5 V supply and all VSS pins must be connected to ground.

The MicroVAX 78032 CPU internally generates its own negative voltage which is brought out on the VBB pin. It is not necessary to filter this voltage, therefore the VBB pin must NOT be connected.

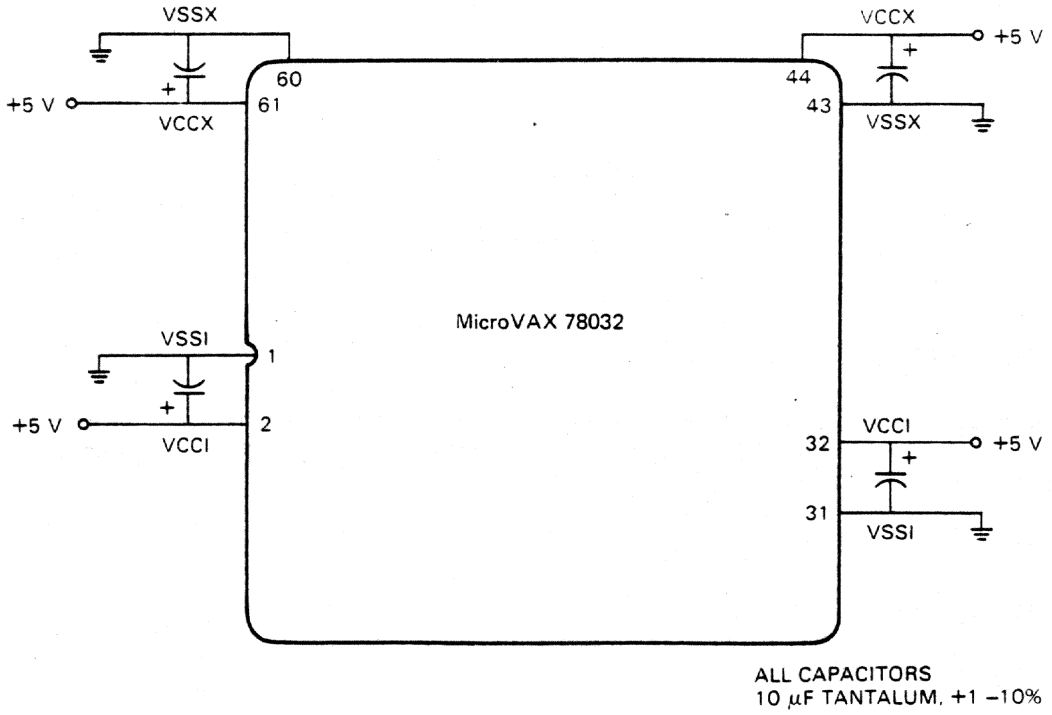


Figure 7-1 Power Supply Decoupling

7.3 RESET/POWER-UP

The MicroVAX 78032 CPU is reset at any time by pulling the $\overline{\text{RESET}}$ pin low as follows:

1. When power is first applied, the $\overline{\text{RESET}}$ pin must be held low for a minimum of 3 msec after VDD has reached a stable +4.75 V. This makes certain that all on chip voltages are stable before beginning operation.

2. $\overline{\text{RESET}}$ must be held low for a minimum of 3.0 usec if $\overline{\text{RESET}}$ is asserted after VDD has been at +4.75 V for more than 3 msec.

When $\overline{\text{RESET}}$ is asserted the processor stops executing instructions and enters the Restart Process. Refer to Section 2.8 for an explanation of the Restart Process.

During reset/power-up the MicroVAX 78032 CPU initializes its internal logic and checks to see if the optional MicroVAX 78132 FPU is present. It checks for the FPU by:

1. Performing an external processor command cycle. This synchronizes the FPU with the CPU.

2. Issuing a valid instruction to the FPU via an external processor command cycle.

3. Transferring data to the FPU using an external processor write data cycle.

4. After waiting a period of time, performing an external processor response/enable cycle. This verifies whether or not the FPU is present.

7.4 HALTING THE PROCESSOR

The MicroVAX 78032 CPU is a dynamic part and cannot be halted by disabling its CLKI input. The MicroVAX 78032 CPU is halted in one of two ways:

1. Execution of a HALT instruction in kernel mode.

2. Assertion of the $\overline{\text{HALT}}$ pin.

Either one of these actions causes the execution of macroinstructions to be suspended and the restart process to be entered. The initiation of the restart process is under control of the processor microcode. The microcode saves the processor state and passes control to user code beginning at physical address 20040000 (hex). For a more detailed explanation of the restart process refer to Section 2.8.

Assertion of the $\overline{\text{HALT}}$ pin results in the execution of a non-maskable interrupt by the CPU. $\overline{\text{HALT}}$ is edge sensitive and must be asserted for a minimum of two microcycles to guarantee its being sensed by the CPU and be deasserted for a minimum of two microcycles before another $\overline{\text{HALT}}$ will be recognized.

7.5 MEMORY SUBSYSTEM

Figure 7-2 shows an implementation of memory subsystem with 32KB of PROM and 128KB of Static RAM (SRAM). This susbsytem consists of an address latch, address decode logic, read/write control logic, $\overline{\text{RDY}}$ logic, 32KB of PROM, and 128KB of SRAM.

The longword address is latched in the LS373 transparent latches by AS. The address is decoded by the LS138 decoder. The output of the decoder is used to select the PROM or one of the four banks of SRAM. The byte(s) to be accessed within the longword are selected by $\overline{\text{BM<3:0>}}$. Note that this system does not do a unique address decode.

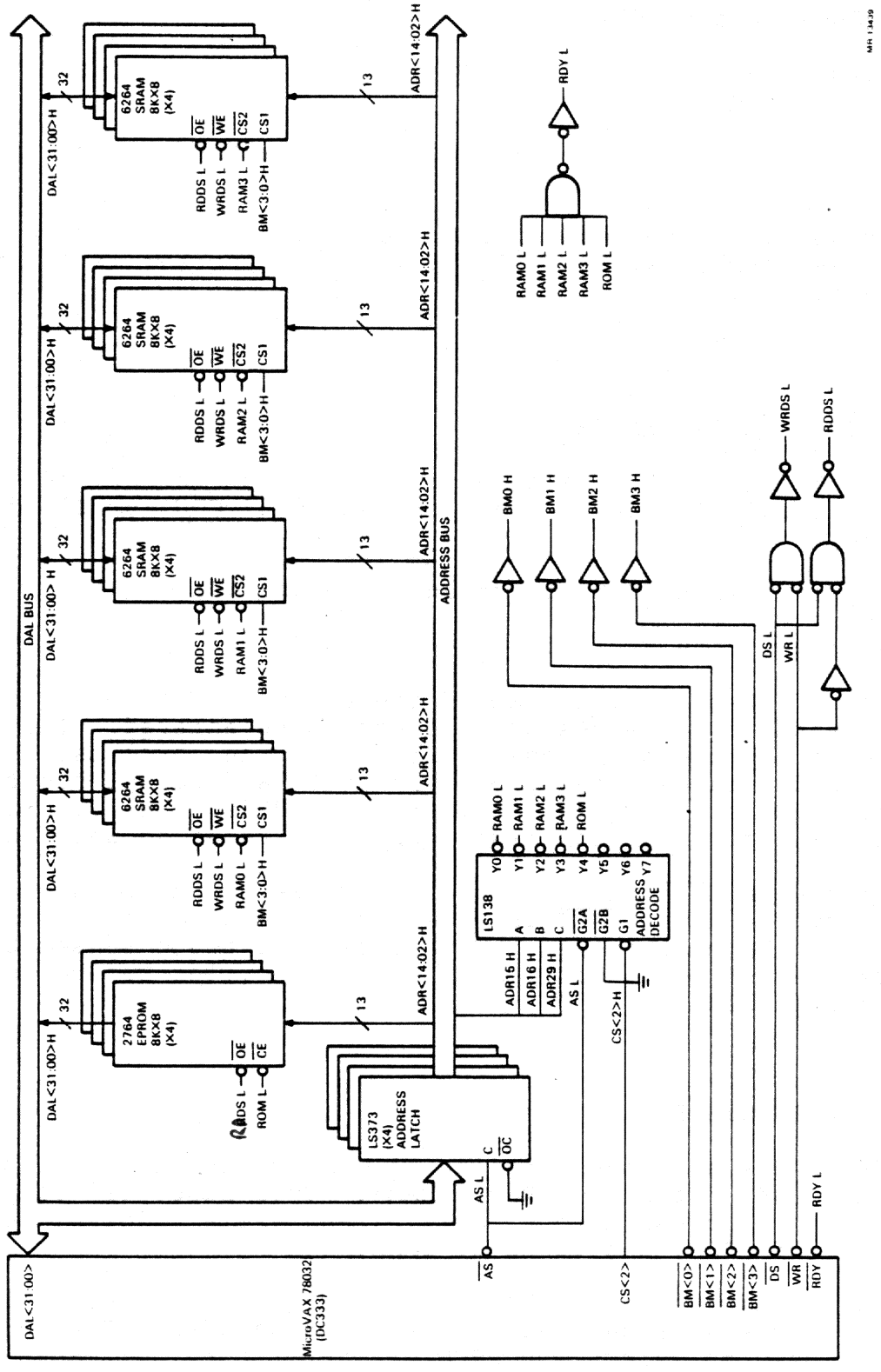


Figure 7-2 Memory Subsystem with 32KB PROM and 128KB SRAM

7.6 BUS ERRORS

Recognition of bus errors (e.g. bus timeout, parity) is implemented in external logic. When a bus error occurs, the external logic notifies the processor of the error by asserting $\overline{\text{ERR}}$. When $\overline{\text{ERR}}$ is asserted one of three things happen:

1. If the bus cycle is a CPU read or write, as determined by CS<2:0>, the current bus cycle is extended one microcycle and then the processor performs a machine check.

2. If the bus cycle is an instruction prefetch (I-stream read) prefetching is halted. When the prefetch buffer is empty, the processor will try to fetch the instruction with a data read. If an error occurs again the processor will perform a machine check.

3. If the bus cycle is an IAK cycle, the processor ends the bus cycle and ignores the interrupt.

If the assertion of $\overline{\text{ERR}}$ results in a machine check, it is up to the executing program to determine the type of error from information pushed on the stack and in external logic. Refer to Section 2.5.4.6.3 for a description of the parameters pushed on the stack for a machine check.

7.7 INTERRUPTS

The MicroVAX 78032 CPU recognizes 6 hardware interrupts. These interrupts are Powerfail, Interval Timer, and $\overline{\text{IRQ}}$<3:0>.

7.7.1 Powerfail ($\overline{\text{PWRFL}}$)

The powerfail interrupt can be used to implement a power fail routine that is located at vector 0C hex in the SCB or as a high priority interrupt (IPL1E) with an internally generated vector (0C). Because the vector is generated by the CPU, there is no external interrupt acknowledge cycle associated with this interrupt.

A powerfail interrupt is initiated by the assertion of the $\overline{\text{PWRFL}}$ pin. $\overline{\text{PWRFL}}$ is edge sensitive and must be asserted for a minimum of two microcycles to guarantee its being sensed by the CPU and be deasserted for a minimum of two microcycles before another powerfail interrupt will be recognized.

7.7.2 Interval Timer ($\overline{\text{INTTIM}}$)

The interval timer interrupt allows external logic to signal an interval timer rollover with a vector of C0 hex in the SCB. This interrupt could also be used as an IPL16 interrupt with an internally generated vector (C0). Because the vector is generated by the CPU, there is no external interrupt acknowledge cycle associated with this interrupt.

An interval timer interrupt is initiated by the assertion of the $\overline{\text{INTTIM}}$ pin. $\overline{\text{INTTIM}}$ is edge sensitive and must be asserted for a minimum of two microcycles to guarantee its being sensed by the CPU and be deasserted for a minimum of two microcycles before another interval timer interrupt will be recognized.

For compatibility with Digital's MicroVMS, ULTRIX, and VAXELN software a 100 Hz oscillator should should be used for the $\overline{\text{INTTIM}}$ input.

7.7.3 General Interrupts ($\overline{\text{IRQ<3:0>}}$)

The MicroVAX 78032 CPU has four interrupt levels for use by peripheral devices. An interrupt is requested by a device asserting one of the four interrupt lines ($\overline{\text{IRQ<3:0>}}$) of the processor. The processor will arbitrate the interrupt and then perform an interrupt acknowledge cycle to acknowledge the highest pending interrupt, if it is higher than the current IPL of the processor. The interrupting device must then provide an interrupt vector to the processor and assert $\overline{\text{RDY}}$.

When interfacing to the interrupt mechanism of the MicroVAX 78032 CPU the user has to consider the following requirements:

1. If more than 4 devices are to be used, some type of priority scheme must be implemented in external logic.

2. External logic has to decode CS<2:0>, $\overline{\text{AS}}$, and $\overline{\text{WR}}$ for an interrupt acknowledge cycle.

3. External logic has to decode the IPL level, in hex, on DAL<04:00>.

4. External logic has to supply a vector to the CPU and assert $\overline{\text{RDY}}$. This vector is an offset into the SCB for the location of the interrupt routine.

   If the vector provided by the interrupting device has DAL<00> = 1, the CPU will set its IPL to IPL17 (hex) before servicing the interrupt.

For a description of an interrupt acknowledge bus cycle, refer to Section 5.2.3.

In its simplest form the MicroVAX 78032 CPU will accept four different devices, one for each interrupt level.. To expand this capability the user must provide prioritization logic, such as a daisy chain, vectored interrupt controller, etc.

APPENDIX A

DC AND AC CHARACTERISTICS

A.1 DC CHARACTERISTICS

Absolute Maximum Ratings

Storage Temperature Range	-55 C to +125 C
Active Temperature Range	0 C to +70 C
Supply Voltage	-0.5 V to +7.0 V
Input or Output Voltage Applied	-0.5 V to +7.0 V
Maximum Power Dissipation	< 3.5 Watts

Electrical Characteristics

Specified Temperature Range	0 C to +70 C
Minimum Air Flow Over Chip	100 linear ft/min
Specified Supply Voltage Range	+4.75 V to +5.25 V
Test Conditions	Temperature = +70 C Vss = 0 V Vdd = +4.75 V (except as noted)

DC AND AC CHARACTERISTICS

Symbol	Parameter	Min	Max	Units	Test Condition
Vih	High level input voltage	2.0		V	
Vil	Low level input voltage		0.8	V	
Voh	High level output voltage	2.4		V	Ioh = - 400 uA
Vol	Low level output voltage		0.4	V	Iol = 2.0 mA
Vohe	High level output voltage ($\overline{EPS}$ only)	2.6		V	Ioh = - 100 ua
Vole	Low level output voltage ($\overline{EPS}$ only)		0.2	V	Iol = 1.0 ma
Iils	Input leakage current (CS<2>)		3.2	mA	Vin = 0.4 V
Iil	Input leakage current	-10	10	uA	0 < Vin < Vdd
Iol	Output leakage current	-10	10	uA	0.4 < Vin < Vdd
Idd	Active supply current		700	mA	Iout = 0, Ta = 0 C
Cin	Input capacitance		8	pF	

A.2 AC CHARACTERISTICS

The following notes apply to Figures A-1 through A-7 and their associated timing tables.

1. Formulas for the timing parameters are stated in terms of the CLKI period. CLKI period = tCIP = P.

2. All times are in nanoseconds except where noted.

3. AC characteristics are measured with a purely capacitive load of 100 pf. Times are valid for loads of up to 100 pf on all pins.

