KA780 Centraler Processor

Technical Description

EK-KA780-TD.001

EK-KA780-TD-001

KA780 Central Processor Technical Description

digital equipment corporation • maynard, massachusetts

First Edition, June 1979

The material in this manual 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.

Printed in U.S.A.

CONTENTS

CONTENTS (Cont)

CONTENTS (Cont)

FIGURES

FIGURES (Cont)

FIGURES (Cont)

TABLES

CHAPTER 1
INTRODUCTION

1.1 MANUAL SCOPE

This manual provides a general description of the VAX-11/780 system in Chapter 1 and a detailed functional description of the KA780 central processor in Chapter 2. For a complete discussion of the KA780 central processor, this manual should be read in conjunction with the Translation Buffer, Cache, SBI, Control technical description and the KC780 Console Interface technical description. The purpose of this manual is to provide a resource for appropriate branch and support level courses of the Field Service and Manufacturing training programs and as a field reference.

Detailed information concerning system components not covered in this manual can be found in the related literature listed in Table 1-1.

1.2 SYSTEM OVERVIEW

The VAX-11/780 system is a high-speed, synchronous microprogrammed computer that represents a significant extension to the PDP-11 family of computers. The processor is capable of executing variable length instructions in native mode and non-privileged PDP-11 instructions in compatibility mode. Compatibility mode enables existing user mode PDP-11 programs to be run without modification.

The major components of the VAX-11/780 system, shown in Figure 1-1, include the following;

a. central processing unit (CPU)
b. memory cache
c. writable diagnostic control store (WDCS)
d. clocks
e. console subsystem
f. main memory and memory controllers
g. optional floating-point accelerator
h. Massbus adapter and Massbus peripherals
i. Unibus adapter and Unibus peripherals

Table 1-1 Related Hardware Manuals

Title	Document Number
Translation Buffer, Cache, SBI Control Technical Description	EK-MM780-TD *
MS780 Memory System Technical Description	EK-MS780-TD *
DW780 Unibus Adaptor Technical Description	EK-DW780-TD *
RH780 Massbus Adapter Technical Description	EK-RH780-TD *
KC780 Console Interface Board Technical Description	EK-KC780-TD *
FP780 Floating-Point Accelerator Technical Description	EK-FP780-TD *
VAX-11/780 Hardware User Guide	EK-UG780-UG **
VAX-11/780 System Maintenance Guide	EK-11780-PG **
VAX-11/780 Power System Technical Description	EK-PS780-TD *
VAX-11/780 Installation Manual	EK-SI780-IN **
DS780 Diagnostic System User's Guide	EK-DS780-UG **
DS780 Diagnostic System Technical Description	EK-DS780-TD *

** Available on hard copy only.
* Available on hard copy and microfiche.

For information concerning microfiche libraries, contact:

Digital Equipment Corporation
Micropublishing Group
12 Crosby Drive
Bedford, MA 01730

Hardcopy documents can be ordered from:

Digital Equipment Corporation
444 Whitney Street
Northboro, MA 01532
Attention: Communication Services (NR2/M15)
Customer Services Section

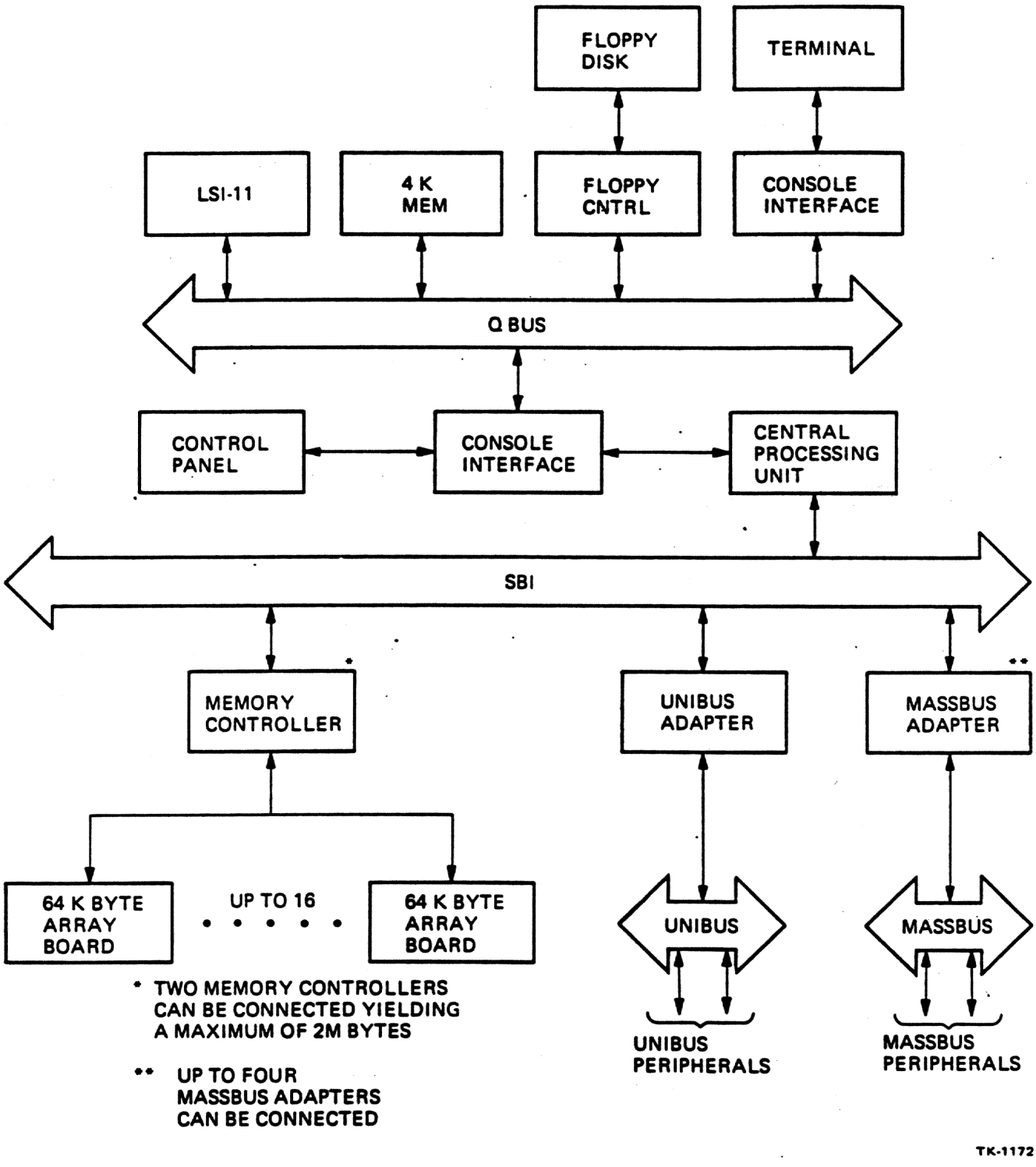


Figure 1-1 VAX-11/780 System Block Diagram

These major hardware components operate on clocked 200 nanosecond cycles. Normal operations are synchronized by the system clock and each event occurs at defined points in time within the machine cycle.

1.2.1 Synchronous Backplace Interconnect

Data exchanges between the CPU, memory, and peripheral bus adapters are made over a high-speed bus called the SBI (Synchronous Backplace Internconnect). The SBI enables checked, parallel information exchanges which are synchronized with the system clock.

The SBI has a physical address space of one billion bytes (2^{30}). The physical address space is all possible memory and I/O addresses accessible to the processor. Allocation of the physical address space is divided in half. The upper half is assigned to I/O addresses and the lower half is reserved for memory.

The VAX-11/780 memory management system provides a mechanism which maps over four billion (2^{32}) bytes of virtual address space to one billion bytes of physical address space. The virtual address space is divided in half. The lower half (per process space) is designated for use by a process. A context switch changes the mapping of all locations in the process space to accommodate the current process. The upper half (system space) is shared by all processes and remains the same during context switching. The memory management system translates the addresses from virtual to physical and also provides the capabilities for paging, swapping, overlaying protection, and sharing.

1.2.2 Central Processing Unit

Logical and arithmetic operations are performed by the central processing unit. The processor provides sixteen 32-bit general purpose registers that can be used for temporary storage, as accumulators, index registers, and base registers.

The processor's microcode performs the virtual to physical address translations. The address translations and associated memory access protection information are stored in a translation buffer. The system also includes a memory cache which reduces the average memory access time.

The microcode is contained in a PROM (programmable read only memory) control store. The standard control store contains 4K 96-bit microwords plus 3 parity bits per microword. Also included is a writable diagnostic control store (WDCS) for updating the microcode. The WDCS is a RAM (random access memory) which contains 1K 96-bit microwords plus 3 parity bits per microword.

1.2.3 MS780 Main Memory

The VAX-11/780 main memory (MS780) is a MOS (metal oxide semiconductor), random access memory subsystem which consists of a controller and one to sixteen array boards. Memory features include an error checking and correction (ECC) scheme which can

detect all double-bit errors and detect and correct all single-bit errors.

Each memory controller can access a maximum of 1M bytes. Two memory controllers can be connected to the SBI yielding a maximum of 2M bytes of physical memory available to the system.

The VAX-11/780 system includes two standard clocks in addition to the system clock. The programmable real-time clock is used by the operating system and by diagnostics. The time-of-year clock is used for system operations and includes a battery back-up for automatic system restart operations.

1.2.4 Console Subsystem

The console subsystem provides the interface between the operator and the processor. The subsystem includes an LSI-11 microprocessor (KD11-F) with a 4K by 16-bit RAM, a floppy disk drive and controller (RXV11), a system terminal and two serial line interface units (DLV11-E). The console interface board (CIB) contains a 4K by 16-bit ROM for the LSI-11 microprocessor. The control panel is located on the VAX-11/780 cabinet.

1.2.5 Peripheral Controllers

Peripherals are connected to the SBI through two types of adapters; the Massbus adapter and Unibus adapter.

The Massbus adapter provides the interface for high speed disk or magnetic tape devices. Up to four Massbus adapters can be placed on the SBI. Each adapter includes a translation map that allows the transfer of physically continuous disk blocks to/from discontiguous blocks of memory. The Massbus adapter includes a silo (32-byte data buffer) to enable efficient transfer of data from memory to the Massbus device. Data being transferred from the Massbus device to memory is assembled into 64-bit quadwords (plus parity) to make use of the SBI bandwidth.

The Unibus adapter provides the interface for a number of I/O devices, including terminals, line printers and card readers. The adapter translates 18-bit Unibus addresses to 30-bit SBI addresses and provides priority arbitration among devices on the Unibus. It allows the processor to access Unibus peripheral device registers and allows Unibus devices access to random main memory locations. High speed Unibus devices can also access consecutive memory locations within a memory page. The adapter provides buffered direct memory access data paths for up to 15 non-processor request (NPR) devices. Each data path has a 64-bit buffer (plus byte parity) which holds four 16-bit PDP-11 data words. Therefore, only one SBI operation is required for every four Unibus transfers.

1.2.6 Floating-Point Accelerator

An optional floating-point accelerator (FPA) is available in the VAX-11/780 system. The FPA extends the processor's capabilities and improves the speed and performance of floating-point instructions. The floating-point accelerator executes addition, subtraction, multiplcation and division instructions that operate on single- and double-precision floating-point operands, including the special EMOD and POLY instructions in both single- and double-precision formats. The floating-point accelerator is not required for the processor to execute floating-point instructions and it can be added or removed without changing existing software.

For a complete discussion of the floating-point accelerator, peripheral adapters, memory system, or console subsystem, reference should be made to the associated manual listed in Table 1-1.

1.3 SYSTEM ARCHITECTURE

1.3.1 Addressing

The native instruction set is capable of operating on a wide range of data types including an 8-bit byte, 16-bit word, 32-bit longword, etc. However, the basic addressable unit in the VAX-11/780 system is the 8-bit byte. The 32-bit virtual address will identify a particular byte location and the data type specified by the instruction will identify the number of bits to be treated as a unit in the operation.

The 4 billion bytes of virtual address space are allocated as shown in Figure 1-2. The upper half is designated system virtual space and shared by all processes. Note that the highest quarter of virtual space is reserved for future use. The lower half of virtual address space is designated for process space and contains information relating to the current process. Per process space is further divided into program space and control space. Program space contains process images currently executing on the system and control space contains user stacks and I/O buffers for the current process.

The virtual addresses generated are translated into physical addresses under operating system control. Each 30-bit physical address corresponds to an actual location in main memory. The physical address space (Figure 1-3) is divided in half. Device control registers are assigned to the upper half and the lower half is reserved for primary memory. A detailed explanation of the address translation process is provided in the TB/CACHE/SBI technical description (refer to Table 1-1).

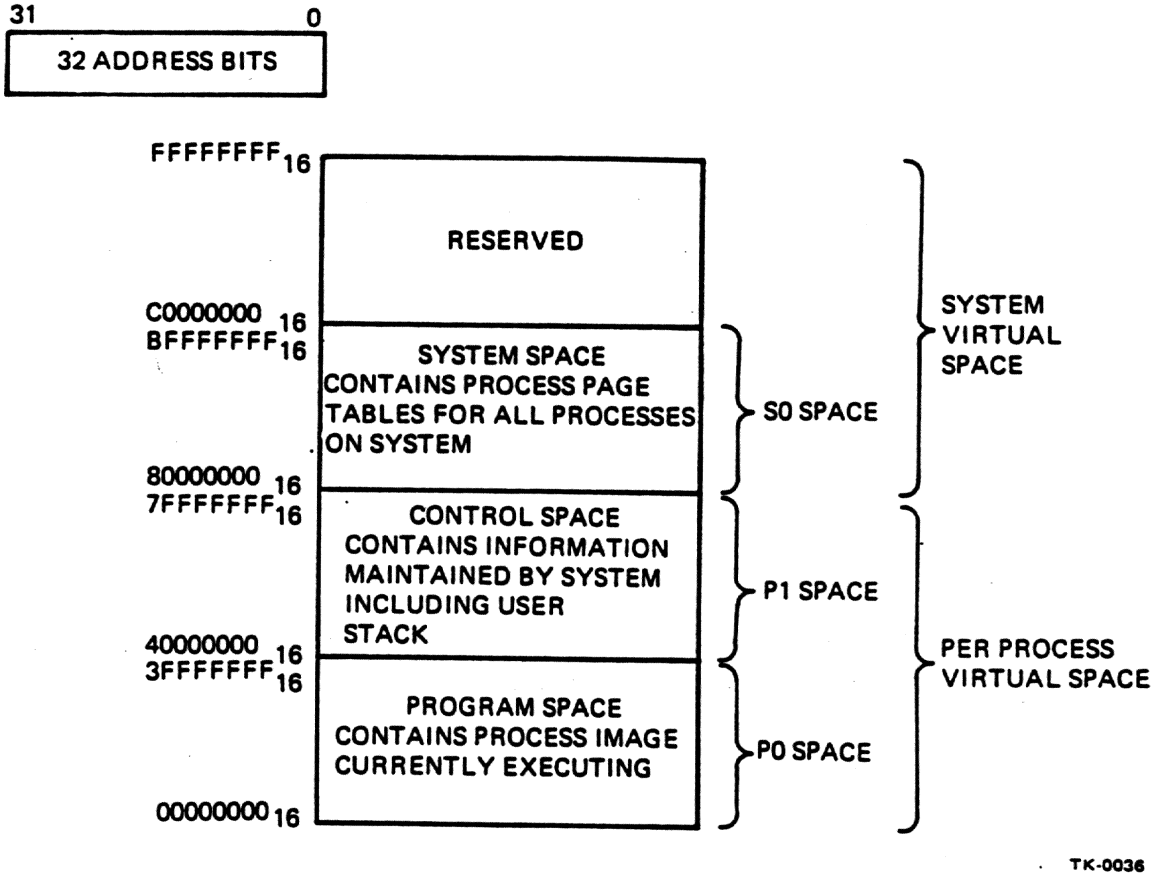


Figure 1-2 Virtual Address Space

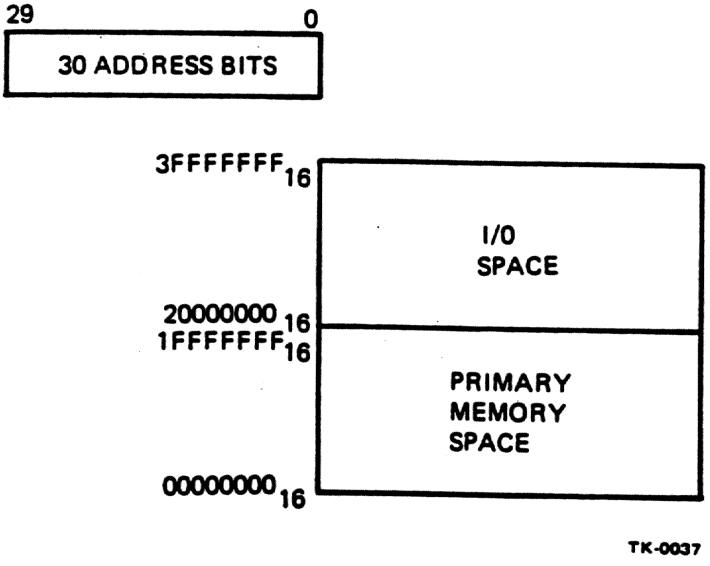


Figure 1-3 Physical Address Space

1.3.2 Data Types

The native instruction set operates on several data types of different sizes and formats including the following;

a. Integer
   Byte
   Word
   Longword
   Quadword
b. Floating/Double Floating
c. Variable Length Bit Field
d. Character String
e. Decimal String
   Trailing Numericl String
   Leading Seperate Numeric String
   Packed Decimal String

The following paragraphs provide a brief description of each of the data types listed above.

1.3.2.1 Integer -- Four integer data types can be specified, each of which represents quantities in a binary format. The quantities can be treated as unsigned or signed (represented in twos complement form). The integer data types are termed byte (8 bits), word (16 bits), longword (32 bits) and quadword (64 bits). Refer to Figure 1-4.

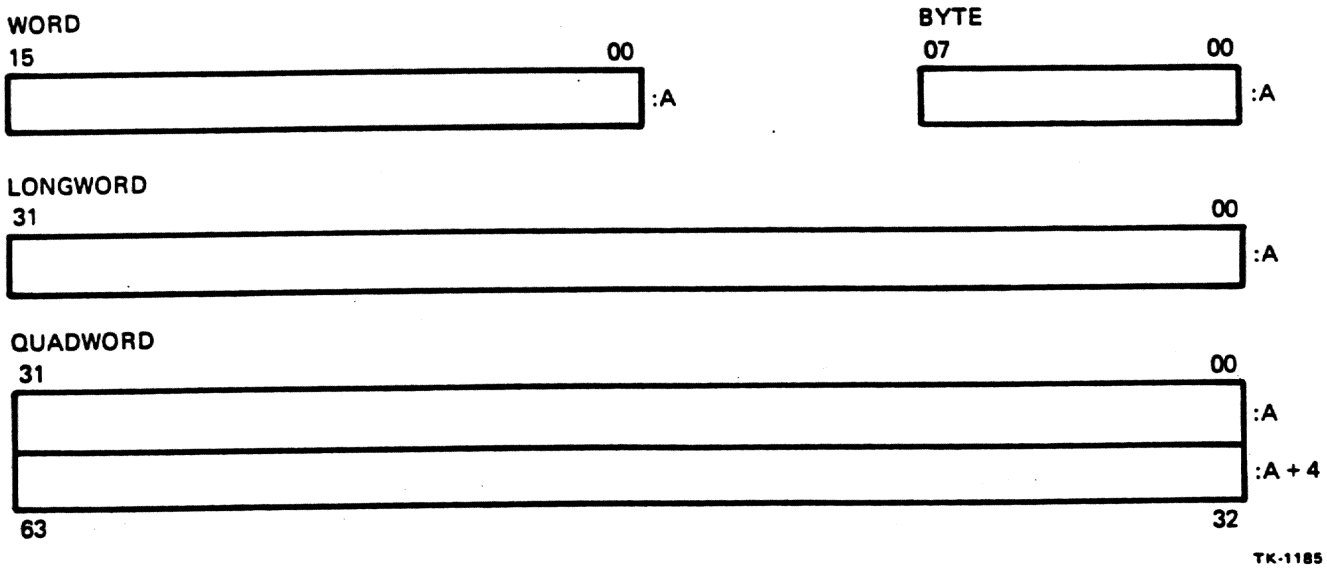


Figure 1-4 Integer Data Formats

Each of the data types is specified by its address A, the address of the byte containing bit Ø. When interpreted arithmetically, the byte, word, longword, or quadword is a 2's complement integer with bit Ø as the least significant bit. The sign of a byte, word, longword, or quadword is bit 7, bit 15, bit 31 or bit 64, respectively. Each of these data types can also be interpreted as an unsigned integer. The following gives the range of integer values that can be represented by each of the data types.

Integer Data Type	Size	Range Signed	Unsigned
Byte	8 bits	-128 to +127	Ø to 255
Word	16 bits	-32768 to +32767	Ø to 65535
Longword	32 bits	-2^{31} to $+2^{31}-1$	Ø to $2^{32}-1$
Quadword	64 bits	-2^{63} to $+2^{63}-1$	Ø to $2^{64}-1$

1.3.2.2 Floating-Point -- The floating-point data types are used to represent approximations to quantities for which scaling is not specified in the program. Floating-point data is stored in scientific notation as a power of two times a fraction in the range of Ø.5 (inclusive) to 1.Ø (exclusive). The data format consists of three fields: the sign, the power of two exponent, and the fractional magnitude. The VAX-11/78Ø provides two types of floating-point data: single precision (32 bits) and double precision (64 bits). These are termed floating and double floating, respectively.

A floating datum is 4 contiguous bytes, specified by its address A (the address of the byte containing bit Ø) and formatted as shown in Figure 1-5.

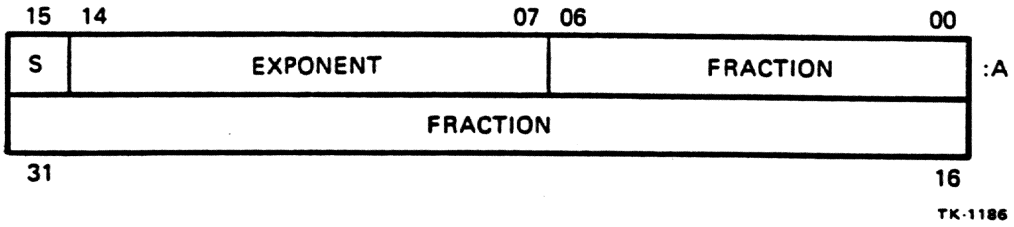


Figure 1-5 Floating Data Formats

The form of a 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 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 a floating datum is in the approximate range $.29*10^{-38}$ through $1.7*10^{38}$. The precision of a floating datum is approximately one part in 2^{23}, i.e., typically 7 decimal digits.

A double floating datum is 8 contiguous bytes, specified by its address A (the address of the byte containing bit 0) and formatted as shown in Figure 1-6.

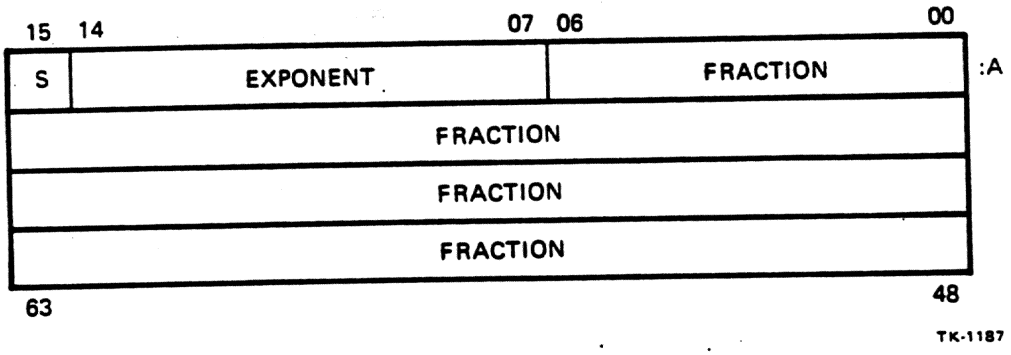


Figure 1-6 Double Floating Data Formats

The format of a double floating datum is the same as the floating datum except for an additional 32 low significance fraction bits. Within the fraction, bits of increasing significance go from 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 double floating. The precision of a double floating datum is approximately one part in 2^{55}, i.e., typically 16 decimal digits.

1.3.2.3 Variable Length Bit Field -- The variable length bit field is a data type used to store small integers packed together in a larger data structure. This saves memory when many small integers are part of a larger structure. A specific case of the variable bit field is that of one bit. This form is used to store and access individual flags efficiently.

A variable bit field is 0 to 32 contiguous bits located arbitrarily with respect to byte boundaries. A variable bit field is specified by three characteristics; the address (A) of a byte, the bit position (P) that is starting location of the field with respect to bit 0 of the byte at A, and the size S of the field. Figure 1-7 illustrates the format of the bit field where the field is the shaded area.

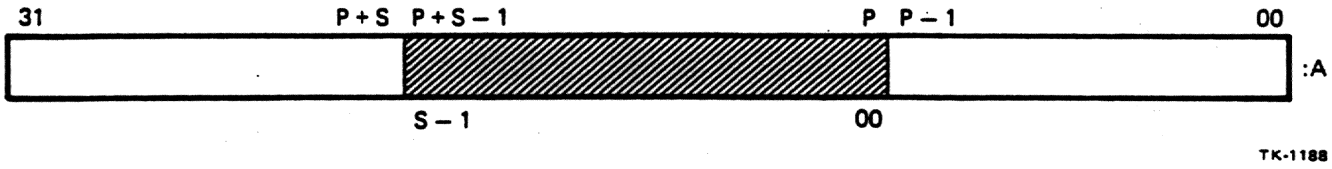


Figure 1-7 Bit Field Format

The VAX-11/780 field instruction provides for the interpretation of a field as a signed or unsigned integer. When interpreted as a signed integer, it is represented in two complement form with bits of increasing significance going from 0 through S-2 with bit S-1 as the sign bit. When interpreted as an unsigned integer, bits of increasing significance go from 0 through S-1.

1.3.2.4 Character String -- The character string is a data type used to represent strings of characters such as names, data records or text. Typical operations include copying, concatenating, searching, and translating the string rather than arithmetic or logical operations.

A character string is a contiguous sequence of bytes in memory specified by the address (A) of the first byte of the string and the length (L) of the string in bytes. Figure 1-8 illustrates the format of a character string.

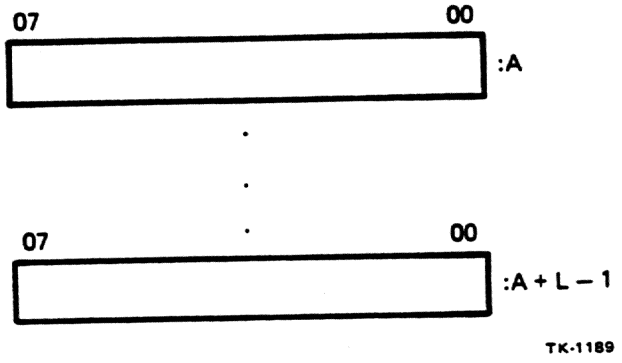


Figure 1-8 Character String Format

The length L of a string is in the range Ø through 65,535. A string with length Ø is termed a null string.

1.3.2.5 Decimal String -- The decimal string data types are used to store scaled quantities in a form that closely resembles their external representation. This data representation is used frequently in programs that simply move or transfer the information rather than perform computation on the information. The decimal string data types include formats in which each decimal digit occupies one byte (numeric string) and a more compact form in which two decimal digits occupy one byte (packed decimal string). The numeric string form is used to represent many external data arrangements exactly and therefore appears in several representations. The most significant difference between the representations is whether the sign, if any, appears before the first digit or whether it is superimposed on the final digit. These are termed leading seperate and trailing numeric strings respectively.

1.3.2.5.1 **Trailing Numeric String** -- A trailing numeric string is a contiguous sequence of bytes in memory specified by two characteristics: the address A of the first byte (most significant digit) of the string, and the length L of the string in bytes.

All bytes of a trailing numeric string, except the last (least significant digit), must contain an ASCII decimal digit character. The representation for digits in all bytes, except the last, is shown as follows:

digit	decimal	hex	ASCII character
0	48	30	0
1	49	31	1
2	50	32	2
3	51	33	3
4	52	34	4
5	53	35	5
6	54	36	6
7	55	37	7
8	56	38	8
9	57	39	9

The highest address byte of a trailing numeric string represents an encoding of both the least significant digit and the sign of the numeric string. The VAX-11 numeric string instructions support any encoding; however there are three preferred encodings used by DIGITAL software. These are (1) unsigned numeric in which there is no sign and the least significant digit contains an ASCII decimal digit character, (2) zoned numeric, and (3) overpunched numeric. Because the overpunch format has been used by compilers of many manufacturers over many years, and because various card encodings are used, several variations in overpunch format have evolved. Typically, these alternate forms are accepted on input. The valid representations of the digit and sign in each of the later two formats is shown in Table 1-2.

Table 1-2 Representation of Least Significant Digit and Sign

	Zoned Numeric Format			Overpunch Format			
digit	decimal	hex	ASCII char.	decimal	hex	ASCII norm	char. alt.
Ø	48	3Ø	Ø	123	7B	{	[?
1	49	31	1	65	41	A	a
2	5Ø	32	2	66	42	B	b
3	51	33	3	67	43	C	c
4	52	34	4	68	44	D	d
5	53	35	5	69	45	E	e
6	54	36	6	7Ø	46	F	f
7	55	37	7	71	47	G	g
8	56	38	8	72	48	H	h
9	57	39	9	73	49	I	i
-Ø	112	7Ø	p	125	7D	}	] ! :
-1	113	71	g	74	4A	J	j
-2	114	72	r	75	4B	K	k
-3	115	73	s	76	4C	L	l
-4	116	74	t	77	4D	M	m
-5	117	75	u	78	4E	N	n
-6	118	76	v	79	4F	O	o
-7	119	77	w	8Ø	5Ø	P	p
-8	12Ø	78	x	81	51	Q	q
-9	121	79	y	82	52	R	r

The length L of a trailing numeric string is in the range of Ø to 31 bytes (Ø to 31 digits). The value of a Ø length string is identically Ø. The address A of the string specifies the byte of the string containing the most significant digit. Digits of decreasing significance are assigned to increasing addresses. Figure 1-9 illustrates the representation of both a positive and negative value in a trailing numeric string format.

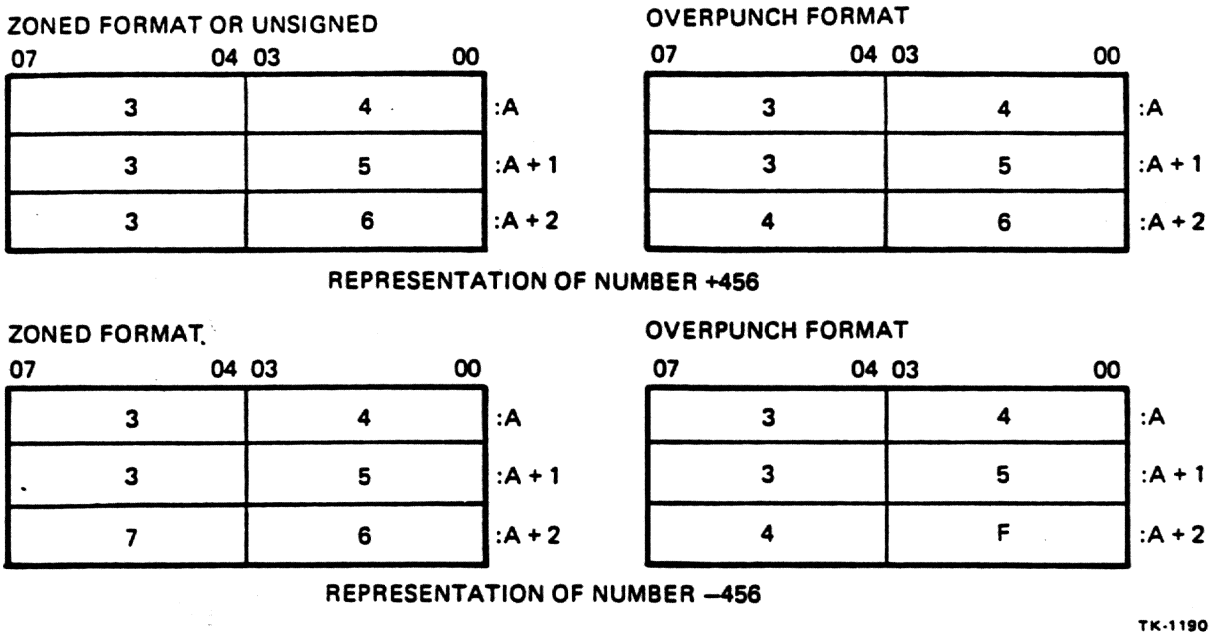


Figure 1-9 Trailing Numeric String Formats

1.3.2.5.2 Leading Separate Numeric String -- A leading separate numeric string is a contiguous sequence of bytes in memory specified by two characteristics; the address A of the first byte (containing the sign character) and a length L that is the length of the string in digits. Note that L is not the length of the string in bytes. The number of bytes in a leading separate numeric string is L + 1.

The sign of a leading separate numeric string is stored in a seperate byte. The following shows valid signs and their byte representation:

sign	decimal	hex	ASCII character
+	43	2B	+
+	32	20	<blank>
-	45	2D	-

All bytes other than the sign byte will contain an ASCII digit character which are represented as follows:

digit	decimal	hex	ASCII character
Ø	48	3Ø	Ø
1	49	31	1
2	5Ø	32	2
3	51	33	3
4	52	34	4
5	53	35	5
6	54	36	6
7	55	37	7
8	56	38	8
9	57	39	9

The length L of a leading separate numeric string is in the range of Ø to 31 bytes (Ø to 31 digits). The value of a Ø length string is identically Ø and contains only the sign bit.

The address A of the string specifies the byte of the string containing the sign. Digits of decreasing significance are assigned to bytes of increasing addresses. Figure 1-1Ø illustrates the representation of both a positive and negative value in a leading separate numeric string format.

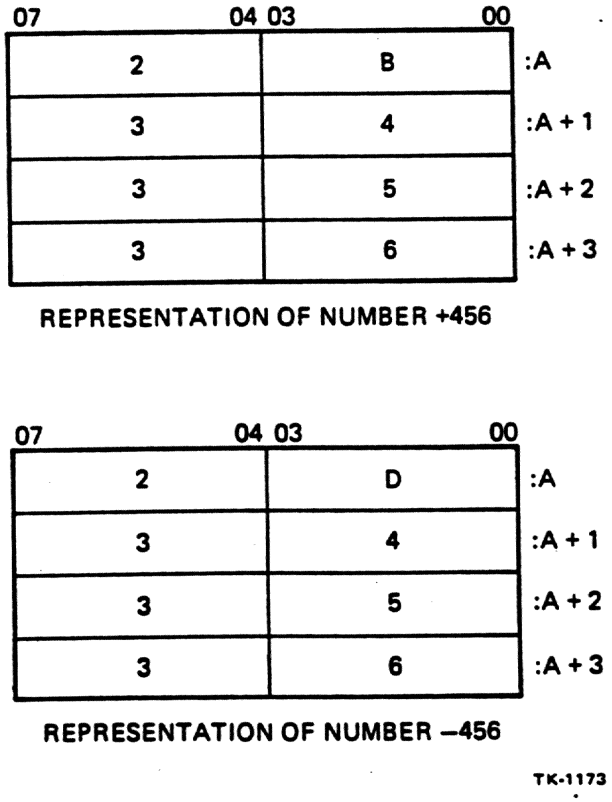


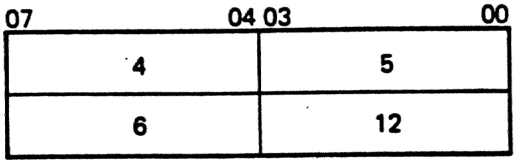
Figure 1-1Ø Leading Separate Numeric String Format

1.3.2.5.3 Packed Decimal String -- A packed decimal string is a contiguous sequence of bytes in memory specified by two characteristics: the address A of the first byte of the string and the length L that is the number of digits in the string. Note that L is not the number of the string in bytes. The bytes of a packed decimal string are divided into two 4-bit fields (nibbles). Each nibble contains a decimal digit, except the low nibble (bits 3:0) of the last (highest addressed) byte which must contain a sign.

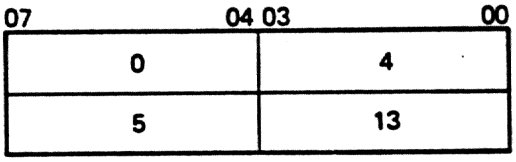
The representation for the digits and sign is as follows:

digit or sign	decimal	hex
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9
+	10,12,14, or 15	A,C,E, or F
-	11 or 13	B or D

The preferred sign representation is 12 for "+" and 13 for "-". The length L is the number of digits in the packed decimal string (not including the sign) and is in the range of 0 through 31. The address A of the string specifies the byte of the string containing the most significant digit in its high nibble. Digits of decreasing significance are assigned to increasing byte addresses and from high nibble to low nibble within a byte. Figure 1-11 illustrates the representation of two values; one value containing an odd number of digits and the other value containing an even number of digits. Note that if the number of digits is even, an extra "0" digit is required in the high nibble (bits 7:4) of the first byte of the string.



REPRESENTATION OF VALUE (+456) WITH ODD NUMBER OF DIGITS



REPRESENTATION OF VALUE (–45) WITH EVEN NUMBER OF DIGITS

TK-1174

Figure 1-11 Packed Decimal String Format

1.3.3 General Registers

The VAX-11/780 provides sixteen general purpose registers, designated R15 through R0. These registers can be used for temporary storage of data and addresses. These registers are accessed when the register number is explicitly identified in an operand specifier or when a register is implicitly identified by the machine operation. Certain general registers are always used by software for a particular purpose and are denoted as follows;

PC — R15 is the Program Counter (PC). The PC points to the next byte of the program and is updated by the processor as the program progresses. The PC cannot be used as a temporary register, an accumulator, or an index register.

SP — R14 is the Stack Pointer (SP). Several instructions make implicit references to SP, and most software assumes that SP points to memory set aside for use as a stack. There is no restriction on the explicit use of other registers (except PC) as stack pointers, though those instructions which make implicit references to the stack always use SP.

FP — R13 is the Frame Pointer (FP). The VAX-11 procedure cal convention builds a data structure on the stack called stack frame. The CALL instructions load FP with the bas address of the stack frame, and the RETurn instructio depends on FP containing the address of a stack frame Further, VAX-11 software depends on maintenance of F for correct reporting of certain exceptional conditions

AP — R12 is the Argument Pointer (AP). The VAX-11 procedur call convention uses a data structure called an argumen list, and uses AP as the base address of the argumen list. The CALL instructions load AP in accordance wit that convention, but there is no hardware or softwar restriction on the use of AP for other purposes.

R6-R11 — Registers R6 through R11 have no special significanc either to hardware or the operating system. Specifi software will assign specific uses for each register.

R0-R5 — Registers R0 through R5 are generally available for an use by software, but are also loaded with specifi values by those instructions whose execution must b interruptable--the character string, decimal arithmetic RC, and POLY instructions. The specific instructio descriptions identify which registers are used, and wha values are loaded into them.

1.3.4 Stacks

Stacks, also called pushdown lists or last-in first-out queues are used in the VAX-11/780 as follows:

a. At the entry to a subroutine, the general register (including the PC) are saved so that they can be restore at exit from the subroutine.

b. The PC, PSL, and general registers are saved at the time of interrupts and exceptions, and during context switches.

c. The stacks are used to create storage space for temporar use or for nesting of recursive routines.

A stack is defined by a block of memory and a general register (stack pointer) which addresses the top of the stack. The top of the stack is that memory location which will be read when an item is removed from the stack. An item is added to the stack (pushed on) by first decrementing the stack pointer and then storing the item at the address contained in the updated stack pointer. The pointer is decremented by the length of item added to the stack so that there is sufficient room to store it.

When an item is removed (popped off) from the stack, the length of the item is added to the stack pointer. These operations are built into the basic addressing mechanisms of the VAX-11/780 instructions.

Many processor operations make use of the stack implicitly without identifying the SP in an operand specifier. This occurs in instructions used in calling and returning from subroutines, and in processor sequences which initiate and terminate interrupt or exception service routines. In this case, the processor uses the stack addressed by R14. This does not mean that exceptions, interrupts, and system services are performed on the same stack as is used by user-mode programs. The processor maintains five internal registers as pointers to separate blocks of memory to be used as stacks, and uses one or another as SP depending on the current access mode and interrupt stack bit in the processor status longword. Whenever the current access mode and/or interrupt stack bits change, the processor saves the contents of SP into the internal register selected by the old value of those bits, and loads SP from the register selected by the new value. There is one interrupt stack for the entire system, but the kernel, executive, supervisor, and user mode stacks are different for each process in the system. Figure 1-12 illustrates the relationship between the five stacks and multiple processes.

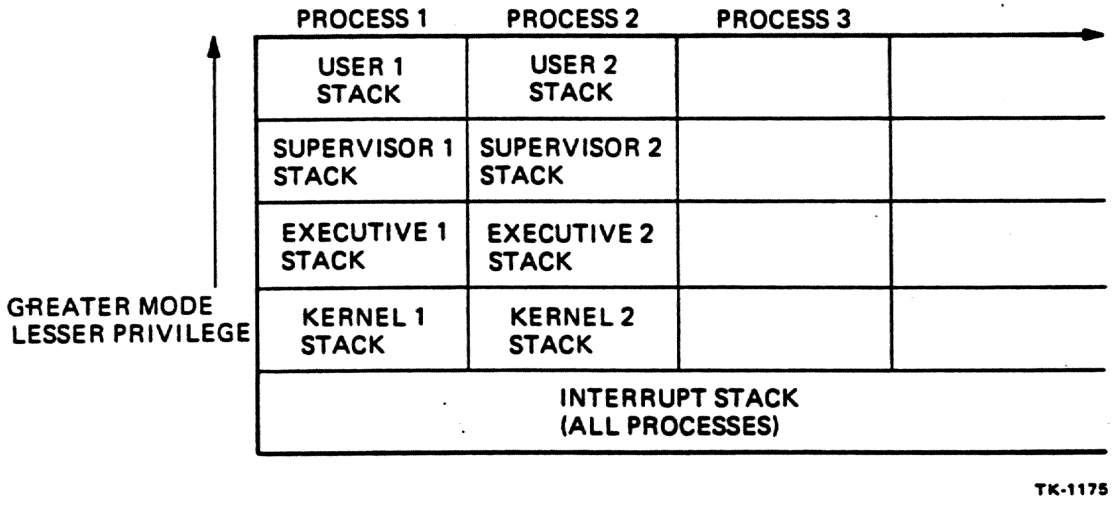


Figure 1-12 Relationship Between Stacks and Processes

The multiple-stack mechanism provides the following advantages over a single stack:

a. User mode programs are not subject to sudden and non-reproducible changes in the data beyond the end of their stack.

b. The integrity of a privileged mode program cannot be compromised by a less privileged caller. Even if the caller has completely filled its own stack, the privileged code is in no danger if running out of space because separate blocks of memory are allocated to the stack associated with each mode.

c. Privileged mode programs are not vulnerable to accidental destruction of the stack pointer by less privileged programs.

1.3.5 Processor Status Longword (PSL)

There are a number of processor state variables associated with each process, which are grouped together into the 32-bit Processor Status Longword or PSL. Bits 15-Ø of the PSL are referred to seperately as the Processor Status Word (PSW). The PSW contains unprivileged information, and those bits of the PSW which have defined meaning are freely controllable by any program. Bits 31-16 of the PSL contain privileged status, and while any program can perform the REI instruction (which loads PSL), REI will refuse to load any PSL which would increase the privilege of a process, or create an undefined state in the processor. Figure 1-13 shows the format of the Processor Status Longword.

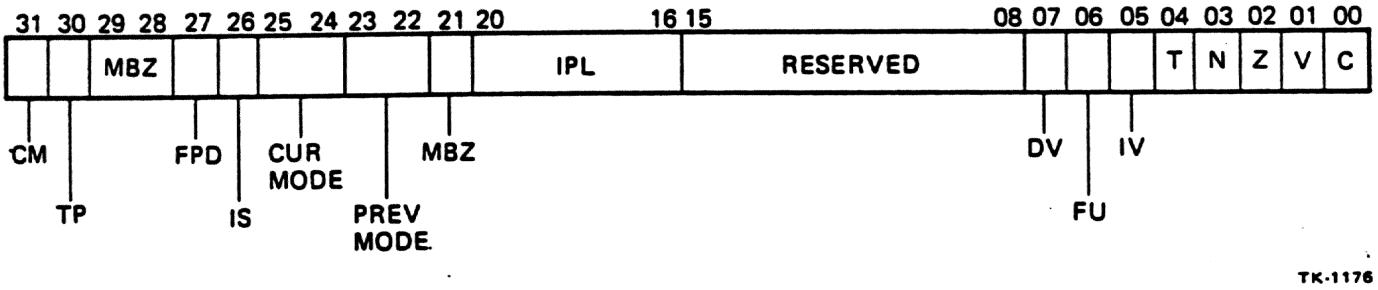


Figure 1-13 Processor Status Longword

Bits 3:Ø of the PSL are termed the condition codes. These bits are used to reflect the status of the result of the most recent instruction. Refer to the VAX-11/78Ø Architecture Handbook for details as to how each individual instruction affects the condition codes. The following provides of brief description of the condition codes:

N--Bit 3 is the Negative condition code; in general it is set by instructions in which the result stored is negative, and cleared by instructions in which the result stored is positive or zero. For those instructions which affect N according to a stored result, N reflects the actual result, even if the sign of the result is algebraically incorrect as a result of overflow.

Z--Bit 2 is the Zero condition code; in general it is set by instructions which store a result that is exactly zero, and cleared if the result is not zero. Again, this reflects the actual result, even if overflow occurs.

V--Bit 1 is the oVerflow condition code; in general it is set after arithmetic operations in which the magnitude of the algebraically correct result is too large to be represented in the available space, and cleared after operations whose result fits. Instructions in which overflow is impossible or meaningless either clear V or leave it unaffected. Note that all overflow conditions which set V can also cause traps if the appropriate trap enable bits are set.

C--Bit Ø is the Carry condition code; in general it is set after arithmetic operations in which a carry out of, or borrow into, the most significant bit occurred. C is cleared after arithmetic operations which have no carry or borrow, and either cleared or unaffected by other instructions. The C bit is unique in that it not only determines the operation of conditional branch instructions, it also serves as an input variable to the ADWC (Add with Carry) and SBWC (Subtract with Carry) instructions used to implement multiple-precision arithmetic.

Bits 4-7 of the PSL are trap enable flags which cause traps to occur under special circumstances and are described as follows:

T--Bit 4 is the Trace bit; when set, it causes a trace trap to occur after execution of the next instruction. The facility is used by debugging and performance analysis software to step through a program one instruction at a time.

IV--Bit 5 is the Integer oVerflow trap enable; when set, it causes an integer overflow trap after any instruction which produced an integer result that could not be correctly represented in the space provided. When bit 5 is clear, no integer overflow trap occurs. The V condition code is set independently of the state of IV (bit 5).

FU--Bit 6 is the Floating Underflow Trap enable. When set, it causes a decimal overflow trap after the execution of any instruction which produces a decimal result whose absolute value is too large to be represented in the destination space provided. When DV is clear, no decimal overflow trap occurs. The result stored consists of the low-order digits and sign of the algebraically correct result.

NOTE

There are other trap conditions for which there are no enable flags-division by zero and floating overflow.

Bits 8-15 of the PSL are unused and reserved.

As previously mentioned, bits 31-16 of the PSL contains privileged status and are described below:

IPL--Bits 16-20 represent the processor's Interrupt Priority Level. An interrupt, in order to be acknowledged by the processor, must be at a priority higher than the current IPL. Virtually all software runs at IPL 0, so the processor acknowledges and services interrupt requests at any priority. The interrupt service routine for any request, however, runs at the IPL of the request, thereby temporarily blocking interrupt requests of lower or equal priority. There are 31 priority levels above zero, numbered in hex 01 through 1F. Interrupt levels 01 through 0F exist entirely for use by software. Levels 10 through 17 are for use by peripheral devices and their controllers, though present systems support only 14 through 17. Levels 18 to 1F are for use for urgent conditions, including the interval clock, serious errors, and power fail.

Previous Mode--Bits 22-23 represents the previous mode field, which contains the value from the current mode field at the most recent exception which transferred from a less privileged mode to this one. Previous mode is of interest only in the PROBE instructions, which enable privileged routines to determine whether a caller at the previous mode is sufficiently privileged to reference a given area of memory.

Current Mode--Bits 24-25 present the current mode field, which determines the privilege level of the currently executing program. The values of mode are:

0--Kernel; most privileged, including the ability to perform all instructions
1--Executive
2--Supervisor
3--User; least privileged

Privilege is granted in two ways by the mode field. Certain instructions (HALT, Move To Processor Register, and Move From Processor Register) are not performed unless the current mode is kernel. The memory management logic controls access to virtual addresses on the basis of the program's current mode, the type of reference (read or write), and the protection code assigned to each page of the address space.

IS--Bit 26 is the Interrupt Stack flag, which indicates that the processor is using the special interrupt stack rather than one of the four stacks associated with the current mode. When IS is set, the current mode is always kernel; thus software operating on the interrupt stack has full kernel-mode privileges.

FPD--Bit 27 is the First Part Done flag, which the processor uses in certain instructions which may be interrupted or page faulted in the middle of their execution.

If FDP is set when the processor returns from an exception or interrupt, it resumes the interrupted operation where it left off, rather that restarting the instruction.

TP--Bit 30 is the Trace Pending bit, which is used by the processor to ensure that one, and only one, trace trap occurs for each instruction performed with the Trace bit (bit 4) set.

CM--Bit 31 is the Compatibility Mode bit. When CM is set, the processor is in PDP-11 compatibility mode, and executes PDP-11 instructions. When CM is clear, the processor is in native mode, and executes VAX-11 instructions.

1.4 INSTRUCTION FORMATS

The VAX-11/780 executes both variable length native mode (VAX-11) instructions and fixed length compatibility mode (PDP-11) instructions. The compatibility mode instructions are in the standard 16-bit, PDP-11 format and are stored in two contiguous bytes in memory.

The native mode instructions vary in length and format depending on the type of instruction and addressing mode used. Figure 1-14 illustrates the general format of a VAX-11 instruction.

Figure 1-14 General Format of VAX-11 Instructions

The presently available instruction set uses a one byte operation code (op code). An instruction may consist of an opcode alone or may consist of an op code and multiple operand specifiers. The operand specifier indicates the manner (addressing mode and register information) in which the operand is to be accessed. Certain addressing modes require an extension to be appended to the operand specifier. The specifier extension can be used as a displacement or can be immediate data. Immediate denotes that the data or address immediately follows the operand specifier.

1.4.1 Op Code

The op code of each instruction specifies the operation to be performed and the number of operands to be used in the operation. The data and access types of each operand are also dictated by the op code. The op code of each instruction is listed in Appendix A. The following lists the possible data and access types of each operand:

- Byte--8-bits
- Word--16-bits
- Longword--32-bits
- Floating--32-bit single-precision floating point (same as longword for addressing mode considerations).
- Quad word--64-bit
- Double--64-bit double-precision floating point (same as quad word for addressing mode considerations).

An operand may be accessed in one of the following ways:

- Read--The specified operand is read only.
- Write--The specified operand is written only.
- Modify--the specified operand is read, may or may not be modified and is written.
- 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 the address calculation. The specified operand is not accessed directly although the instruction may subsequently use the address to access that operand.
- Variable field--If just Rn is specified, the field is in the general register (R[n]). Otherwise, address calculation occurs until the actual address of the operand is obtained. This address specified the base to which the field position (offset) is applied.
- Branch--No operand is accessed. The operand specifier itself is a branch displacement. In this specifier, the data type indicates the size of the branch displacement.

1.4.2 Operand Specifier

The operand specifier provides the information required to locate the operand. In literal modes, the operand specifier actually includes the operand value. The format of the operand specifier includes the addressing mode and any register designators that are required. In certain addressing modes, the operand specifier is extended with additional data. The specifier extension can be used as displacement data, immediate data, or an absolute address. The format of the operand specifiers are shown with each associated addressing mode in paragraphs 1.5.2.1.1 through 1.5.2.2.4.

1.5 NATIVE MODE ADDRESSING

Native Mode Addressing can be broadly categorized into branch mode addressing and general mode addressing.

1.5.1 Branch Mode Addressing

The two types of addressing modes used with branch instructions are termed byte displacement and word displacement. Figure 1-15 illustrates the operand specifier formats used with each of the two addressing modes.

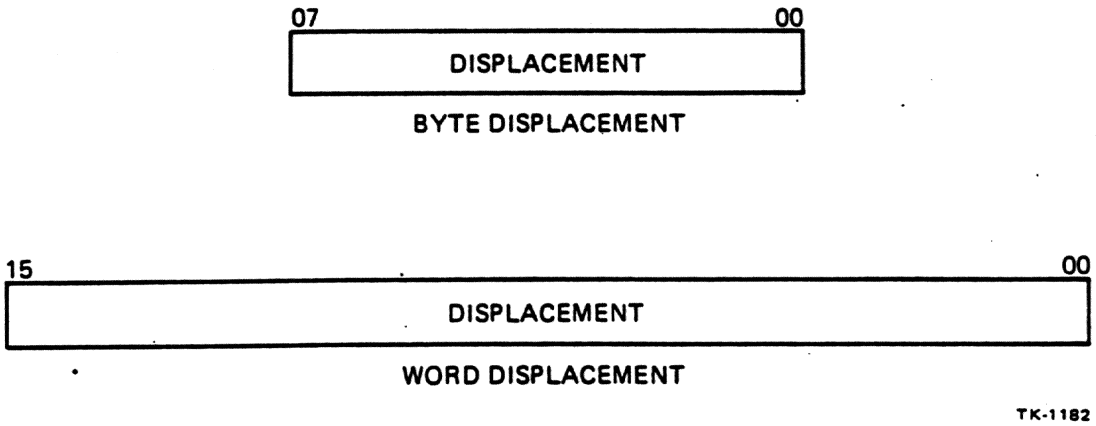


Figure 1-15 Operand Specifier Formats For Branch Mode Addressing

1.5.2 General Mode Addressing

General mode addressing implements the processor's sixteen general purpose registers. The only general addressing mode which does not use a general purpose register is literal mode, in which the operand actually contained in the operand specifier.

Table 1-3 summarizes the addressing modes, listing the value of the mode specifier, the assembler notation, the modes which may be indexed, and the access types which may be used with each mode. Also shown is the result of using the program counter (R15) or stack pointer (R14) in each of the general addressing modes. The program counter addressing modes use R15 (PC) as the general register.

1.5.2.1 General Register Addressing -- General register mode addressing excludes use of the PC (R15) in the operand specifier but implements the remaining fifteen general purpose registers (RØ through R15). When these general registers are used for temporary storage or as an accumulator, the data is stored in the register in the same format as it would be in memory. If a quadword or floating datum is stored in a general register, it is actually stored in two adjacent registers.

If the registers are used as pointers, the content of the register is the address of the operand rather than the operand itself. The register is referred to as a base register if it contains the address of a data structure such as a table or queue. The registers can also be used as pointers which automatically step through memory locations. Automatically stepping forward through consecutive locations is called autoincrement addressing and automatically stepping backwards is called autodecrement addressing. These addresssing modes are useful for processing tabular data and manipulating stacks. When the general registers are used as an index register, an offset is generated and added to the base operand address to yield the indexed location. This is called index mode addressing.

The following paragraphs provide a brief description of each general register addressing mode and the associated operand specifier format.

1.5.2.1.1 Register Mode -- In 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.

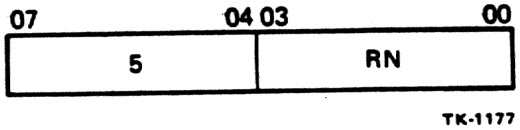


Figure 1-16 Operand Specifier Format in Register Mode

Table 1-3 Summary of Addressing Modes

GENERAL REGISTER ADDRESSING

Hex	Dec	Name	Assembler	r	m	w	a	v	PC	SP	Indexable?
0-3	0-3	literal	S^#literal	y	f	f	f	f	-	-	f
4	4	indexed	i [Rx]	y	y	y	y	y	f	y	f
5	5	register	Rn	y	y	y	f	y	u	uq	f
6	6	register deferred	(Rn)	y	y	y	y	y	u	y	y
7	7	autodecrement	-(Rn)	y	y	y	y	y	u	y	ux
8	8	autoincrement	(Rn)+	y	y	y	y	y	p	y	ux
9	9	autoincrement deferred	@(R)+	y	y	y	y	y	p	y	ux
A	10	byte displacement	B^D (Rn)	y	y	y	y	y	p	y	y
B	11	byte displacement deferred	@B^D (Rn)	y	y	y	y	y	p	y	y
C	12	word displacement	W^D (Rn)	y	y	y	y	y	p	y	y
D	13	word displacement deferred	@W^D (Rn)	y	y	y	y	y	p	y	y
E	14	longword displacement	L^D (Rn)	y	y	y	y	y	p	y	y
F	15	longword displacement deferred	@L^D (Rn)	y	y	y	y	y	p	y	y

PROGRAM COUNTER ADDRESSING

Hex	Dec	Name	Assembler	r	m	w	a	v	Indexable?
8	8	immediate	I^#constant	y	u	u	y	y	y
9	9	absolute	@#address	y	y	y	y	y	y
A	10	byte relative	B^address	y	y	y	y	y	y
B	11	byte relative deferred	@B^address	y	y	y	y	y	y
C	12	word relative	W^address	y	y	y	y	y	y
D	13	word relative deferred	@W^address	y	y	y	y	y	y
E	14	longword relative	L^address	y	y	y	y	y	y
F	15	longword relative deferred	@L^address	y	y	y	y	y	y

D -- displacement
i -- any indexable addressing mode
- -- logically impossible
f -- reserved addressing mode fault
p -- Program Counter addressing
u -- Unpredictable
up -- Unpredictable for quad and double (and field if position + size greater than 32)
ux -- Unpredictable for index register same as base register
y -- yes, always valid addressing mode
r -- read access
m -- modify access
w -- write access
a -- address access
v -- field access

1.5.2.1.2 Register Deferred Mode -- 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.

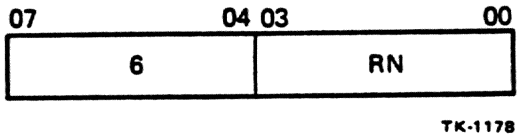


Figure 1-17 Operand Specifier Format in Register Deferred Mode

1.5.2.1.3 Autoincrement Mode -- In autoincrement mode addressing, the contents of Rn contain 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 floating and 8 for quadword or double floating) is added to the contents of register 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. It assumes the contents of the selected general register to be the address of the operand. 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 completely general and may be used for a variety of purposes.

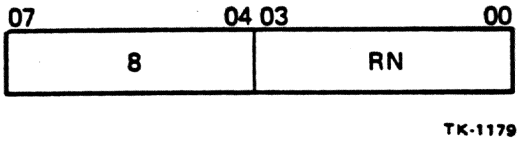


Figure 1-18 Operand Specifier Format in Autoincrement Mode

1.5.2.1.4 Autoincrement Deferred Mode -- In autoincrement deferred addressing, register Rn contains a longword address which is a pointer to the operand address. After the operand address has been determined, 4 is added to the contents of register Rn and the content of register Rn is replaced with the result. The quantity 4 is used since there are 4 bytes in an address.

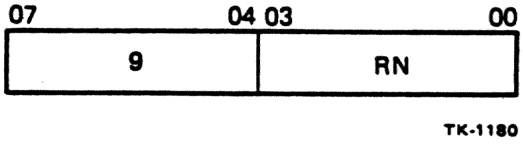


Figure 1-19 Operand Specifier Format in Autoincrement Deferred Mode

1.5.2.1.5 Autodecrement Mode -- In autodecrement mode, the size of the operand in bytes (1 for byte, 2 for word, 4 for longword or floating and 8 for quadword or double) is subtracted from the contents of register Rn and the contents of register Rn are replaced by the result. The updated contents of register Rn are the address of the operand. The contents of the selected general register are decremented and then used as the address of the operand.

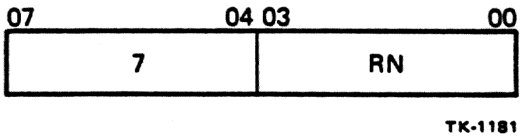


Figure 1-20 Operand Specifier Format in Autodecrement Mode

1.5.2.1.6 Displacement Mode -- 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. This mode is the equivalent of index mode in the PDP-11 addressing.

The VAX-11 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 substantial saving of space (that is, fewer bits are required).

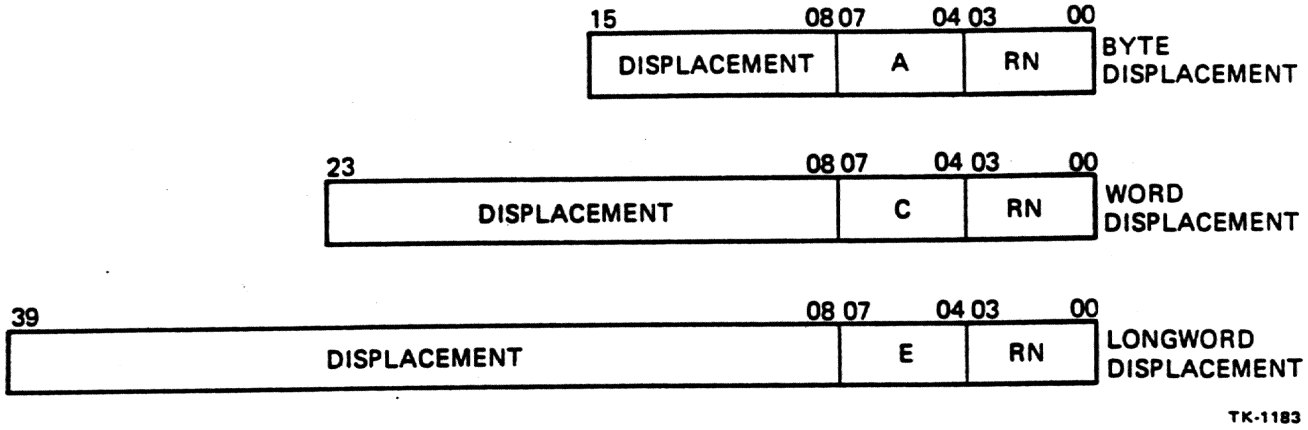


Figure 1-21 Operand Specifier Format in Displacement Mode

1.5.2.1.7 Displacement Deferred Mode -- In displacement deferred mode addressing, the displacement (after being sign-extended to 32 bits if it is a byte or word) is added to the contents of the selected general register and the result is a longword address of the operand address.

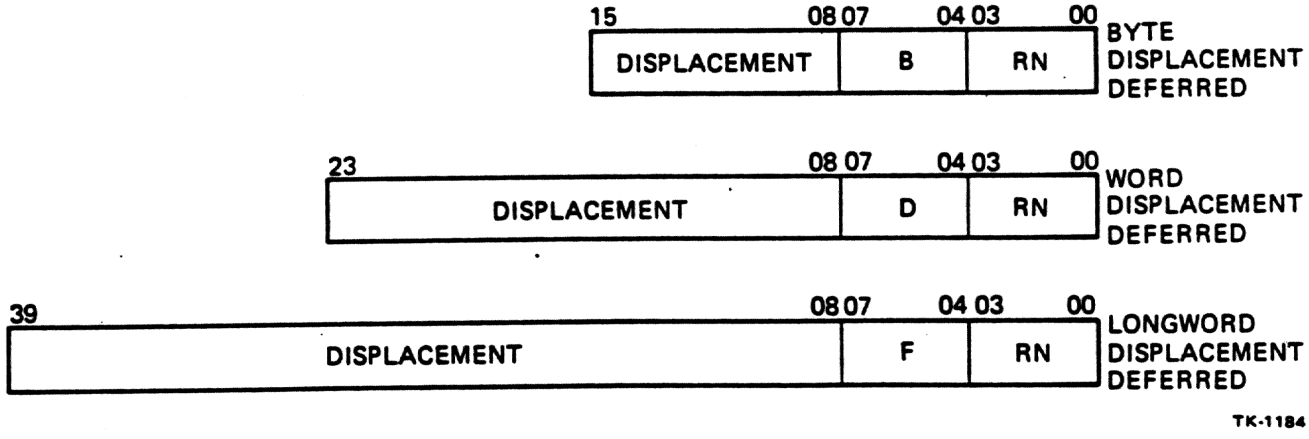


Figure 1-22 Operand Specifier Format in Displacement Mode

1.5.2.1.8 Index Mode -- In index mode, the operand specifier consists of at least two bytes--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 index register Rx by the size of the primary operand in bytes (1 for byte, 2 for word, 4 for longword or floating, and 8 for quadword or double). 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.