4. AC highs are measured at 2.0 volts and AC lows at 0.8 volts except for $\overline{EPS}$.

5. AC high for $\overline{EPS}$ is measured at 2.2 volts and AC low at 0.6 volts.

6. S = the number of slipped microcycles during a bus cycle.

7. The sampling window is used to sample the following asynchronous signals: $\overline{RDY}$, $\overline{ERR}$, and $\overline{DMR}$. $\overline{RDY}$ and $\overline{ERR}$ are qualified by $\overline{AS}$ being asserted. $\overline{DMR}$ is qualified by $\overline{AS}$ being deasserted. The effect of these signals on the current bus cycle is as follows:

   - The bus cycle will conclude at the end of the current microcycle if $\overline{RDY}$ (and NOT $\overline{ERR}$) is asserted throughout the sampling window while $\overline{AS}$ is asserted.

   - If $\overline{ERR}$ is asserted throughout the sampling window while $\overline{AS}$ is asserted, the current microcycle becomes an extension cycle and the bus cycle ends after the next microcycle.

   - If $\overline{RDY}$ or $\overline{ERR}$ go through a transition during the sampling window while $\overline{AS}$ is asserted, the result is indeterminate.

   - $\overline{DMR}$ is sampled at every microcycle boundary.

   - If $\overline{DMR}$ is asserted throughout the sampling window, and $\overline{AS}$ is not asserted, and the CPU has not locked the bus the next microcycle will be the beginning of a DMA cycle.

   - If $\overline{DMR}$ is asserted throughout the sampling window, $\overline{AS}$ is asserted, and the CPU has not locked the bus, the first microcycle after the end of the current bus cycle will be the beginning of a DMA cycle

   - A DMA cycle will conclude at the end of the current microcycle if $\overline{DMR}$ is deasserted throughout the sampling window.

A.2.1 CLKI Timing

Table A-1 CLKI Timing

SYMBOL	DEFINITION	MIN	MAX
tCIF	Clock In fall time		4.5
tCIH	Clock In high	8	
tCIL	Clock In low	8	
tCIP	Clock Period	25	250
tCIR	Clock In rise time		4.5

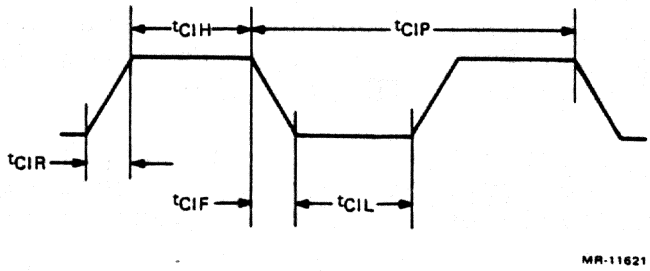


Figure A-1 CLKI Timing

A.2.2 CPU Read Cycle, CPU Write Cycle

Table A-2 CPU Read Cycle, CPU Write Cycle Timing

SYMBOL	DEFINITION	MIN	MAX	NOTES
tAAS	Address set up time to $\overline{AS}$ assertion	2P - 28		
tASA	Address hold time after $\overline{AS}$ assertion	2P - 15		
tASHC	$\overline{AS}$ rising through 2.0V to CLKO rising through 0.8V	P - 23		
tASLC	$\overline{AS}$ falling through 0.8V to CLKO rising through 0.8V	P - 20		
tASDB	$\overline{AS}$ assertion to $\overline{DBE}$ and $\overline{DS}$ (read) assertion	3P - 15	3P + 20	
tASDI	$\overline{AS}$ assertion to read data valid		11P - 30 + 8PS	1
tASDSO	$\overline{AS}$ assertion to $\overline{DS}$ assertion (write)	5P - 15	5P + 20	
tASDZ	$\overline{AS}$ and $\overline{DBE}$ deassertion to data 3-state		2P - 20	
tASHW	$\overline{AS}$ deassertion width	3P		
tASLW	$\overline{AS}$ assertion width	12P - 15 + 8PS		
tASWB	$\overline{AS}$ assertion to beginning of $\overline{RDY}$, $\overline{ERR}$, and $\overline{DMR}$ sampling window		(6P - 45) + 8PS	2
tASWE	$\overline{AS}$ assertion to end of $\overline{RDY}$, $\overline{ERR}$, and $\overline{DMR}$ sampling window	6P + 10 + 8PS		3
tASWR	$\overline{WR}$, $\overline{BM<3:0>}$, CS<2:0> hold time from $\overline{AS}$ deassertion	P - 20		
tBMAS	$\overline{BM<3:0>}$ set up time before $\overline{AS}$ assertion	2P - 25		
tCASH	CLKO rising through 2.0V to $\overline{AS}$ rising through 0.8V	P - 7	P - 15	

Table A-2 CPU Read Cycle, CPU Write Cycle Timing (Continued)

SYMBOL	DEFINITION	MIN	MAX	NOTES
tCASL	CLKO rising through 2.0V to $\overline{AS}$ falling through 2.0V	P - 9	P + 16	
tCDI	CLKO rising through 2.0V to read data valid		P - 5	
tCDO	Write data hold time from CLKO rising through 2.0V	P - 15		
tCF	CLKO fall time		12.5	
tCH	CLKO high	(2P - 25) x .5		
tCL	CLKO low	(2P - 25) x .5		
tCP	CLKO period	50	500	
tCR	CLKO rise time		12.5	4
tCWB	T4 CLKO rising through 2.0V to beginning of RDY, ERR, and DMR sampling window		3P - 45	2
tCWE	T4 CLKO rising through 0.8V to end of RDY, ERR, and DMR sampling window	3P + 15		3
tDBLW	DBE assertion width	9P - 20 + 8PS		
tDOC	Write data set-up time to CLKO rising through 0.8V	3P - 42		
tDODS	Write data set-up time to DS assertion	3P - 30		
tDSAS	DS deassertion to AS and DBE deassertion	P - 15		
tDSD	Read data hold time after DS deassertion	0		
tDSDI	DS assertion to read data valid		8P - 35 - 8PS	1

Table A-2 CPU Read Cycle, CPU Write Cycle Timing (Continued)

SYMBOL	DEFINITION	MIN	MAX	NOTES
tDSDO	Write data hold time from DS deassertion	3P - 20		
tDSDZ	DS deassertion to read data 3-state		3P - 20	
tDSHW	DS deassertion width	6P		
tDSLWI	DS assertion width (read)	8P - 20 + 8PS		
tDSLWO	DS assertion width (write)	6P - 20 + 8PS		
tWEDI	Sampling window end to read data valid		5P - 25	
tWRAS	WR, CS<2:0> set up time before AS assertion	3P - 35		

Notes:

1. Read data is valid early enough if tASDI or tDSDI or tCDI is satisfied.
2. Requirements for the beginning of the sampling window are satisfied if either tASWB or tCWB is satisfied.
3. Requirements for the end of the sampling window are satisfied if either tASWE or tCWE is satisfied.

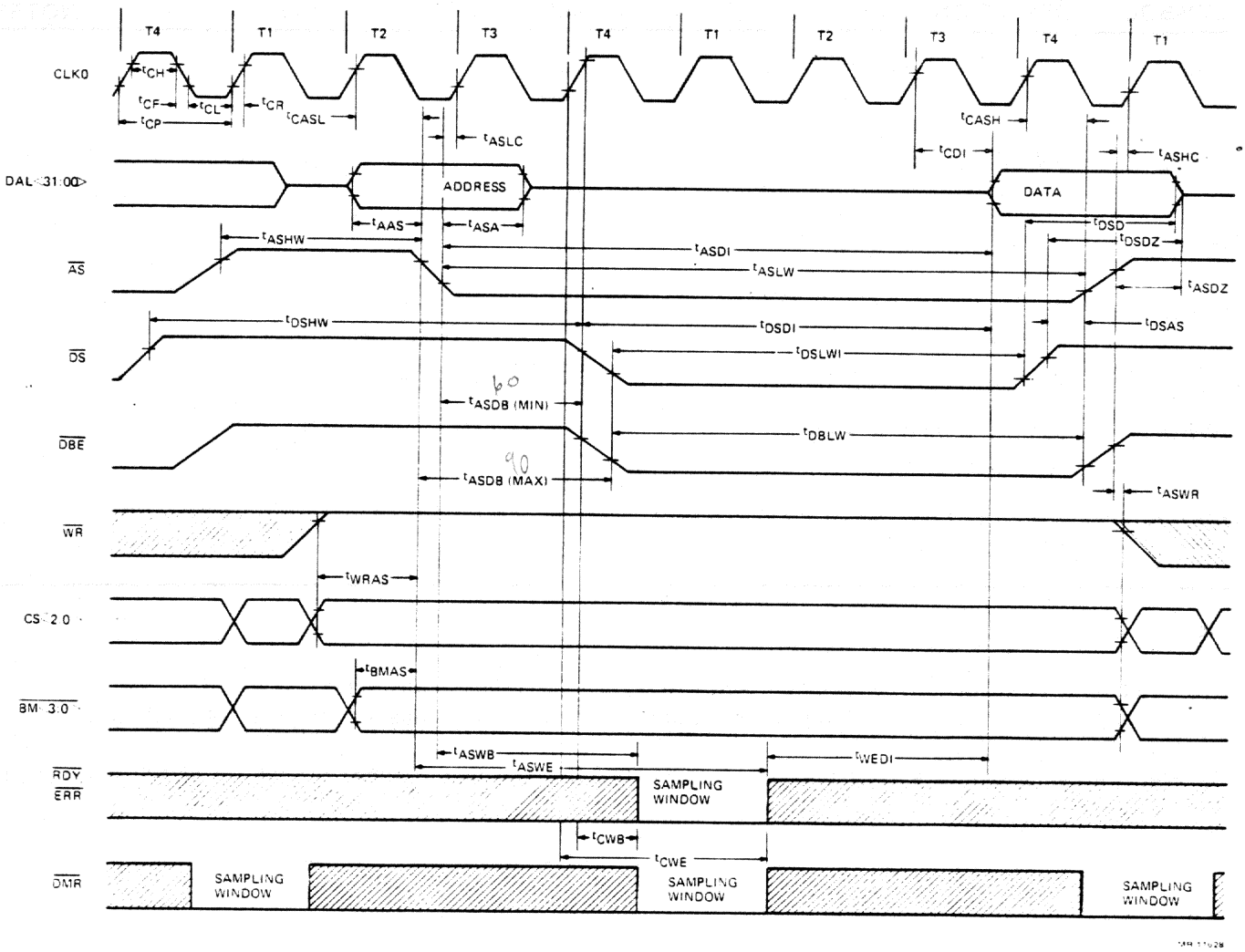


Figure A-2 CPU Read Timing

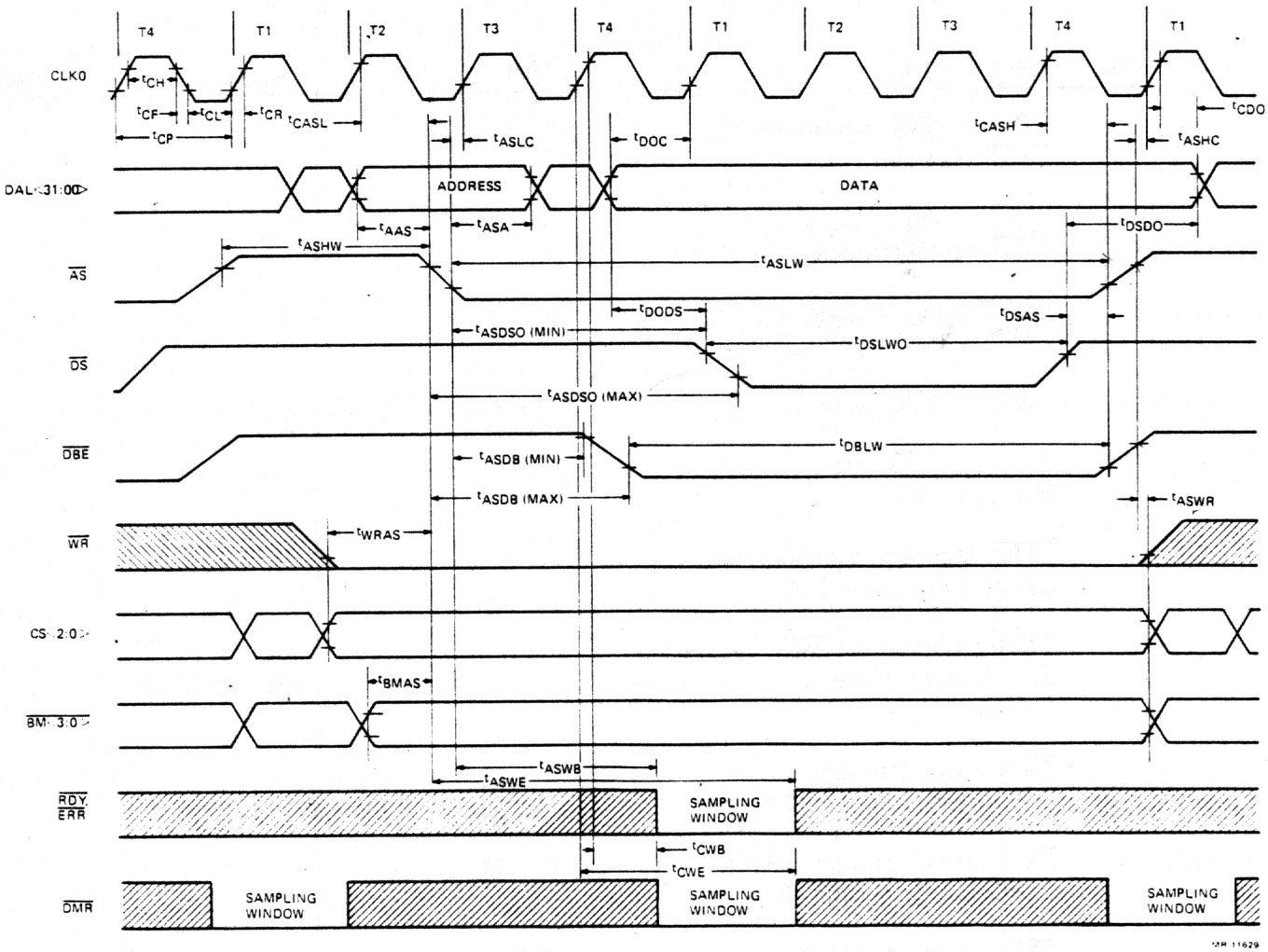


Figure A-3 CPU Write Timing

A.2.3 DMA Cycle

Table A-3 DMA Cycle Timing

SYMBOL	DEFINITION	MIN	MAX	NOTES
tASG	$\overline{AS}$ and $\overline{DBE}$ deassertion to $\overline{DMG}$ assertion	4P - 25		
tCGH	CLKO rising through 2.0V to $\overline{DMG}$ rising through 0.8V	P - 7	P + 16	
tCGL	CLKO rising through 2.0V to $\overline{DMG}$ falling through 2.0V	P - 7	P + 18	
tDMRG	$\overline{DMR}$ to $\overline{DMG}$ latency	10P - 25	60P + 20 + 16PS	
tDMRGU	$\overline{DMR}$ to $\overline{DMG}$ latency with bus unlocked	10P - 25	28P - 20 + 8PS	
tGDALZ	$\overline{DMG}$ deassertion to external device three-state of DALS.		4P - 20	
tGDMR	$\overline{DMG}$ assertion to $\overline{DMR}$ deassertion such that no more DMA cycles are requested.		6P - 45 + ((N - 2) x 8P)	1
tGHC	$\overline{DMG}$ rising through 2.0V to CLKO rising through 0.8V	P - 25		
tGLC	$\overline{DMG}$ falling through 0.8V to CLKO rising through 0.8V	P - 23		
tGLW	$\overline{DMG}$ minimum assertion width	10P - 25 + ((N - 2) x 8P)		1
tGSZ	$\overline{DMG}$ assertion to three-state of $\overline{AS}$, $\overline{DS}$, $\overline{DBE}$, $\overline{WR}$, CS<2:0>, and $\overline{BM}$<3:0>	-10	0	
tGZ	$\overline{DMG}$ deassertion to external device three-state of $\overline{AS}$, $\overline{DS}$, $\overline{DBE}$, $\overline{WR}$, CS<2:0>, and $\overline{BM}$<3:0>.		3P - 20	2

Notes:

1. N = the number of microcycles that a DMA grant lasts. A DMA grant is issued for a minimum of two microcycles.
2. At the conclusion of a DMA grant the external logic MUST deassert $\overline{AS}$, $\overline{DS}$, and $\overline{DBE}$ before the external bus drivers are put in the high impedance state.