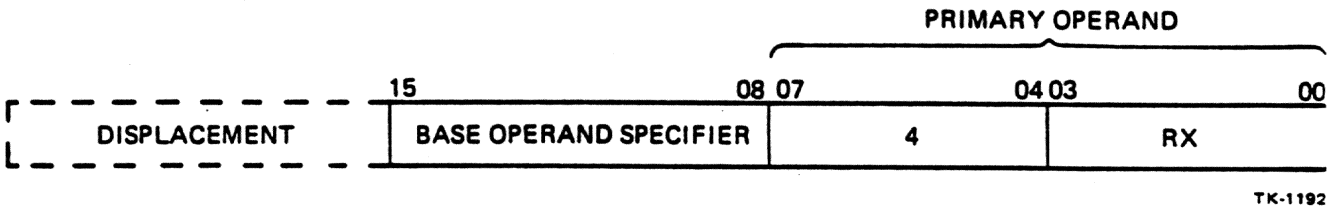


Figure 1-23 Operand Specifier Format in Index Mode

The chief advantage of index mode addressing is to provide very general and efficient accessing of arrays. The VAX-11 architecture provides for context indexing whereby the number in the index register is shifted left by the context of the data type specified (once for byte, twice for word, three times for longword, four times for quadword). This allows loop control variables to be used in the address calculation without first shifting them the appropriate number of times, thus minimizing the number of instructions required.

The following restrictions are placed on the index register (Rx):

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

b. If the base operand specifier uses 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 1-4 lists the various forms of index mode addressing. The name of the addressing mode results from the addressing mode of the base operand specifier. Specifying register, literal, or index mode for the base operand specifier will result in an illegal addressing mode fault. The general register is designated Rn and the indexed register is Rx.

Table 1-4 Index Mode Addressing

MODE	ASSEMBLER NOTATION
Register deferred index	(Rn) [Rx]
Autoincrement indexed Immediate indexed	(Rn) + [Rx] I# constant [Rx] which is recognized by assembler but is not generally useful. Operand address is independent of value of constant.
Autoincrement deferred indexed Absolute indexed	@(Rn) + [Rx] @#address [Rx]
Autodecrement indexed	-(Rn) [Rx]
Byte, word or longword displacement indexed	B^D(Rn) [Rx] W^D(Rn) [Rx] L^D(Rn) [Rx]
Byte, word or longword displacement deferred indexed	@B^D(Rn) [Rx] @W^D(Rn) [Rx] @L^D(Rn) [Rx]

1.5.2.1.9 Literal Mode -- Literal mode provides an efficient means of specifying integer constants in the range of 0 to 63 (decimal). Values in the range of 0 to 63 are called short literals. Values above 63 (long literals) can be obtained using immediate mode (autoincrement mode using PC). The format of the operand specifier is shown in Figure 1-24. The value of the mode specifier (bits 07:04) is 0, 1, 2, or 3, and depends on the value of the short literal (bits 05:00). Bits 07 and 06 of the mode specifier are always zero.

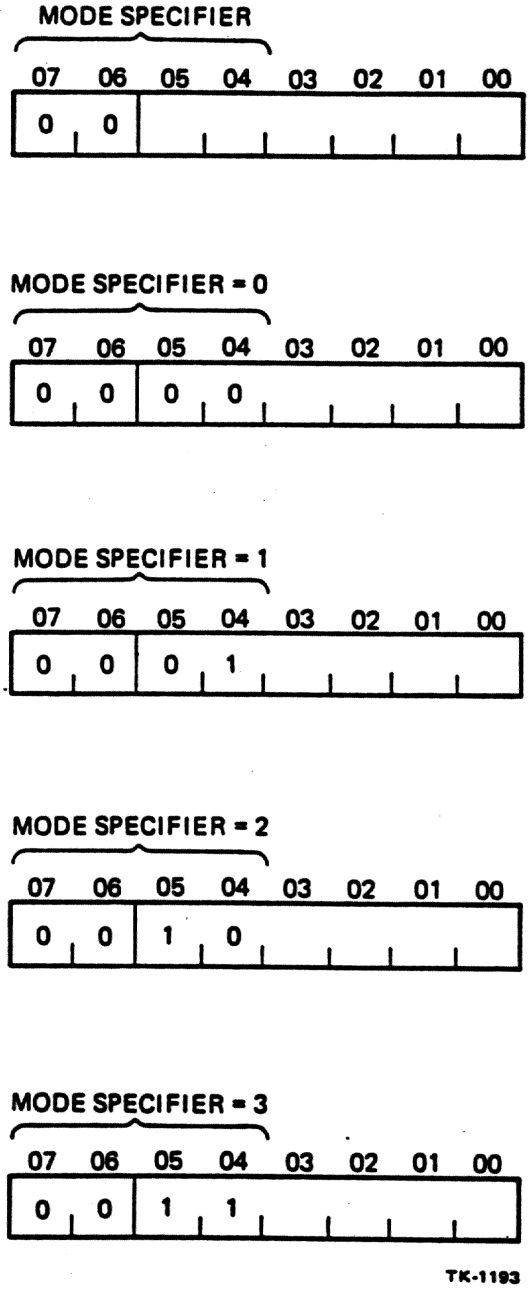


Figure 1-24 Operand Specifier Formats in Literal Mode

Literal mode can also be used to express floating point values (listed in Table 1-5). For floating point operands, the 6-bit literal is composed of a 3-bit exponent (EXP) field and a 3-bit fraction (FRAC) field. Refer to Figure 1-25.

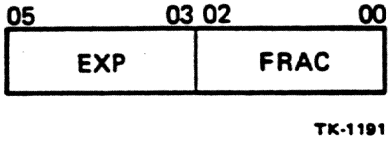


Figure 1-25 Floating Literal Format

The 3-bit EXP field and 3-bit FRAC field are used to form floating or double operands as shown in Figure 1-26. Note that bits 63:32 are not present in single-precision floating point operands.

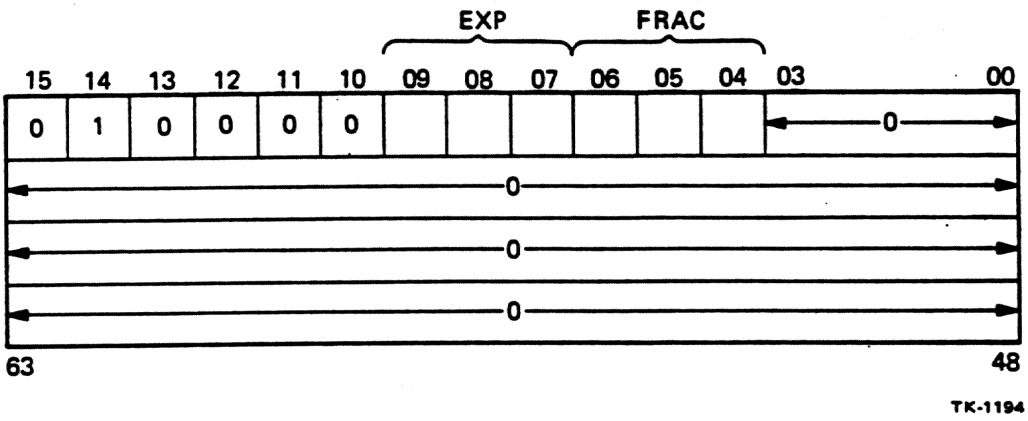


Figure 1-26 Literal Fields in Floating/Double Floating Operands

Table 1-5 lists the possible floating literals that can be expressed in the operand specifier.

Table 1-5 Floating Literals

FRAC EXP	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

1.5.2.2 Program Counter Addressing -- The program counter addressing modes use the PC (R15) as the general register in the operand specifier. Since the program counter is incremented as the instruction is evaluated, the addressing modes have special significance when the PC is used. The PC can be used with all of the general register addressing modes except register or index mode. Refer to Table 1-3 for a list of to program counter addressing modes and associated assembler notation. The following paragraphs provide a brief description of each program counter addressing mode and the associated operand specifier format.

1.5.2.2.1 Immediate Mode -- Immediate mode is autoincrement mode with the PC as the general register. The operand constant is contained in the location immediately following the operand specifier.

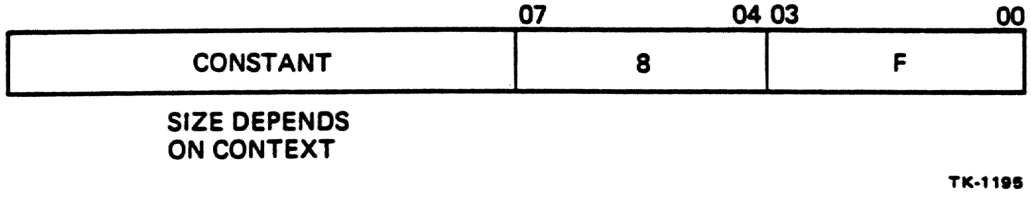


Figure 1-27 Operand Specifier Format in Immediate Mode

1.5.2.2.2 Absolute Mode -- Absolute mode is autoincrement deferred mode with the PC as the general register. The contents of the location following the operand specifier are taken as the operand address. This is interpreted as an absolute address (i.e., an address that remains constant regardless of where in memory the assembled instruction is executed).

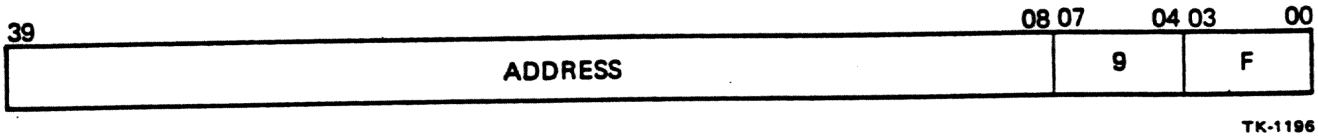


Figure 1-28 Operand Specifier Format in Absolute Mode

1.5.2.2.3 Relative Mode -- Relative mode is displacement mode with the PC as the general register. The displacement which follows the operand specifier is added to the PC and the result is 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.

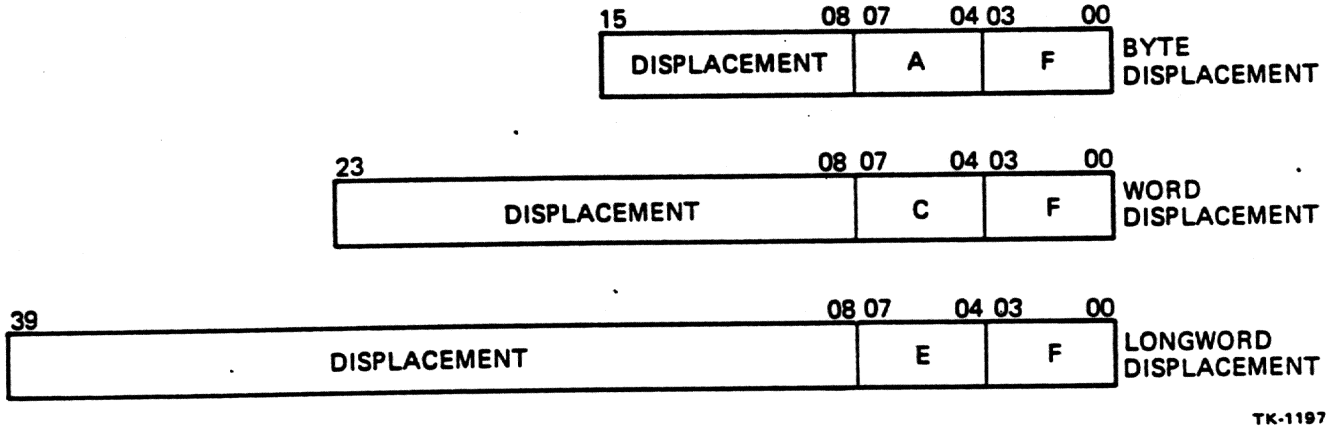


Figure 1-29 Operand Specifier Format in Relative Mode

1.5.2.2.4 **Relative Deferred Mode** -- Relative deferred mode is displacement deferred mode with the PC as the general register. The displacement which follows the operand specifier is added to the PC and the result is a longword address of the address of the operand. This addressing mode is useful when processing tables of addresses.

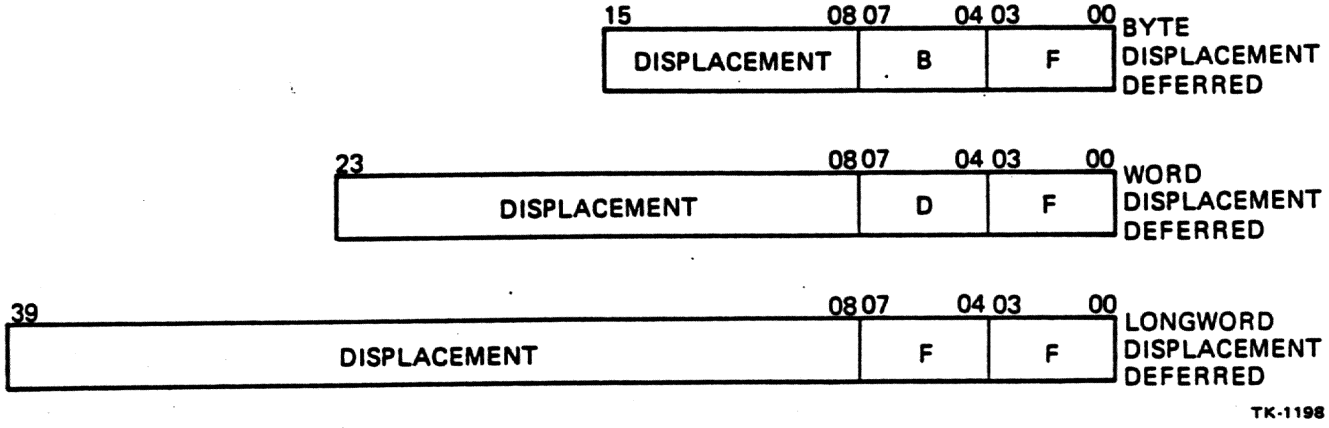


Figure 1-30 Operand Specifier Format in Relative Deferred Mode

1.6 NATIVE MODE INSTRUCTION SET

The VAX-11/780 processor is capable of executing instructions in either of two modes, native (VAX-11) or compatibility (PDP-11). The primary mode of instruction execution is native mode. The variable-length native mode instructions are based on over 200 op codes listed in Appendix A. The following paragraphs provide a brief description of each class of instructions.

1.6.1 Integer and Floating Point Instructions

The logical and arithmetic instructions operate on all available data types. Most of the operations provided for integer data are also provided for floating point and packed decimal data. Exceptions are the strictly logical operations for integer data (e.g., bit clear, bit set, complement), the multiword arithmetic instructions for integer data (e.g., Add/Subtract with Carry, Extended Multiply, and Extended Divide), and the Extended Modules and Polynomial instructions for floating point data.

The arithmetic instructions include both 2-operand and 3-operand forms that eliminate the need to move data to and from temporary operands. The 2-operand instructions store the result in one of the two operands, as in "Set A equal to A plus B." The 3-operand instructions effectively implement the high-level language statements in which two different variables are used to calculate a third, such as "Set C equal to A plus B." The 3-operand instructions are applicable to both integer and floating point data, and equivalent instructions exist for packed decimal data.

Some of the arithmetic instructions are used for extending the accuracy of repeated computations. The Extended Multiply (EMUL) instruction takes longword integer arguments and produces a quadword result. The instruction effectively implements a high-level language statement such as "Set D equal to (A times B) plus C." The Extended Divide (EDIV) instruction divides a quadword integer by a longword and produces a longword quotient and a longword remainder.

The Extended Modulus (EMOD) instructions multiply a floating point number with an extended precision floating point number (extended by eight bits for an effective 9 or 19 digits of accuracy) and return the integer portion and the fractional portion separately. This instruction is particularly useful for preserving the precision of input throughout trigonometic and exponential function evaluation.

The Polynomial Evaluation (POLY) instructions evaluate a polynomial from a table of coefficients using Horner's method. This instruction is used extensively in the high-level languages' math library for operations such as sine and cosine.

The following lists the integer and floating point instructions.

Integer and Floating Point Logical Instructions

MOV_	Move (B, W, L, F, D, Q)*
MNEG_	Move Negated (B, W, L, F, D)
MCOM_	Move Complemented (B, W, L)
MOVZ_	Move Zero-Extended (BW, BL, WL)
CLR_	Clear (B, W, L, F, Q, D)
CVTR_L	Convert Rounded (F, D) to Longword
CMP_	Compare (B, W, L, F, D)
TST_	Test (B, W, L, F, D)
BIS_2	Bit Set (B, W, L) 2-Operand
BIS_3	Bit Set (B, W, L) 3-Operand
BIC_2	Bit Clear (B, W, L) 2-Operand
BIC_3	Bit Clear (B, W, L) 3-Operand
BIT_	Bit Test (B, W, L)
XOR_2	Exclusive OR (B, W, L) 2-Operand
XOR_3	Exclusive OR (B, W, L) 3-Operand

Integer and Floating Point Arithmetic Instructions

INC_	Increment (B, W, L)
DEC_	Decrement (B, W, L)
ASH_	Arithmetic Shift (L, Q)
ADD_2	Add (B, W, L, F, D) 2-Operand
ADD_3	Add (B, W, L, F, D) 3-Operand
ADWC	Add with Carry
ADAWI	Add Aligned Word Interlocked
SUB_2	Subtract (B, W, L, F, D) 2-Operand
SUB_3	Subtract (B, W, L, F, D) 3-Operand
SBWC	Subtract with Carry
MUL_2	Multiply (B, W, L, F, D) 2-Operand
MUL_3	Multiply (B, W, L, F, D) 3-Operand
EMUL	Extended Multiply
DIV_2	Divide (B, W, L, F, D) 2-Operand
DIV_3	Divide (B, W, L, F, D) 3-Operand
EDIV	Extended Divide
EMOD_	Extended Modulus (F, D)
POLY_	Polynomial Evaluation (F, D)

*B = byte, W = word, L = longword, F = floating, D = double floating, Q = quadword.

1.6.2 Character String Instructions

The character string instructions operate on strings of bytes. They include:

- move string instructions, with translation options
- string compare instructions
- single character search instructions
- substring search instructions

There are two basic forms of Move instructions for character strings. The Move Character instructions (MOVC3 and MOVC5) simply copy character strings from one location to another. They are optimized for block transfer operations.

The Move Translated Characters (MOVTC) and Move Translated Until Character (MOVTUC) instructions actually create new character strings. A string is supplied which the instruction uses as a list of offsets into a translation table. The instruction selects characters from the table in the order that the offset list points to the table. The MOVTC instruction allows a fill character to be supplied that the instruction uses to pad out the resultant string to a given size with an arbitrary character. The MOVTUC instruction allows any number of escape characters to be supplied. When the next offset points to an escape character in the table, translation stops.

The Compare Characters (CMPC) instructions provide character-by-character byte string compares. CMPC has a 3-operand form and a 5-operand form. Both instructions compare two strings from beginning to end and acknowledge that it reached the first character that is different between the strings, or when it gets to the end of either string. The 5-operand variation enables a fill character to be supplied which it uses to effectively pad out a string when comparing it with a longer one.

The Locate Character (LOCC) and Skip Character (SKPC) instructions are search instructions for single characters within a string. LOCC searches a given string for a character that matches the search character supplied. This is useful, for example, when searching for the delimiter at the end of a variable-length string. SKPC, on the other hand, finds the first character in the string that is different from the search character supplied. This is useful for skipping through fill characters at the end of a field to find the beginning of the next field.

The Match Characters (MATCHC) instruction is similar to the Locate Character instruction, but it locates multiple-character substrings. MATCHC searched a string for the first occurrence of a substring supplied.

The Span characters (SPANC) and Scan Characters (SCANC) instructions are search instructions that look for members of character classes. For these instructions the following are supplied: a character string, a mask, and the address of a 256-byte table of character type definitions. For each character in the given string, the instruction looks up the type code in the table for that character, and then AND's the given mask with the character's type code. SPANC finds the first character in the string which is of the type indicated by its mask. SCANC finds the first character in the string which is of any type other the one indicated by its mask.

The character string instructions are listed as follows:

MOVC3	Move Character 3-Operand
MOVC5	Move Character 5-Operand
MOVTC	Move Translated Characters
MOVTUC	Move Translated Until Character
CMPC3	Compare Characters 3-Operand
CMPC5	Compare Characters 5-Operand
LOCC	Locate Character
SKPC	Skip Character
SCANC	Scan Characters
SPANC	Span Characters
MATCHC	Match Characters

1.6.3 Packed Decimal Instructions

Many of the operations for integer and floating point data also apply to packed decimal strings. They include:

- Move Packed (MOVP) for copying a packed decimal string from one location to another, and Arithmetic Shift Packed (ASHP) for scaling a packed decimal up or down by a given power of 10 while moving it, and optionally rounding the value.

- Compare Packed (CMPP) for comparing two packed decimal strings. Compare packed has two vairations: a 3-operand (CMPP3) instructions for strings of equal length, and a 4-operand instruction (CMPP4) for strings of differing lengths.

- Convert Instructions, including Convert Long to Packed (CVTLP). Convert Packed to Long (CVTPL), Convert Packed to numeric with Trailing sign (CVTPT), Convert numeric with Trailing sign to Packed (CVTTP), Convert Packed to numeric with Separate overpunched sign (CVTPS), and Convert numeric with Separate overpunched sign to Packed (CVTSP). These instructions enable conversion of packed decimal format to commonly used numeric formats. Numeric with trailing sign allows various sign encodings including zoned and overpunched.

- Add Packed (ADDP) and Subtract Packed (SUBP) for adding or subtracting two packed decimal strings, with the option of replacing the addend or subtrahend with the result (ADDP4 and SUBP4), or storing the result in a third string (ADDP6 or SUBP6).

- Multiply Packed (MULP) and Divide Packed (DIVP) for multiplying or dividing two packed decimal strings and storing the result in a third string.

In addition, the packed decimal instructions include a special packed decimal string to character string conversion instruction that provides output formatting: the Edit instruction.

The Edit Packed to Character String (EDITPC) instruction supplies formatted numeric output functions. The instruction converts a given packed decimal string to a character string using selected pattern operators. The pattern operators enable creation of numeric output fields with any of the following characteristics:

- leading zero fill
- leading zero protection
- leading asterisk fill protection
- a floating sign
- a floating currency symbol
- special sign representations
- insertion characters
- blank when zero

The packed decimal instructions are listed as follows:

MOVP	Move Packed
CMPP3	Compare Packed 3-Operand
CMPP4	Compare Packed 4-Operand
ASHP	Arithmetic Shift Packed and Round
ADDP4	Add Packed 4-Operand
ADDP6	Add Packed 6-Operand
SUBP4	Subtract Packed 4-Operand
SUBP6	Subtract Packed 6-Operand
MULP	Multiply Packed
DIVP	Divide Packed
CVTLP	Convert Long to Packed
CVTPL	Convert Packed to Long
CVTPT	Convert Packed to Trailing
CVTTP	Convert Trailing to Packed
CVTPS	Convert Packed to Separate
CVTSP	Convert Separate to Packed
EDITPC	Edit Packed to Character String

1.6.4 Index Instruction

The Index instruction (INDEX) calculates an index for an array of fixed length data types (integer and floating) and for arrays of bit fields, character strings, and decimal strings. It accepts as arguments: a subscript, lower and upper subscript bounds, an array element size, a given index, and a destination for the calculated index. It incorporates range checking within the calculation for high-level languages using subscript bounds, and it allows index calculation optimization by removing invarient expressions.

1.6.5 Variable-Length Bit Field Instructions

The bit field instructions enable the definition, access, and modification of fields whose size and location can be specified. The location is determined from a base address or from a register and signed bit offset$_{32}$. If the field is in memory, the offset range can be as large as $2^{32}-1$ (approximately 16 million bytes). If the field is in a register, the offset can be as large as 31. Fields of arbitrary lengths (0 to 32 bits) can be used to store data structure header information compactly or for storing status codes.

The Insert Field and Extract Field instructions store data and retrieve data from fields. Insert Field (INSV) stores data in a field by taking a specified number of bits of a longword (starting from the low-order bit) and writing them into a field, which may start at any bit relative to a given base address. The field can either be signed (EXTV) or unsigned (EXTZV).

The Compare Field and Find First instructions enable the contents of a field to be tested. Compare Field extracts a field and then compares it with a given longword. The field can be interpreted as signed (CMPV), or as unsigned (CMPZV). The Find First instructions locate the first bit in a field that is clear (FFC) or set (FFS), scanning from low-order bit to high-order bit. These instructions are particularly useful for scanning a status control longword. For example, the longword may represent a set of queues processed in order by priority 0 (high) to 31 (low). Each set bit represents an active queue. The Find First Set instruction quickly returns the highest priority queue that is active. Together with the SKPC instructions, the Find First instructions are also useful for scanning an allocation table (bit map) of arbitrary length.

The variable-length bit field instructions are listed as follows:

EXTV	Extract Field
EXTZV	Extract Zero-Extended Field
INSV	Insert Field
CMPV	Compare Field
CMPZV	Compare Zero-Extended Field
FFS	Find First Set
FFC	Find First Clear

1.6.6 Queue Instructions

Two instructions are provided that enable construction and maintenance of queue data structures. Queues manipulated using the queue instructions are circular, doubly linked lists of data items.

The first longword of a queue entry contains the forward pointer to the next entry in the queue, and the next longword contains the backware pointer to the preceding entry in the queue. One queue entry is arbitrarily treated as the head of the queue. Since a

list is circular, the tail of a queue is the entry that points to the head of the queue. In practice, the first entry of a queue is a "permanently allocated" listhead containing only the pointers to the first and last elements.

The INSQUE instruction inserts entries into queues. If an entry is the first item in a queue, INSQUE effectively creates a queue. The REMQUE instruction removes entries from a queue, and effectively deletes a queue if an entry is the last item removed. Entries can be inserted or removed at the head or tail of a queue, or anywhere in between.

Cooperating processes can access a queue at the same time without external synchronization. However, if more than one process is allowed to access a given queue at the same time, each process should insert or remove entries only at the head or tail of the queue. If only one process at a time accesses a queue, entries can be inserted or removed anywhere in the queue.

1.6.7 Address Manipulation Instructions

Two instructions are provided that enable an address to be fetched without actually accessing the data at that location:

a. The Move Address (MOVA) instruction which stores the address of a byte, word, longword (and floating), or quadword (and double floating) datum in a specified register or memory location.

b. The Push Address (PUSHA) instructions which store the address of a byte, word, longword (and floating) or quadword (and double floating) datum on the stack.

1.6.8 General Register Manipulation Instructions

The general register manipulation instructions enable any user program to save or load the general purpose registers in one operation, examine the Processor Status Longword, and set or clear status bits in the Processor Status Word.

The Push Longword (PUSHL) instruction pushes a longword on the stack. This instruction is the same as a Move Longword using the Stack Pointer in register deferred mode, but is a byte shorter. It is a consistent and convenient way to move data to the stack.

The Push Registers (PUSHR) instruction pushes a set of registers on the stack in one operation. A mask word is supplied in which each bit set (Ø-14) represents a register (RØ through R14) that is to be saved on the stack. The only general register that cannot be saved using this instruction is R15, the Program Counter. Pop Registers (POPR) reverse the operation, loading each register from successive longwords on the stack according to the given mask word. The PUSHR and POPR instructions replace the need to write a sequence of Move instructions to save and restore registers upon entry and exit from a subroutine.

The Move from Processor Status Longword (MOVPSL) instruction allows examination of the contents of the processor's status register by loading its contents into a specified location. The Bit Set (BISPSW) and Bit Clear (BICPSW) Processor Status Word instructions allow the setting or clearing of the PSW condition codes and trap enable bits.

The general register manipulation instructions are listed as follows:

PUSHL	Push Longword on Stack
PUSHR	Push Registers on Stack
POPR	Pop Registers from Stack
MOVPSL	Move from Processor Status Longword
BISPSW	Bit Set Processor Status Word
BICPSW	Bit Clear Processor Status Word

1.6.9 Branch, Jump, and Case Instructions

The two basic types of control transfer instructions are the branch and jump instructions, both of which load new addresses in the Program Counter. In branch instructions, a displacement (offset) is supplied and added to the current Program Counter to obtain the new address. In jump instructions, the new address is loaded into the Program Counter using one of the normal addressing modes.

A number of branch instructions are offered since most transfers are to locations relatively close to the current instruction and branch instructions are more efficient than jump instructions. There are two unconditional branch instructions and several conditional branch instructions.

The unconditional branch instructions allow byte (BRB) or word (BRW) displacements to be specified. This allows branching to locations a maximum of 32,767 bytes in either direction from the current location. For control transfers to locations farther away, the Jump (JMP) instruction can be used.

The condition branch instructions include:

a. branch on bit instructions

b. set and clear bit instructions with a branch if it is already set or cleared

c. loop instructions that increment or decrement a counter, compare it with a limit value, and branch on a relational condition

d. computed branch instruction in which a branch may take place to one of several locations depending on a computed value

The branch or condition instructions enable transfer of control to another location depending on the status of one or more of the condition codes in the Processor Status Word (PSW). There are three groups of Branch on Condition instructions:

a. the signed relational branches, which are used to test the outcome of instructions operating on integer and field data types being treated as signed integers, floating point data types, and decimal strings

b. the unsigned relational branches, which are used to test the outcome of instructions operating on integer and field data types being treated as unsigned integers, character strings, and addresses

c. the overflow and carry test branches, which are used for checking overflow when traps are not enabled, for multiprecision arithmetic, and for the results of special instructions

The instruction mnemonics indicate the choice between a signed and unsigned integer data type interpretation for relational testing. The relational tests determine if the result of the previous operation is less than, less than or equal, equal, not equal, greater than or equal, or greater than zero. For example, the Branch on Less than or Equal Unsigned (BLEQU) instruction branches if either the Carry or Zero bit is set. The Branch on Greater Than (BGTR) instruction branches if neither the Negative nor the Zero bit is set.

There are also general purpose Branch on Bit instructions similar to Branch on Condition. The Branch on Low Bit Set (BLBS) and Branch on Low Bit Clear (BLBC) instructions test bit Ø of an operand, which is useful for testing Boolean values. The Branch on Bit Set (BBS) and Branch on Bit Clear (BBC) instructions test any selected bit.

There are special kinds of Branch on Bit instructions that are actually bit set/clear instructions. The Branch on Bit Set and Set (BBSS) is an example. The instruction branches if the indicated bit is set, otherwise it falls through. In either case, the instruction sets the given bit. The BBSS instruction can thus be thought of as a Bit Set instruction with a branch side effect if the bit was already set. There are four such instructions:

a. Branch on Bit Set and Set (BBSS)
b. Branch on Bit Clear and Clear (BBCC)
c. Branch on Bit Set and Clear (BBSC)
d. Branch on Bit Clear and Set (BBCS)

These instructions are particularly useful for keeping track of procedure completion or initialization, and for signaling the completion or initialization of a procedure to a cooperating process. In addition, there are two Branch on Bit Interlocked instructions that provide control variable protection:

a. Branch on Bit Set and Set Interlocked (BBSSI)
b. Branch on Bit Clear and Clear Interlocked (BBCCI)

The SBI bus provides a memory interlock on these instructions. No other BBSSI or BBCCI operation can interrupt these instructions to gain access to the byte containing the control variable between the testing of the bit and the setting or clearing of the bit.

Three types of branch instructions can be used to write efficient loops. The first type provides a basic subtract-one-and-branch loop. A counter variable is supplied which is decremented each time the loop is executed. In the Subtract One and Branch Greater

Than (SOBGTR) instruction, the loop repeats until the counter equals zero. In the Subtract One and Branch Greater Than or Equal (SOBGEQ) instruction, the loop repeats until the counter becomes negative.