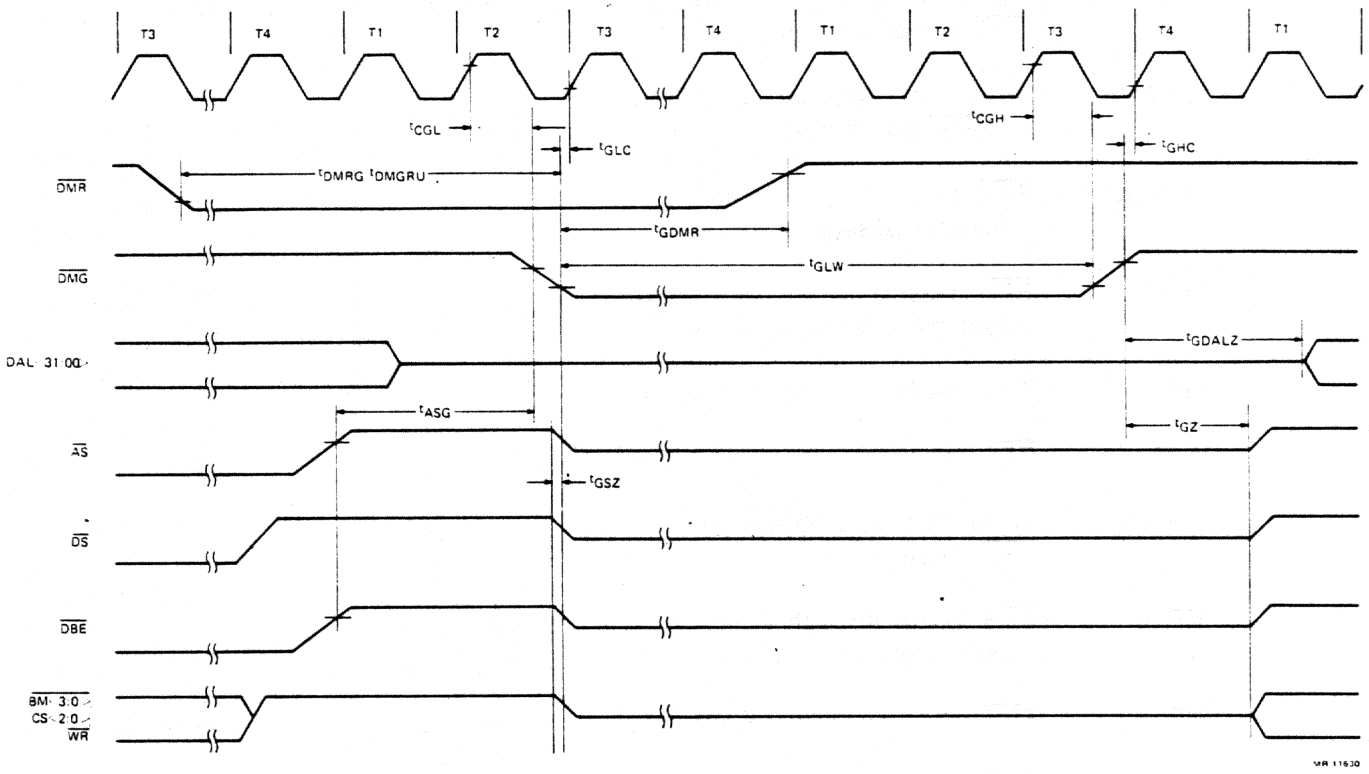


Figure A-4 DMA Timing

A.2.4 External Processor Read/Response Enable Cycle, External Processor Write/Command Cycle

Table A-4 External Processor Cycle Timing

SYMBOL	DEFINITION	MIN	MAX
tCEP	CLKO falling through 0.8V to $\overline{EPS}$ falling through 2.2V	P - 5	P + 19
tDOEPH	Write data valid set up time to $\overline{EPS}$ deassertion	2P - 35	
tEPCSL	$\overline{EPS}$ assertion to external processor assertion of CS<2>	0	3P - 40
tEPCSZ	$\overline{EPS}$ deassertion to CS<2> three-stated by external processor	0	2P - 20
tEPDI	$\overline{EPS}$ assertion to read data valid		4P - 40
tEPF	$\overline{EPS}$ fall time from 2.2V to 0.6V	0	10
tEPHDO	Write data hold time from $\overline{EPS}$ deassertion	2P - 25	
tEPLC	$\overline{EPS}$ falling through 0.6V to CLKO falling through 2.0V	P - 25	
tEPLWI	$\overline{EPS}$ assertion width (read)	4P - 20	4P + 20
tEPLWO	$\overline{EPS}$ assertion width (write)	5P - 20	5P + 20
tEPWR	$\overline{WR}$ and CS<1:0> hold time from $\overline{EPS}$ deassertion	P - 20	
tEPZ	$\overline{EPS}$ deassertion to read data three-state		3P - 20
tWREP	$\overline{WR}$ and CS<1:0> set up time before $\overline{EPS}$ assertion	2P - 35	

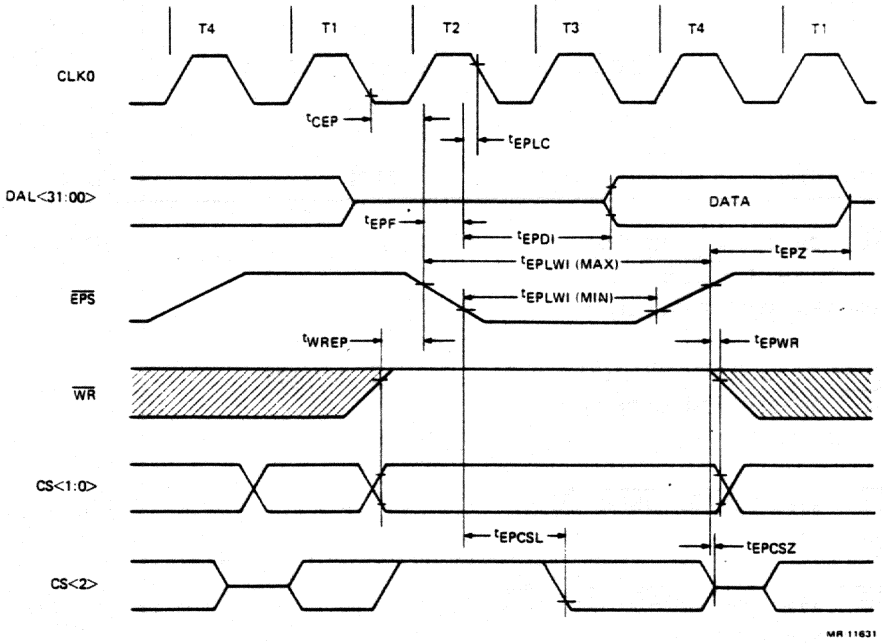


Figure A-5 External Processor Read/Response Timing

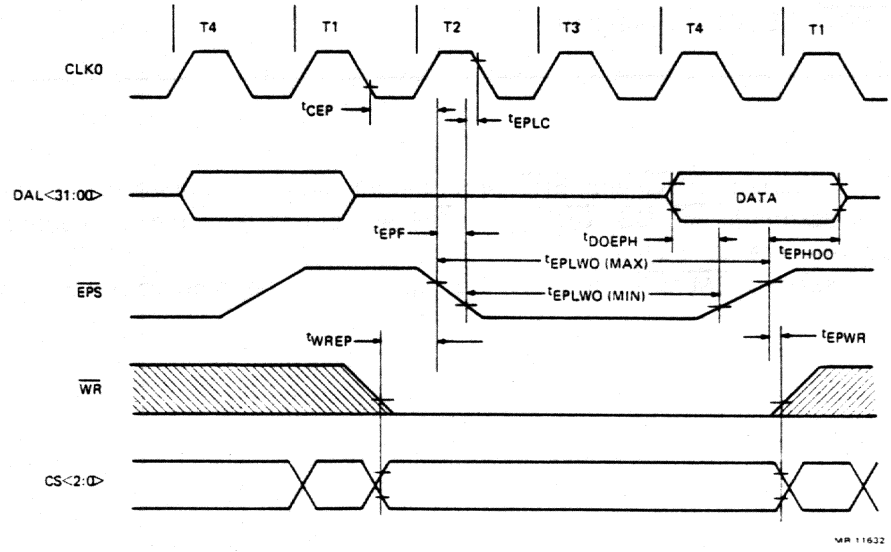


Figure A-6 External Processor Write/Command Timing

A.2.5 Reset Timing

Table A-5 Reset Timing

SYMBOL	DEFINITION	MIN	MAX	NOTES
tRES	$\overline{\text{RESET}}$ deassertion to first CLKO pulse if $\overline{\text{RESET}}$ is deasserted synchronously	3P + 10	3P + 85	
tRESC	Number of CLKO periods from $\overline{\text{RESET}}$ deassertion until first DAL activity	32 periods		
tRESGH	$\overline{\text{RESET}}$ assertion to $\overline{\text{DMG}}$, $\overline{\text{EPS}}$ deassertion		150	1
tRESH	$\overline{\text{RESET}}$ assertion to $\overline{\text{AS}}$, $\overline{\text{DS}}$, $\overline{\text{DBE}}$, $\overline{\text{WR}}$ deassertion		1.0 usec	2
tRESW	$\overline{\text{RESET}}$ assertion width after VDD = 4.75V	3.0 msec		
tRESWB	$\overline{\text{RESET}}$ assertion width if VDD has already been at 4.75V for 3 msec when $\overline{\text{RESET}}$ is asserted	3.0 usec		
tRESZ	$\overline{\text{RESET}}$ assertion to DAL<31:00>, $\overline{\text{BM}}$<3:0>, CS<2:0> three-state		100	3

Notes:

1. When $\overline{\text{RESET}}$ is asserted, $\overline{\text{DMG}}$ and $\overline{\text{EPS}}$ are brought high and held high by their ouput drivers.
2. When $\overline{\text{RESET}}$ is asserted, $\overline{\text{AS}}$, $\overline{\text{DS}}$, $\overline{\text{DBE}}$, and $\overline{\text{WR}}$ are put in the high impedance state and brought high by low current internal pull-ups.
3. Whe $\overline{\text{RESET}}$ is asserted $\overline{\text{BM}}$<3:0> and CS<2:0> are put in the high impedance state.

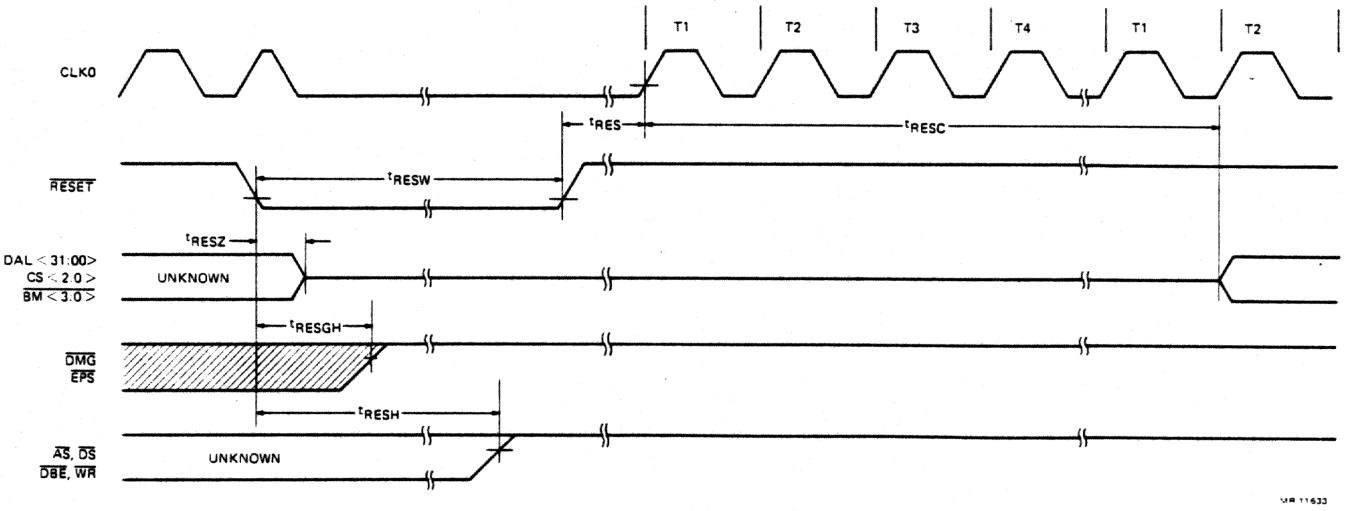


Figure A-7 Reset Timing

APPENDIX B

INSTRUCTION SET SUMMARY

B.1 INTRODUCTION

This section provides a summary of the VAX-11 instructions implemented by the MicroVAX 78032 CPU, the floating point instructions supported by the floating point unit, and the emulated instructions that are assisted by the MicroVAX 78032 CPU's microcode.

The standard notation for operand specifiers is:

<name>.<access type><data type>

where:

1. Name is a suggestive name for the operand in the context of the instruction. It is the capitalized name of a register or block for implied operands.

2. Access type is a letter denoting the operand specifier access type.

a	=	address operand
b	=	branch displacement
m	=	modified operand (both read and written)
r	=	read only operand
v	=	if not "Rn", same as a, otherwise R[n+1]'Rn
w	=	write only operand

3. Data type is a letter denoting the data type of the operand.

b	=	byte
d	=	D_floating
f	=	F_floating
g	=	G_floating
l	=	longword
q	=	quadword
v	=	field (used only in implied operands)
w	=	word
*	=	multiple longwords (used only in implied operands)

4. Implied operands, that is, locations that are accessed by the instruction, but not specified in an operand, are denoted by braces {}.

The abbreviations for condition codes are:

*	=	conditionally set/cleared
-	=	not affected
0	=	cleared
1	=	set

The abbreviations for exceptions are:

rsv	=	reserved operand fault
iov	=	integer overflow trap
idvz	=	integer divide by zero trap
fov	=	floating overflow fault
fuv	=	floating underflow fault
fdvz	=	floating divide by zero fault
dov	=	decimal overflow trap
ddvz	=	decimal divide by zero trap
sub	=	subscript range trap
prv	=	privileged instruction fault

Opcode values are given in hexadecimal.

B.2 INSTRUCTION SUMMARY

The following is a summary of the VAX-11 instructions implemented by the MicroVAX 78032 CPU.

OP	Mnemonic & Arguments	Description	N	Z	V	C	Exceptions
9D	ACBB limit.rb, add.rb, index.mb, displ.bw	Add compare and branch byte	*	*	*	-	iov
F1	ACBL limit.rl, add.rl, index.ml, displ.bw	Add compare and branch long	*	*	*	-	iov
3D	ACBW limit.rw, add.rw, index.mw, displ.bw	Add compare and branch word	*	*	*	-	iov
58	ADAWI add.rw, sum.mw	Add aligned word interlocked	*	*	*	*	iov
80	ADDB2 add.rb, sum.mb	Add byte 2-operand	*	*	*	*	iov
81	ADDB3 add1.rb, add2.rb, sum.wb	Add byte 3-operand	*	*	*	*	iov
C0	ADDL2 add.rl, sum.ml	Add long 2-operand	*	*	*	*	iov
C1	ADDL3 add1.rl, add2.rl, sum.wl	Add long 3-operand	*	*	*	*	iov
A0	ADDW2 add.rw, sum.mw	Add word 2-operand	*	*	*	*	iov
A1	ADDW3 add1.rw, add2.rw, sum.ww	Add word 3-operand	*	*	*	*	iov
D8	ADWC add.rl, sum.ml	Add with carry	*	*	*	*	iov
F3	AOBLEQ limit.rl, index.ml, displ.bb	Add one and branch on less or equal	*	*	*	-	iov
F2	AOBLSS limit.rl, index.ml, displ.bb	Add one and branch on less	*	*	*	-	iov
78	ASHL cnt.rb, src.rl, dst.wl	Arithmetic shift left	*	*	*	0	iov
79	ASHQ cnt.rb, src.rq, dst.wq	Arithmetic shift quad	*	*	*	0	iov
E1	BBC pos.rl, base.vb, displ.bb, {field.rv}	Branch on bit clear	-	-	-	-	rsv
E5	BBCC pos.rl, base.vb, displ.bb, {field.mv}	Branch on bit clear and clear	-	-	-	-	rsv
E7	BBCCI pos.rl, base.vb, displ.bb, {field.mv}	Branch on bit clear and clear interlocked	-	-	-	-	rsv
E3	BBCS pos.rl, base.vb, displ.bb, {field.mv}	Branch on bit clear and set	-	-	-	-	rsv
E0	BBS pos.rl, base.vb, displ.bb, {field.rv}	Branch on bit set	-	-	-	-	rsv
E4	BBSC pos.rl, base.vb, displ.bb, {field.mv}	Branch on bit set and clear	-	-	-	-	rsv
E2	BBSS pos.rl, base.vb, displ.bb, {field.mv}	Branch on bit set and set	-	-	-	-	rsv
E6	BBSSI pos.rl, base.vb, displ.bb, {field.mv}	Branch on bit set and set interlocked	-	-	-	-	rsv
1E	BCC{=BGEQU} displ.bb	Branch on carry clear	-	-	-	-	
1F	BCS{=BLSSU} displ.bb	Branch on carry set	-	-	-	-	
13	BEQL{=BEQLU} displ.bb	Branch on equal	-	-	-	-	
18	BGEQ displ.bb	Branch on greater or equal	-	-	-	-	
14	BGTR displ.bb	Branch on greater	-	-	-	-	
1A	BGTRU displ.bb	Branch on greater unsigned	-	-	-	-	
8A	BICB2 mask.rb, dst.mb	Bit clear byte 2-operand	*	*	0	-	
8B	BICB3 mask.rb, src.rb, dst.wb	Bit clear byte 3-operand	*	*	0	-	
CA	BICL2 mask.rl, dst.ml	Bit clear long 2-operand	*	*	0	-	
CB	BICL3 mask.rl, src.rl, dst.wl	Bit clear long 3-operand	*	*	0	-	

OP	Mnemonic & Arguments	Description	N	Z	V	C	Exceptions
B9	BICPSW mask.rw	Bit clear processor status word	*	*	*	*	rsv
AA	BICW2 mask.rw, dst.mw	Bit clear word 2-operand	*	*	0	-	
AB	BICW3 mask.rw, src.rw, dst.ww	bit clear word 3-operand	*	*	0	-	
88	BISB2 mask.rb, dst.mb	Bit set byte 2-operand	*	*	0	-	
89	BISB3 mask.rb, src.rb, dst.wb	Bit set byte 3-operand	*	*	0	-	
C8	BISL2 mask.rl, dst.ml	Bit set long 2-operand	*	*	0	-	
C9	BISL3 mask.rl, src.rl, dst.wl	Bit set long 3-operand	*	*	0	-	
B8	BISPSW mask.rw	Bit set processor status word	*	*	*	*	rsv
A8	BISW2 mask.rw, dst.mw	Bit set word 2-operand	*	*	0	-	
A9	BISW3 mask.rw, src.rw, dst.ww	Bit set word 3-operand	*	*	0	-	
93	BITB mask.rb, src.rb	Bit test byte	*	*	0	-	
D3	BITL mask.rl, src.rl	Bit test long	*	*	0	-	
B3	BITW mask.rw, src.rw	Bit test word	*	*	0	-	
E9	BLBC src.rl, displ.bb	Branch on low bit clear	-	-	-	-	
E8	BLBS src.rl, displ.bb	Branch on low bit set	-	-	-	-	
15	BLEQ displ.bb	Branch on less or equal	-	-	-	-	
1B	BLEQU displ.bb	Branch on less or equal unsigned	-	-	-	-	
19	BLSS displ.bb	Branch on less	-	-	-	-	
12	BNEQ{=BNEQU} displ.bb	Branch on not equal	-	-	-	-	
03	BPT {-(KSP).w*}	Break point fault	0	0	0	0	
11	BRB displ.bb	Branch with byte displacement	-	-	-	-	
31	BRW displ.bw	Branch with word displacement	-	-	-	-	
10	BSBB displ.bb, {-(SP).wl}	Branch to subroutine with byte displacement	-	-	-	-	
30	BSBW displ.bw, {-(SP).wl}	Branch to subroutine with word displacement	-	-	-	-	
1C	BVC displ.bb	Branch on overflow clear	-	-	-	-	
1D	BVS displ.bb	Branch on overflow set	-	-	-	-	
FA	CALLG arglist.ab, dst.ab, {-(SP).w*}	Call with general argument list	0	0	0	0	rsv
FB	CALLS numarg.rl, dst.ab, {-(SP).w*}	Call with argument list on stack	0	0	0	0	rsv
8F	CASEB selector.rb, base.rb, limit.rb, displ.bw-list	Case byte	*	*	0	*	
CF	CASEL selector.rl, base.rl, limit.rl, displ.bw-list	Case long	*	*	0	*	
AF	CASEW selector.rw, base.rw, limit.rw, displ.bw-list	Case word	*	*	0	*	
BD	CHME param.rw, {-(ySP).w*}	Change mode to executive	0	0	0	0	
BC	CHMK param.rw, {-(ySP).w*}	Change mode to kernel	0	0	0	0	
BE	CHMS param.rw, {-(ySP).w*}	Change mode to supervisor	0	0	0	0	
BF	CHMU param.rw, {-(ySP).w*} Where y = MINU(x,PSL<current_mode>)	Change mode to user	0	0	0	0	
94	CLRB dst.wb	Clear byte	0	1	0	-	
D4	CLRL dst.wl	Clear long	0	1	0	-	
7C	CLRQ dst.wq	Clear quad	0	1	0	-	
B4	CLRW dst.ww	Clear word	0	1	0	-	

OP	Mnemonic & Arguments	Description	N	Z	V	C	Exceptions
91	CMPB src1.rb, src2.rb	Compare byte	*	*	0	*	
D1	CMPL src1.rl, src2.rl	Compare long	*	*	0	*	
EC	CMPV pos.rl, size.rb, base.vb, {field.rv}, src.rl	Compare field	*	*	0	*	rsv
B1	CMPW src1.rw, src2.rw	Compare word	*	*	0	*	
ED	CMPZV pos.rl, size.rb, base.vb, {field.rv}, src.rl	Compare zero-extended field	*	*	0	*	rsv
98	CVTBL src.rb, dst.wl	Convert byte to long	*	*	0	0	
99	CVTBW src.rb, dst.wl	Convert byte to word	*	*	0	0	
F6	CVTLB src.rl, dst.wb	Convert long to byte	*	*	*	0	iov
F7	CVTLW src.rl, dst.ww	Convert long to word	*	*	*	0	iov
33	CVTWB src.rw, dst.wb	Convert word to byte	*	*	*	0	iov
32	CVTWL src.rw, dst.wl	Convert word to long	*	*	0	0	
97	DECB dif.mb	Decrement byte	*	*	*	*	iov
D7	DECL dif.ml	Decrement long	*	*	*	*	iov
B7	DECW dif.mw	Decrement word	*	*	*	*	iov
86	DIVB2 divr.rb, quo.mb	Divide byte 2-operand	*	*	*	0	iov, idvz
87	DIVB3 divr.rb, divd.rb, quo.wb	Divide byte 3-operand	*	*	*	0	iov, idvz
C6	DIVL2 divr.rl, quo.ml	Divide long 2-operand	*	*	*	0	iov, idvz
C7	DIVL3 divr.rl, divd.rl, quo.wl	Divide long 3-operand	*	*	*	0	iov, idvz
A6	DIVW2 divr.rw, quo.mw	Divide word 2-operand	*	*	*	0	iov, idvz
A7	DIVW3 divr.rw, divd.rw, quo.ww	Divide word 3-operand	*	*	*	0	iov, idvz
7B	EDIV divr.rl, divd.rq, quo.wl, rem.wl	Extended divide	*	*	*	0	iov, idvz
7A	EMUL mulr.rl, muld.rl, add.rl,prod.wq	Extended multiply	*	*	0	0	
EE	EXTV pos.rl, size.rb, base.vb, {field.rv}, dst.wl	Extract field	*	*	0	-	rsv
EF	EXTZV pos.rl, size.rb, base.vb, {field.rv}, dst.wl	Extract zero-extended field	*	*	0	-	rsv
EB	FFC startpos.rl, size.rb, base.vb, {field.rv}, findpos.wl	Find first clear bit	0	*	0	0	rsv
EA	FFS startpos.rl, size.rb, base.vb, {field.rv}, findpos.wl	Find first set bit	0	*	0	0	rsv
00	HALT {-(KSP).w*}	Halt (kernel mode only)	-	-	-	-	prv
96	INCB sum.mb	Increment byte	*	*	*	*	iov
D6	INCL sum.ml	Increment long	*	*	*	*	iov
B6	INCW sum.mw	Increment word	*	*	*	*	iov
0A	INDEX subscript.rl, low.rl, high.rl, size.rl, indexin.rl, indexout.wl	Index calculation	*	*	0	0	sub
5C	INSQHI entry.ab, header.aq	Insert at head of queue, interlocked	0	*	0	*	rsv
5D	INSQTI entry.ab, header.aq	Insert at tail of queue, interlocked	0	*	0	*	rsv
0E	INSQUE entry.ab, pred.ab	Insert into queue	*	*	0	*	
F0	INSV src.rl, pos.rl, size.rb, base.vb, {field.wv}	Insert field	-	-	-	-	rsv
17	JMP dst.ab	Jump	-	-	-	-	
16	JSB dst.ab, {-(SP).wl}	Jump to subroutine	-	-	-	-	

OP	Mnemonic & Arguments	Description	N	Z	V	C	Exceptions
06	LDPCTX {PCB.r*, -(KSP).w*}	Load process context (kernel mode only)	-	-	-	-	rsv, prv
92	MCOMB src.rb, dst.wb	Move complemented byte	*	*	0	-	
D2	MCOML src.rl, dst.wl	Move complemented long	*	*	0	-	
B2	MCOMW src.rw, dst.ww	Move complemented word	*	*	0	-	
DB	MFPR procreg.rl, dst.wl	Move from processor register (kernel mode only)	*	*	0	-	rsv, prv
8E	MNEGB src.rb, dst.wb	Move negated byte	*	*	*	*	iov
CE	MNEGL src.rl, dst.wl	Move negated long	*	*	*	*	iov
AE	MNEGW src.rw, dst.ww	Move negated word	*	*	*	*	iov
9E	MOVAB src.ab, dst.wl	Move address of byte	*	*	0	-	
DE	MOVAL{=F} src.al, dst.wl	Move address of long	*	*	0	-	
7E	MOVAQ{=D=G} src.aq, dst.wl	Move address of quad	*	*	0	-	
3E	MOVAW src.aw, dst.wl	Move address of word	*	*	0	-	
90	MOVB src.rb, dst.wb	Move byte	*	*	0	-	
28	MOVC3 len.rw, srcaddr.ab, dstaddr.ab, {R0-5.wl}	Move character 3-operand	0	1	0	0	
2C	MOVC5 srclen.rw, srcaddr.ab, fill.rb, dstlen.rw, dstaddr.ab, {R0-5.wl}	Move character 5-operand	*	*	0	*	
D0	MOVL src.rl, dst.wl	Move long	*	*	0	-	
DC	MOVPSL dst.wl	Move processor status longword	-	-	-	-	
7D	MOVQ src.rq, dst.wq	Move quad	*	*	0	-	
B0	MOVW src.rw, dst.ww	Move word	*	*	0	-	
9B	MOVZBL src.rb, dst.wl	Move zero-extended byte to long	0	*	0	-	
9A	MOVZBW src.rb, dst.wb	Move zero-extended byte to word	0	*	0	-	
3C	MOVZWL src.rw, dst.ww	Move zero-extended word to long	0	*	0	-	
DA	MTPR src.rl, procreg.rl	Move to processor register (kernel mode only)	*	*	0	-	rsv, prv
84	MULB2 mulr.rb, prod.mb	Multiply byte 2-operand	*	*	*	0	iov
85	MULB3 mulr.rb, muld.rb, prod.wb	Multiply byte 3-operand	*	*	*	0	iov
C4	MULL2 mulr.rl, prod.ml	Multiply long 2-operand	*	*	*	0	iov
C5	MULL3 mulr.rl, muld.rl, prod.wl	Multiply long 3-operand	*	*	*	0	iov
A4	MULW2 mulr.rw, prod.mw	Multiply word 2-operand	*	*	*	0	iov
A5	MULW3 mulr.rw, muld.rw, prod.ww	Multiply word 3-operand	*	*	*	0	iov
01	NOP	No operation	-	-	-	-	
BA	POPR mask.rw, {(SP)+.r*}	Pop registers	-	-	-	-	
0C	PROBER mode.rb, len.rw, base.ab	Probe read access	0	*	0	-	
0D	PROBEW mode.rb, len.rw, base.ab	Probe write access	0	*	0	-	
9F	PUSHAB src.ab, {-(SP).wl}	Push address of byte	*	*	0	-	
DF	PUSHAL{=F} src.al, {-(SP).wl}	Push address of long	*	*	0	-	
7F	PUSHAQ{=D=G} src.aq, {-(SP).wl}	Push address of quad	*	*	0	-	
3F	PUSHAW src.aw, {-(SP).wl}	Push address of word	*	*	0	-	
DD	PUSHL src.rl,	Push long	*	*	0	-	
BB	PUSHR mask.rw, {-(SP).w*}	Push registers	-	-	-	-	
02	REI {(SP)+.r*}	Return form exception or interrupt	*	*	*	*	rsv