The counterpart to subtract-one-and-branch is add-one-and-branch. In this case, a counter and a limit are incremented at the end of the loop. In the Add One and Branch Less Than (AOBLSS) instruction, the loop repeats until the counter equals the limit set. In the Add One and Branch Less Than or Equal (AOBLEQ) instruction, the loop repeats until the counter exceeds the limit set.

The third type of loop instruction efficiently implements the FORTRAN language DO statement and the BASIC language FOR statement: Add Compare and Branch (ACB). A limit, a counter, and a step value are supplied. For each execution of the loop, the instruction adds the step value to the counter and compares the counter to the limit. The sign of the step value determines the logical relation of the comparison: the instruction loops on a less than or equal comparison if the step value is positive, on a greater than or equal comparison if the step value is negative.

The processor provides a branch instruction that implements higher-level language computed GO TO statements: the CASE instruction. For CASE, a list of displacements are supplied that generate different branch addresses indexed by the value obtained as a selector. The branch falls through if the selector does not fall within the limits of the list.

The branch, jump, and case instructions are listed as follows:

Unconditional Branch and Jump Instructions

BR	Branch with (Byte, Word) Displacement
JMP	Jump

Branch on Condition Code

BLSS	Less Than
BLSSU	Less than Unsigned
BLEQ	Less than or Equal
BLEQU	Less than or Equal Unsigned
BEQL	Equal
BEQLU	Equal Unsigned
BNEQ	Not Equal
BNEQU	Not Equal Unsigned
BGTR	Greater than
BGTRU	Greater than Unsigned
BGEQ	Greater than or Equal
BGEQU	Greater than or Equal Unsigned
BCC	Carry Cleared
BVS	Overflow Set
BVC	Overflow Clear

Branch on Bit

BLB_	Branch on Low Bit (Set, Clear)
BB_	Branch on Bit (Set, Clear)
BBS_	Branch on Bit Set and (Set, Clear) bit
BBC_	Branch on Bit Clear and (Set, Clear) bit
BBSSI	Branch on Bit Set and Set bit Interlocked
BBCCI	Branch on Bit Clear and Clear bit Interlocked

Loop and Case Branch

ACB_	Add, Compare and Branch (B, W, L, F, D)
AOBLEQ	Add One and Branch Less Than or Equal
AOBLSS	Add One and Branch Less Than
SOBGEQ	Subtract One and Branch Greater Than or Equal
SOBGTR	Subtract One and Branch Greater Than
CASE	Case on (B, W, L)

1.6.10 Subroutine Branch, Jump, and Return Instructions

Two special types of branch and jump instruction are provided for calling subroutines: the Branch to Subroutine (BSB) and Jump to Subroutine (JSB) instructions. Both BSB and JSB instructions save the contents of the Program Counter on the stack before loading the Program Counter with the new address. With Branch to Subroutine, either a byte (BSBB) or word (BSBW) displacement are supplied. With Jump to Subroutine, regular addressing is used.

The subroutine call instructions are complemented by the Return from Subroutine (RSB) instruction. RSB pops the first longword off the stack and loads it into the Program Counter. Since the Branch to Subroutine instruction is either two or three bytes long, and the Return from Subroutine instruction is one byte long, it is possible to write extremely efficient programs using subroutines.

1.6.11 Procedure Call and Return Instructions

Procedures are general purpose routines that use argument lists passed automatically by the processor. The procedure Call instructions enable language processors and the operating system to provide a standard calling interface. They:

a. save all the registers that the procedures use, and only those registers, before entering the procedure

b. pass an argument list to a procedure

c. maintain the Stack, Frame, and Argument Pointer registers

d. initialize the arithmetic trap enables to a given state

When a Call procedure instruction is issued, the address of the procedure is supplied.

The first word of a procedure contains an entry mask that is used in the same way as the entry mask defined for the Push Registers instruction. Each set bit of the 12 low-order bits in the word represents one of the general register, RØ through R11, that the procedure uses. The Call instruction examines this word and saves the indicated registers on the stack. In addition, the Call instruction also automatically saves the contents of the Frame Pointer, Argument Pointer, and Program counter registers. This is an extremely efficient way to ensure that registers are saved across procedure calls. No general register is saved that does not have to be saved.

The Call Procedure with General Argument List (CALLG) instruction accepts the address of an argument list and passes the address to the procedure in the Argument Pointer register. The Call Procedure with Stack Argument List (CALLS) passes the argument list, if any, which you have placed on the stack, by loading the Argument Pointer register with its stack address.

When a procedure completes execution, it issues the Return from Procedure Instruction (RET). Return uses the Frame Pointer register to find the saved registers that it restores, and to clean up any data left on the stack, including nested routine linkages. A procedure can return values using the argument list or other registers.

1.6.12 Miscellaneous Special Purpose Instructions

Native mode includes a number of special purpose instructions, including:

a. Cyclic Redundancy Check (CRC)
b. Breakpoint Fault (BPT)
c. Extended Function Call (XFC)
d. No Operation (NOP)
e. Halt

The Cyclic Redundancy Check (CRC) instruction calculates a cyclic redundancy check for a given string using any CRC polynomial up to 32 bits long. The operating system library includes tables for standard CRC functions, such as CRC-16.

The Breakpoint Fault (BPT) instruction makes the processor execute the kernel mode condition handler associated with the Breakpoint Fault exception vector. BPT is used by the operating system debugging utilities but can also be used by any process that sets up a Breakpoint Fault condition handler.

The Extended Function Call (XFC) instruction allows escapes to customer-defined instructions in writable control store. The NOP instruction is useful for debugging. The HALT instruction is a privileged instruction issued only by the operating system to halt the processor when bringing the system down by operator request.

1.6.13 Protected and Privileged Instructions

The processor provides three types of instructions that enable user mode software to obtain operating system services without jeopardizing the integrity of the system. They include:

a. the Change Mode instructions
b. the PROBE instructions
c. the Return from Exception or Interrupt instruction

User mode software can obtain privileged services by calling operating system service procedures with a standard CALL instruction. The operating system's service dispatcher issues an appropriate Change Mode instruction before actually entering the procedure. Change Mode allows access mode transitions to take place from one mode to the same or more privileged mode only. When the mode transition takes place the previous mode is saved in the Previous Mode field of the Processor Status Longword, allowing the more privileged code to determine the privilege of its caller.

A Change Mode instruction is simply a special trap instruction that can be thought of as an operating system service call instruction. User mode software can explicitly issue Change Mode instructions, but since the operating system receives the trap, non-privileged users can not write any code to execute in any of the privileged access modes. User mode software can include a condition handler for Change Mode to User traps, however, and this instruction is useful for providing general purpose services for user mode software. The system manager ultimately grants the privilege to write any code that handles Change Mode traps to more privileged access modes.

For service procedures written to execute in privileged access modes (kernel, executive, and supervisor), the processor provides address access privilege validation instructions. The PROBE instructions enable a procedure to check the read (PROBER) and write (PROBEW) access protection of pages in memory against the privileges of the caller who requested to access a particular location. This enables the operating system to provide services that execute in privileged modes to less privileged callers and still prevent the caller from accessing protected areas of memory.

The operating system's privileged service procedures and interrupt and exception service routines exit using the Return from Exception or Interrupt (REI) instruction. REI is the only way in which the privilege of the processor's access mode can be decreased. Like the procedure and subroutine return instructions, REI restores the Program Counter and the processor state to resume the process at the point where it was interrupted.

REI performs special services, however, that normal return instructions do not. For example, REI checks to see if any asynchronous system traps have been queued for the currently executing process while the interrupt or exception service routine was executing, and ensures that the process will receive them. Furthermore, REI checks to ensure that the mode to which it is returning control is the same as or less privileged than the mode in which the processor was executing when the exception or interrupt occurred. Thus REI is available to all software including user-written trap handling routines, but a program can not increase its privilege by altering the processor state to be restored.

When the operating system schedules a context switching operation, the context switching procedure uses the Save Process Context (SVPCTX) and Load Process Context (LDPCTX) instructions to save the current process context and load another. The operating system's context switching procedure identifies the location of the hardware context to be loaded by updating an internal processor register.

Internal processor registers not only include those that identify the process currently executing, but also the memory management and other registers, such as the console and clock control registers. The Move to Processor Register (MTPR) and Move from Processor Register (MFPR) instructions are the only instructions that can explicitly access the internal processor register. MTPR and MFPR are privileged instructions that can be issued only in kernel mode. Table 1-6 provides a complete list of the processor registers.

The protected and privileged instructions are listed as follows:

Protected Procedure Call and Return Instructions

CHM_	Change Mode to (Kernel, Executive, Supervisor, User)
REI	Return from Exception or Interrupt
PROBER	Probe Read
PROBEW	Probe Write

Privileged Processor Register Control Instructions

SVPCTX	Save Process Context
LDPCTX	Load Process Context
MTPR	Move to Process Register
MFPR	Move from Processor Register

Table 1-6 Processor Registers

Register Name	Mnemonic	Number (Hex)
Kernel Stack Pointer	KSP	00
Executive Stack Pointer	ESP	01
Supervisor Stack Pointer	SSP	02
User Stack Pointer	USP	03
Interrupt Stack Pointer	ISP	04
P0 Base Register	P0BR	08
P0 Length Register	P0LR	09
P1 Base Register	P1BR	0A
P1 Length Register	P1LR	0B
System Base Register	SBR	0C
System Limit Register	SLR	0D
Process Control Block Base	PCBB	10
System Control Block Base	SCBB	11
Interrupt Priority Level	IPL	12
AST Level	ASTLVL	13
Software Interrupt Request	SIRR	14
Software Interrupt Summary	SISR	15
Interval Clock Control	ICCS	18
Next Interval Count	NICR	19
Interval Count	ICR	1A
Time of Year	TODR	1B
Console Receiver C/S	RXCS	20
Console Receiver D/B	RXDB	21
Console Transmit C/S	TXCS	22
Console Transmit D/B	TXDB	23
Memory Management Enable	MAPEN	38
Trans. Buf. Invalidate All	TBIA	39
Trans. Buf. Invalidate Single	TBIS	3A
Performance Monitor Enable	PMR	3D
System Identification	SID	3E
Accelerator Control/Status	ACCS	28
Accelerator Maintenance	ACCR	29
WCS Address	WCSA	2C
WCS Data	WCSD	2D
SBI Fault/Status	SBIFS	30
SBI Silo	SBIS	31
SBI Silo Comparator	SBISC	32
SBI Maintenance	SBIMT	33
SBI Error Register	SBIER	34
SBI Timeout Address	SBITA	35
SBI Quadword Clear	SBIQC	36
Micro Program Breakpoint	MBRK	3C

1.7 COMPATIBILITY MODE

Under control of the operating system, the processor can execute PDP-11 instruction streams within the context of any process. When executing in compatibility mode, the processor interprets the instruction stream executing in the context of the current process as a subset of PDP-11 code that does not include floating point hardware instructions or privileged instructions.

In general, compatibility mode enables the operating system to provide an environment for executing most user mode programs written for a PDP-11 except stand-alone software. The processor expects all compatibility mode software to rely on the services of the native operating system for I/O processing, interrupt and exception handling, and memory management. There are some restrictions, however, on the environment that the native operating system can provide a PDP-11 program. For example, the PDP-11 memory management instructions Move To/From Previous Instruction/Data Space can not be simulated by the operating system since they do not trap to native mode software.

PDP-11 addresses are 16-bit byte addresses. There is a one-to-one correspondence between compatibility mode virtual addresses and the first 64K bytes of virtual address space available to native mode processes. As in the PDP-11, a compatibility mode program is restricted to referencing only these addresses. It is possible for the operating system to provide most of the PDP-11 memory management mechanisms. For example, compatibility mode automatically supports PDP-11 memory segment protection, but in 512 byte rather than 64-byte segments.

All of the PDP-11 general registers and addressing modes are available in compatibility mode. Compatibility mode registers R0 through R6 are the low-order 16 bits of native mode registers R0 through R6. Compatibility mode R7 (the Program Counter) is the low-order bits of native mode register 15 (the Program Counter). Native mode registers 8 through 14 are not affected by compatibility mode. Note that the compatibility mode register R6 acts as the Stack Pointer for program-local temporary data storage, but that the program-local stack is allocated address space in the program region, not the control region.

A subset of the PDP-11 Processor Status Word is defined for compatibility mode. Only the condition codes and the trace trap bit are relevant for the PDP-11 instruction stream.

All interrupts and exceptions that occur when the processor is executing in compatibility mode cause the processor to enter native mode. As in native mode, it is the operating system's responsibility to handle interrupt and exceptions. There are a few types of exceptions that apply only to compatibility mode. They include illegal instruction exceptions and odd address trap.

The compatibility mode instruction set is that of the PDP-11 with the following exceptions:

a. the privileged and floating-point option instructions are illegal (this includes HALT, WAIT, RESET, SPL, MARK, the floating instruction set, and the floating-point processor instructions)

b. the trap instructions (BPT, IOT EMT, and TRAP) cause the processor to enter native mode, where either the trap may be serviced, or the instruction simulated

c. the move from/to previous instruction/data space instructions (MFPI, MTPI, MFPD, and MTPD) execute exactly as they would on a PDP-11 in user mode with instruction and data space overmapped. They ignore the previous access level and act as PUSH and POP instructions referencing the current stack.

All other instructions execute exactly as they would on a PDP-11/70 processor running in user mode.

1.8 CENTRAL PROCESSING UNIT (CPU) HARDWARE INTRODUCTION

This section provides a general description of the following function areas of the central processor:

a. buses
b. clocks
c. microsequencer
d. control store
e. data path
f. instruction buffer and instruction decode
g. interrupts and exceptions

Chapter 2 provides a detailed description of each of the above areas. The translation buffer, cache, SBI control, and console subsystem are described in the associated manuals listed in Table 1-1. Figure 1-31 illustrates the interconnection of the major units of the CPU.

1.8.1 Bus Summary

The following paragraphs describe each of the buses which interconnect the CPU. The major buses are:

Synchronous Backplane Interconnect (SBI)
Physical Address Bus (PA Bus)
Control Store Bus (CS Bus)
Internal Data Bus (ID Bus)
Memory Data Bus (MD Bus)
Visibility Bus (V Bus)
LSI-11 Bus (Q Bus)

1.8.1.1 Synchronous Backplane Internconnect

The Synchronous Backplane Interconnect (SBI) is the bidirectional information path and communication protocol for data exchanges between the CPU, memory, and adapters of the VAX-11/780 system. The SBI provides checked, parallel information exchanges synchronous with a common system clock.

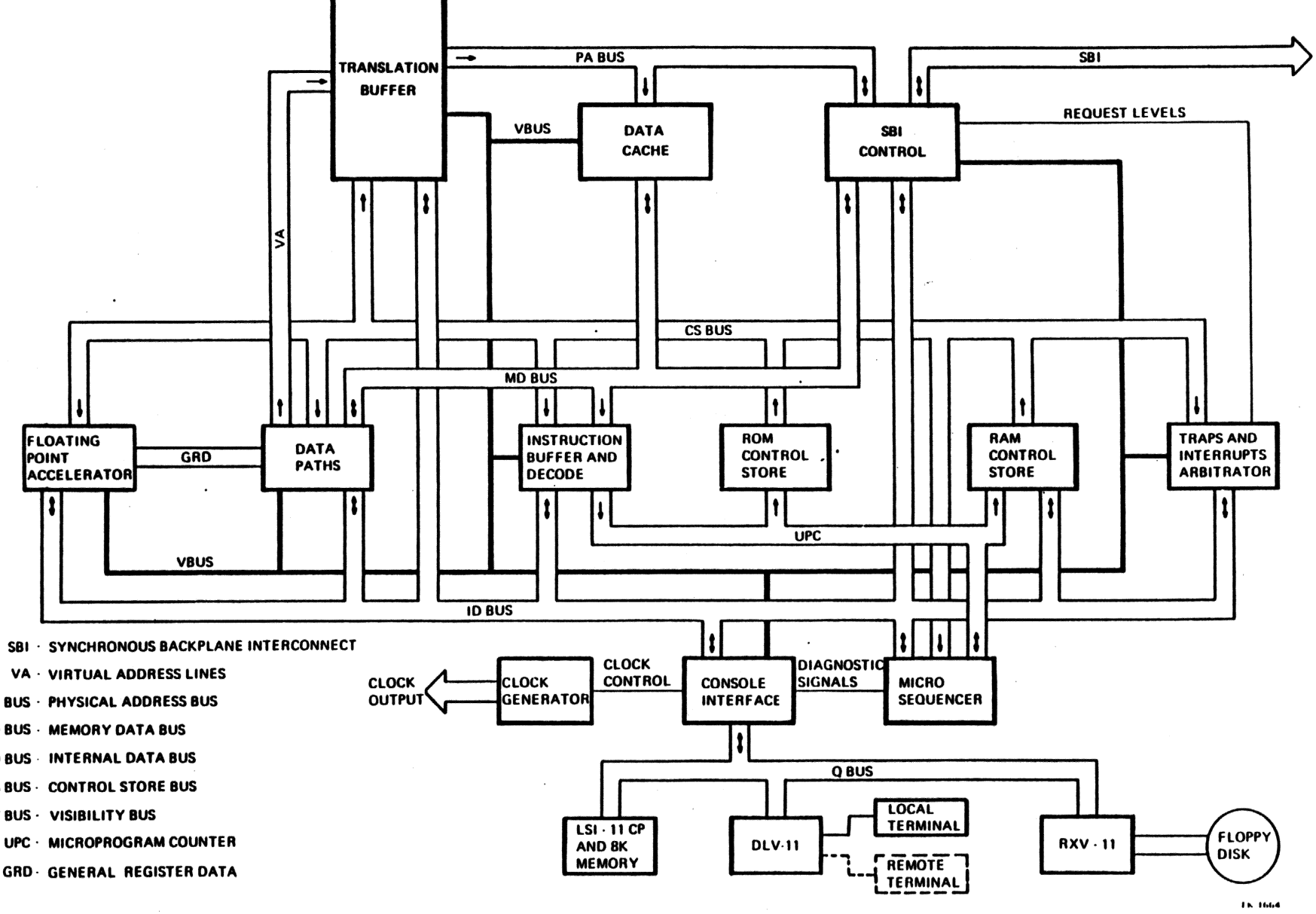


Figure 1-31 CPU Block Diagram

The communications protocol allows the information path to be time multiplexed, so that a number of information exchanges may be in progress simultaneously. During each clock period (or cycle), interconnect arbitration, information transfer, and transfer confirmation may occur in parallel.

SBI signals are clocked into data latches. All checking and subsequence decision making is based on these latched signals. Error checking logic detects single failures in the information path. However, multiple SBI system failures are not necessarily detected.

A nexus, which is any physical connection to the SBI, is capable of performing one or more of the functions listed:

1. Commander -- A nexus which transmits command and address information.

2. Responder -- A nexus which recognizes command and address information as directed to it and transmits a response.

3. Transmitter -- A nexus which drives the information lines.

4. Receiver -- A nexus which samples and examines the information lines.

As an example, consider the CPU which issues a read-type command. It may be considered one of three nexus types, depending on the point in the information exchange.

When the CPU issues the read command, it is a commander since it is issuing command/address information. At the same time it is a transmitter since it is driving the information lines. When the device (responder) returns the requested data, the CPU is considered a receiver, since it examines the information lines and the data is specifically directed to it. In the strict sense, each nexus is a receiver (i.e., examining information lines) in every SBI cycle.

In the case of a memory read exchange, the memory is the responder since it recognizes and responds to a command/address signal. Also, since it examines the information lines, it is a receiver (along with every other nexus on the SBI). When the memory returns the requested data by driving the information lines, it is a transmitter.

The 84 lines of the SBI are divided into these functional groups:

1. Aribitration
2. Information
3. Confirmation
4. Interrupt
5. Control.

1.8.1.1.1 Aribtration Group -- The arbitration group sets nexus priority to access the SBI. It determines which nexus of those requesting access to the SBI in a particular cycle will perform an information transfer in the following cycle.

1.8.1.1.2 Information Group -- The information group exchanges command/address, data, and interrupts summary information. Each exchange consists of one to three information transfers.

For write-type commands, the commander uses two or three successive SBI cycles. The number of successive cycles required depends on whether one or two data longwords are to be written in the exchange. In the first case, the commander transmits the command/address in the first cycle, and a data longword in the second cycle. In the second case, the commander transmits the command/address in the first cycle, data longword 0 in the second cycle, and data longword 1 in the third cycle.

Read-type commands are also initiated with a command/address transmitted from the commander. However, since data emanates from the responder, the requested data may be delayed by the characteristic access time of the responder. As in a write exchange, the read data will be transmitted using one or two successive cycles depending on whether one or two data longwords were requested.

An interrupt summary exchange is response to a device-generated interrupt to the CPU. The exchange is initiated with an interrupt summary read transfer from the CPU. The exchange is completed two cycles later with an interrupt summary response transfer containing the interrupt information.

1.8.1.1.3 Confirmation Group -- The confirmation group provides a path to inform the transmitter whether the infomration transfer was correctly received and, in the case of a command/address transfer, whether the receiver can process the command.

Each command/address or information transfer is confirmed by the responder (or receiver) two cycles after transmission by the commander. During a write-type exchange, command/address and data transfers are confirmed by the responder. During a read-type exchange, the command/address transfer is confirmed by the responder; the reception of read data is confirmed by the commander.

Interrupt summary transfers are not confirmed.

1.8.1.1.4 Interrupt Request Group -- The interrupt request group provides a path for nexus to interrupt the CPU to service a condition requiring processor intervention. In addition, the group includes a special line for nexus which interrupts the CPU only for changes in power or operating conditions.

1.8.1.1.5 Control Group -- The control group provides a path to synchronize system activity and provides specialized system comminucation. The group includes the system clock which provides the universal time base for any nexus connected to the SBI. The group also provides initialization, power fail, and restart functions for the system. In addition, an interlock line is provided for coordination of memory sharing in multiprocessor systems.

1.8.1.2 Physical Address Bus -- The physical address (PA) bus is a bidirectional internal bus 28 bits wide [PA (29:Ø2)]. The PA bus transfers the translated physical address from the TB to the Cache and SBI Control. In the case when the memory management enable function is off, the address transferred is not translated. The PA bus is also used to transfer a physical address from the SBI Control to Cache for Cache refill and SBI invalidated sequences.

1.8.1.3 Control Store Bus -- The control store (CS) bus provides the path for the transfer of each microword field to various areas of the central processor. The CS bus is comprised of 96 data bits and 3 parity bits (1 for each 32-bit data segment). The 96 data bits represent the VAX-11/78Ø control word or microword. The microword is divided into fields, each of which provides control for some area of the processor. Paragraph 2.2 defines each of the microword fields.

1.8.1.4 Internal Data Bus -- The internal data (ID) bus is the high speed, bidirectional data path of the CPU. The ID bus is used to perform the following;

a. data transfers to and from the internal registers of the CPU.

b. data transfers in the form of displacements and short literals from the instruction buffer to the CPU data paths and the FPA.

c. data transfers between the CPU data paths and the FPA.

d. data transfers from the internal register to the console under console control during maintenance operation.

1.8.1.4.1 ID Bus Operation -- The ID bus consists of 32 data lines, 6 address lines, and 1 write control line. The address lines specify which internal register has been designated as the source or destination. The internal register address assignments are listed in Table 1-7. The write control line specifies directional control, indicating whether an internal register is to be read onto the bus or data is to be clocked from the bus into an internal register.

During a normal read operation, data is transferred from the addressed internal register to the Q register of the data paths via the ID bus. During a normal write operation, data is transferred from the D register of the data paths to the addressed internal register. During maintenance operation, the console can read or write the internal register via the ID bus. In this mode, the D and Q registers of the data paths may be addressed as internal registers.

Table 1-7 ID Bus Register Address Assignment

Address (Hex)	Register Name	Address (Hex)	Register Name
00	IBUF DATA	20	USTACK
01	TIME OF DAY	21	UBREAK
02	-RSVD-	22	WCS ADDRESS
03	SYSTEM ID	23	WCS DATA/STATUS
04	CNSL RXCS	24	P0BR
05	CNSL RXDB (TO ID)	25	P1BR
06	CNSL TXCS	26	SBR
07	CNSL TXDB (FROM ID)	27	RSVD FOR SYS SPACE
08	DQ (ID MAINT ONLY)	28	KSP
09	NEXT INTERVAL REGISTER	29	ESP
0A	CLOCK CS	2A	SSP
0B	INTERVAL COUNTER	2B	USP
0C	CES	2C	ISP
0D	VECT	2D	FPDA
0E	SIR	2E	D.SV
0F	PSL	2F	Q.SV
10	TBUF DATA	30	T0
11	-RSVD-	31	T1
12	TBUF REG 0	32	T2
13	TBUF REG 1	33	T3
14	ACC REG 0	34	T4
15	ACC REG 1	35	T5
16	ACC MAINT REGISTER	36	T6
17	ACC CONTROL/STATUS	37	T7
18	SBI SILO	38	T8
19	SBI ERR REGISTER	39	T9
1A	SBI TIMEOUT ADDRESS	3A	PCBB
1B	SBI FAULT/STATUS	3B	SCBB
1C	SBI SILO COMPARATOR	3C	P0LR
1D	MAINTENANCE	3D	P1LR
1E	CACHE PARITY	3E	SLR
1F	-RSVD-	3F	RSVD FOR SYS SPACE

NOTE

Data formats and bit descriptions of each of the ID bus registers is provided in the VAX-11/780 Maintenance Handbook.

1.8.1.4.2 ID Bus Control -- Control of the ID bus is derived from two fields of the microword (UFS and UCID). Function Select (UFS) is a one bit field. If this bit is clear, the ID address and write signals are zero and the instruction buffer data is gated onto the ID bus. This data will be clocked into the Q register of the data path when selected.

If the UFS bit is set, the console ID (UCID) field of the microword controls the ID bus. The UCID field specifies the type of data transfer (read or write) and the address source. Table 1-8 lists the address source and operation selected by each UCID field value.

During normal operation, the internal register addresses are generated in either the Shift Count (SC) register in the data paths or the UKMX field of the microword. During maintenance operation, the address is generated by the console which controls the operation. Console control allows access to the Writable Control Store and provides visibility of the internal registers from the console.

Table 1-8 ID Bus Control

UCID Field (Hex)	Operation	Address Source
Ø	NO-OP	--
1	UNUSED	--
2	CNSL ACK	Console
3	CNSL CONT	Console
4	ID DATA←ID REG	Data Paths
5	ID DATA←ID REG	Microcode
6	ID REG←ID DATA	Data Paths
7	ID REG←ID DATA	Microcode

CNSL ACK is used to notify the console that the CPU is not using the ID bus and that the console may assert its ID maintenance bit and an internal register address CNSL ACK also sets the Console Command Mode flip-flop. CNSL CONT is used to clear the Console Command Mode flip-flop in order to relinquish control of the ID bus.

1.8.1.5 Memory Data Bus -- The memory data (MD) bus is the bidirectional information path for longword aligned data exchanges. The MD bus connects the data path portion of the CPU and the instruction buffer to the cache and SBI control. The bus consists of 40 lines: 32 data lines, 4 parity lines, and 4 mask lines. The parity lines provide parity for each of the four data bytes (i.e., parity bit 0 associated with byte 0, bits 7--0, etc.). The mask bits are associated with the data bytes similar to the parity bits. The mask bits inform the system which bytes are to be read or written.

Data is transferred over the MD bus in the following circumstances:

a. Data requested by the data path or instruction buffer is found in cache (hit) and is transferred back to the data path or instruction buffer via the MD bus.

b. Data requested by the data path or instruction buffer is not found in cache (miss) and is retrieved from main memory. The data is transferred from the SBI control to cache and the data path (or instruction buffer) simultaneously via the MD bus.

c. CPU write data is transferred to the SBI control via the MD bus and sent over the SBI to be written in memory. If the location is in cache, the data is also updated in cache simultaneously via the MD bus.

d. Interrupt Summary Read Responses are transferred over the MD bus to the data path.

1.8.1.6 Visibility Bus -- The visibility (V) bus is used for diagnostic isolation of CPU system failures. The V bus consists of eight serial data lines, a load signal line, a clock signal line, and a self-test line. Each of the participating CPU modules contains at least one V bus shift register. The data input lines to the shift register monitor specific test points on the CPU module. The load signal causes the shift register to parallel load from the test points when the CPU is in a stable condition. The clock signal can then be used to read the latched data serially from each of the shift registers into a register on the console interface board (CIB) where it can be read by LSI-11 software.

1.8.1.7 LSI-11 Bus (Q Bus) -- The Q bus connects the LSI-11 processor (and its ROM and RAM memories), the console terminal interfaces, and the floppy disk interface to the Console Interface

Board, and thus to the CPU. The 16 address signals and 16 data signals share the same bus lines. Fourteen other LSI-11 signal lines are used in the VAX-11/780 configuration for control signals (note that the DMA control lines are not used).

A master-slave relationship defines communication between the processor and the other devices on the bus. Each control signal issued by a master device must be acknowledged by a slave device in order to complete a transfer. The LSI-11 processor must therefore become bus master in order to read or write any interface register or memory location on the Q bus. The Q bus permits an addressing structure in which control, status, and data registers for peripheral devices are directly addressed as memory locations. No system clock is used on the Q bus, and all communications on it are asynchronous. However, when one of the interface units such as the serial line interface for the console terminal must transfer data (i.e., a character) to or from the LSI-11 processor, it must interrupt the processor and thereby invoke a service routine which will handle the actual data transfer.

Note that the serial line interfaces and the floppy disk interface cannot communicate directly with the Console Interface Board, nor can the CIB communicate directly with them. All transfers initiated from the interface begin with interrupts to the LSI-11 processor.

1.8.2 Clocks

The following provides a brief description of each of the VAX-11/780 clocks.

1.8.2.1 Processor Clock -- The processor clock provides the circuitry required for the generation of SBI timing signals, distribution and decoding of SBI signals to the processor modules, and power up/power fail sequencing.

The synchronous operation of the VAX-11/780 is based on a clock cycle of 200 ns. There are four 50 ns time states per cycle (T0, T1, T2, and T3). The CPU time states and SBI time states are derived from SBI signals called TP (timing pulse), PCLK (clock phase), and PDCLK (clock phase delayed). Figure 1-32 shows the relationship between the SBI/CPU timing signals and the derived time states.

NOTE

CPU and SBI time states are not the same. CPT0=SBI T1, CPT1=SBI T2, CPT2=SBI T3, and CPT3=SBI T0.