OP	Mnemonic & Arguments	Description	N	Z	V	C	Exceptions
5E	REMQHI header.aq, addr.wl	Remove from head of queue, interlocked	0	*	*	*	rsv
5F	REMQTI header.aq, addr.wl	Remove from tail of queue, interlocked	0	*	*	*	rsv
0F	REMQUE entry.ab, addr.wl	Remove from queue	*	*	*	*	
04	RET {(SP)+.r*}	Return form procedure	*	*	*	*	rsv
9C	ROTL cnt.rb, src.rl, dst.wl	Rotate long	*	*	0	-	
05	RSB {(SP)+.rl}	Return from subroutine	-	-	-	-	
D9	SBWC sub.rl, dif.ml	Subtract with carry	*	*	*	*	iov.
F4	SOBGEQ index.ml, displ.bb	Subtract one and branch on greater or equal	*	*	*	-	iov
F5	SOBGTR index.ml, displ.bb	Subtract one and branch on greater	*	*	*	-	iov
82	SUBB2 sub.rb, dif.mb	Subtract byte 2-operand	*	*	*	*	iov
83	SUBB3 sub.rb, min.rb, dif.wb	Subtract byte 3-operand	*	*	*	*	iov
C2	SUBL2 sub.rl, dif.ml	Subtract long 2-operand	*	*	*	*	iov
C3	SUBL3 sub.rl, min.rl, dif.wl	Subtract long 3-operand	*	*	*	*	iov
A2	SUBW2 sub.rw, dif.mw	Subtract word 2-operand	*	*	*	*	iov
A3	SUBW3 sub.rw, min.rw, dif.ww	Subtract word 3-operand	*	*	*	*	iov
07	SVPCTX {(SP)+.r*, PCB.w*}	Save process context (kernel mode only)	-	-	-	-	prv
95	TSTB src.rb	Test byte	*	*	0	0	
D5	TSTL src.rl	Test long	*	*	0	0	
B5	TSTW src.rw	Test word	*	*	0	0	
FC	XFC {unspecified operands}	Extended function call	0	0	0	0	
8C	XORB2 mask.rb, dst.mb	Exclusive or byte 2-operand	*	*	0	-	
8D	XORB3 mask.rb, src.rb, dst.wb	Exclusive or byte 3-operand	*	*	0	-	
CC	XORL2 mask.rl, dst.ml	Exclusive or long 2-operand	*	*	0	-	
CD	XORL3 mask.rl, src.rl, dst.wl	Exclusive or long 3-operand	*	*	0	-	
AC	XORW2 mask.rw, dst.mw	Exclusive or word 2-operand	*	*	0	-	
AD	XORW3 mask.rw, src.rw, dst.ww	Exclusive or word 3-operand	*	*	0	-	

B.3 FLOATING POINT INSTRUCTION SUMMARY

These instructions are implemented in hardware if the optional MicroVAX 78132 Floating Point Unit (FPU) is present in the system. If the MicroVAX 78132 FPU is not present in the system a reserved operand fault will be taken and system software may emulate these instructions.

OP	Mnemonic & Arguments	Description	N	Z	V	C	Exceptions
6F	ACBD limit.rd, add.rd, index.rd	Add compare and branch D_floating	*	*	0	-	rsv, fov, fuv
4F	ACBF limit.rf, add.rf, index.rf	Add compare and branch F_floating	*	*	0	-	rsv, fov, fuv
4FFD	ACBG limit.rg, add.rg, index.rg	Add compare and branch G_floating	*	*	0	-	rsv, fov, fuv
60	ADDD2 add.rd, sum.md	Add D_floating 2-operand	*	*	0	0	rsv, fov, fuv
61	ADDD3 add1.rd, add2.rd, sum.wd	Add D_floating 3-operand	*	*	0	0	rsv, fov, fuv
40	ADDF2 add.rf, sum.mf	Add F_floating 2-operand	*	*	0	0	rsv, fov, fuv
41	ADDF3 add1.rf, add2.rf, sum.wf	Add F_floating 3-operand	*	*	0	0	rsv, fov, fuv
40FD	ADDG2 add.rg, sum.mg	Add G_floating 2-operand	*	*	0	0	rsv, fov, fuv
41FD	ADDG3 add1.rg, add2.rg, sum.wg	Add G_floating 3-operand	*	*	0	0	rsv, fov, fuv
71	CMPD src1.rd, src2.rd	Compare D_floating	*	*	0	0	rsv
51	CMPF src1.rf, src2.rf	Compare F_floating	*	*	0	0	rsv
51FD	CMPG src1.rg, src2.rg	Compare G_floating	*	*	0	0	rsv
6C	CVTBD src.rb, dst.wd	Convert byte to D_floating	*	*	0	0	
4C	CVTBF src.rb, dst.wf	Convert byte to F_floating	*	*	0	0	
4CFD	CVTBG src.rb, dst.wg	Convert byte to G_floating	*	*	0	0	
68	CVTDB src.rd, dst.wb	Convert D_floating to byte	*	*	*	0	rsv, iov
76	CVTDF src.rd, dst.wf	Convert D_floating to F_floating	*	*	0	0	rsv, fov
6A	CVTDL src.rd, dst.wl	Convert D_floating to long	*	*	*	0	rsv, iov
69	CVTDW src.rd, dst.ww	Convert D_floating to word	*	*	*	0	rsv, iov
48	CVTFB src.rf, dst.wb	Convert F_floating to byte	*	*	*	0	rsv, iov
56	CVTFD src.rf, dst.wd	Convert F_floating to D_floating	*	*	0	0	rsv
99FD	CVTFG src.rf, dst.wg	Convert F_floating to G_floating	*	*	0	0	rsv
4A	CVTFL src.rf, dst.wl	Convert F_floating to to long	*	*	*	0	rsv, iov
49	CVTFW src.rf, dst.ww	Convert F_floating to word	*	*	*	0	rsv, iov
48FD	CVTGB src.rg, dst.wb	Convert G_floating to byte	*	*	*	0	rsv, iov
33FD	CVTGF src.rg, dst.wf	Convert G_floating to F_floating	*	*	0	0	rsv, fov, fuv
4AFD	CVTGL src.rg, dst.wl	Convert G_floating to to long	*	*	*	0	rsv, iov
49FD	CVTGW src.rg, dst.ww	Convert G_floating to to word	*	*	*	0	rsv, iov
6E	CVTLD src.rl, dst.wd	Convert long to D_floating	*	*	0	0	
4E	CVTLF src.rl, dst.wf	Convert long to F_floating	*	*	0	0	
4EFD	CVTLG src.rl, dst.wg	Convert long to G_floating	*	*	0	0	
6D	CVTWD src.rw, dst.wd	Convert word to D_floating	*	*	0	0	
4D	CVTWF src.rw, dst.wf	Convert word to F_floating	*	*	0	0	
4DFD	CVTWG src.rw, dst.wg	Convert word to G_floating	*	*	0	0	
6B	CVTRDL src.rd, dst.wl	Convert rounded D_floating to long	*	*	*	0	rsv, iov
4B	CVTRFL src.rf, dst.wl	Convert rounded F_floating to long	*	*	*	0	rsv, iov
4BFD	CVTRGL src.rg, dst.wl	Convert rounded G_floating to long	*	*	*	0	rsv, iov
66	DIVD2 divr.rd, quo.md	Divide D_floating 2-operand	*	*	0	0	rsv, fov, fuv, fdvz
67	DIVD3 divr.rd, divd.rd, quo.wd	Divide D_floating 3-operand	*	*	0	0	rsv, fov, fuv, fdvz
46	DIVF2 divr.rf, quo.mf	Divide F_floating 2-operand	*	*	0	0	rsv, fov, fuv, fdvz
47	DIVF3 divr.rf, divd.rf, quo.wf	Divide F_floating 3-operand	*	*	0	0	rsv, fov, fuv, fdvz
46FD	DIVG2 divr.rg, quo.mg	Divide G_floating 2-operand	*	*	0	0	rsv, fov, fuv, fdvz
47FD	DIVG3 divr.rg, divd.rg, quo.wg	Divide G_floating 3-operand	*	*	0	0	rsv, fov, fuv, fdvz
74	EMODD mulr.rd, mulrx.rb, muld.rd, int.wl, fract.wd	Extended modulus D_floating	*	*	*	0	rsv, fov, fuv, iov

OP	Mnemonic & Arguments	Description	N	Z	V	C	Exceptions
54	EMODF mulr.rf, mulrx.rb, muld.rf, int.wl, fract.wf	Extended modulus F_floating	*	*	*	0	rsv, fov, fuv, iov
54FD	EMODG mulr.rg, mulrx.rw, muld.rg, int.wl, fract.wg	Extended modulus G_floating	*	*	*	0	rsv, fov, fuv, iov
72	MNEGD src.rd, dst.wd	Move negated D_floating	*	*	0	0	rsv
52	MNEGF src.rf, dst.wf	Move negated F_floating	*	*	0	0	rsv
52FD	MNEGG src.rg, dst.wg	Move negated G_floating	*	*	0	0	rsv
70	MOVD src.rd, dst.wd	Move D_floating	*	*	0	-	rsv
50	MOVF src.rf, dst.wf	Move F_floating	*	*	0	-	rsv
50FD	MOVG src.rg, dst.wg	Move G_floating	*	*	0	-	rsv
64	MULD2 mulr.rd, prod.md	Multiply D_floating 2-operand	*	*	0	0	rsv, fov, fuv
65	MULD3 mulr.rd, muld.rd, prod.wd	Multiply D_floating 3-operand	*	*	0	0	rsv, fov, fuv
44	MULF2 mulr.rf, prod.mf	Multiply F_floating 2-operand	*	*	0	0	rsv, fov, fuv
45	MULF3 mulr.rf, muld.rf, prod.wf	Multiply F_floating 3-operand	*	*	0	0	rsv, fov, fuv
44FD	MULG2 mulr.rg, prod.mg	Multiply G_floating 2-operand	*	*	0	0	rsv, fov, fuv
45FD	MULG3 mulr.rg, muld.rg, prod.wg	Multiply G_floating 3-operand	*	*	0	0	rsv, fov, fuv
75	POLYD arg.rd, degree.rw, table.ab	Evaluate polynomial D_floating	*	*	0	0	rsv, fov, fuv
55	POLYF arg.rf, degree.rw, table.ab	Evaluate polynomial F_floating	*	*	0	0	rsv, fov, fuv
55FD	POLYG arg.rf, degree.rw, table.ab	Evaluate polynomial G_floating	*	*	0	0	rsv, fov, fuv
62	SUBD2 sub.rd, dif.md	Subtract D_floating 2-operand	*	*	0	0	rsv, fov, fuv
63	SUBD3 sub.rd, min.rd, dif.wd	Subtract D_floating 3-operand	*	*	0	0	rsv, fov, fuv
42	SUBF2 sub.rf, dif.mf	Subtract F_floating 2-operand	*	*	0	0	rsv, fov, fuv
43	SUBF3 sub.rf, min.rf, dif.wf	Subtract F_floating 3-operand	*	*	0	0	rsv, fov, fuv
42FD	SUBG2 sub.rg, dif.mg	Subtract G_floating 2-operand	*	*	0	0	rsv, fov, fuv
43FD	SUBG3 sub.rg, min.rg, dif.wg	Subtract G_floating 3-operand	*	*	0	0	rsv, fov, fuv
73	TSTD src.rd	Test D_floating	*	*	0	0	rsv
53	TSTF src.rf	Test F_floating	*	*	0	0	rsv
53FD	TSTG src.rg	Test G_floating	*	*	0	0	rsv

B.4 EMULATED INSTRUCTION WITH MICROCODE ASSIST SUMMARY

The MicroVAX 78032 CPU provides microcode assistance for the emulation of these instructions by system software.

OP	Mnemonic & Arguments	Description	N	Z	V	C	Exceptions
20	ADDP4 addlen.rw, addaddr.ab, sumlen.rw, sumaddr.ab	Add packed 4-operand	*	*	*	0	rsv, dov
21	ADDP6 add1len.rw, add1addr.ab, add2len.rw, add2addr.ab, sumlen.rw, sumaddr.ab	Add packed 6-operand	*	*	*	0	rsv, dov
F8	ASHP cnt.rb, srclen.rw, srcaddr.ab, round.rb, dstlen.rw, dstaddr.ab	Arithmetic shift and round packed	*	*	*	0	rsv, dov
29	CMPC3 len.rw, src1addr.ab, src2addr.ab	Compare character 3-operand	*	*	0	*	
2D	CMPC5 src1len.rw, src1addr.ab, fill.rb, src2len.rw, src2addr.ab	Compare character 5-operand	*	*	0	*	
35	CMPP3 len.rw, src1addr.ab, src2addr.ab	Compare packed 3-operand	*	*	0	0	
37	CMPP4 src1len.rw, src1addr.ab, src2len.rw, src2addr.ab	Compare packed 4-operand	*	*	0	0	
0B	CRC tbl.ab, inicrc.rl, strlen.rw, stream.ab	Calculate cyclic redundancy check	*	*	0	0	
F9	CVTLP src.rl, dstlen.rw, dstaddr.ab	Convert long to packed	*	*	*	0	rsv, dov
36	CVTPL srclen.rw, srcaddr.ab, dst.wl	Convert packed to long	*	*	*	0	rsv, iov
08	CVTPS srclen.rw, srcaddr.ab, dstlen.rw, dstaddr.ab	Convert packed to leading separate	*	*	*	0	rsv, dov
09	CVTSP srclen.rw, srcaddr., dstlen.rw, dstaddr.ab	Convert leading separate to packed	*	*	*	0	rsv, dov
24	CVTPT srclen.rw, srcaddr.ab, tbladdr.ab, dstlen.rw, dstaddr.ab	Convert packed to trailing	*	*	*	0	rsv, dov
26	CVTTP srclen.rw, srcaddr.ab, tbladdr.ab, dstlen.rw, dstaddr.ab	Convert packed to trailing	*	*	*	0	rsv, dov
27	DIVP divrlen.rw, divraddr.ab, divdlen.rw, quolen.rw, quoaddr.ab	Divide packed	*	*	*	0	rsv, dov, ddvz
38	EDITPC srclen.rw, srcaddr.ab, pattern.ab, dstaddr.ab	Edit packed to character string	*	*	*	*	rsv, dov
3A	LOCC char.rb, len.rw, addr.ab	Locate character	0	*	0	0	
39	MATCHC objlen.rw, objaddr.ab, srclen.rw, srcaddr.ab	Match characters	0	*	0	0	
34	MOVP len.rw, srcaddr.ab, dstaddr.ab	Move packed	*	*	0	0	
2E	MOVTC srclen.rw, srcaddr.ab, fill.rb, tbladdr.ab, dstlen.rw, dstaddr.ab	Move translated characters	*	*	0	*	
2F	MOVTUC srclen.rw, srcaddr.ab, esc.rb, tbladdr.ab, dstlen.rw, dstaddr.ab	Move translated until character	*	*	*	*	
25	MULP mulrlen.rw, mulraddr.ab, muldlen.rw, muldaddr.ab, prodlen.rw, prodaddr.ab	Multiply packed	*	*	*	0	rsv, dov
2A	SCANC len.rw, addr.ab, tbladdr.ab, mask.rb	Scan for character	0	*	0	0	
3B	SKPC char.rb, len.rw, addr.ab	Skip character	0	*	0	0	

OP	Mnemonic & Arguments	Description	N	Z	V	C	Exceptions
2B	SPANC len.rw, addr.ab, tbladdr.ab, mask.rb	Span characters	0	*	0	0	
22	SUBP4 sublen.rw, subaddr.ab, diflen.rw, difaddr.ab	Subtract packed 4-operand	*	*	*	0	rsv, dov
23	SUBP6 sublen.rw, subaddr.ab, minlen.rw, minaddr.ab, diflen.rw, difaddr.ab	Subtract packed 6-operand	*	*	*	0	rsv, dov

APPENDIX C

CONSOLE ENTRY AND EXIT ROUTINES

C.1 INTRODUCTION

This appendix contains an example of a console entry and exit routine and a routine for simulating the memory management process. These routines are written in VAX-11 Macro.

C.2 CONSOLE ENTRY AND EXIT ROUTINE

The following are routines for entering and exiting the console. The console typically would be entered from the restart process as described in Section 2.8 of this user's guide. The console entry routine saves the volatile internal registers and process state, performs a stack swap if necessary, sets up the console stack pointer and calls the main console routine. The console exit routine is a mirror of the entry routine and restores the saved state of the processor back to the running state.

```
        .TITLE   Example of a Console Entry and Exit Routine

;
; Include files:  These files are a part of MicroVMS and can be
;                 used to define the processor registers and the
;                 PSL fields.

;
;       $PRDEF                          ; Define processor registers
;       $PSLDEF                         ; Define PSL fields
;

;
; MACROS:
;


        .macro  mtpr_pic src,dst
         moval  src,r0
         mtpr   r0,#dst
        .endm   mtpr_pic


        .macro  mcheck dst
         .if blank dst
         clrl   machine_check_cont
         .if_false
         movab  dst,machine_check_cont
         .endc
         .external machine_check_cont
        .endm   mcheck

;
;This macro calls the routine that simulates memory management
;

        .macro  poke    data,dst
         pushl  #19.
         pushal data
         pushal dst
         calls  #3,memory
         .external memory
        .endm   poke
```

```
;
; Equated Symbols:
;
;

        scb_a_mcheck            = 4
        scb_a_write_timeout     = ^x60


        k_parity.error          = 1
        k_bus.timeout           = 2


        PR$_P0BR        = ^d8
        PR$_P0LR        = ^d9
        PR$_P1BR        = ^d10
        PR$_P1LR        = ^d11
        PR$_SBR         = ^d12
        PR$_SLR         = ^d13
        PR$_PCBB        = ^d16
        PR$_SCBB        = ^d17
        PR$_RXCS        = ^d32
        PR$_TXCS        = ^d34
        PR$_MCESR       = ^d38
        PR$_SAVISP      = ^d41      ; MicroVAX special saved isp reg
        PR$_SAVPC       = ^d42      ; MicroVAX special saved pc reg
        PR$_SAVPSL      = ^d43      ; MicroVAX special saved psl reg
        PR$_IORESET     = ^d55      ; Bus init
        PR$_MAPEN       = ^d56

        PSL$V_CURMOD    = ^d24
        PSL$S_CURMOD    = ^d2


;
; declare the psects
;

        .psect  $$$$$0boot,page         ; main code psect for start
        .psect  $scb$,page              ; scb is located here
```

CONSOLE ENTRY AND EXIT ROUTINES

```
        .psect  $scb$                           ; the console scb

;+
; Console Program SCB.
;-

        .align  page                            ; SCB must be aligned.
SCB::
        .long   scb_int_00+1        ; #00   Unused.
        .long   machine_check_detect+1  ; #04   MCHK
        .long   scb_int_08+1        ; #08   KSP invalid.
        .long   scb_int_0c+1        ; #0C   Power failure.
        .long   scb_int_10+1        ; #10   Reserved/priv instr.
        .long   scb_int_14+1        ; #14   XFC instr.
        .long   reserved_operand_int+1  ; #18   Reserved oprnd.
        .long   scb_int_1c+1        ; #1C   Reserved addr mode.
        .long   scb_int_20+1        ; #20   Access violation.
        .long   scb_int_24+1        ; #24   Trans invalid.
        .long   scb_int_28+1        ; #28   Trace pending.
        .long   scb_int_2c+1        ; #2c   BPT
        .long   scb_int_30+1        ; #30   Compatibility mode
        .long   scb_int_34+1        ; #34   Arithmetic
        .long   scb_int_38+1        ; #38   Unused
        .long   scb_int_3c+1        ; #3C   Unused
        .long   scb_int_40+1        ; #40   CHMK.
        .long   scb_int_44+1        ; #44   CHME
        .long   scb_int_48+1        ; #48   CHMS
        .long   scb_int_4c+1        ; #4C   CHMU
        .long   scb_int_50+1        ; #50   SBI SILO
        .long   scb_int_54+1        ; #54   Corrected Mem Read
        .long   scb_int_58+1        ; #58   SBI Alert
        .long   scb_int_5c+1        ; #5C   SBI Fault
        .long   write_timeout_int+1         ; #60   Write timeout
        .long   scb_int_64+1        ; #64   Unused
        .long   scb_int_68+1        ; #68   Unused
        .long   scb_int_6c+1        ; #6C   Unused
        .long   scb_int_70+1        ; #70   Unused
        .long   scb_int_74+1        ; #74   Unused
        .long   scb_int_78+1        ; #78   Unused
        .long   scb_int_7c+1        ; #7C   Unused
        .long   scb_int_80+1        ; #80   Unused
        .long   scb_int_84+1        ; #84   Software level 1.
        .long   scb_int_88+1        ; #88   Software level 2.
        .long   scb_int_8c+1        ; #8C   Software level 3.
        .long   scb_int_90+1        ; #90   Software level 4.
        .long   scb_int_94+1        ; #94   Software level 5.
        .long   scb_int_98+1        ; #98   Software level 6.
        .long   scb_int_9c+1        ; #9C   Software level 7.
        .long   scb_int_a0+1        ; #A0   Software level 8.
        .long   scb_int_a4+1        ; #A4   Software level 9.
        .long   scb_int_a8+1        ; #A8   Software level 10.
        .long   scb_int_ac+1        ; #AC   Software level 11.
        .long   scb_int_b0+1        ; #B0   Software level 12.
```

```
.long   scb_int_b4+1    ; #B4   Software level 13.
.long   scb_int_b8+1    ; #B8   Software level 14.
.long   scb_int_bc+1    ; #BC   Software level 15.
.long   scb_int_c0+1    ; #C0   Interval timer.
.long   scb_int_c4+1    ; #C4   Unused
.long   scb_int_c8+1    ; #C8   Emulation start.
.long   scb_int_cc+1    ; #CC   Emulation continue.
.long   scb_int_d0+1    ; #D0   Unused
.long   scb_int_d4+1    ; #D4   Unused
.long   scb_int_d8+1    ; #D8   Unused
.long   scb_int_dc+1    ; #DC   Unused
.long   scb_int_e0+1    ; #E0   CSS
.long   scb_int_e4+1    ; #E4   CSS
.long   scb_int_e8+1    ; #E8   CSS
.long   scb_int_ec+1    ; #EC   CSS
.long   scb_int_f0+1    ; #F0   Console TU58
.long   scb_int_f4+1    ; #F4   Console TU58
.long   scb_int_f8+1    ; #F8   Console Transmit.
.long   scb_int_fc+1    ; #FC   Console Receive.
```

```
        .psect  $$$$$0boot                      ; get us to the right psect
;++
;       CONSOLE
;       It all starts right here.
;       We get here on a halt condition. The system has been
;       brought to a stable/known state by microcode, and the
;       volatile system registers have been saved away in known
;       MicroVAX registers.  This code copies the saved registers
;       to our private scratch RAM.  The GPR's are saved as well.
;       It is assumed that the RAM is at a known address and good.
;
;       The processor state is as follows:
;
;               savpc    = saved pc
;               savpsl   = saved psl + saved mapen + error code
;               savisp   = "real" isp
;               sp              = stack pointer at time of error
;               pc              = 20040000
;               psl             = 041F0000
;               astlvl          = ??? (except RESET = 4)
;               sisr            = ??? (except RESET = 0)
;               iccs            = ??? (except RESET = 0)
;
;       The console is now entered with the state of the processor
;       saved in limited life internal registers.  THIS STATE IS
;       VALID FOR A LIMITED TIME ONLY.  IN PARTICULAR, MEMORY
;       MANAGEMENT MUST BE LEFT DISABLED, AND ONLY A SUBSET OF THE
;       INSTRUCTION SET CAN BE USED (no emulated instructions)
;       before the saved values are moved to permanent memory.
;--

CONSOLE::

; save
;  PC
;  ISP
;  PSL
;  HALT CODE
;  MAPPEN
;  SP
;  R0 - R14

    mfpr    #pr$_savpc,saved_pc     ; save saved pc
    mfpr    #pr$_savisp,saved_isp   ; save saved isp
    mfpr    #pr$_savpsl,saved_psl   ; save saved psl
    movzbl  saved_psl+1,saved_halt  ; move the saved halt code + mappen
    clrb    saved_psl+1             ; and take it out of the saved psl
    movl    sp,temp                 ; save sp, for now
    moval   saved_regs,sp           ; point at end of saved reg space
    pushr   #-1                     ; save all the regs
    cmpzv   #0,#7,saved_halt,#3     ; see if this is a RESET
    bneq    5$                      ; no, so ISP is real
    moval   ram_stack_end+<10*4>,saved_isp ; yes, use end of our stack
```

```
5$:

;  Set up
;   new SP

        moval   ram_stack,sp            ; set us up on known ok stack

; save
;  SCBB
;  PCBB
;  update ?SP from before trap

        mfpr    #pr$_scbb,saved_scbb    ; save users scb
        mfpr    #pr$_pcbb,saved_pcbb    ; save users pcb
        cmpzv   #0,#7,saved_halt,#3     ; see if this is a RESET
        bneq    6$                      ; no, so PCBB is real
        clrl    saved_pcbb              ; yes, so PCBB is undefined,
                                        ; set it to 0
6$:     bbs     #26.,saved_psl,isp_ok   ; br if we were running on ISP
        movl    saved_pcbb,r1           ; get pcb addr handy,
        extzv   #psl$v_curmod,-         ; get current mode as well,
                #psl$s_curmod,-         ; for stack swap
                saved_psl,r2
        moval   (r1)[r2],r1             ; get addr of ?SP cell
        movl    temp,(r1)               ; and copy SP there - was
                                        ; saved in temp

isp_ok:
; save
;  IE bit of RXCS
;  IE bit of TXCS

        mfpr    #pr$_rxcs,r1            ; get term status
        movl    r1,saved_rxcs           ; and save it away
        bbcc    #6,r1,7$                ; clear IE if set
        mtpr    r1,#pr$_rxcs
7$:     mfpr    #pr$_txcs,r1            ; get term status
        movl    r1,saved_txcs           ; and save it away
        bbcc    #6,r1,8$                ; clear IE if set
        mtpr    r1,#pr$_txcs
8$:
; setup
;  SCB
;  PCB
;  trappers
; and call the console

        mcheck                          ; disable mcheck trapper
                                        ; i.e., mchecks are fatal
        mtpr_pic scb,pr$_scbb           ; set up the scb
        mtpr_pic ram_pcb,pr$_pcbb       ; set up the pcb
        calls   #0,console_main         ; we are up, go figure out
                                        ; what to do
```