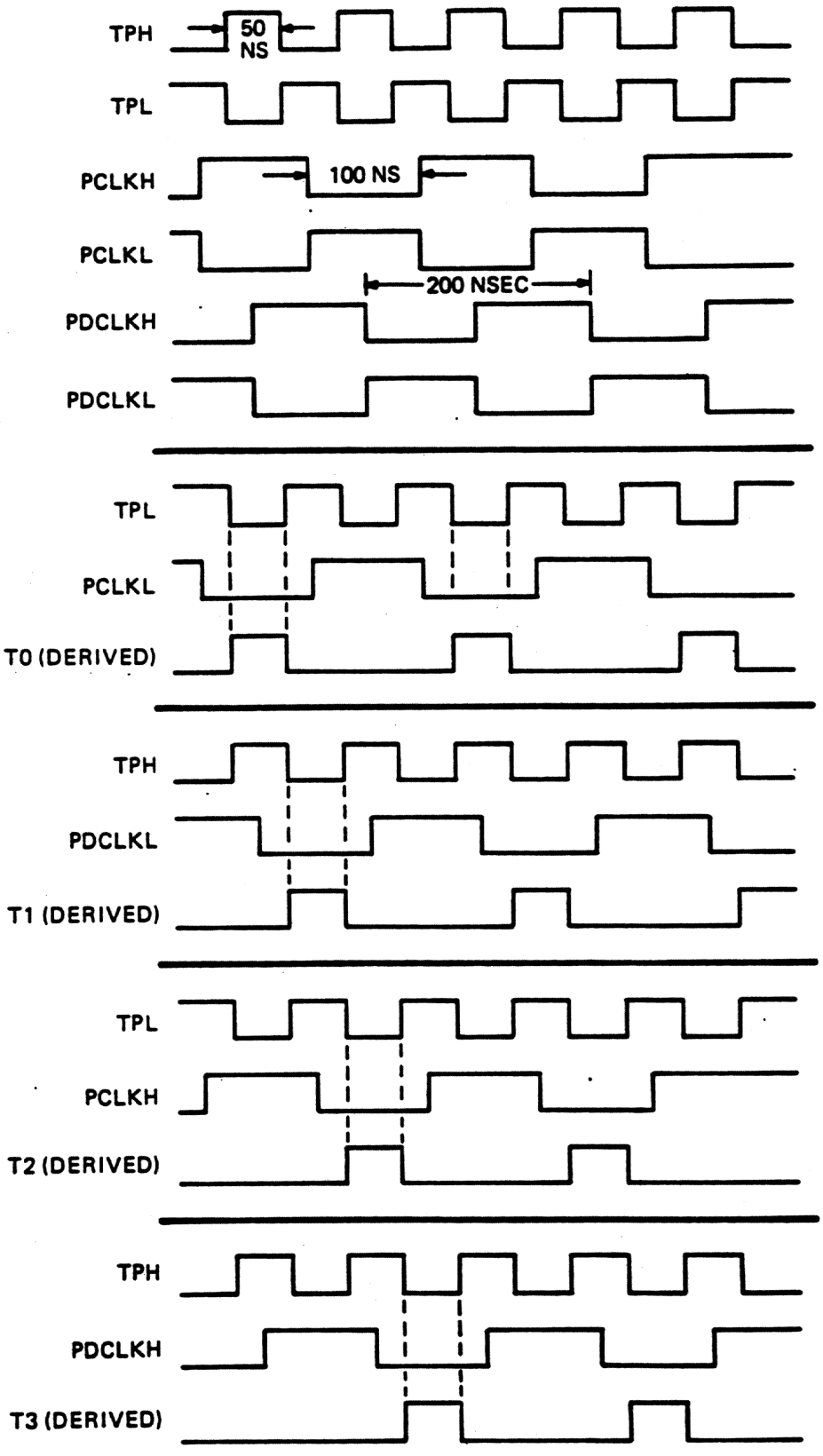


Figure 1-32 CPU and SBI Time States

1.8.2.2 Time of Year Clock -- The time of year clock is used by software to perform various timekeeping functions but its primary purpose is to provide the correct time to the system after power failures. This feature eliminates the need for an operator to enter the time at system restart.

The time which is initially input by the operator is converted by software into a binary number that represents the month, day, hour, etc. This value increases at time elapses. At the end of each year, software will reset the clock to the beginning of the year value.

1.8.2.3 Interval Time Clock -- The interval time clock provides a method of accurately measuring time intervals. The processor is notified of the completion of the time interval via an interrupt. This feature is used by software to perform time dependent events, accounting, and maintenance of software date and time.

There are three registers associated with the interval time clock operation; interval count register, next interval register, and clock control status register. Chapter 2 provides a description of these registers and clock operation.

1.8.3 Microsequencer

The microsequencer contains the logic required to generate the next microword address. The mode in which the address is generated is determined by a number of conditions (e.g., microtraps, stalls, console operations, etc.). The microsequencer monitors and prioritizes these conditions to select the proper source for the 13 microword address lines. If a decision point fork is reached in the microprogram, the instruction decode logic provides the source for the lower 8 address bits. The most significant address bit (bit 12) determines which control store will be addressed. If bit 12 equals 0, the PCS is accessed; if bit 12 is 1, the WDCS is accessed.

1.8.4 Control Store

The basic microprogram of the VAX-11/780 is contained in a standard 4K 99-bit PROM control store (PCS). The 99-bit control word (microword) is comprised of 96 data bits and 3 parity bits (1 for each 32-bit segment). Each microword is addressed by BUS UPC bits 12:00, generated by the microsequencer or the instruction decode logic.

The standard system configuration includes a 1K 99-bit writable diagnostic control store (WDCS). The control store is used to contain diagnostic microprogram routines and also updates to the basic microprogram.

Parity is checked on each microword read from either the PCS or WDCS. One parity bit corresponds to each 32 bit section of the 96-bit microword. Detection of an odd number of ones in any 33 bit field will result in a microtrap.

1.8.5 Data Path

The data path is divided into four functional areas: address, arithmetic, data, and exponent section. Each of the sections operates independently, allowing simultaneous processing of data and addresses.

1.8.5.1 Arithmetic Section -- The arithmetic section provides the circuitry for arithmetic and logic operations, bit mask and constant generation, shifting, and temporary storage of data or addresses. This section also provides the focal point of the data path. Data or address information is transferred between other sections of the data path via the ALU of the arithmetic section.

The arithmetic logic unit (ALU) is the main processing unit of the arithmetic section. The ALU performs arithmetic or logic operations on longword (32 bit) data types. Byte or word data types are sign or zero extended prior to being input to the ALU. The input sources of the ALU provide a number of operations including the following.

Generation of new PC -- The program counter is routed to the ALU through one of its input multiplexers to allow modification of the PC in certain addressing modes.

Operations on stored data -- Data to be used during instruction execution can be stored in the scratch pad register sets or in the D and Q registers of the data section. The operands stored in these registers are input to the ALU to allow performance of operations required by the current instruction. Multiplication and division of operands is accomplished by the shifter at the output of the ALU.

Restarting of instructions -- The register log (RLOG) and PC save (PCSV) inputs to the ALU allow instructions to be restarted after a fault. The RLOG stack contains a record of changes made to the scratch pad register set during instruction execution. The PCSV register contains the lower 8 bits of the PC at the beginning of an instruction.

Assembly of floating point data types -- During the execution of floating point instructions, inputs from the data, exponent, and control section are assembled by one input multiplexer of the ALU to form a packed floating point data type.

1.8.5.2 Address Section -- The address section contains the virtual address register (VA), instruction buffer address register (VIBA), and the program counter (PC).

The VA holds the address of the memory data referenced by the processor which is to be read or written into the data section. The VA will generally contain a virtual address which must be translated to a physical address to reference memory. However, the VA may hold a physical address which was generated during the

translation process or when the memory management mechanisms have been disabled. The VA can be incremented by four to advance the address by one longword.

The VIBA holds the address of the instruction stream data which is to be loaded into the instruction buffer. The VIBA is loaded with a new address whenever the instruction execution changes sequence such as a JUMP or successful BRANCH instruction. The instruction buffer control logic increments the VIBA by four each time instruction data has been successfully fetched from memory.

The PC holds the address of the instruction op code each time a new execution sequence is started. As operand specifiers of the instruction are evaluated, the PC is incremented by an appropriate value. As previously mentioned, a new PC can be generated by routing the contents through the ALU.

1.8.5.3 Data Section -- The data section contains the two major 32-bit holding registers (Q and D registers) used for temporary storage of operand data. This section provides the interface for the transfer of data to and from memory (via the memory data bus) and between internal registers (via the internal data bus). Also included is the circuitry required for the unpacking of floating point data types and the shifting and byte alignment of operand data.

The Q register holds the data transferred from the internal data (ID) bus and the D register holds data to be transmitted to the ID bus. Data received from memory or to be transmitted to memory is stored in the D register. The D and Q registers are used in conjunction to hold data types larger than 32 bits.

1.8.5.4 Exponent Section -- The exponent section of the data path processes the exponent value of floating point numbers. Exponent processing is performed in parallel with fraction processing performed in the arithmetic and data sections.

1.8.6 Instruction Buffer and Instruction Decode

The instruction buffer is basically an 8-byte register used to store instructions for evaluation by the processor. The op code of each instruction is stored in the first byte (byte 0) of the buffer register. The remainder of the instruction (operand specifier and extensions) is stored in subsequent bytes of the buffer register. The op code is kept in byte 0 while operand specifiers are evaluated. As each evaluation is completed, the operand specifier and associated data is removed from the buffer register and replaced with a new operand specifier. The process continues until all evaluations are complete and the instruction can be executed. The current op code is then removed and replaced by the op code of the next instruction. The structure of the buffer allows new instruction stream data to be prefetched and stored in upper byte locations while the current instruction is being evaluated for execution in the lower byte locations. The

ability to prefetch instruction stream data greatly enhances the overall performance of the processor.

The instruction decode logic evaluates the instruction stream data stored in the first two bytes of the buffer register. As previously mentioned, byte 0 contains the op code of the instruction and byte 1 contains an operand specifier. These bytes are decoded to generate the lower eight bits of the next microaddress when the microprogram reaches a decision point fork. Each time a fork is reached, the decoded instruction provides an entry point in the microprogram to a flow which evaluates an operand specifier or to an execution flow unique to the instruction.

1.8.7 Interrupts and Exceptions

Interrupts and exceptions are the result of events within the system which require the execution of software outside the current flow of control. Exceptions are the notification of events which are relevent to the currently executing process whereas interrupts are the notification of events which are generally independent of the current process.

Interrupts and exceptions are prioritized to determine the order in which events will be serviced. The processor has 31 interrupt priority levels (IPL), divided into 15 software levels and 16 hardware levels. Most exception service routines execute at the lowest interrupt priority level (IPL0). However, exceptions which represent serious system failures raise the IPL to the highest level (IPL 1F, hex). Interrupt levels 01 through 0F (hex) are dedicated for use by software. Interrupt levels 10 through 17 (hex) are for use by devices and controllers, including Unibus devices. Unibus levels BR4 to BR7 correspond the VAX-11 interrupt levels 14 to 17. Interrupt levels 18 to 1F (hex) are for use by urgent conditions, including the interval clock, serious errors, and power fail.

1.9 MODULE LOCATIONS

Table 1-9 lists the slot locations of each module in the KA780 Central Processing Unit backplane.

Table 1-9 KA780 Module Utilization

Module Number	Board Mnemonic	Function	Slot Location
M8236	CIB	Console Interface Board	29
M8289	FCT	Floating Point Accelerator*	28
M8288	FAD	Floating Point Accelerator*	27
M8287	FML	Floating Point Accelerator*	26
M8286	FMH	Floating Point Accelerator*	25
M8285	FNM	Floating Point Accelerator*	24
M8235	USC	Microsequencer	23
M8234	PCS	PROM Control Store	22
			21
M8233	WCS	Writable Diagnostic Control Store	20
			19
M8233 or M8234	OCS	Optional Control Store*	18
			17
M8232	CLK	Processor Clock	16
M8231	ICL	Interrupt Control	15
M8230	CEH	Condition Codes/Exceptions	14
M8229	DAP	Data Path	13
M8228	DCP	Data Path	12
M8227	DDP	Data Path	11
M8226	DEP	Data Path	10
M8225	DBP	Data Path	09
M8224	IRC	Instruction Decode	08
M8223	IDP	Instruction Buffer	07
M8222	TBM	Translation Buffer Matrix	06
M8221	CDM	Cache Data Matrix	05
M8220	CAM	Cache Address Matrix	04
M8219	SBH	SBI High Bits Interface	03
M8218	SBL	SBI Low Bits Interface	02
M8237	TRS	SBI Terminator plus Silo	01

*These are optional modules and if not included in the system, are replaced by blank modules.

CHAPTER 2
FUNCTIONAL/LOGIC DESCRIPTION

2.1 INTRODUCTION

Chapter 2 provides a detailed functional description of each major area of the KA780 central processing unit shown in Figure 2-1, excluding the following:

1. Translation Buffer, Cache, SBI Control
2. Console Interface and Q Bus Devices
3. Floating-point Accelerator

These functional areas are fully discussed in their associated manuals listed in Table 1-1.

2.2 MICROPROGRAM CONTROL

Execution of each VAX-11 or PDP-11 instruction requires the performance of a sequence of operations. This sequence is determined by the microprogram contained in the PROM control store (PCS) or writable diagnostic control store (WDCS). The PCS provides storage for 4K microwords and the WDCS provides storage for 1K. Each 96-bit microword is comprised of several fields which control particular functions in the processor. Figure 2-2 illustrates the format of the entire control word and defines each of the fields in the word. Descriptions of individual control word fields are provided in this chapter. They are included with the discussion of the logic which is affected by each particular field. The address of each microword is generated by the microsequencer or instruction decode logic. Address generation is described in Paragraph 2.3.

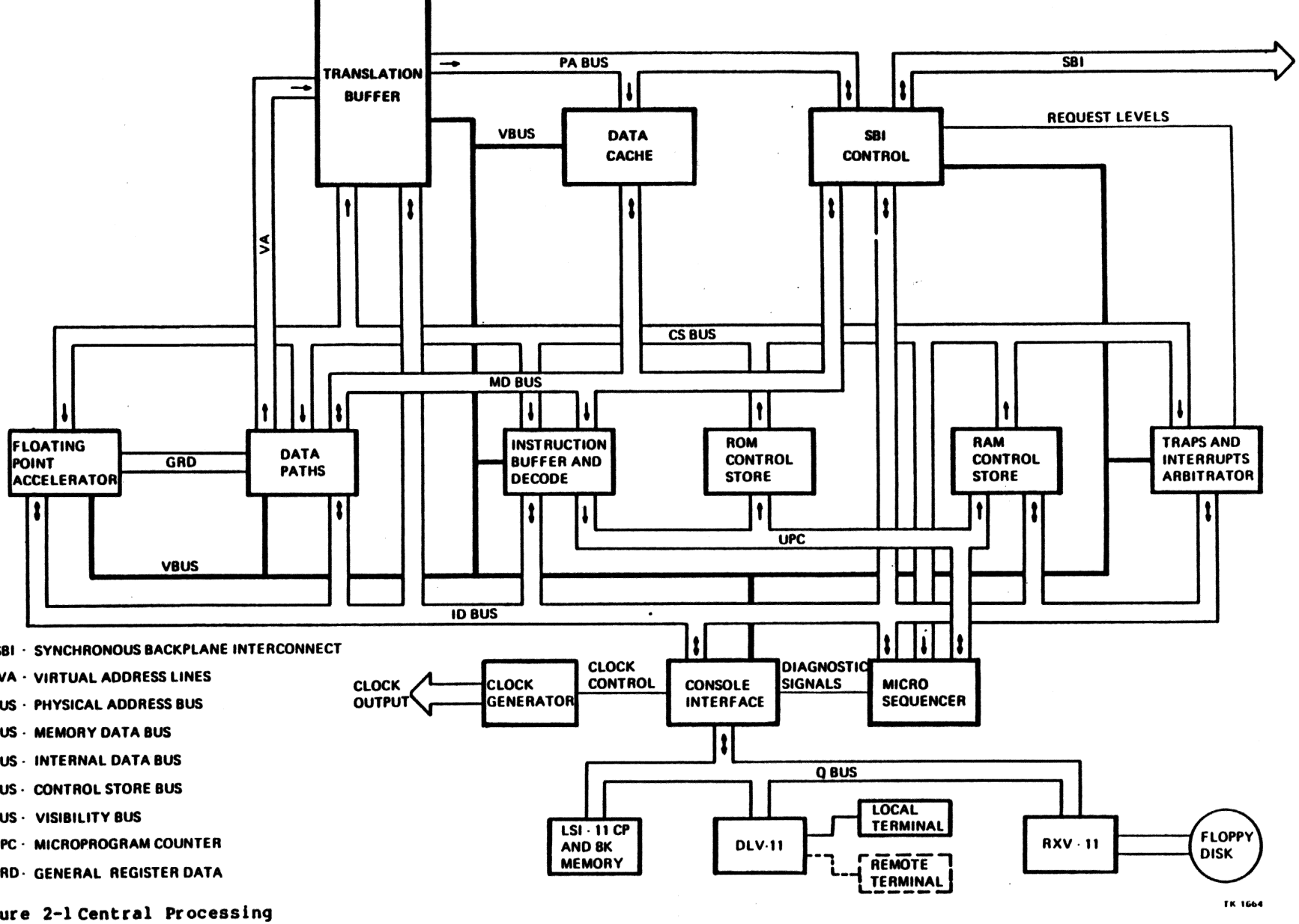


Figure 2-1 Central Processing Unit Block Diagram

Figure 2-1 Central Processing Unit Block Diagram

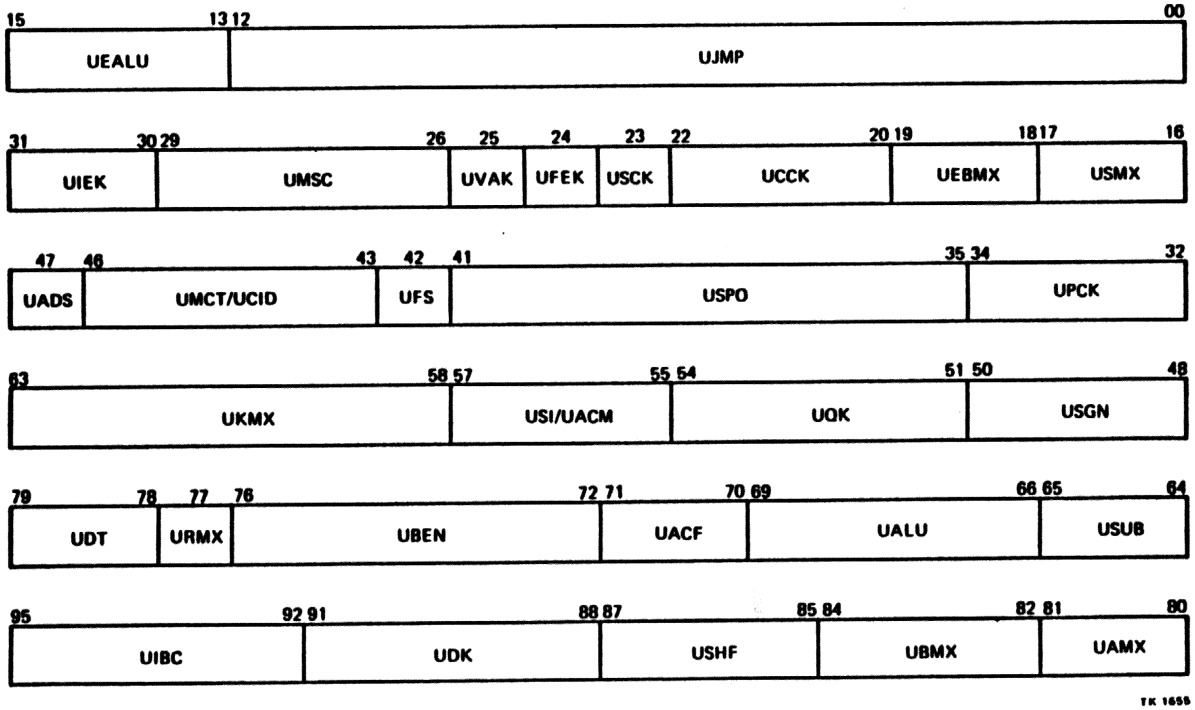


Figure 2-2 Microword Format and Field Definitions
(Sheet 1 of 5)

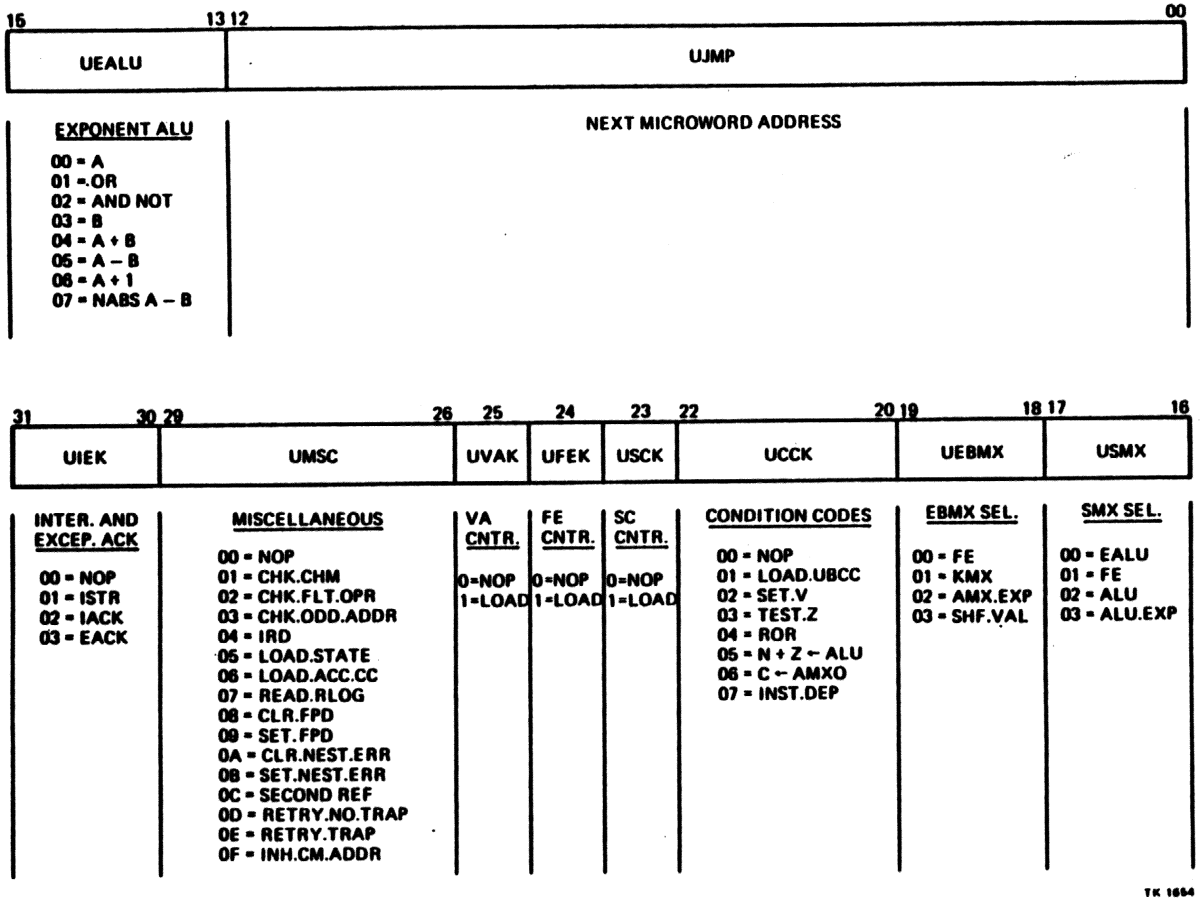


Figure 2-2 Microword Format and Field Definitions (Sheet 2 of 5)

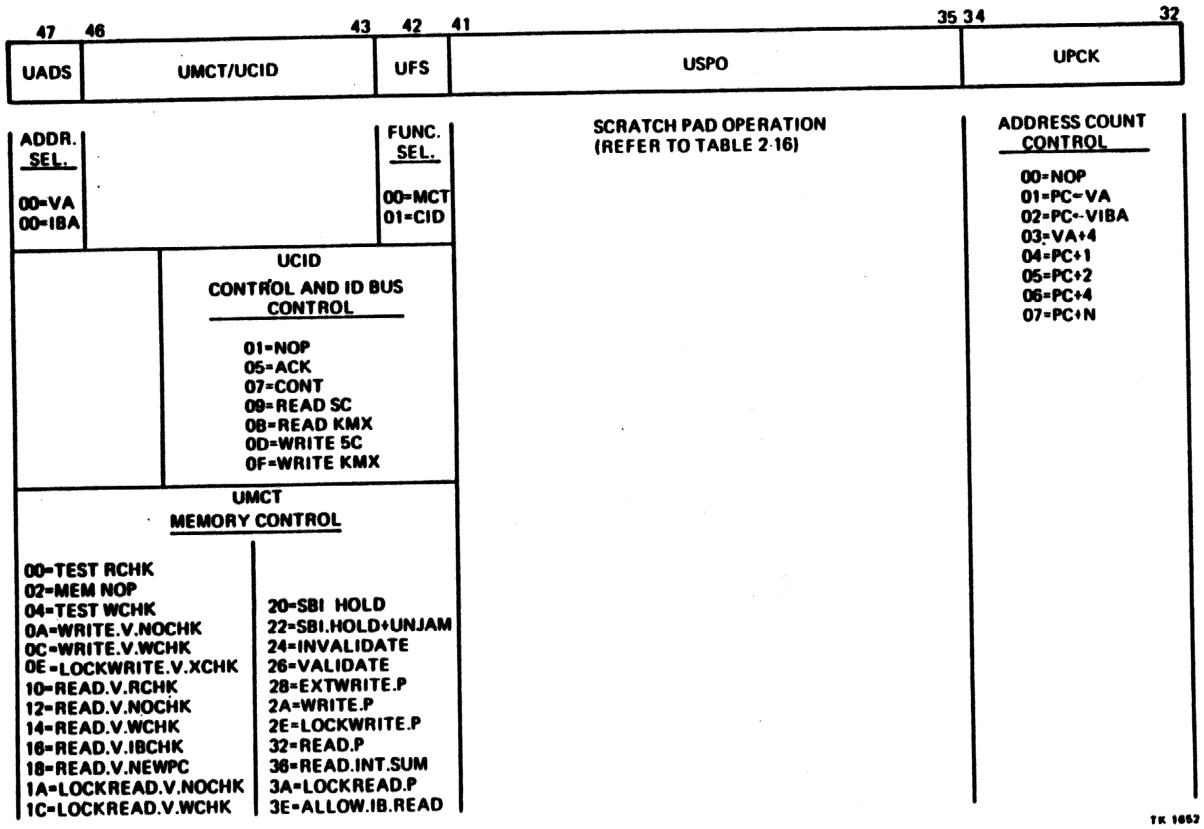


Figure 2-2 Microword Format and Field Definitions
(Sheet 3 of 5)

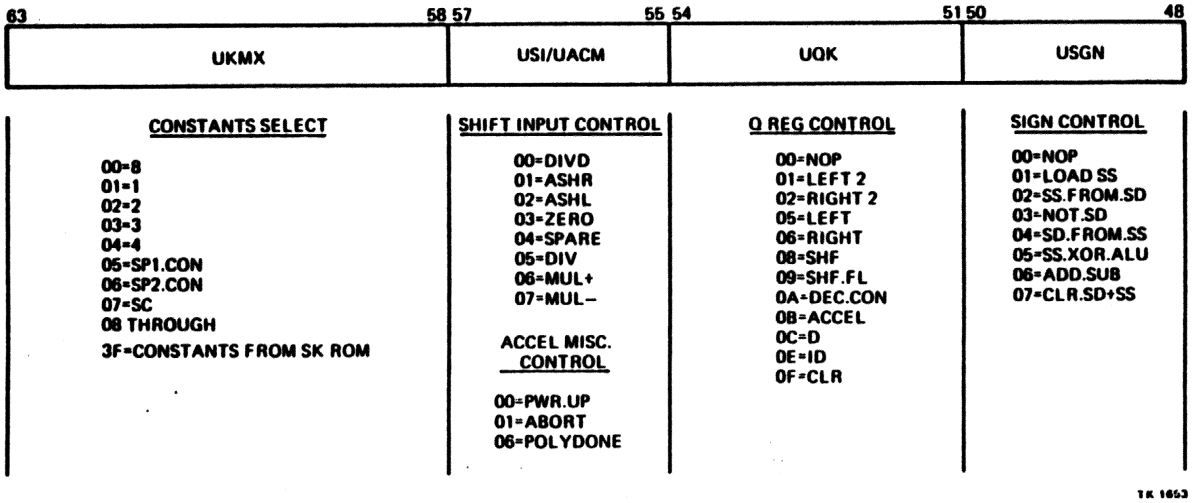


Figure 2-2 Microword Format and Field Definitions (Sheet 4 of 5)

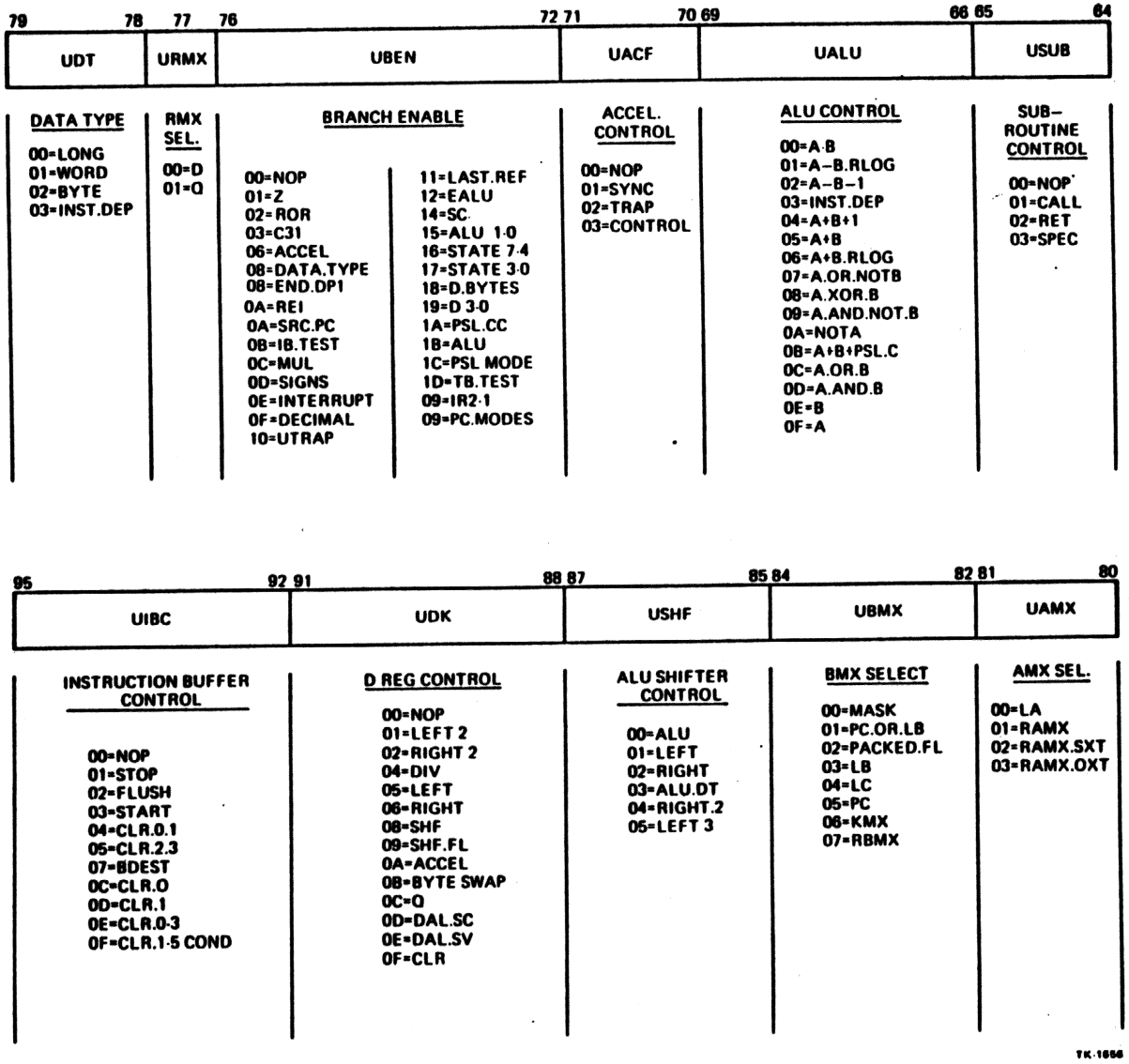


Figure 2-2 Microword Format and Field Definitions (Sheet 5 of 5)

2.2.1 How to Read the Microcode

This section introduces the microcode by describing the field, value, label and microinstruction definitions. Also included are definitions of macros, pseudo-operators, and location control.

2.2.1.1 Field Definitions -- Microcode field definitions have the form SYMBOL/=J,K,L,M. The J parameter is only meaningful when "D" is specified as the default mechanism. In that case, J gives the default value of the field in hexadecimal. The K parameter defines the field size in the number of bits (in decimal). The L parameter defines the field position (in decimal) as the bit number of the rightmost bit of the field. Bits are numbered from Ø on the right. The M parameter is optional, and selects a default mechanism for the field. The legal values of this parameter are the characters "D", or "+", where:

- D — Indicates that J is the default value of the field if no explicit value is specified.
- + — Is used on the jump address field to specify that the default jump address is the address of the next instruction assembled (not, in general, the current location +1).

In general, a field corresponds to the set of bits that provide select inputs for multiplexers or decoders, or controls for the ALU. For example:

```
ALU/=Ø,4,66,D
```

The microcode field which controls the ALU is four bits wide and the rightmost bit is shown in the listing as bit 66 of the microinstruction. If no value is specifically requested for the field, the microassembler will ensure that the field is Ø.

```
AMX/=Ø,2,8Ø
```

The field which controls the AMX is two bits wide, beginning on bit 8Ø. The fourth parameter of the field is omitted. Therefore, the field is available to the microassembler for modification if no value is explicitly called out for the field.

2.2.1.2 Value Definitions -- Following any field definition, symbols may be created in that field to correspond to values of the field. The form is:

```
SYMBOL=N
```

"N" is the value of the symbol (in hex) when used in the field. The following is an example:

```
ALU/=0,4,66,D   ;field definition in which one of the
                 following symbols exist.

XOR=8
A+B=5
```

Here the symbols "XOR" and "A+B" are defined for the ALU field. To the assembler, therefore, writing "ALU/XOR" means put the value 8 into the 4 bit field beginning on bit 66 of the microword. The symbols are chosen for mnemonic significance so that one reading the microcode would interpret "ALU/XOR" as "the output of the ALU shall be the exclusive OR or its A and B inputs." We could write "ALU/NOP" in every microinstruction in which we did not want the ALU to change. However, the default mechanism is used unless a microinstruction explicitly specifies a change to the ALU. The assembler will make the value of this field 0.

2.2.1.3 Label Definitions -- A microinstruction may be labeled by a symbol followed by a colon preceding the microinstruction definition. The address of the microinstruction becomes the value of the symbol in the field named "J". For example:

```
F00:J/F00
```

This is a microinstruction whose J field (jump address) contains the value "F00". It also defines the symbol "F00" to be the address of itself. Therefore, if executed by the microprocessor, it would loop on itself.

2.2.1.4 Comments -- A semicolon anywhere on a line causes the remainder of the line to be ignored by the assembler. It is only information for the reader. For example:

```
ALU/=0,4,66,D   ;field definition in which one of the
                 following symbols exist.
```

Only ALU/=0,4,66,D is relevant to the assembler.

2.2.1.5 Microinstruction Definition -- A word of microcode is defined by specifying a field name, followed by a slash (/), followed by a value. The value may be a symbol defined for that field, a hex digit string, or a decimal digit string (distinguished by the fact that it is terminated by a period). Several fields may be specified in one microinstruction by separating field/value specifications with commas. For example:
AMX/LA,BMX/D,ALU/A-B

The field named "AMX" is given the value LA (to cause the multiplexer on the A side of the ALU to select LA). The field "BMX" has value "D", and the field "ALU" has value "A-B".

2.2.1.6 Continuation -- The definition of a microinstruction may be continued on two or more lines by breaking it after any comma. If the last non-blank character on a line is a comma, the instruction specification is continued on the following line. For example:

```
AMX/LA,BMX/D      ;Select LA and D as ALU inputs
ALU/A-B           ;Select ALU to perform A-B
```

By convention, a blank line and a line of hyphens appears between microinstructions. This makes it easier for the reader to distinguish between a continuation and separate microinstructions.

2.2.1.7 Macros -- A macro is a symbol whose value is one or more field/value specifications. A macro definition is a line containing the macro name followed by a quoted string which is the value of the macro. For example:

```
D__LA-D           "AMX/LA,BMX/D,ALU/A-B,D__ALU""
```

The appearance of a macro in a microinstruction definition is equivalent to the appearance of its value. Macros may have parameters enclosed in square brackets ("[" and "]"). The definition of a macro with parameters includes paired brackets to indicate where the parameters should go. It uses "@" followed by a decimal digit string to indicate which symbols in the macro body should be replaced by the parameters. For example:

```
RC[]__D+K[]       "AMX/D,KMX/@2,BMX/KMX,ALU/A+8,SPO.RC/@1"
```

This macro indicates that the first parameter (selected by @1) should be used as the value in the "SPO.RC" field and the second parameter should be used as the value in the KMX field. A typical use of this macro might look like:

```
RC[T1]__D+K[34]
```

In this case, the expansion would be:

```
"AMX/D,KMX/34,BMX/KMX,ALU/A+B,SPO.RC/TI"
```

2.2.1.8 Pseudo-Operators -- The microassembler contains the following pseudo-operators listed in Table 2-1:

Table 2-1 Microassembler Pseudo-Operators

Pseudo Ops	Function
.UCODE and .DCODE	Selects the RAM into which subsequent microcode will be loaded and, therefore, the field definitions and macros that are meaningful in subsequent microcode.
.TITLE	Defines a string of text to appear in the page header.
.TOC	Defines an entry for the table of contents at the beginning.
.SET	Defines the value of a conditional assembly parameter.
.CHANGE	Redefines a conditional assembly parameter.
.DEFAULT	Assigns a value to an undefined parameter.
.IF	Enables assembly if the value of the parameter is not zero.
.IFNOT	Enables assembly if the parameter value is zero.
.ENDIF	Re-enables assembly.
.RTOL	Enables bits numbered from Ø on the right of the microinstruction.
.HEXADECIMAL	Enables radix to be 16 instead of default radix 8.
.REGION	Defines preferred ports of the .UCODE space.
.CREF and .NOREF	Enable and disable the collection of cross-reference information on symbol usage.
.LIST and .NOLIST	Enable and disable output listing.
.BIN and .NOBIN	Enable and disable leaving room at the left margin for binary output.
.MACHINE	Selects microassembler features needed for special microprocessor.

2.2.1.9 Location Control -- A microinstruction labeled with a number is assigned to that address. The character "=" at the beginning of a line, followed by a string of Øs, 1s, and/or *s, specifies a constraint on the address of the following microinstructions. The number of characters in the constraint string (excluding the "=") is the number of low-order bits contained in the address. The microassembler attempts to find an unused location whose address has zero bits in the positions corresponding to Øs in the constraint string and one bits where the constraint has 1s. Asterisks denote "don't care" bit positions.

If any zeros are in the constraint string, the constraint implies a block of (2 * * N) microwords, where N is the number of Øs in the string. All locations in the block have 1s in the address bits corresponding to 1s in the string. Bit positions denoted by *s are the same in all block locations.

In such a constraint block, the default address progression is counting in the "Ø" positions of the constraint string, but a new constraint string occurring within a block may force skipping over some locations of the block. Within a block, a new constraint string does not change the pattern of default address progression, it merely advances the location counter over those locations. The microassembler fills them in later.

A NULL constraint string ("=" followed by anything except Ø, 1, or *) serves to terminate a constraint block. For example:

a. = Ø

This specifies that the low-order address bit must be zero. The microassembler finds an even-odd pair of locations and places the next two microinstructions into them.

b. = 11

This specifies that the two low-order bits of the address must both be ones. Since there are no Øs in this constraint, the assembler finds only one locoation meeting the constraint.

c. = Ø*****

This specifies a pair of addresses; the first having a zero in thee "2Ø" bit, and the second having a one in that position. All other bit positions are the same.

2.3 MICROSEQUENCER AND CONTROL STORE

The primary function of the microsequencer is to provide the address of the control store word. The microsequencer controls entry into the microprogram during normal program flow and during the following operations;

Power Up/Power Down
Microtraps
Stalls
Microword ECOs
Console Operations

The address of the control store word is transferred to the PROM Control Store (PCS) and Writable Diagnostic Control Store (WDCS) over the Microprogram Counter (UPC) Bus. Under certain conditions (during a Decision Point Branch), a portion of the control store address (UPC bits 07:00) is generated by the instruction decode logic. Refer to the functional block diagram in Figure 2-3.

2.3.1 Microsequencer Mode Control (Picosequencer)

The source of control word address is dependent on the microsequencer mode of operation. The picosequencer determines the microsequencer mode by latching conditions generated in other sections of the CPU. These conditions are input to a priority decoder which determines the source of the control store address. The following lists the modes (conditions) in order of their priority.

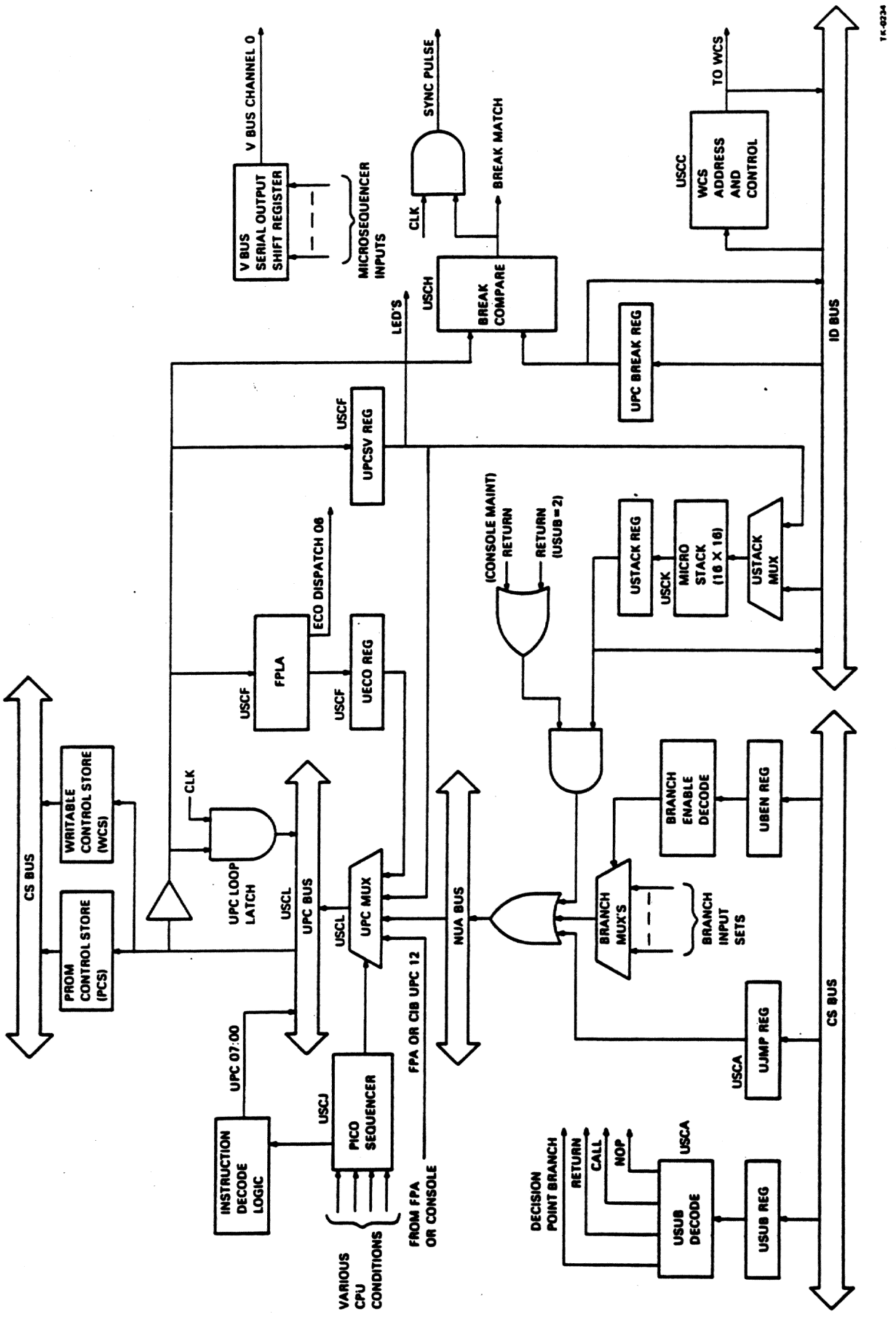


Figure 2-3 Microsequencer Functional Block Diagram

Highest Priority	Initialize (Power Up or Power Down)
	Maintenance Return (Console Operation)
	Cache Stall
	Microtrap
	Micro ECO
Lowest Priority	Normal

The outputs of the priority decoder are used to generate the select lines for Microprogram Counter Multiplexer (UPC MUX). The UPC MUX provides the address source for the control store word (via the UPC BUS). Refer to Figure 2-4.

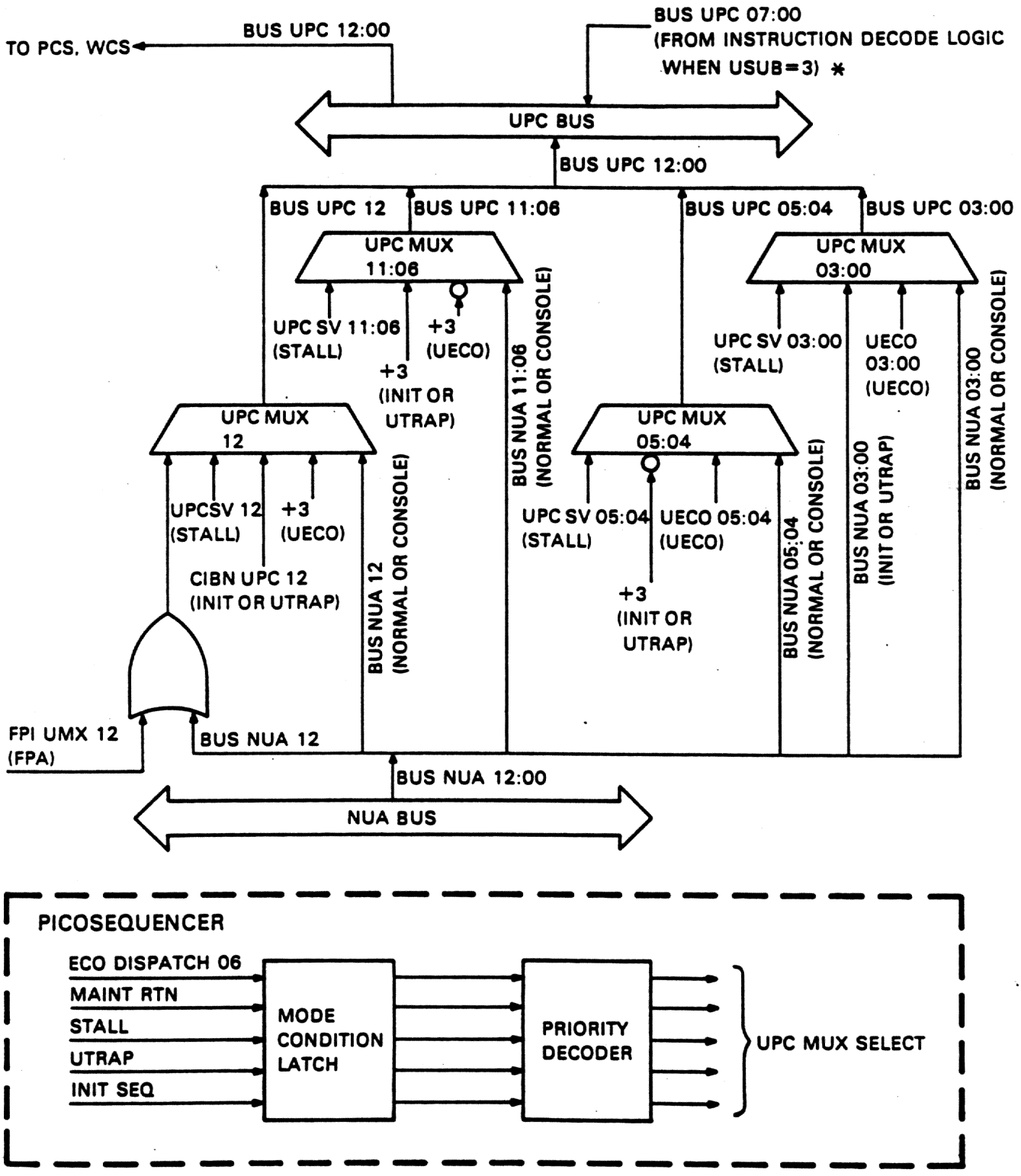
NOTE

Bit 12 of the UPC MUX determines which control store will be addressed. If bit 12 of the address equals Ø, the PCS is accessed; if bit 12 equals 1, the WCS is accessed.

The Microsubroutine (USUB) field of the current control word is also used to select the address source of the next control word.

USUB FIELD HEX	BUS CS 65	BUS CS 64	FUNCTION
Ø	L	L	NO-OP
1	L	H	CALL
2	H	L	RETURN
3	H	H	DECISION POINT BRANCH

Refer to Paragraph 2.3.3 for a description of the USUB field and its effect on microsequencer operation.



* NOTE:

WHEN USUB = 3 (DECISION POINT BRANCH), UPC MUX BITS 07:00 ARE DISABLED.

TK-0237

Figure 2-4 Picosequencer and UPC Mux

Table 2-2 shows the relationship between the conditions (mode) present and the data selected by the UPC MUX as the address source.

2.3.2 Microsequencer Mode Descriptions

This section provides a description of the operations performed by the microsequencer in each of the possible modes.

2.3.2.1 Normal Mode -- Under normal operating conditions, the UPC MUX selects the Next Microword Address (NUA) bus for the control word address. The data source for the NUA bus is the Jump (UJMP) field and the Branch Enable (UBEN) field of the current microword.

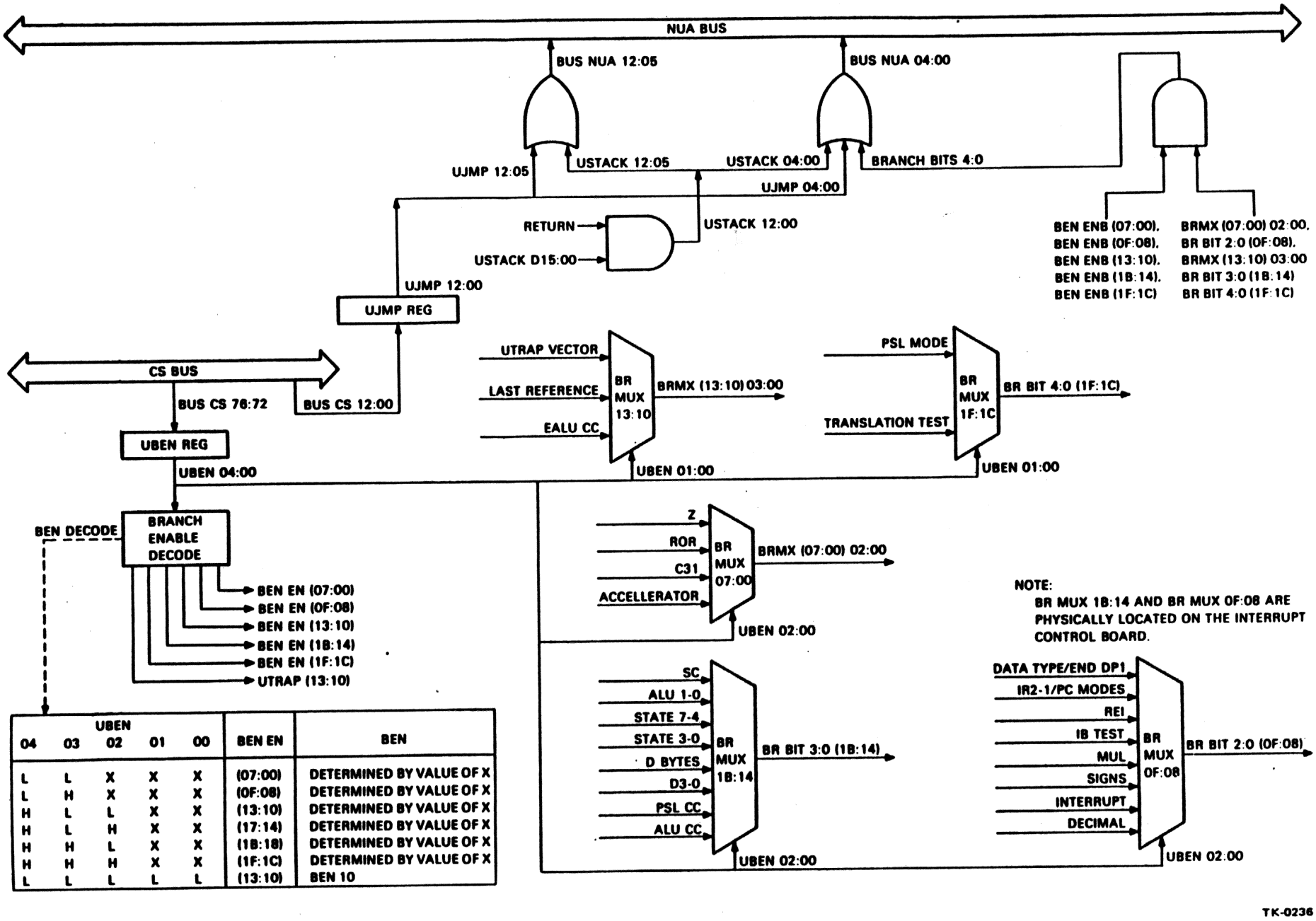
The J field comprises the lower 13 bits of the microword (BUS CS 12:00). BUS CS bits 12:00 are stored in the UJMP register of the microsequencer. The lower five bits of the UJMP register (UJMP 04:00) are ORed with the branch bits generated by the branch enable logic. The value of the BEN field of the microword (BUS CS 76:72) determines which branch conditions (generated in other areas of the CPU) will be used to modify the next microaddress. Refer to Figure 2-5.

The branch bits which are ORed with UJMP bits 04:00 are signals which monitor processor conditions and data values. The microaddress which is generated depends on which group of signals is selected by the branch enable logic. The following shows the relationship between the types of branch sets available and BEN field value.

	BEN Field Values	# of Selectable Branch Sets	UPC Lines Effected	Possible Branch Addresses Per Set
Group 1	0F:00	16	02:00	8
Group 2	1B:10	12	03:00	16
Group 3	1F:1C	4	04:00	32

Table 2-2 Control Word Address Source

MODE	CONDITION	UPC MUX DATA SOURCE SELECTED
POWER UP OR DOWN	INIT	12 | 11 09 | 08 | 07 04 | 03 00 = [ ] | 0 | 1 | 0 | 0 ; CIBN UPC 12
CONSOLE OPERATION	MAINT RET	12 00 = BUS NUA 12:00
CACHE STALL	STALL	12 00 = UPCSV 12:00
MICRO TRAP	UTRAP	12 | 11 09 | 08 | 07 04 | 03 00 = [ ] | 0 | 1 | 0 | BUS NUA 03:00 ; CIBN UPC 12
MICRO ECO	UECO	12 | 11 09 | 08 | 07 | 06 | 05 00 = 1 | 0 | 1 | 0 | 1 | UECO 05:00
NORMAL	NO COND.	12 00 = BUS NUA 12:00
NORMAL	USUB = 3	12 | 11 08 | 07 00 = [ ] | BUS NUA 11:08 | BUS UPC 07:00 ; FPA UMX 12



UBEN 04	03	02	01	00	BEN EN	BEN
L	L	X	X	X	(07:00)	DETERMINED BY VALUE OF X
L	H	X	X	X	(0F:08)	DETERMINED BY VALUE OF X
H	L	L	X	X	(13:10)	DETERMINED BY VALUE OF X
H	L	H	X	X	(17:14)	DETERMINED BY VALUE OF X
H	H	L	X	X	(1B:18)	DETERMINED BY VALUE OF X
H	H	H	X	X	(1F:1C)	DETERMINED BY VALUE OF X
L	L	L	L	L	(13:10)	BEN 10

Figure 2-5 Next Microaddress (NUA) Bus and Branch Logic

Table 2-3 lists each BEN value and branch set selected.

Table 2-3 Branch Condition Sets

Group 1 (8-way branches)		Group 2 (16-way branches)		Group 3 (32-way branches)	
BEN Value	**Branch Set Selected**	**BEN Value**	**Branch Set Selected**	**BEN Value**	**Branch Set Selected**
Ø	NOP	1Ø	UTrap Vector	1C	PSL Mode
1	Z	11	Last Reference	1D	Translation Test
2	ROR	12	EALU CC	1E	Not Used
3	C31	13	Not Used	1F	Not Used
4	Not Used	14	SC		
5	Not Used	15	ALU1-Ø		
6	Accelerator	16	State 7-4		
7	Not Used	17	State 3-Ø		
8	Data Type (VAX)	18	D Bytes		
8	END DPI (11)	19	D 3-Ø		
9	IR2-1 (VAX)	1A	PSL CC		
9	PC Modes (11)	1B	ALU CC		
A	REI				
B	IB Test				
C	MUL				
D	Signs				
E	Interrupt				
F	Decimal				

If the Subroutine Control (USUB) field of the current microinstruction equals 3, the lower 8 bits of the UPC MUX (UPC Ø7:ØØ) are disabled. When the microprogram reaches a Decision Point Fork, the low 8 bits of the next microaddress must be specified by the instruction decode logic via BUS UPC bits Ø7:ØØ. Also, when the USUB=3, bit 12 of the microaddress is defined by the OR condition of UJMP bit 12 and a line from the Floating Point Accelerator (FP1 UMX 12). This allows the Floating Point Accelerator (FPA), if present in the system, to direct the microaddress to the Writable Diagnostic Control Store (WDCS).

2.3.2.2 Microword ECO Mode -- In UECO mode, the microsequencer generates microaddresses which access the Writable Diagnostic Control Store. The ECO logic and WDCS allow sections of the microcode, contained in the PROM Control Store (PCS), to be rewritten without actually replacing PROM chips. The new or updated microcode is stored in the WDCS and is accessed when a changed PCS address is encountered. The Field Programmable Logic Array (FPLA) holds the PCS addresses which require changes and the corresponding WDCS addresses which contain the new microcode (refer to Figure 2-3).

When the current microaddress (BUS UPC 12:00) matches an address stored in the FPLA, and ECO dispatch signal is generated. This signal causes microword registers in various sections of the CPU to be cleared and also causes an abort cycle signal to be sent to the translation buffer. These events effectively create a NO-OP cycle. The NO-OP cycle allows the new microaddress to be formed using the ECO bits read from the FPLA. The ECO dispatch signal also enables the picosequencer to select ECO mode if no higher priority conditions are enabled.

When the UPC bits match the address stored in the FPLA, the corresponding bits for the new microaddress are loaded into the UECO register. The contents of the UECO register (UECO 05:00) are loaded into the UPC MUX to form the next microaddress. In UECO mode, the UPC MUX selects UECO 05:00 as the source for the lower 6 bits of the microaddress and the remaining seven bits are hardwired. Figure 2-6 shows the format of the microaddress formed during ECO mode.

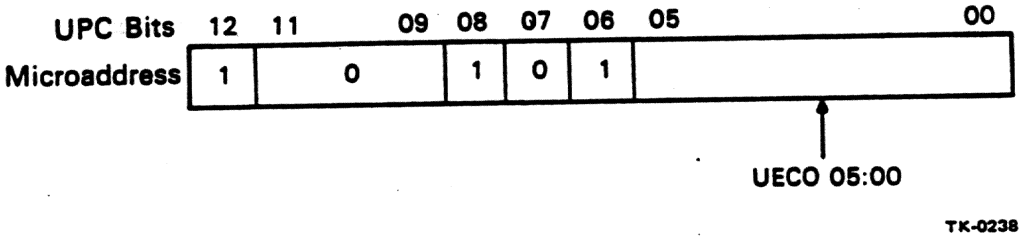


Figure 2-6 Microaddress Format in UECO Mode

The FPLA provides storage for 48 addresses, thereby allowing 48 different changes to be made to the microprogram in PCS. However, more than 48 microinstructions can be replaced. The FPLA only provides the starting address of the revised program section in WDCS. Each ECO may include several microinstructions. The microprogram will be directed back to the PROM Control Store by the UJMP and UBEN fields of the final microinstruction of each revised section.

2.3.2.3 Microtrap Mode -- In utrap mode, the microsequencer generates specific vector locations (in WDCS or PCS) which address trap handling conditions in the CPU. The presence of these error conditions causes the picosequencer to select utrap mode if no higher mode conditions are present. The reception of the utrap signal by the microsequencer causes microword registers to be cleared and an abort cycle to be generated. As in UECO mode, these conditions create a NO-OP cycle in which a new microaddress (vector) can be formed.

Microtrap mode forces the UPC MUX to select BUS NUA 03:00 as the source for the lower 4 bits of the vector address. Bits 11:04 of the vector are hardwired and bit 12 is determined by the console. Refer to Figure 2-4. If bit 12 is a 1, the vector address is in the WDCS and if bit 12 equals 0, the vector is in PCS. BUS NUA bits 03:00 are formed by the branch bits. The utrap signal causes the UJMP and UBEN fields of the microword to be cleared. If the UBEN field equals 0, a particular branch enable set (BEN 10) is selected. A BEN 10 selects inputs from the interrupts and exception logic. This logic will control the value of BUS NUA bits 03:00 and thereby select the proper vector location for the utrap encountered. The UJMP and USUB fields will be cleared and will not effect the vector generation. Figure 2-7 shows the format of the microaddress (vector location formed during microtrap mode).

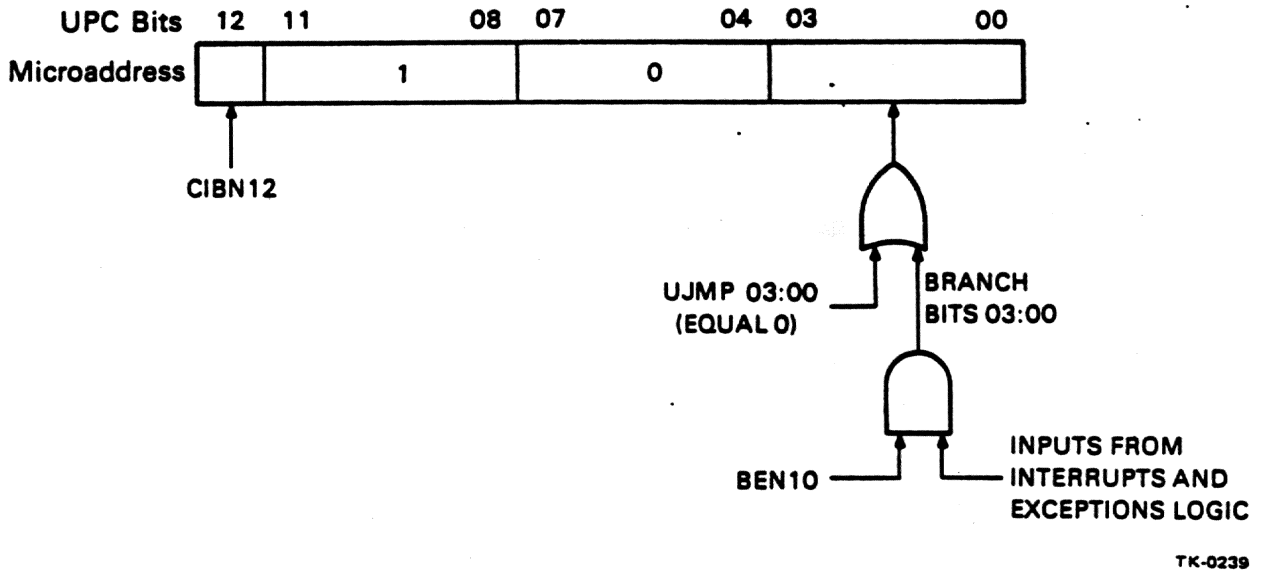


Figure 2-7 Microaddress Format in UTrap Mode

When a utrap sequence is initiated, the contents of the Microprogram Counter Save Register (UPCSV) are pushed onto the Microstack. The UPCSV contents specify the address of the next microword that would have been accessed under normal conditions. This address is saved so that the microprogram can be returned to normal flow after the trap has been serviced. The ustack pointer (USP) address is decremented prior to writing data onto the stack. Once the utrap routine has been completed, the microstack location containing the next normal address is read from the stack and loaded into the ustack register. The output of the ustack register (ustack 12:00) is then routed to the NUA bus. The contents of the ustack register are enabled onto the NUA bus by the RETURN signal, generated when the USUB field equals 2. The UPC MUX will select BUS NUA 12:00 as the source for the next microaddress.

The Control Store Parity Error microtrap is an exception with regard to the storage of the UPCSV register contents on the microstack. If there is a control store parity error, the clocking of the UPCSV register is inhibited. Therefore, the UPCSV data which is loaded onto the stack is the failing microword address rather than the address of the next executable microword. The next executable microword is loaded on the stack if any other utrap occurs.

The range of utrap vector addresses is as follows:

Control Store	Vector Address Range (Hex)
PCS	0100--010F
WCS	1100--110F

A signal from the console (CIBN UPC 12) controls the value of the most significant address bit (UPC 12) and therefore determines which control store will be accessed.

The following lists each microtrap and its associated vector location.