CONSOLE ENTRY AND EXIT ROUTINES

```
; now externalize all the externals we need

        .external saved_pc,saved_psl,saved_isp,temp
        .external saved_regs,saved_regs_end
        .external saved_pcbb,saved_scbb,ram_pcb
        .external ram_stack,ram_stack_end,saved_rxcs,saved_txcs
        .external console_main,saved_halt
        .external machine_check_cont
```

```
;++
;       CONSOLE_EXIT
;       This code is a mirror of CONSOLE, and restores the
;       saved state back to the running state.
;--

        .entry  console_exit,^m<r2,r3,r4,r5,r6,r7,r8,r9,r10,r11>
        mtpr    #^xff,#pr$_mcesr        ; clear machine check error
        mcheck  50$                     ; if anything goes wrong, go back

        movl    saved_isp,r2            ; get the stack pointer
        poke    saved_psl,-(r2)         ; put PSL on stack
                                        ; the PSL and PC must be put on
                                        ; the interrupt stack, prior to
                                        ; the REI, if MAPEN is also on,
                                        ; then the stack must be mapped,
                                        ; and the PSL and PC put back on
                                        ; the mapped stack...the poke
                                        ; macro calls a routine that
                                        ; handles the problems of
                                        ; dealing with the mapped stack
        tstl    r0                      ; any errors?
        bneq    50$                     ; yes, back we go
        poke    saved_pc,-(r2)          ; put pc on stack
        tstl    r0                      ; any errors?
        bneq    50$                     ; yes, back we go

        movl    r2,temp                 ; save stack
        mtpr    saved_rxcs,#pr$_rxcs    ; restore terminal status (IE)
        mtpr    saved_txcs,#pr$_txcs    ; restore terminal status (IE)
        mcheck                          ; disable mcheck trapper
        mtpr    saved_scbb,#pr$_scbb    ; set up the scb
        mtpr    saved_pcbb,#pr$_pcbb    ; set up the pcb
        moval   saved_regs_end,sp       ; point at start of saved reg
                                        ; space
        popr    #-1                     ; restore all the registers
        movl    temp,sp                 ; restore saved sp to sp
        bbc     #7,saved_halt,console_rei ; if sys not mapped, skip
                                        ; mapping.
        brb     map_on                  ; goto long word aligned map
                                        ; turn on.

50$:    mcheck                          ; disable mcheck trapper
        ret                             ; if we fail, go back to user.

        .align  long                    ; in order for the enable of
                                        ; MAPEN to work, it MUST be in
                                        ; the same longword as the REI,
                                        ; thus this .align long is
                                        ; MOST critical

MAP_ON: mtpr    #1,#pr$_mapen           ; Mapping is turned on
```

```
CONSOLE ENTRY AND EXIT ROUTINES

CONSOLE_REI::
        rei                             ; and back to user
```

```
; now externalize all the externals we need

        .external saved_pc,saved_psl,saved_isp
        .external saved_regs,saved_regs_end
        .external saved_pcbb,saved_scbb,ram_pcb
        .external ram_stack,saved_rxcs,saved_txcs
```

```
;++
; machine_check_detect
;
; functional description:
;
; This sequence is the error trapper.  This sequence runs when a
; machine check occurs.  If machine_check_cont is not 0 the reason
; for the machine check is moved to r0 and then an REI to the
; machine_check_cont address is executed.  If machine_check_cont
; is 0 the trap is handled as an unexpected scb interrupt.
;
; inputs:
;
;       machine check stack
;       machine_check_cont      = address of the continuation code or 0
;
; outputs:
;
;       r0 = machine check code
;--

        .align  long

machine_check_detect:

        mtpr    #^xff,#pr$_mcesr        ; clear machine check error
        tstl    machine_check_cont      ; change return PC?
        beql    unfielded_scb_int       ; unexpected error if no
                                        ; continue addr
        movl    4(sp),r0                ; load reason
        addl    (sp)+,sp                ; pop stack
        movl    machine_check_cont,(sp) ; actually change return PC
        rei                             ; continue
```

```
;++
; write_timeout
; reserved_operand
;
; functional description:
;
; This sequence runs when a write timeout or reserved operand occurs.
;
; inputs:
;
;       PC/PSL are on the stack
;       machine_check_cont      = address to continue at or 0
;
; outputs:
;
;       r0 = error code
;--


        .align  long

write_timeout_int:
reserved_operand_int:

        movl    machine_check_cont,(sp) ;reset PC
        beql    unfielded_scb_int       ;unexpected error if no
                                        ;continue addr
        movl    #k_bus.timeout,r0       ;set code
        rei                             ;done
```

```
;++
; unfielded_scb_int
;
; functional description:
;
; This routine is executed if an unwanted SCB interrupt occurs during
; booting. System is halted.
;
; inputs:
;
;       scb interrupt stack
;
; outputs:
;
;       r0 has scb offset for unfielded scb interrupt otherwise
;          r0 is undefined
;       r1 has sp at time of trap
;--

        .align  long

unfielded_scb_int:

        movl    sp,r1
        halt
```

```
                .align long
scb_int_00:     movl    #^x0,r0
                brw     unfielded_scb_int
                .align long
scb_int_04:     movl    #^x4,r0
                brw     unfielded_scb_int
                .align long
scb_int_08:     movl    #^x8,r0
                brw     unfielded_scb_int
                .align long
scb_int_0c:     movl    #^xc,r0
                brw     unfielded_scb_int

                .align long
scb_int_10:     movl    #^x10,r0
                brw     unfielded_scb_int
                .align long
scb_int_14:     movl    #^x14,r0
                brw     unfielded_scb_int
                .align long
scb_int_18:     movl    #^x18,r0
                brw     unfielded_scb_int
                .align long
scb_int_1c:     movl    #^x1c,r0
                brw     unfielded_scb_int

                .align long
scb_int_20:     movl    #^x10,r0
                brw     unfielded_scb_int
                .align long
scb_int_24:     movl    #^x14,r0
                brw     unfielded_scb_int
                .align long
scb_int_28:     movl    #^x18,r0
                brw     unfielded_scb_int
                .align long
scb_int_2c:     movl    #^x1c,r0
                brw     unfielded_scb_int

                .align long
scb_int_30:     movl    #^x10,r0
                brw     unfielded_scb_int
                .align long
scb_int_34:     movl    #^x14,r0
                brw     unfielded_scb_int
                .align long
scb_int_38:     movl    #^x18,r0
                brw     unfielded_scb_int
                .align long
scb_int_3c:     movl    #^x1c,r0
                brw     unfielded_scb_int

                .align long
```

```
scb_int_40:     movl    #^x10,r0
                brw     unfielded_scb_int
                .align long
scb_int_44:     movl    #^x14,r0
                brw     unfielded_scb_int
                .align long
scb_int_48:     movl    #^x18,r0
                brw     unfielded_scb_int
                .align long
scb_int_4c:     movl    #^x1c,r0
                brw     unfielded_scb_int

                .align long
scb_int_50:     movl    #^x10,r0
                brw     unfielded_scb_int
                .align long
scb_int_54:     movl    #^x14,r0
                brw     unfielded_scb_int
                .align long
scb_int_58:     movl    #^x18,r0
                brw     unfielded_scb_int
                .align long
scb_int_5c:     movl    #^x1c,r0
                brw     unfielded_scb_int

                .align long
scb_int_60:     movl    #^x10,r0
                brw     unfielded_scb_int
                .align long
scb_int_64:     movl    #^x14,r0
                brw     unfielded_scb_int
                .align long
scb_int_68:     movl    #^x18,r0
                brw     unfielded_scb_int
                .align long
scb_int_6c:     movl    #^x1c,r0
                brw     unfielded_scb_int

                .align long
scb_int_70:     movl    #^x10,r0
                brw     unfielded_scb_int
                .align long
scb_int_74:     movl    #^x14,r0
                brw     unfielded_scb_int
                .align long
scb_int_78:     movl    #^x18,r0
                brw     unfielded_scb_int
                .align long
scb_int_7c:     movl    #^x1c,r0
                brw     unfielded_scb_int

                .align long
scb_int_80:     movl    #^x10,r0
```

```
                brw     unfielded_scb_int
                .align long
scb_int_84:     movl    #^x14,r0
                brw     unfielded_scb_int
                .align long
scb_int_88:     movl    #^x18,r0
                brw     unfielded_scb_int
                .align long
scb_int_8c:     movl    #^x1c,r0
                brw     unfielded_scb_int

                .align long
scb_int_90:     movl    #^x10,r0
                brw     unfielded_scb_int
                .align long
scb_int_94:     movl    #^x14,r0
                brw     unfielded_scb_int
                .align long
scb_int_98:     movl    #^x18,r0
                brw     unfielded_scb_int
                .align long
scb_int_9c:     movl    #^x1c,r0
                brw     unfielded_scb_int

                .align long
scb_int_a0:     movl    #^x10,r0
                brw     unfielded_scb_int
                .align long
scb_int_a4:     movl    #^x14,r0
                brw     unfielded_scb_int
                .align long
scb_int_a8:     movl    #^x18,r0
                brw     unfielded_scb_int
                .align long
scb_int_ac:     movl    #^x1c,r0
                brw     unfielded_scb_int

                .align long
scb_int_b0:     movl    #^x10,r0
                brw     unfielded_scb_int
                .align long
scb_int_b4:     movl    #^x14,r0
                brw     unfielded_scb_int
                .align long
scb_int_b8:     movl    #^x18,r0
                brw     unfielded_scb_int
                .align long
scb_int_bc:     movl    #^x1c,r0
                brw     unfielded_scb_int

                .align long
scb_int_c0:     movl    #^x10,r0
                brw     unfielded_scb_int
```

```
                .align long
scb_int_c4:     movl    #^x14,r0
                brw     unfielded_scb_int
                .align long
scb_int_c8:     movl    #^x18,r0
                brw     unfielded_scb_int
                .align long
scb_int_cc:     movl    #^x1c,r0
                brw     unfielded_scb_int

                .align long
scb_int_d0:     movl    #^x10,r0
                brw     unfielded_scb_int
                .align long
scb_int_d4:     movl    #^x14,r0
                brw     unfielded_scb_int
                .align long
scb_int_d8:     movl    #^x18,r0
                brw     unfielded_scb_int
                .align long
scb_int_dc:     movl    #^x1c,r0
                brw     unfielded_scb_int

                .align long
scb_int_e0:     movl    #^x10,r0
                brw     unfielded_scb_int
                .align long
scb_int_e4:     movl    #^x14,r0
                brw     unfielded_scb_int
                .align long
scb_int_e8:     movl    #^x18,r0
                brw     unfielded_scb_int
                .align long
scb_int_ec:     movl    #^x1c,r0
                brw     unfielded_scb_int

                .align long
scb_int_f0:     movl    #^x10,r0
                brw     unfielded_scb_int
                .align long
scb_int_f4:     movl    #^x14,r0
                brw     unfielded_scb_int
                .align long
scb_int_f8:     movl    #^x18,r0
                brw     unfielded_scb_int
                .align long
scb_int_fc:     movl    #^x1c,r0
                brw     unfielded_scb_int

        .END
```

```
        .TITLE  BOOTRAM - MicroVAX BOOT RAM

;++
;
;
; Abstract:  BOOTRAM.:.provides initial setup of scratch
;            space in RAM
;
; This sequence defines the scratch area in RAM used by the
; ROM code.  This sequence uses preallocated space in RAM, an
; alternative is to search for known good RAM and define it
; as the scratch are for ROM.
;
; Environment:  Mode=Kernel
;
;
;--
        .psect  BOOTRAM, WRT, PAGE


        .align page
ram_start::
ram_pcb::       .blkl   128     ;process control block for rom code

machine_check_cont::            ; contains 0 or addr to xfer to
                .blkl   1       ; after a machine check


        .align long
saved_rxcs::    .blkl   1       ; save rxcs IE bit here
saved_txcs::    .blkl   1       ; save txcs IE bit here
saved_halt::    .blkl   1       ; save salted halt code/mapen here
saved_isp::     .blkl   1       ; save salted isp here
saved_scbb::    .blkl   1       ; save scbb here
saved_pcbb::    .blkl   1       ; save pcbb here
saved_regs_end::
                .blkl   15.
saved_regs::                    ; save GPR's starting here
saved_pc::      .blkl   1       ; save salted pc here
saved_psl::     .blkl   1       ; save salted psl here
temp::          .blkl   1       ; temp scratch cell for SP
ram_stack_end::
                .blkl   128.    ; mini stack to use while in rom code
ram_stack::


ram_end::

        .end
```