Vector Address	Microtrap
X100	System Init
X101	Unaligned Data
X102	Page
X103	M bit
X104	Protection Violation
X105	TB Miss
X106	Reserved Floating
X107	TB Parity
X108	Cache Parity
X109	Reserved
X10A	Reserved
X10B	Reserved
X10C	Read Data Substitute
X10D	Time Out
X10E	Odd Address
X10F	Control Store Parity

If X = 0, the vector address is in PCS
If X = 1, the vector address is in WCS

Multiple utrap conditions can be present at the same time. Therefore, the conditions are input to a priority decoder to determine which microtrap will be serviced first. The relative priorities of each utrap are listed listed as follows:

Highest	System Init
	CS Parity Error
	Odd Address Error
	Time Out
	Read Data Substitute
	Cache Parity Error
	Translation Buffer Error
	Reserved Floating Operand
	Translation Buffer Miss
	Protection Violation
	Modify Bit
	Page Trap
Lowest	Unaligned Data

2.3.2.4 Cache Stall Mode -- Cache stall mode is initiated if the signal SBLT STALL is sent to the microsequencer from the SBI control. In this mode, the execution of the next microinstruction is temporarily prevented. A number of conditions can cause the stall signal to be generated (e.g., if during a read operation, the requested data is not in cache).

If the cache stall mode is enabled, the clear u word signal is generated. This effectively puts the microprogram in a NO-OP state. The NO-OP state may last for several microcycles until the condition causing the stall is negated.

Once the stall condition is negated, the UPC MUX selects the UPCSV register contents (UPCSV 12:00) as the source for the next microaddress. The UPCSV register will contain the address of the next instruction that would have been executed if the stall had not occurred.

NOTE

The Microsequencer board provides 13 LEDs which display the contents of the UPCSV register and a single LED which displays the stall condition.

2.3.2.5 Maintenance Mode -- In maintenance mode the console can control a number of functions, including the selection of the next microaddress. Data can be written from the console to the microsequencer via the Internal Data (ID) bus. The destination of the data written over the ID bus is determined by the ID bus control logic. Refer to Figure 2-8.

The ID bus consists of 32 data lines. Control of this data is determined by an additional 7 lines. One control line (ID WRITE) specifies whether data is being written into a specified register from the ID bus or if data is being read from the register onto the ID bus. The remaining 6 lines (ID ADDR 5:0) provide the address of the register to be read or written. It should also be noted that the ID bus is divided in half. That is, half of the addressable registers on the ID bus are viewed as being to the right of the ID bus control and the other half to the left of the ID bus control. The right and left lines are buffered separately to accommodate the loading on the ID bus.

The source of the address lines and the write control line is the same for both the right and left halfs of the bus. The right or left designation simply indicates the position of the register relative to the control logic. The microsequencer is to the left of the ID bus control and therefore its registers are addressed by the lines ID LEFT ADDR 5:0 and the direction of the data flow is controlled by the ID LEFT WRITE line. In maintenance mode, the value of the address lines and control line is determined by the console. The console initiates maintenance mode by asserting the ID MAINT signal. The following shows the address of the microsequencer registers which can be read or written from the ID bus.

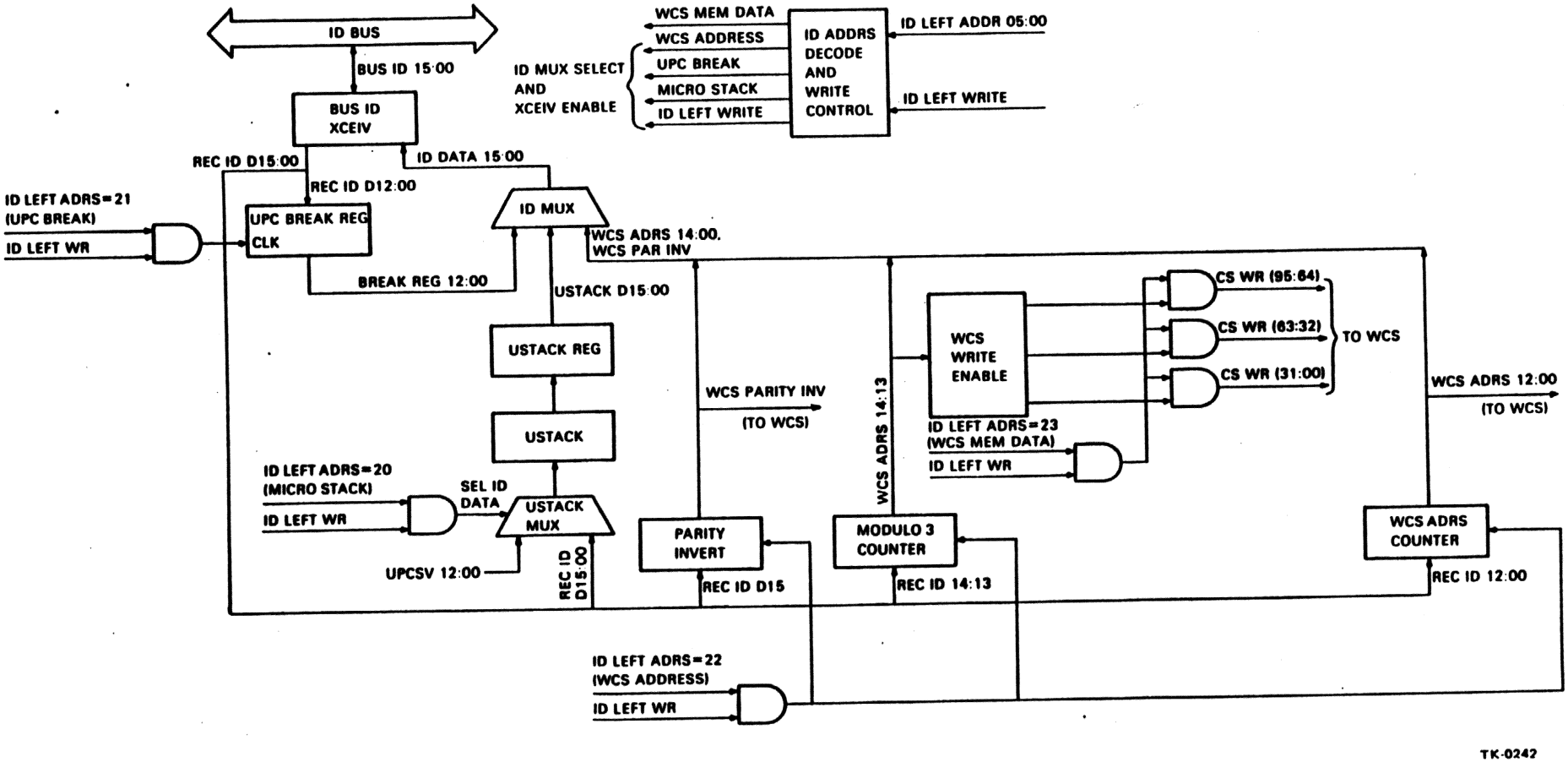


Figure 2-8 Microsequencer Internal Data (ID) Registers

Microsequencer Register	ID Address (Hex)
USTACK	20
UBREAK	21
WCS ADDRESS	22
WCS MEM DATA	23

In maintenance mode, the console can specify the next microaddress by writing into the microstack over the ID bus. If ID LEFT ADDR 5:0 equals 20 (hex) and the ID LEFT WR signal is asserted, the ustack mux will select data (REC ID 15:00) from the ID bus. Refer to Figure 2-8. The microstack pointer address is decremented and then the ID data is written into the stack. The MAINT RTN signal from the console will enable the picosequencer to select maintenance mode, providing the INIT condition is not present. Maintenance return will cause the ustack data to be loaded onto the NUA bus and will enable the UPC MUX to select the NUA bus as the source for the next microinstruction. Note that the ustack register data can also be loaded onto the ID bus when a read microstack function is indicated by the ID BUS and control lines.

The UPC Break register is also read or written over the ID bus under console control. When ID LEFT ADDR 5:0 equals 21 (hex) and the ID LEFT WR signal is asserted, data from the ID bus (REC ID 12:00) is loaded into the Break register. The output of the Break register (BRK REG 12:00) is routed to a compare network and is also fed back to the ID bus so the register contents can be read. The other input (BUF UPC 12:00) to the compare network is the address of the next microinstruction to be performed. When the address loaded into Break register matches the address of the next microinstruction, the comparator will generate a break match (BRK MAT) signal. The BRK MAT signal will stop the clock if the enable bit in the console is set. The BRK MAT signal is also routed to the back panel (SYNC PULSE). If the console enable bit is not set, the SYNC PULSE signal can be used for an oscilloscope sync on the UPC address specified in the Break register. The console can use the microsequencer to stop the clock at a specific microaddress by writing that address over the ID bus and into the Break register. The console can also force the microsequencer into a NO-OP cycle by asserting the ROM NOP signal. This signal will cause the microword registers to be cleared and the abort cycle signal to be generated.

The two remaining microsequencer registers that are addressable on the ID bus are WCS Address and WCS Memory Data. These two registers are used to write data into the Writable Control Store and are discussed in Paragraph 2.3.8.

The V bus serial output shift register is also used for maintenance purposes. A number of microsequencer lines are parallel loaded into the V bus register and then read out serially. The V bus has 8 serial channels that are selectable. Channel Ø is designated for the microsequencer. The V bus allows observation of numerous microsequencer conditions provided that the clock is stopped.

2.3.2.6 Initialize Mode -- Power Up or Power Down cause the Init signal to be generated. The initialize condition forces the microsequencer to place a constant microtrap vector (xlØØ) on the UPC bus. The value of x, determined by the console, indicates whether the vector is in PCS (x = Ø) or WCS (x = 1). During the init condition, all microsequencer registers are clear except the V bus, UPCSV, UPC Break and UECO registers.

The microsequencer remains in the initialize mode for one microcycle after the system INIT level is negated. When power becomes good, the J and Ben fields of the microinstruction at vector location xlØØ will determine the microinstruction address for start up.

2.3.3 Micro Subroutine (USUB) FIELD
The USUB field of the microinstruction will specify a CALL subroutine, RETURN from subroutine, or a Decision Point Branch.

USUB FIELD

HEX	BUS CS 65	BUS CS 64	Function
Ø	L	L	NO-OP
1	L	H	CALL
2	H	L	RETURN
3	H	H	DECISION POINT BRANCH

If a CALL subroutine is specified (USUB = 1), the contents of the UPCSV register are pushed onto the ustack. The saved address will later be used to form the return address. The microstack pointer (USP) is decremented prior to push operation.

When the return from subroutine is specified, the UPCSV contents are popped from the stack and ORed with the J field and branch condition of the return microinstruction. The result of the OR condition will specify the correct address of the next microinstruction past the CALL instruction. The USP is incremented after the data is popped from the microstack.

If the USUB field equals 3, a Decision Point Branch is enabled. The lower eight bits of the UPC MUX are disabled, thereby allowing the instruction decode logic to determine UPC bits 07:00 of the microaddress.

2.3.4 UPC Loop Latch

Each new microaddress is latched for a specific time period to prevent a false uword parity error. During the critical period, the UPC MUX tristates are disabled and the UPC Loop Latch is closed. This latching network keeps the UPC lines stable and prevents the control store (CS) bus lines from changing when parity is being checked. After the critical period the latches are opened and the UPC MUX is enabled which allows the next microaddress to be put on the UPC bus.

2.3.5 Microstack Operation

The microstack functions as a Last On/First Off storage unit. The data popped from the stack in a read operation will be the same data which was pushed on the stack in the last write operation. To perform the correct push/pop sequence, the microstack pointer (USP 03:00) is decremented before data is written and incremented after data is read. The conditions which cause the microstack to be written are a utrap ANDed with NOT STALL, a BUS ID write to the microstack or the USUB field equal to 1 (CALL). Refer to Figure 2-9. These conditions enable the USTACK DEC and USTACK LOAD signals. USTACK LOAD causes the microstack pointer to be loaded into the Current Address register which is clocked at the beginning of a CPU cycle (T0). USTACK DEC selects the microstack pointer decremented by 1, and also enables a write into the microstack. The data is not clocked into the stack until approximately T125 of the same CPU cycle. Therefore, the microstack pointer is decremented before data is written onto the stack.

A microstack read operation is initiated if MAINT RET is issued by the console or if the USUB field equals 2 (RETURN). The signals USTACK INC and USTACK LOAD are generated during the read sequence. USTACK LOAD causes the Current Address register and USTACK register to be loaded. Both registers are clocked at T0 which causes the data to be read from the stack into the USTACK register before the stack pointer address is incremented.

Since the microstack can be used to store up to 16 words (16 bits each), the stack user must keep track of the number of writes performed in order to read the correct location when a pop microstack is specified.

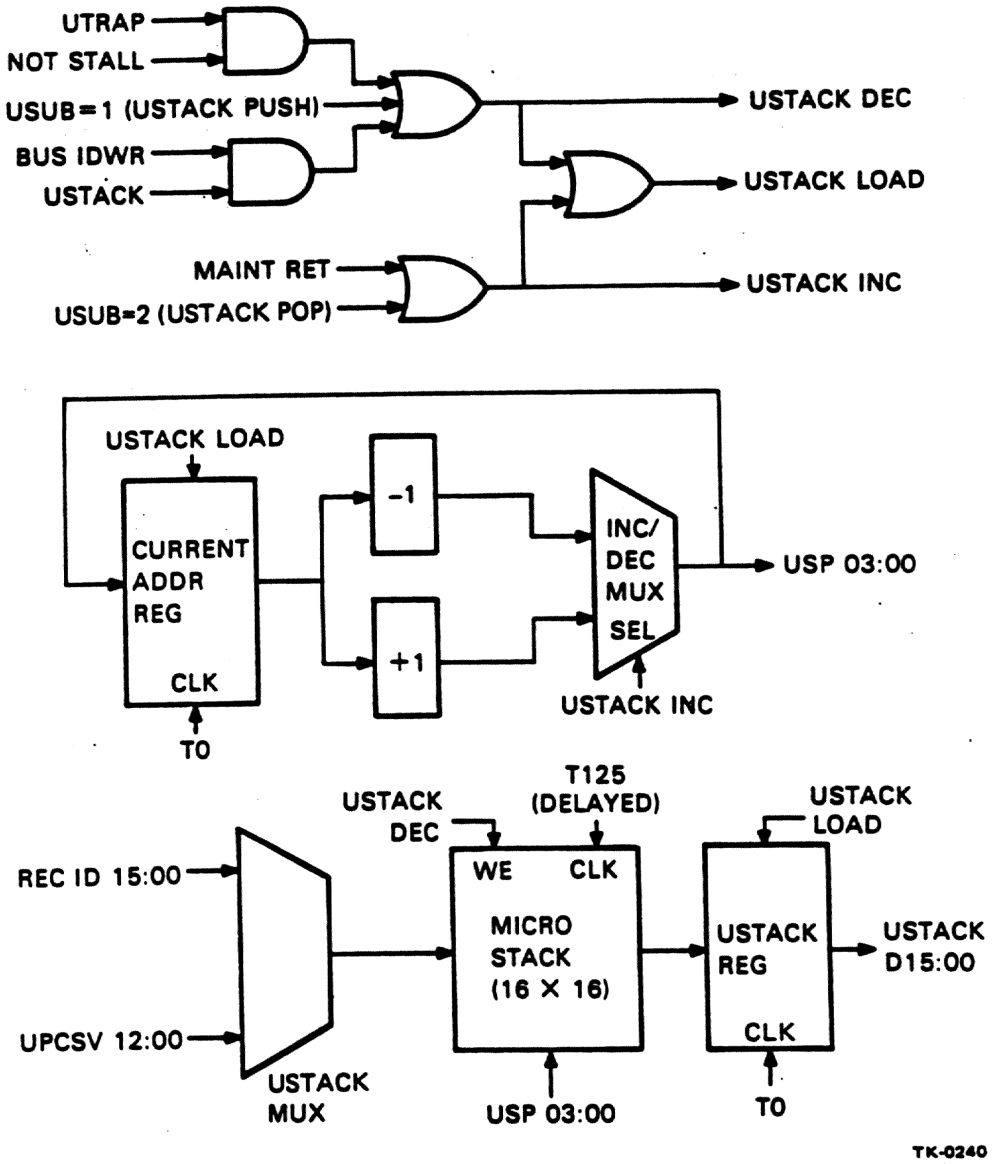


Figure 2-9 Microstack Operation

2.3.6 Control Store Configuration

Three slots in the processor backplane are dedicated for control store modules. The maximum control store allowed is 8K 99-bit words. The standard configuration (using two slots) will contain 4K of PROM Control Store (PCS) and 1K of Writable Control Store (WCS). Optional configurations can use the third slot for an additional 1K of WCS. Refer to Figure 2-10.

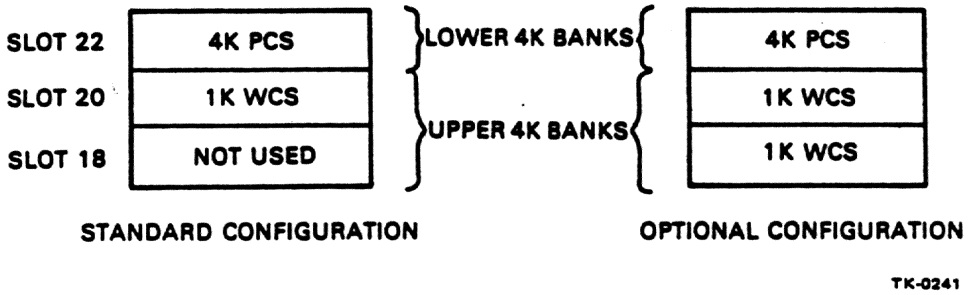


Figure 2-10 Control Store Configuration

Each backplane slot contains jumpers (J5 through J1) which are used to enable or disable 1K banks of control store. Jumper J5 will enable either the upper or lower 4K group. Jumpers J4 through J1 enable specific banks within the 4K group. Table 2-4 shows the correspondence between jumper placement and bank selected. The modules can be interchanged within the three slots since the jumpers are on the backplane and can be connected accordingly.

Table 2-4 Control Store Bank Selection

Bank Selected	Jumper Connection					
	J5(PCS)	J5(WCS)	J4	J3	J2	J1
1K	IN	OUT	IN	IN	IN	OUT
2K	IN	OUT	IN	IN	OUT	IN
3K	IN	OUT	IN	OUT	IN	IN
4K	IN	OUT	OUT	IN	IN	IN
5K	OUT	IN	IN	IN	IN	OUT
6K	OUT	IN	IN	IN	OUT	IN
7K	OUT	IN	IN	OUT	IN	IN
8K	OUT	IN	OUT	IN	IN	IN

2.3.7 PROM Control Store (PCS)

The standard PROM Control Store contains 4K 99-bit words. The 99-bit control word (microword) is comprised of 96 data bits and three parity bits (1 for each 32-bit segment). Refer to Figure 2-11.

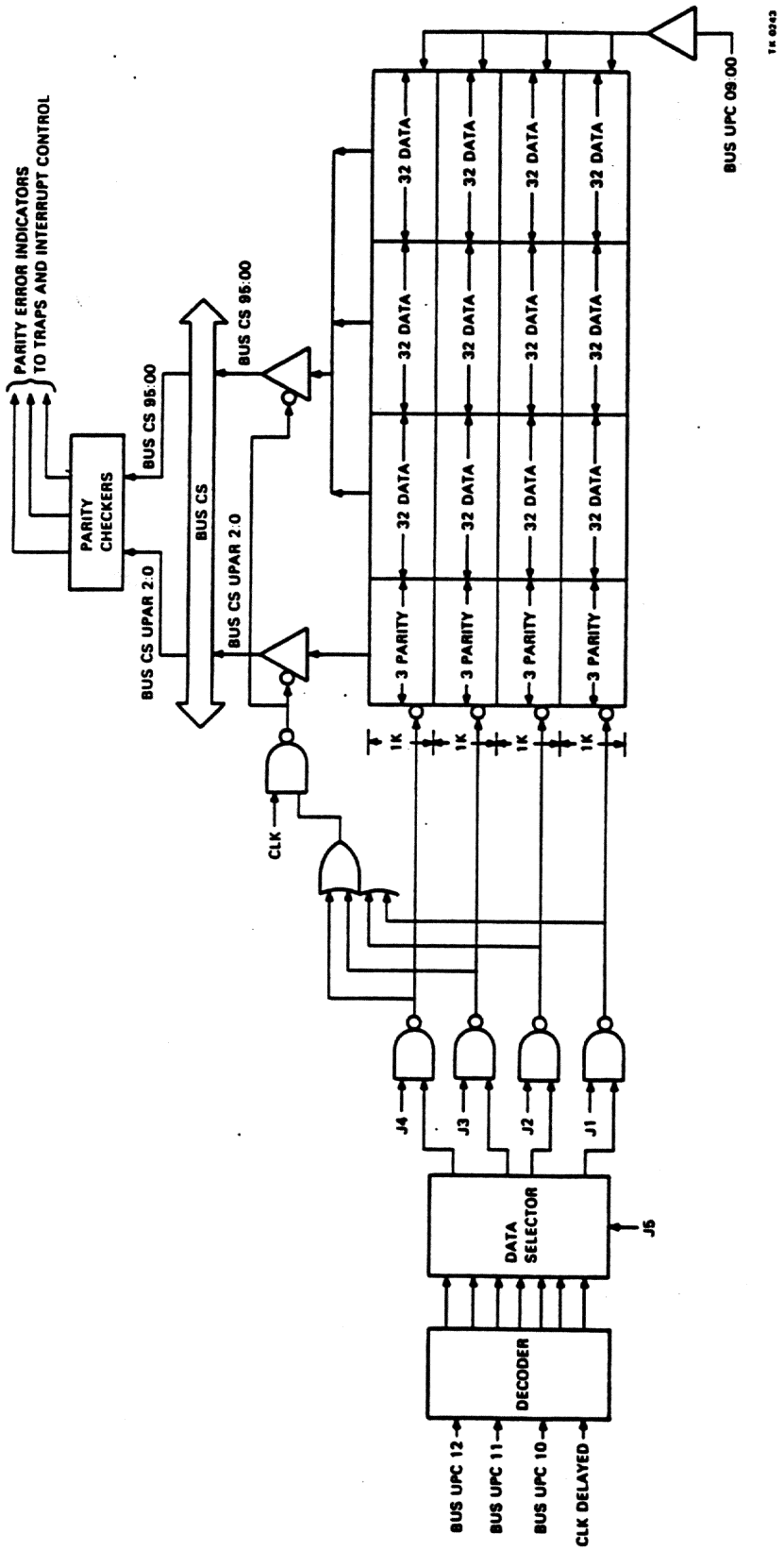


Figure 2-11 PROM Control Store (PCS)

Each microword is addressed by BUS UPC bits 12:00. These lines are generated by the microsequencer or the instruction decode logic (during a decision point branch). BUS UPC bits 12:01 enable a specific 1K bank of control store and the CS bus drivers if the UPC lines match the corresponding address jumpers. BUS UPC 12 addresses either the lower 4K bank (PCS) or the upper 4K bank (WCS and optional control store). Bits 11 and 10 determine which 1K bank of the 4K group is enabled. BUS UPC 09:00 address each microword in the 1K bank selected.

Parity is checked on each microword that is read from the PCS. One parity bit corresponds to each 32 bit section of the 96 bit microword. The parity bit is used to make an even number of ones in the 33 bit field (1 parity and 32 data). If the parity checkers detect an odd number of ones in any 33 bit field, a control store parity error is sent to the Interrupt Control logic, resulting in a microtrap.

NOTE

The PCS parity checkers are also used to detect parity errors in the Writable Control Store.

2.3.8 Writable Control Store (WCS)

The standard Writable Control Store contains 1K 99-bit words. As in the PCS, each microword contains 96 data bits and 3 parity bits. Refer to Figure 2-12.

The WCS is addressed in the same manner as the PCS during read operations. BUS UPC bits 12:10 enable the CS bus drivers if the UPC lines match to corresponding address jumpers. The chip enable for this 1K control store is always turned on. BUS UPC bits 09:00 address each microword in the 1K WCS.

During write operations, both the address and data to be written into WCS are generated by the console and transferred over the ID bus. Data must be written into WCS in three 32 bit segments since the ID bus is only 32 bits wide and the microword is 96 bits. The address generated in the console is transferred to the WCS Address register in the microsequencer. Refer to Figure 2-8.

The WCS address register consists of a 13 bit address counter, a modulo 3 counter, and a parity invert bit. This register is loaded from the ID bus when ID LEFT ADDR 05:00 equals 22 (hex) and the ID LEFT WR signal is asserted. The parity invert bit is not part of the WCS address counter. This bit is used for diagnostic purposes to generate odd parity and force a parity error microtrap. The signal WCS PAR INV is sent from the microsequencer to the parity generator in the WCS. Refer to Figure 2-12. WCS PAR INV controls whether even or odd parity is generated for each 32 bit segment written. The normal parity written is even.

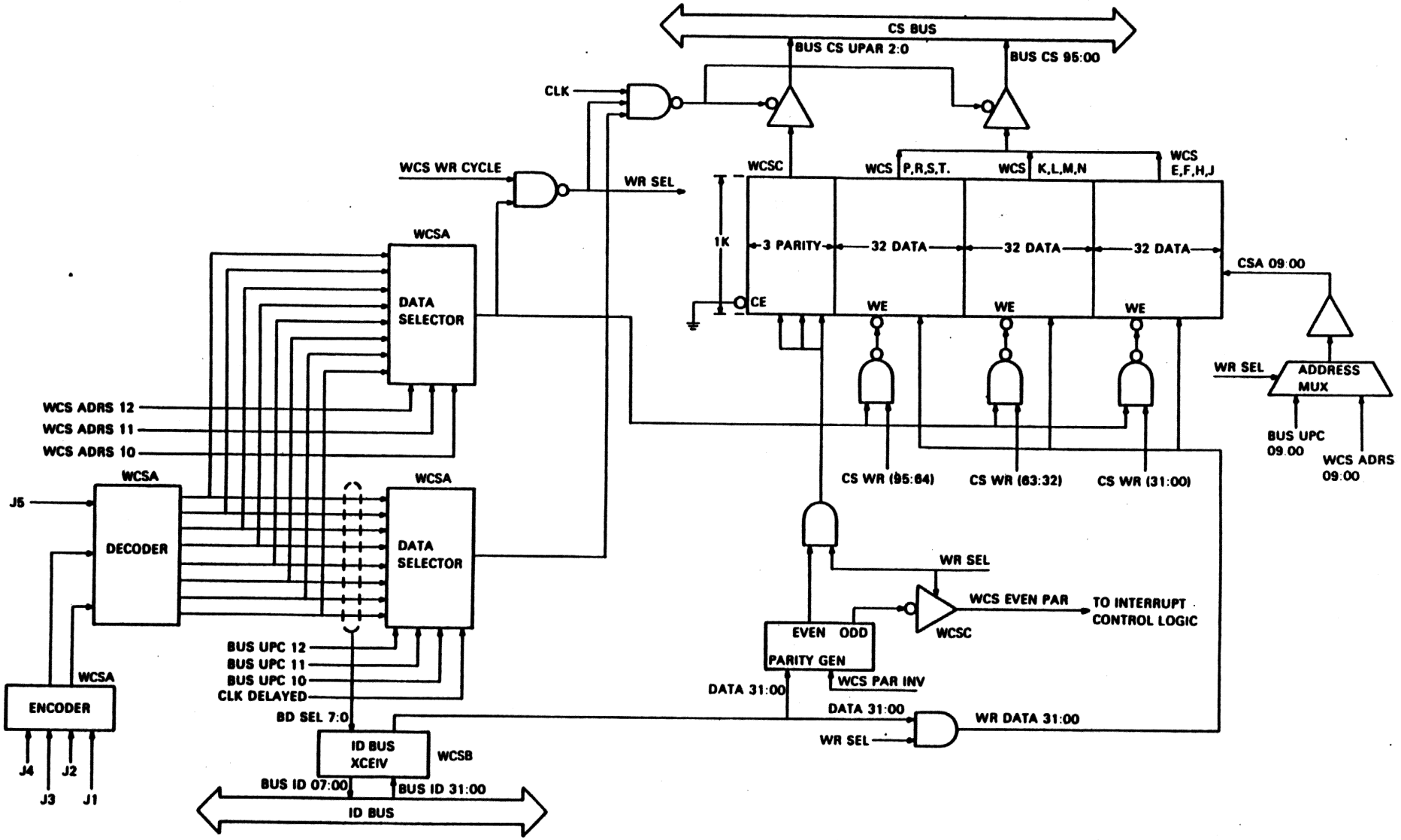


Figure 2-12 Writable Control Store (WCS)

The WCS Address and Modulo 3 counters (Figure 2-14) generate the address of the microword and the specific 32 bit section of the microword to be written. The Modulo 3 counter generates the lines which enable each 32 bit segment. The following shows the relationship between the value of the Modulo 3 counter bits and the segment written.

Modulo 3 Counter Bit 14	Counter Bit 13	Microword Segment Written
0	0	Bits 31:00
0	1	Bits 63:32
1	0	Bits 95:64
1	1	WCS Write Inhibited and WCS Address Increment Inhibited

The counter can be loaded with 11 but nothing will be written. Normal incrementation of the counter will result in the following sequence: 00,01,10,00 (refer to Figure 2-13). At the end of each WCS write data command, the counter is incremented. When the Modulo 3 counter overflows (from 10 to 00), the WCS Address counter is incremented, thereby specifying the next microword address.

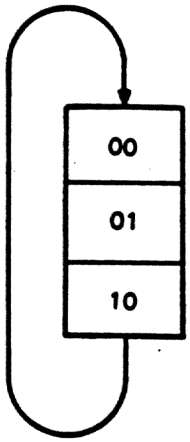


Figure 2-13 Incrementation of Modulo 3 Counter

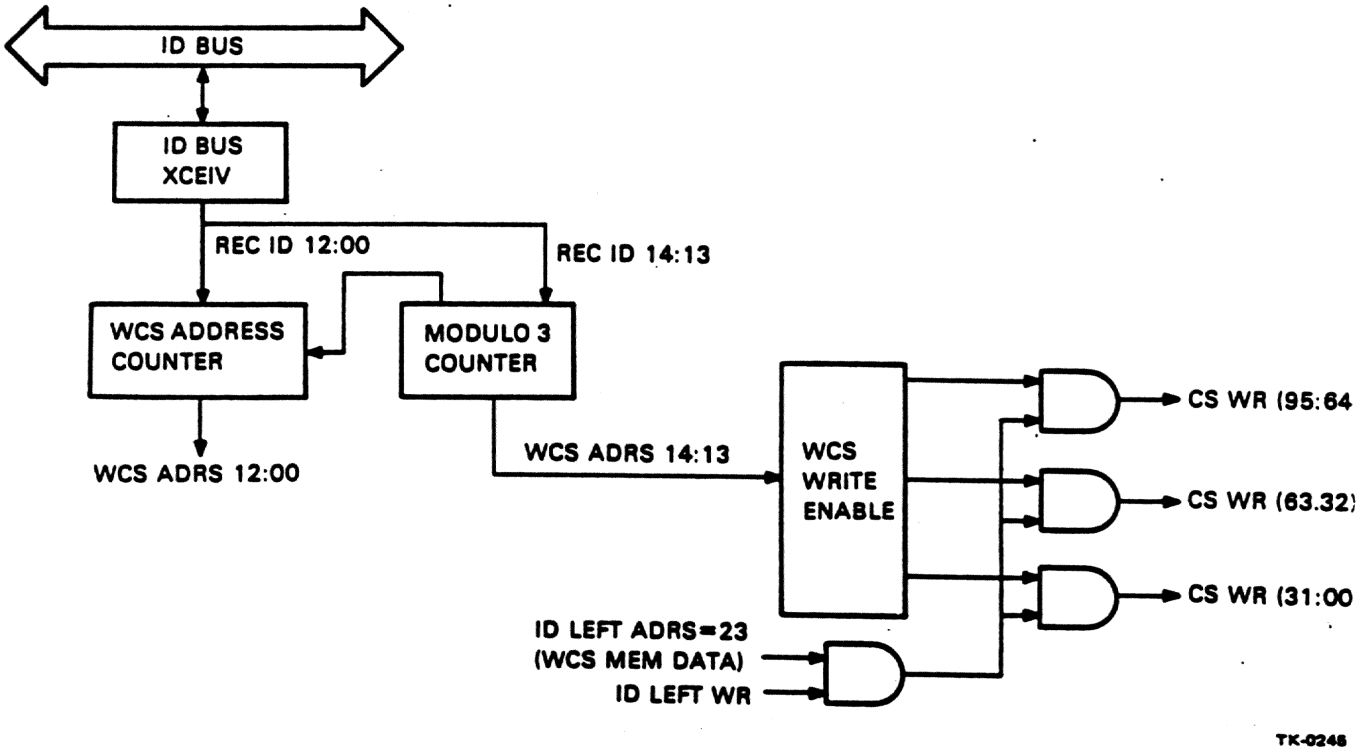


Figure 2-14 WCS Address Counter and Modulo 3 Counter

As previously mentioned, the address counters and parity invert bits are loaded when the ID register address equals 22 (hex) and an ID LEFT write operation was specified. The actual data to be written is received from the ID bus (refer to Figure 2-8) when ID LEFT ADDR = 23 (WCS MEM AVAIL) and ID LEFT WR is asserted. These conditions will also generate the microsequencer signal WCS WR CYCLE which is used to inhibit the CS drivers in the Writable Control Store. Reading of the WCS is thereby prevented if a write operation is in progress.