C.3 MEMORY MANAGEMENT SIMULATION

When memory management is to be enabled when exiting from the console, the environment that the console exits to must have a validly mapped interrupt stack with at least to spare longwords at the bottom. These routines simulate read/writes to physical memory with memory management on or off. These routines are called by the poke macro in the console exit routine.

```
        .TITLE   Example of MicroVAX Memory Management Simulation

;
; Include files:
;
;       $PTEDEF                         ; Define PTE fields
;       $VADEF                          ; Define virtual address fields
;       $PRDEF                          ; Define processor registers
;       $PSLDEF                         ; Define PSL fields

;
; MACROS:
;

        .macro  mcheck dst
         .if blank dst
         clrl   machine_check_cont
         .if_false
         movab  dst,machine_check_cont
         .endc
         .external machine_check_cont
        .endm   mcheck

        .macro  push_mcheck
         movl   machine_check_cont,-(sp)
         .external machine_check_cont
        .endm   push_mcheck

        .macro  pop_mcheck
         movl   (sp)+,machine_check_cont
         .external machine_check_cont
        .endm   pop_mcheck
```

```
;
; Equated Symbols:
;
;

        success         = 0
        err_acc_vio     = 100
        err_bad_addr    = 101
        err_len_vio     = 103
        err_trans_nv    = 104
        err_nxm         = 105

        PR$_P0BR        = ^d8
        PR$_P0LR        = ^d9
        PR$_P1BR        = ^d10
        PR$_P1LR        = ^d11
        PR$_SBR         = ^d12
        PR$_SLR         = ^d13

        PSL$V_CURMOD    = ^d24
        PSL$S_CURMOD    = ^d2

;
; declare the psects
;

        .psect  $memman,page            ; main code psect for memory
                                        ; management
```

```
        .psect  $memman                 ; get us to the right psect
;++
; PROBE
;       arguments are
;               R2      Virtual address
;               R3      Physical address
;               R4      <1> Read/Write indicator  (0 = read, 1 = write)
;
;       This routine translates a virtual addresses to a physical
;       address.  If write intent, and MAPEN, then also sets the M bit
;       in PTE.
;
;       returned arguments are
;               R2      unchanged
;               R3      Physical address (filled in)
;               R4      unchanged
;       return status
;               success         = 0
;               err_acc_vio     = 1
;               err_len_vio     = 2
;               err_trans_nv    = 3
;
;--
        .entry  probe,^m<R2>
        movl    r2,r1                   ; get Vaddr
        movl    r4,r2                   ; get quals

        extzv   #30,#2,r1,r0            ; get P0/P1/S0/S1 dispatch
        caseb   r0,#0,#2                ; go to correct length checker
9$:     .word   10$-9$                  ; P0
        .word   20$-9$                  ; P1
        .word   30$-9$                  ; S0
        movl    #err_acc_vio,r0         ; assume failure
        brb     40$                     ; err

10$:    calls   #0,get_p0_Vaddr         ; map a p0 addr
        brb     40$                     ; join common

20$:    calls   #0,get_p1_Vaddr         ; map a p1 addr
        brb     40$                     ; join common

30$:    calls   #0,get_sys_Vaddr        ; map a sys addr
40$:    movl    r1,r3                   ; return paddr
        ret
```

```
;+
; GET_SYS_VADDR
;       arguments are
;               R1      Virtual address
;               R2      <1> Read/Write indicator  (0 = read, 1 = write)
;       returns
;               R0 = status ( 0 = success, else err)
;               R1 =    physical address if success
;
;  regs are used in this routine as follows
;
;       R3 = BOFF (byte offset within page)
;       R4 = VPN  (virtual page number, or VPN * 4, or PTE_Addr)
;       R6 = scratch
;
;_

        .entry GET_SYS_VADDR,^m<R3,R4,R6>
        movl    #err_acc_vio,r0         ; assume failure
        push_mcheck
        mcheck  20$                     ; any problem is an access
                                        ; violation
; first get the byte offset and VPN
        extzv   #0,#9.,r1,r3            ; offset within page
        extzv   #9.,#21.,r1,r4          ; page number
; check page range
        mfpr    #pr$_slr,r6             ; get length (in ptes)
        cmpl    r4,r6                   ; is pte in the table?
        bgequ   20$                     ; no so error
        rotl    #2,r4,r4                ; now is offset to pte in table
        mfpr    #pr$_sbr,r6             ; get sbr base address
        addl    r6,r4                   ; got the pte address
        movl    (r4),r6                 ; and the pte
        bgeq    20$                     ; br if valid bit not set.
        bbc     #1,r2,10$               ; do we have write intent?
        bisl    #1@26.,(r4)             ; yes, set the Modified bit
10$:    extzv   #0,#21.,r6,r1           ; get <29:9> of physical address
        rotl    #9.,r1,r1               ; now is in place
        bisl    r3,r1                   ; and is correct physical
                                        ; address
        clrl    r0                      ; signal success
20$:    pop_mcheck
        ret
```

```
;+
; GET_P0_VADDR
;       arguments are
;               R1      Virtual address
;               R2      <1> Read/Write indicator  (0 = read, 1 = write)
;       returns
;               R0 = status ( 0 = success, else err)
;               R1 =    physical address if success
;
;  regs are used in this routine as follows
;
;       R3 = BOFF (byte offset within page)
;       R4 = VPN  (virtual page number, or VPN * 4, or PTE_Addr)
;       R6 = scratch
;
;_

        .entry GET_P0_VADDR,^m<R3,R4,R6>
        movl    #err_acc_vio,r0         ; assume failure
        push_mcheck
        mcheck  20$                     ; any problem is an access
                                        ; violation
; first get the byte offset and VPN

        extzv   #0,#9.,r1,r3            ; offset within page
        extzv   #9.,#21.,r1,r4          ; page number

; check page range

        mfpr    #pr$_p0lr,r6            ; get length (in ptes)
        cmpl    r4,r6                   ; is pte in the table?
        bgequ   20$                     ; no so error
        rotl    #2,r4,r4                ; now is offset to pte in table
        mfpr    #pr$_p0br,r6            ; get p0br base address
        addl    r6,r4                   ; got the pte address

; now translate the system virtual pte addr to physical

        movl    r2,-(sp)                ; save r2
        clrl    r2                      ; say no write access
        calls   #0,get_sys_vaddr        ; translate the pte address
        movl    (sp)+,r2                ; restore r2
        tstl    r0                      ; did it translate?
        bneq    20$                     ; no, return error
        movl    r1,r4                   ; physical address of p0 pte
        movl    (r4),r6                 ; and the pte
        bgeq    20$                     ; br if valid bit not set.
        bbc     #1,r2,10$               ; do we have write intent?
        bisl    #1@26.,(r4)             ; yes, set the Modified bit
10$:    extzv   #0,#21.,r6,r1           ; get <29:9> of phy addr
        rotl    #9.,r1,r1               ; now is in place
        bisl    r3,r1                   ; and is correct physical
                                        ; address
```

```
        clrl    r0                      ; signal success
20$:    pop_mcheck
        ret
```

```
;+
; GET_P1_VADDR
;       arguments are
;               R1      Virtual address
;               R2      <1> Read/Write indicator  (0 = read, 1 = write)
;       returns
;               R0 = status ( 0 = success, else err)
;               R1 =    physical address if success
;
;  regs are used in this routine as follows
;
;       R3 = BOFF (byte offset within page)
;       R4 = VPN  (virtual page number, or VPN * 4, or PTE_Addr)
;       R6 = scratch
;
;-

        .entry GET_P1_VADDR,^m<R3,R4,R6>
        movl    #err_acc_vio,r0         ; assume failure
        push_mcheck
        mcheck  20$                     ; any problem is an access
                                        ; violation
; first get the byte offset and VPN
        extzv   #0,#9.,r1,r3            ; offset within page
        extzv   #9.,#21.,r1,r4          ; page number
; check page range
        mfpr    #pr$_p1lr,r6            ; get length (in ptes)
        cmpl    r4,r6                   ; is pte in the table?
        blssu   20$                     ; no so error
        rotl    #2,r4,r4                ; now is offset to pte in table
        mfpr    #pr$_p1br,r6            ; get p1br base address
        addl    r6,r4                   ; got the pte address

; now translate the system virtual pte addr to physical

        movl    r2,-(sp)                ; save r2
        clrl    r2                      ; say no write access
        calls   #0,get_sys_vaddr        ; translate the pte address
        movl    (sp)+,r2                ; restore r2
        tstl    r0                      ; did it translate?
        bneq    20$                     ; no, return error
        movl    r1,r4                   ; physical address of p1 pte
        movl    (r4),r6                 ; and the pte
        bgeq    20$                     ; br if valid bit not set.
        bbc     #1,r2,10$               ; do we have write intent?
        bisl    #1@26.,(r4)             ; yes, set the Modified bit
10$:    extzv   #0,#21.,r6,r1           ; get <29:9> of physical address
        rotl    #9.,r1,r1               ; now is in place
        bisl    r3,r1                   ; and is correct physical address
        clrl    r0                      ; signal success
20$:    pop_mcheck
        ret
```

```
;+
; exits
; status code to r0
;-

done:   clrl    r0                      ; return success
        ret

acc_vio:
        movl    #err_acc_vio,r0         ; failure return err
        ret

nxm:
        movl    #err_nxm,r0             ; failure return err
        ret
```

```
;++
; MEMORY
;       arguments are
;               4(ap)     Physical/Virtual address
;               8(ap)     Address of Data to read/write
;               12(ap)
;                       <0>   Physical/Virtual (0 = physical, 1 = virt)
;                       <1>   Read/Write indicator (0 = read, 1 = write)
;                       <4:2> Data length
;                               0 = err
;                               1 = byte
;                               2 = word
;                               3 = err
;                               4 = long
;
;       This routine does not really do much it self, it is only an
;       interface to the real memory access routine, RMEMORY.
;       The RMEMORY routine does physical/virtual IO to memory (both
;       read and write).
;
;       returned arguments are
;               4(ap)   unchanged
;               8(ap)   Data read or written.
;               12(ap)  unchanged
;       return status
;               success         = 0
;               err_acc_vio     = 1
;
;--
        .entry  memory,^m<R2,R3,R4,R5>
        movl    4(ap),r2                ; get Addr
        movl    @8(ap),r5               ; get Data in case we want to
                                        ; write
        movl    12(ap),r4               ; get quals
        calls   #0,rmemory              ; now go do the real work
        bbs     #1,r4,10$               ; if write, no need to write
                                        ; back data
        movl    r5,@8(ap)               ; return data
10$:    ret
```

```
;++
; RMEMORY
;       arguments are
;               R2      Physical/virtual address
;               R5      Data to read/write
;               R4
;                    <0>    Physical/Virtual  (0 = physical, 1 = virt)
;                    <1>    Read/Write indicator  (0 = read, 1 = write)
;                    <4:2>  Data length
;                              0 = err
;                              1 = byte
;                              2 = word
;                              3 = err
;                              4 = long
;
;       This routine does physical/virtual IO to memory. (both read
;       and write)
;
;       returned arguments are
;               R2      unchanged
;               R5      Data read or written.
;               R4      unchanged
;       return status   in R0
;               success         = 0
;               err_acc_vio     = 1
;
;--

;+
; register usage while in this routine....
;       r0 = status or SCRATCH
;       r2 = original virtual address
;       r3 = Physical address to pass to memio or SCRATCH
;       r4 = Qual (original or temporary)
;       r5 = data to read or write
;       r6 = DL for total access
;       r7 = DL for page 2
;       r8 = data accumulator
;       r9 = original r5
;-
        .entry  rmemory,^m<R2,R3,R4,R6,R7,R8,R9>
        movl    r5,r9
        movl    r2,r3                   ; assume we want phyio
        blbs    r4,31$                  ; all ok if doing phyio
32$:    brw     30$                     ; br if phyio
31$:    bbc     #7.,saved_halt,32$      ; if mapen off, V is P, ok too.

; we are doing a virtual io, see if we span a page boundary

        extzv   #0,#9.,r2,r1            ; get low 9 bits of address
        extzv   #2,#3,r4,r6             ; get data length
        decl    r1                      ;  - 1
        addl2   r6,r1                   ; now is VA<9:0>+DL-1
```

```
        bbc     #9.,r1,20$              ; does not span page boundary
                                        ; branch to address translation

; we now have a virtual access that spans a page boundary

        bicl3   #^c511.,r1,r7           ; save number of bytes on
                                        ; second page-1
        incl    r7                      ; now is correct
        calls   #0,probe                ; see if access is legal to
                                        ; first page
        tstl    r0                      ; any errors?
        bneq    25$                     ; yes, return the error
        addl    r6,r2                   ; now point to last byte
        calls   #0,probe                ; see if access is legal to
                                        ; last page
        subl    r6,r2                   ; put the first Vio addr back
        tstl    r0                      ; any errors?
        bneq    25$                     ; yes, return the error

; the access is legal to both parts, now we split the Vio up into 2
; seperate Vio's one for each page

        subl3   r7,r6,r1                ; get DL for part 1
        insv    r1,#2,#3,r4             ; set up qual for the access
                                        ; page 1
        calls   #0,rmemory              ; now recurse to do page one's
                                        ; Vio
        tstl    r0                      ; any errors?
        bneq    25$                     ; yes, return the error
        movl    r5,r8                   ; save the partial data

        subl3   r7,r6,r1                ; get DL for part 1 again
        addl    r1,r2                   ; point to first byte of part 2
        rotl    #3,r1,r1                ; get number of bits in part 1
        subl3   r1,#32.,r1              ; number of bits to left shift
                                        ; data
        rotl    r1,r9,r5                ; r5 is now data for part 2
        insv    r7,#2,#3,r4             ; set up qual for the access
                                        ; page 2
        calls   #0,rmemory              ; now recurse to do page two's
                                        ; Vio
        ashl    #3,r7,r3                ; number of bits in part 2
        subl3   r3,#32.,r3              ; now number of bits to shift by
        ashl    r3,r5,r5                ; shift part two to where it
        bisl    r8,r5                   ; belongs and set in the low
                                        ; bytes
25$:    ret                             ; we are done now

20$:    calls   #0,probe                ; is simple Vio, do probe
        tstl    r0                      ; any errors?
        bneq    25$                     ; yes, return the error
                                        ; else now have phyio to do
                                        ; so fall through and do it.
```

```
30$:    calls   #0,memio                ; go do the IO
        mcheck                          ; turn off the trapper
ret$:   ret

        .external saved_halt
```

```
;+
; MEMIO
;       arguments are
;               R3  Physical address
;               R4  <1>     Read/Write indicator  (0 = read, 1 = write)
;                   <4:2>   Data length
;                               0 = err
;                               1 = byte
;                               2 = word
;                               3 = word+byte
;                               4 = long
;               R5  Data
;
;       This routine does the actual read/write to physical memory
;
;       returns
;               R0 = status ( 0 = success, else err)
;               R3 =    Unchanged
;               R4 =    Unchanged
;               R5 =    Data if success
;_
        .entry MEMIO,0


        mcheck  nxm                     ; error is nxm
        extzv   #2,#3,r4,r1             ; get data length
        bbc     #1,r4,100$              ; dispatch to read service
        caseb   r1,#0,#4                ; go to correct writer
9$:     .word   30$-9$                  ; err
        .word   10$-9$                  ; writeB
        .word   20$-9$                  ; writeW
        .word   30$-9$                  ; writeW+writeB
        .word   40$-9$                  ; writeL
        brw     acc_vio                 ; err

10$:    movb    r5,(r3)                 ; write it out
        brw done

20$:    movw    r5,(r3)                 ; write it out
        brw done

30$:    pushl   r5
        pushl   r3
        movb    r5,(r3)+                ; write out first byte
        rotl    #-8.,r5,r5              ; shift the data for next part
        movw    r5,(r3)                 ; write it out
        popl    r3
        popl    r5
        brw done

40$:    movl    r5,(r3)                 ; write it out
        brw done
```

```
100$:   caseb   r1,#0,#4                ; go to correct reader
109$:   .word   130$-109$               ; err
        .word   110$-109$               ; readB
        .word   120$-109$               ; readW
        .word   130$-109$               ; readW+readB
        .word   140$-109$               ; readL
        brw     acc_vio                 ; err

110$:   movzbl  (r3),r5                 ; read data
        brw done

120$:   movzwl  (r3),r5                 ; read data
        brw done

130$:   pushl   r3
        movzwl  (r3)+,r5                ; read data
        movzbl  (r3),r3
        rotl    #16.,r3,r3
        bisl    r3,r5
        popl    r3
        brw done

140$:   movl    (r3),r5                 ; read data
        brw done

        .END
```

APPENDIX D

MECHANICAL SPECIFICATIONS

D.1 PACKAGING

The MicroVAX 78032 CPU is available in two different packages; surface mount and socket mount. Figures D-1 and D-2 give the mechanical specifications for these packages.

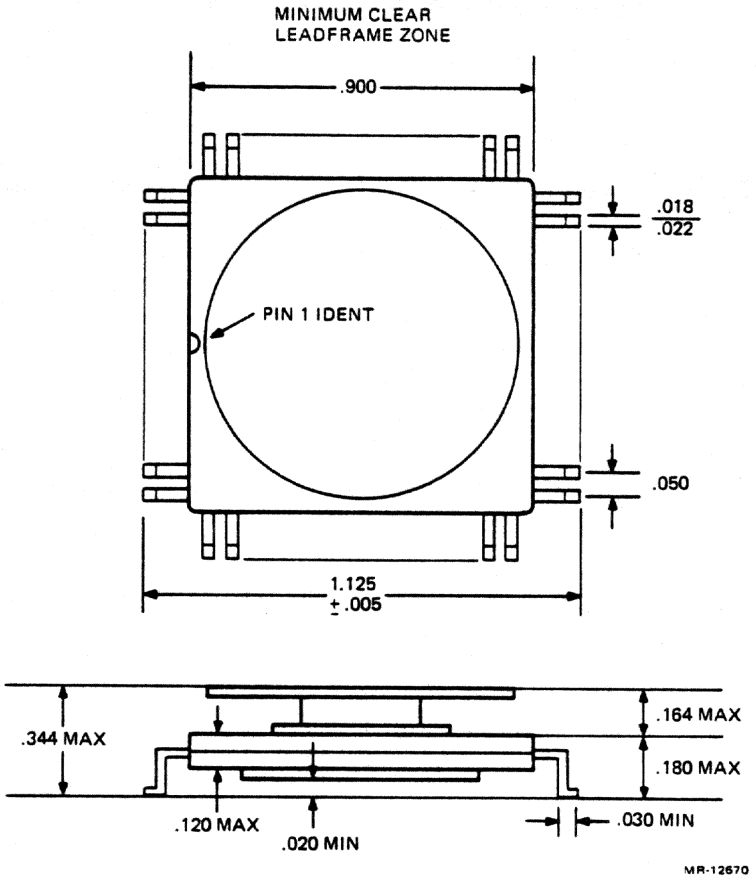


Figure D-1 68 Pin CERQUAD, Surface Mount

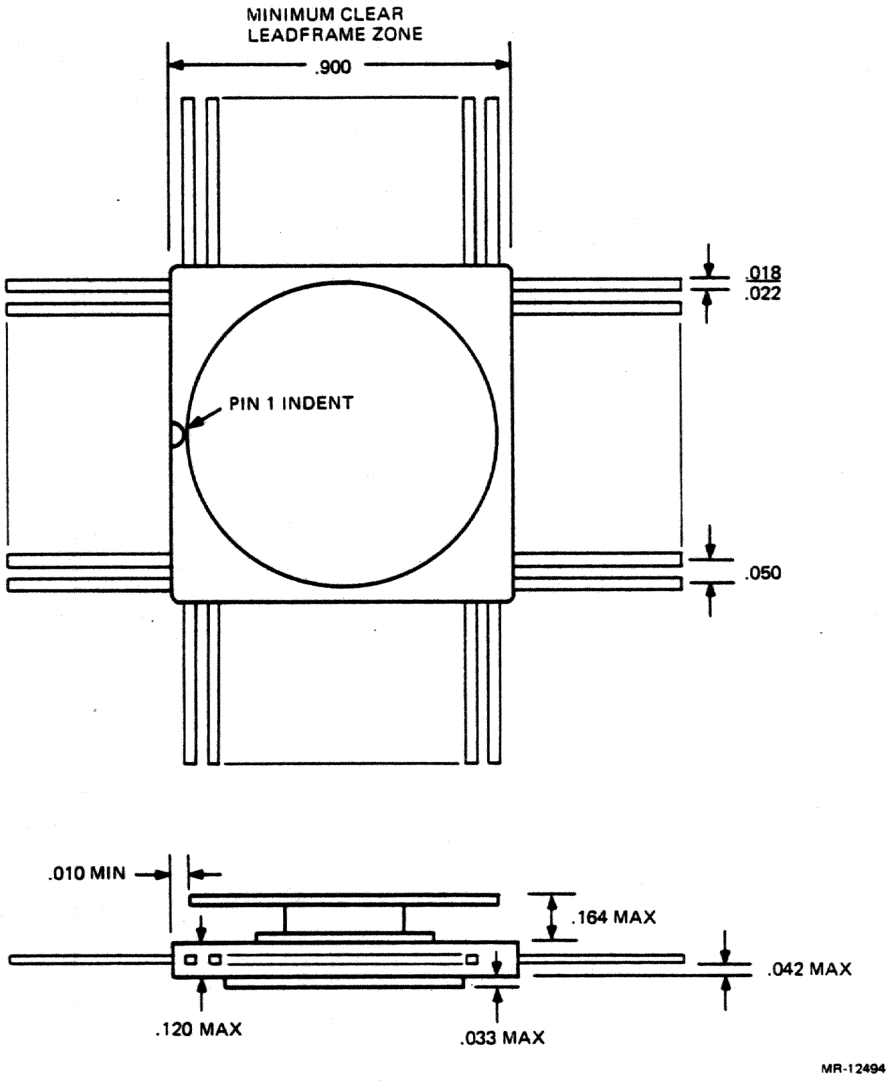


Figure D-2 68 Pin CERQUAD, Socket Mount

INDEX