The address bits WCS ADRS 12:00 are used for write operations in a manner similar to BUS UPC bits 12:00 in a read operation. However, WCS ADRS bits 12:10 inhibit the CS bus drivers if the address lines match the corresponding jumpers, unlike BUS UPC bits 12:10 which enable the drivers. WCS ADRS bits 09:00 address the specific microword in the 1K WCS.

The jumpered address of the WCS module can be read over the ID bus if an ID bus read of the WCS Memory Data Register is performed. This operation causes the microsequencer to generate a signal which enables the board select lines (BD SEL 07:00) onto the ID bus.

2.4 INSTRUCTION BUFFER

The instruction buffer logic consists of an 8-byte register, shifters and multiplexers which enable the central processor to fetch instructions for evaluation. Refer to the instruction buffer block diagram (Figure 2-15). The structure of the instruction buffer allows prefetching of instruction stream data. The new data can be stored in the buffer register while the current instruction is being executed. Prefetching of instructions increases the efficiency of the system by reducing the time required to access new instruction data once the current data has been evaluated. The op code of each instruction is kept in byte 0 of the buffer register while operand specifiers are being evaluated. As the specifiers are evaluated, they are removed from the buffer register along with associated data and replaced with new operand specifiers. This process continues until the instruction can be executed. The current op code is then removed and replaced by the op code of the next instruction. Instructions which consist of an op code only (e.g., NOP) can be executed immediately since no specifiers must be evaluated.

The VAX-11/780 has the capability of operating in two instruction modes, native and compatibility. In compatibility mode, a subset of 16-bit, PDP-11 instruction can be executed. Native mode enables the execution of variable length VAX-11 instructions. Instructions are stored in contiguous byte locations in memory and are aligned on byte boundaries. The contiguous bytes of instructions are referred to as the instruction stream.

The instruction stream is transferred to the buffer over the Memory Data (MD) bus. Data loaded into the lower byte locations of the buffer can be evaluated while additional bytes of data are fetched from memory, thereby increasing overall performance.

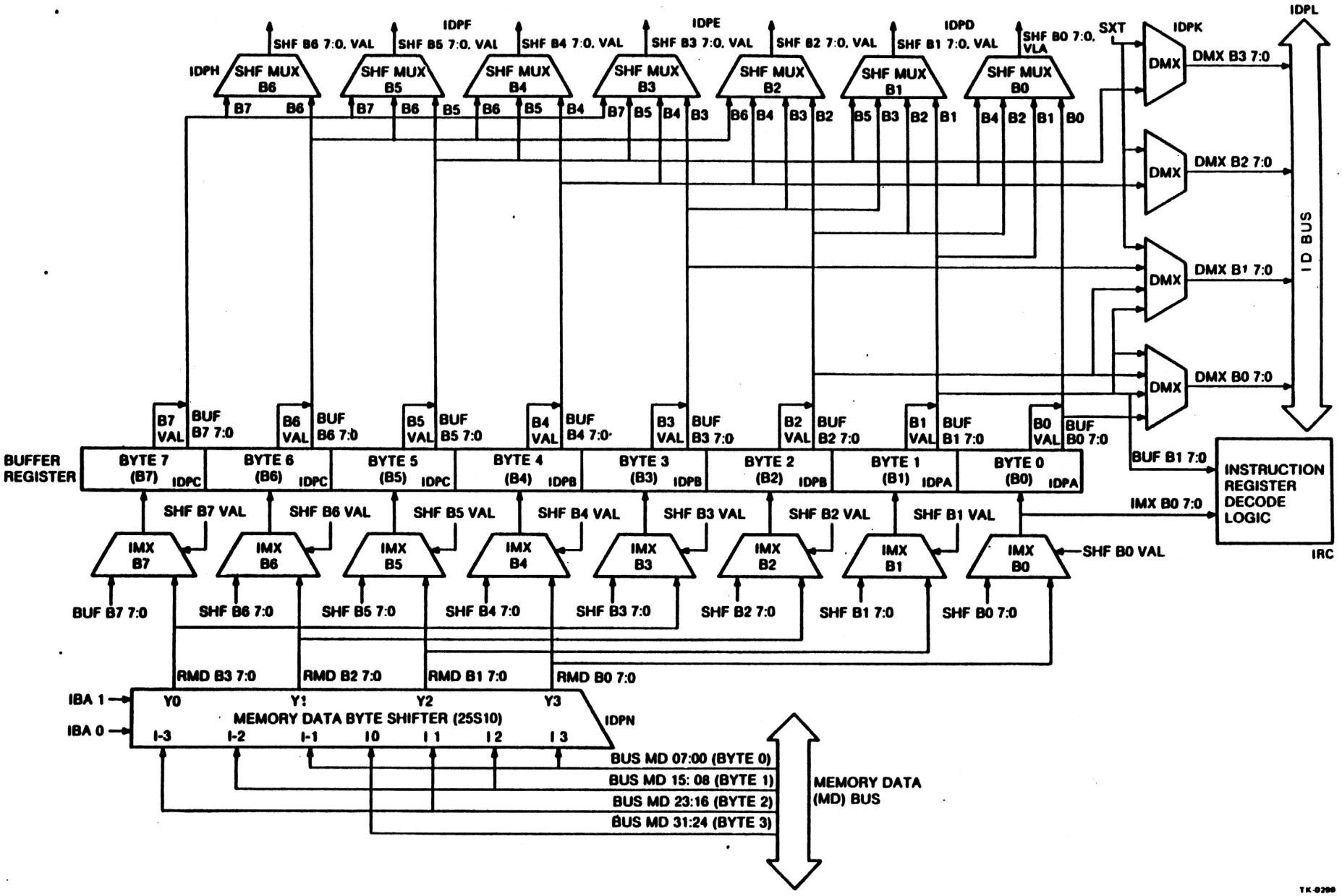


Figure 2-15 Instruction Buffer Block Diagram

The PDP-11 instructions are 16-bits and occupy two contiguous bytes. Refer to Paragraph 1.7 for a description of PDP-11 instructions which can be executed.

The format of the variable length VAX-11 instruction is illustrated in Figure 2-16.

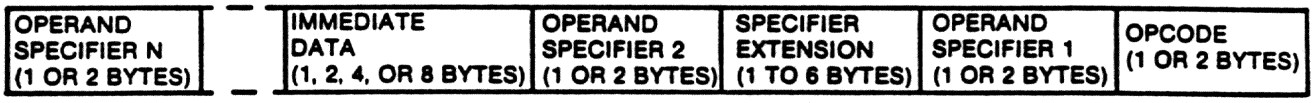


Figure 2-16 General Format of VAX-11 Instruction

The presently available instruction set uses a one byte operation code (op code). An instruction may consist of an op code alone or may consist of an op code and multiple operand specifiers. The operand specifier indicates the manner (addressing mode) in which the operand is to be accessed. Certain addressing modes require an extension to be appended to the operand specifier. The specifier extension can be used as a displacement or can be immediate data. Immediate denotes that the data or address immediately follows the operand specifier.

The variable length and format of the VAX-11 instructions require that the buffer be able to shift and align instruction stream data. The first byte of the instruction (op code) must be loaded into byte 0 of the buffer register before instruction decode can begin. The operand specifier to be evaluated must be loaded in byte 1 of the register. The memory data byte shifter ensures that information read from the MD bus is loaded in to the register in the correct byte positions. As operand specifiers are evaluated, bytes of data are read or cleared from the buffer. The shift multiplexers (SHF MUX) enable data from the higher order bytes of the buffer to be shifted into the vacant positions allowing further evaluation. The Input Multiplexer (IMUX) determines whether the buffer register will receive data from the memory data byte shifter or from the shifter multiplexer. Each byte of the buffer register has an associated valid bit which when set indicates that the byte has been loaded with valid data. Each IMUX is controlled independently by the valid bit associated with the SHF MUX input.

When a byte is cleared from the buffer, the shift multiplexer selects the data and the valid bit from the higher order byte which is to be loaded into the vacant position. If the valid bit is set in the byte selected by the SHF MUX, the IMUX will load that data into the vacant byte location. If the valid bit is not

set, the SHF MUX has selected an invalid byte and the IMUX will load data from the memory data byte shifter. The capability of storing information in the upper bytes of the buffer register allows prefetching of instruction stream data. While the lower bytes of the register are being evaluated, new data can be fetched from memory. This data is then available and can be shifted into the lower byte locations when required.

The instruction buffer also provides the capability of transferring displacement or literal data to other sections of the CPU via the Internal Data (ID) bus. The data multiplexer (DMX) selects bytes from the buffer register to be transferred over the ID bus. The DMX can sign extend or shift the data before it is enabled onto the bus.

The following paragraphs provide a more detailed explanation of each functional area of the instruction buffer logic in relation to overall operation.

2.4.1 Memory Data Byte Shifter (MD Byte Shifter)

Instructions stored in memory are byte aligned; i.e., an instruction can begin or end on any byte boundary. However, because memory is longword aligned, all instruction stream data read from memory is referenced on longword boundaries. The memory data byte shifter will always receive four bytes of longword aligned data from the memory data (MD) bus. The memory data must be rotated so that the desired information is loaded into the correct register byte. The positioning of the bytes is determined by the low two bits of the instruction address (IBA 0 and IBA 1). These bits specify a particular byte within a longword. The relationship between the byte address and format of the shifted data is shown in Table 2-5.

Table 2-5 Memory Data Shift Format

Byte Address Bits		MD Shifter Output			
IBA 1	IBA 0	Y0	Y1	Y2	Y3
0	0	Byte 3	Byte 2	Byte 1	Byte 0
0	1	Byte 0	Byte 3	Byte 2	Byte 1
1	0	Byte 1	Byte 0	Byte 3	Byte 2
1	1	Byte 2	Byte 1	Byte 0	Byte 3

The function of the MD byte shifter can be illustrated by the following example. Assume instruction stream data is stored in the memory byte locations listed, and a program branched to location CHECK.

Memory Address (Byte Locations)	I-Stream Data (Hex Code)	Assembler Notation
203	C0	ADDL #09,R2
204	09	
205	52	
206	D1	CHECK : CMPL (R2),R4
207	62	
208	54	

The program branch would result in the instruction buffer address equal to 206. Since the memory reference will always be on a longword boundary, the lower two bits (IBA 1 and IBA 0) of the instruction buffer address are ignored by memory. The fetch from memory would result in the MD byte shifter being loaded with four bytes of data beginning at longword boundary 204. The MD byte shifter will position the data so that byte 0 of the buffer register will be loaded with the contents of byte location 206. The memory data will be shifted as shown in Figure 2-17.

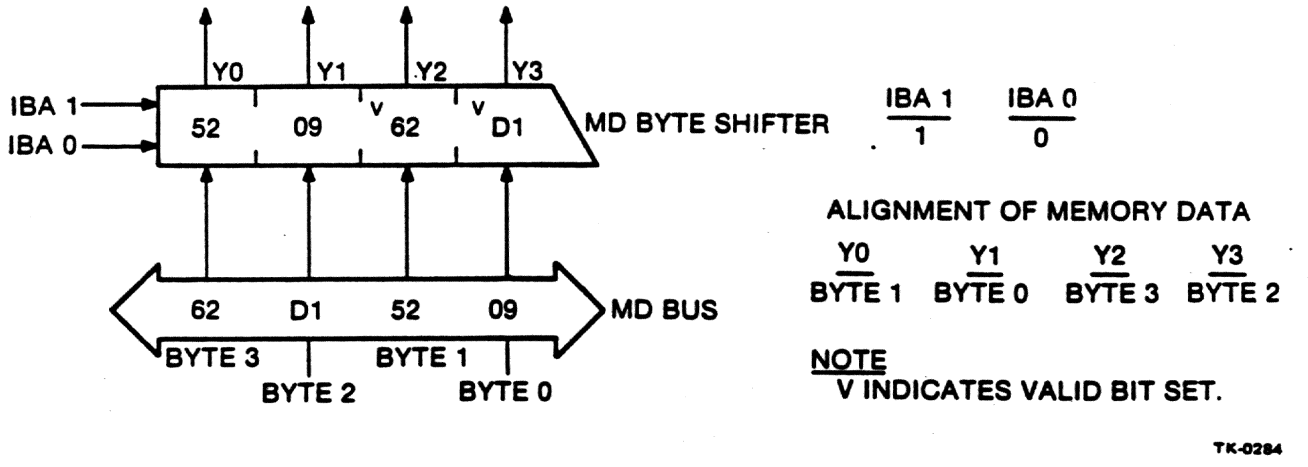


Figure 2-17 Memory Data Shift Example

An expanded example of loading data from the MD bus is provided in Paragraph 2.4.5

2.4.2 Buffer Register

The buffer register consists of eight 9-bit bytes, designated as byte 7 through byte 0. Byte 0 is loaded with data read from the lowest memory address and byte 7 is loaded from the highest address.

In native mode, the op code of an instruction must always be loaded in byte 0 to be decoded and the operand specifier must be in byte 1 to be evaluated. Bytes 2 through 7 can contain literal or displacement data associated with the specifier in byte 1, or these bytes can contain other instruction stream data (e.g., the next specifier, next op code, etc.).

In compatibility mode, the 16-bit PDP-11 instruction is loaded into bytes 0 and 1 of the buffer register. Bytes 2 and 3 can contain literal data associated with the current instruction (e.g., data used in index mode) or can contain the next instruction. Bytes 3 through 7 can also be loaded with new instructions while the current instruction is being executed.

2.4.2.1 Valid Bits -- Each byte of the buffer register contains eight data bits and one valid bit. The valid bit, when set, indicates that useful data has been loaded into the associated byte. Valid data can be loaded through the IMUX from the shift multiplexer (SHF MUX). When the SHF MUX is selected as the input to a particular byte, its associated valid bit is loaded into the register byte with the data. If the MD byte shifter is selected as the input source, the valid bit can be set depending on the value of address bits IBA 1 and IBA 0 and whether or not the lower bytes in the register have valid data. Paragraph 2.4.5 provides an example of loading the buffer register and illustrates when the valid bits are set.

The buffer register holds the instruction stream data while the op code is decoded and specifiers are being evaluated. The op code of an instruction indicates how many specifier evaluations must be performed and each specifier indicates how much literal or displacement data will follow. The register bytes are changed as literal and displacement data is loaded on the Internal Data bus and as instruction execution is completed. The moving of bytes within the buffer register is controlled by the UIBC field of the microword.

2.4.3 Shift Multiplexer (SHF MUX)

As instruction stream data is evaluated, contents of the upper register bytes are shifted into the lower byte locations. The shift multiplexers are configured to allow shifting of the 9-bit bytes by 0, 1, 2, or 4 positions to the right each microcycle. The SHF MUX data is transferred to the buffer register through the input multiplexer (IMUX). If the valid bit for a particular SHF MUX is set, that shift data will be selected by the IMUX and stored in the buffer register. The shift multiplexers are controlled by the UIBC field (BUS CS 95:92) of the microinstruction. Table 2-6 shows the relationship between the UIBC field value and the function selected. The entire buffer register is clocked at the beginning of every microcycle.

Table 2-6 Microword Control of Instruction Buffer

UIBC Field (Hex Value)	Function	Comment
Ø	NOP	Shift by Ø. Does not affect shift mux or data input to IMUX.
1	STOP	Prevents further memory requests by instruction buffer.
2	FLUSH	Clears all valid bits of buffer register. Used for conditions which cause a change in PC (e.g., jump or branch). The VIBA register (in the data path) is also loaded when this field value is specified.
3	START	Enables prefetch operations by the instruction buffer.
4	CLR Ø, 1	In compatibility mode, the next instruction is shifted over the current instruction in bytes Ø and 1. This function is performed in native mode to optimize short literal to register and register to register transfers and also to execute 16-bit branch destinations.
5	CLR 2, 3	In compatibility mode, 16-bit displacement or literal data contained in bytes 2 and 3 are shifted over with new data.
6	Reserved	
7	READ BDEST	Used during ACBX, AOB, SOB and Branch on Bit instructions to indicate branch displacement specifiers.
8-B	Reserved	
C	CLR Ø	In native mode, the current op code in byte Ø is shifted over with the next op code.
D	CLR 1	In native mode, the operand specifier in byte 1 is shifted over with new data. This function is performed for instructions using displacement mode addressing, absolute addressing or long literals. In these modes, the operand specifier is the last byte removed from the register during operand evaluation.

Table 2-6 Microword Control of Instruction Buffer (Continued)

UIBC Field (Hex Value)	Function	Comment
E	CLR 0,1,2,3	This function is performed in compatibility mode to optimize instructions which execute literal data to register transfers.
F	CLR 1-5 Conditional	This function is performed in native mode to remove long literals, displacement data, or specifiers from the buffer register. The op code and/or specifier evaluation will indicate if the data is 8, 16, or 32 bits. In addressing modes that do not have instruction stream data other than the specifier, the specifier itself is cleared from the buffer register.

Lower bytes of the buffer register are actually changed when data is shifted over a byte location. The UIBC field controls the data selection of the shift multiplexers and effectively controls which byte locations will be written over. Instruction execution results in lower byte locations being removed by the shifting in of new data. As indicated from the comments in Table 2-6, the bytes which are cleared depend on the instruction mode, op code, specifier, and context.

2.4.4 Data Multiplexer (DMX)

The data multiplexer (Figure 2-18) provides the capability of transferring the contents of the buffer register to other areas of the CPU via the Internal Data (ID) bus. The DMX selects bytes from the buffer register and can sign extend or shift the data if required. This capability enables the evaluation of displacement and literal specifiers to be optimized by allowing the direct transfer of information from the buffer register. The data read onto the ID bus is transferred to the Q register of the data path and can then be transferred from the data path to a specified destination. Refer to Paragraph 1.8.1.4 for an explanation of the ID bus. The DMX is selected as the source of ID bus data if the address lines (ID RIGHT ADDR 5:0) specify the DMX (hex address = 00) and the control line ID RIGHT WRITE L is not asserted.

Selection of data by the DMX requires that the op code and specifier of the instruction first be decoded. The following shows the relationship between the addressing modes of VAX instructions and the data transferred onto the ID bus.

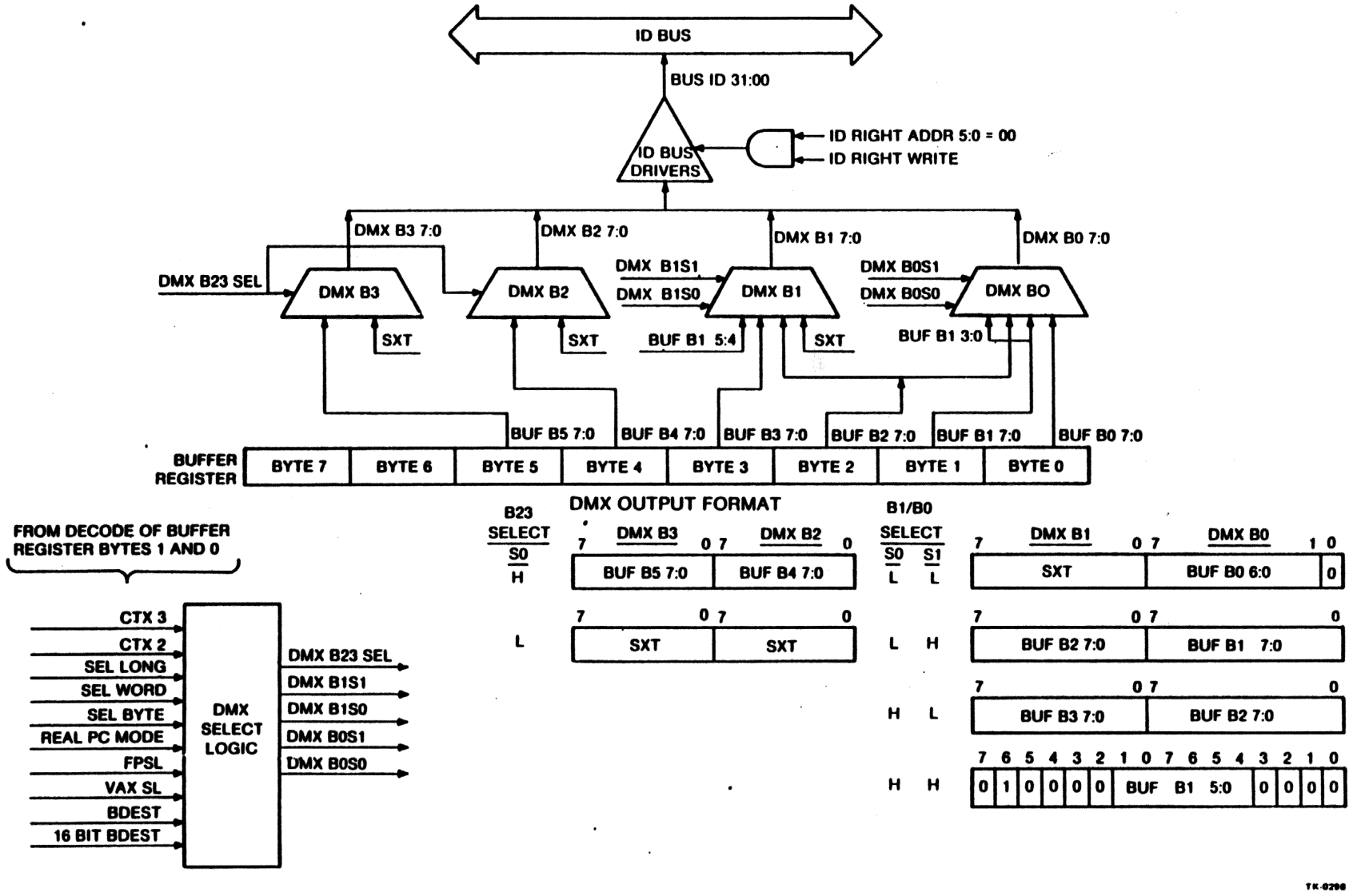


Figure 2-18 Data Multiplexer (DMX)

General Addressing Modes	ID Bus Data	Comment
Short Literal (Note 1)	B1 = 07:00	Zero Extended
Indexed	N/A	
Register	N/A	
Register deferred	N/A	
Autoincrement (R = PC)	B5:B2 = 31:00	1, 2, or 4 bytes depending on context, sign extended
Autoincrement deferred (R = PC)	B5:B2 = 31:00	32 bit address
Byte displacement	B2 = 07:00	Sign extended on bit 07
Byte displacement deferred	B2 = 07:00	Sign extended on bit 07
Word displacement	B3,B2 = 15:00	Sign extended on bit 15
Word displacement deferred	B3,B2 = 15:00	Sign extended on bit 15
Longword displacement	B5:B2 = 31:00	
Longword displacement deferred	B5:B2 = 31:00	

Branch Addressing Modes	ID Bus Data	Comment
8-bit byte displacement	B1 = 07:00	Sign extended on bit 07
16-bit word displacement	B2,B1 = 15:00	Sign extended on bit 15

NOTE 1

If the operand is a floating data type, the short literal is in the format shown in Figure 2-19.

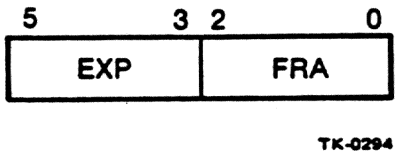


Figure 2-19 Floating-Point Short Literal

The DMX formats the data as shown in Figure 2-20.

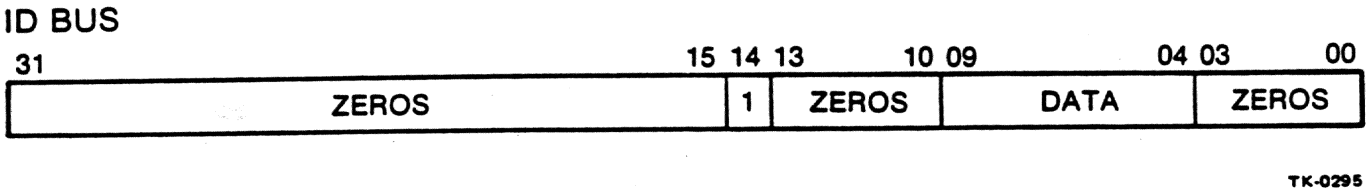


Figure 2-20 DMX Format of Floating-Point Short Literal

If the system is operating in compatibility mode, the DMX will select data from the buffer register for the following PDP-11 instructions:

a. Instructions which are followed by 16-bit displacement or 16-bit literal data. The information is stored in the buffer register as shown in Figure 2-21.

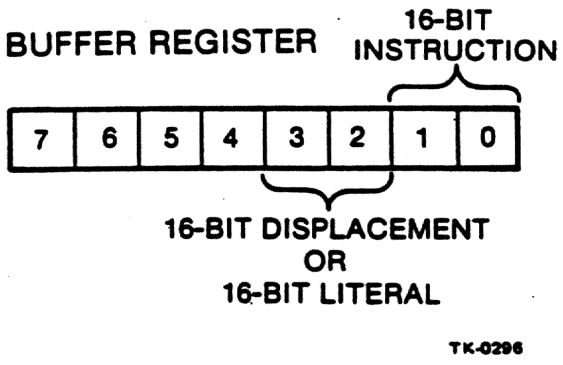


Figure 2-21 Format of PDP-11 Instruction in Buffer Register

The DMX selects the data from bytes 2 and 3 of the registers to be transferred onto the ID bus. The DMX shifts these bytes so that they are transferred as ID bits 15:00, sign extended on bit 15.

b. Branch instructions which are stored in the buffer register as shown in Figure 2-22.

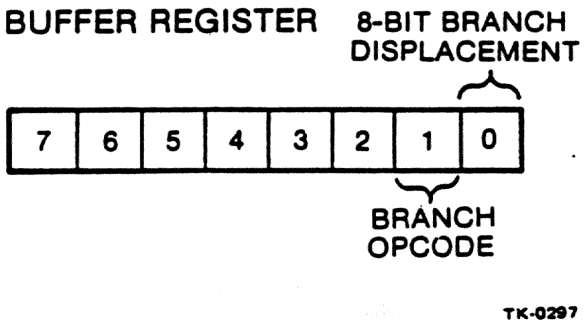


Figure 2-22 Format of Branch Instruction in Buffer Register

The DMX will select the branch displacement from byte 0, left shift the data by one bit and sign extend the data on bit 7.

The DMX decreases execution time by optimizing the format of certain addressing modes in both VAX-11 and PDP-11 instructions. Once the op code and specifier have been decoded, the DMX can sort out instruction stream data from the buffer register and arrange it in a usable form. This hardware capability increases overall system performance.

2.4.5 Loading the Instruction Buffer

This section provides an example of loading an instruction (ADDL #09,R2) into the instruction buffer from memory. Assume the following program is in the memory locations indicated:

Memory Address (Byte locations)	Hex Code	Assembler Notation
200	D0	MOVL (R3), (R4)+
201	63	
202	84	
203	C0	ADDL #09,R2
204	09	
205	52	
206	D1	CMPL (R2), R4
207	62	
208	54	
209	13	BEQL, DONE
20A	05	
20B	01	NOP
20C	90	MOVB (R7), R8
20D	67	
20E	58	
20F	B4	CLRW R4

	Contents			
Longword memory address	200/C0	84	63	D0
	204/62	D1	52	09
	208/01	05	13	54
	20C/B4	58	67	90

As previously mentioned, the op code of the instruction must be loaded into byte 0 of the buffer register to be decoded. The instruction in this example (ADDL #09,R2) does not begin on a longword boundary. Therefore, in order to load the op code into byte 0 of the buffer register, the MD byte shifter will have to align the memory data received.

The instruction buffer address (IBA) will equal 203, however, the low two address bits (IBA 1 and IBA 0) are ignored when a memory reference is made. References to memory can be made on longword boundaries only. The first memory fetch will load four bytes of

data into the MD byte shifter and will be read from memory address 200. The MD byte shifter will align the data so that the op code (C0) is loaded into byte 0 of the buffer register. The alignment of memory data is controlled by the low two bits of the instruction buffer address. The data is loaded into the buffer register as illustrated in Figure 2-23. To simplify the diagram, only the buffer register and MD byte shifter are shown.

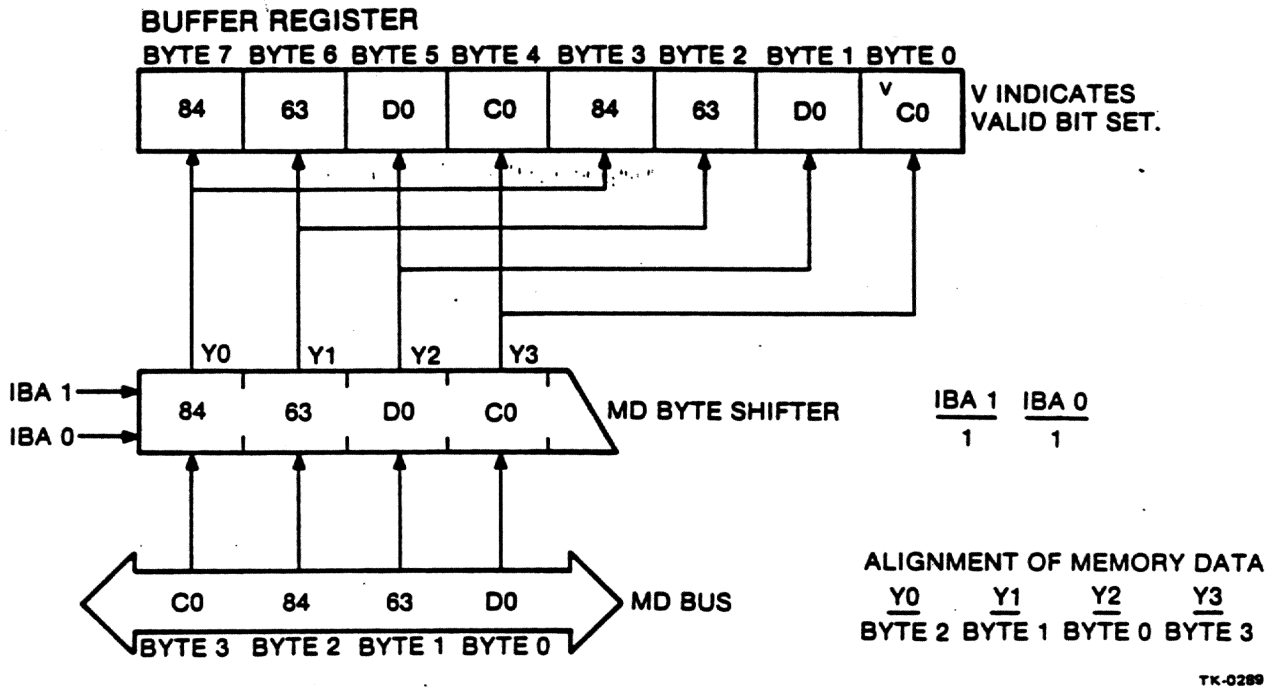


Figure 2-23 Result of First Memory Fetch

The setting of each valid bit depends on whether or not the lower byte locations contain valid data and also depends on the value of the low two bits of the instruction buffer address. After the first fetch, the instruction buffer address is incremented by four.

Since the instruction began on the last byte of a longword, only byte 0 of the buffer register was loaded with valid data. The remaining bytes of data are loaded into the buffer register but the associated valid bits are not set. Setting the valid bit in byte 0 will prevent it from being written over during the next memory fetch.

The instruction buffer address will equal 207 after incrementation. The value of the low two address bits remains the same. Adding four to the VIBA register increments the address by a longword and does not affect the byte position. The second memory fetch will load four bytes of data beginning at longword boundary 204. The MD byte shifter will align the data exactly as it did in the first fetch because bits IBA 1 and IBA 0 are at the same value. Figure 2-24 shows the result of the second fetch.

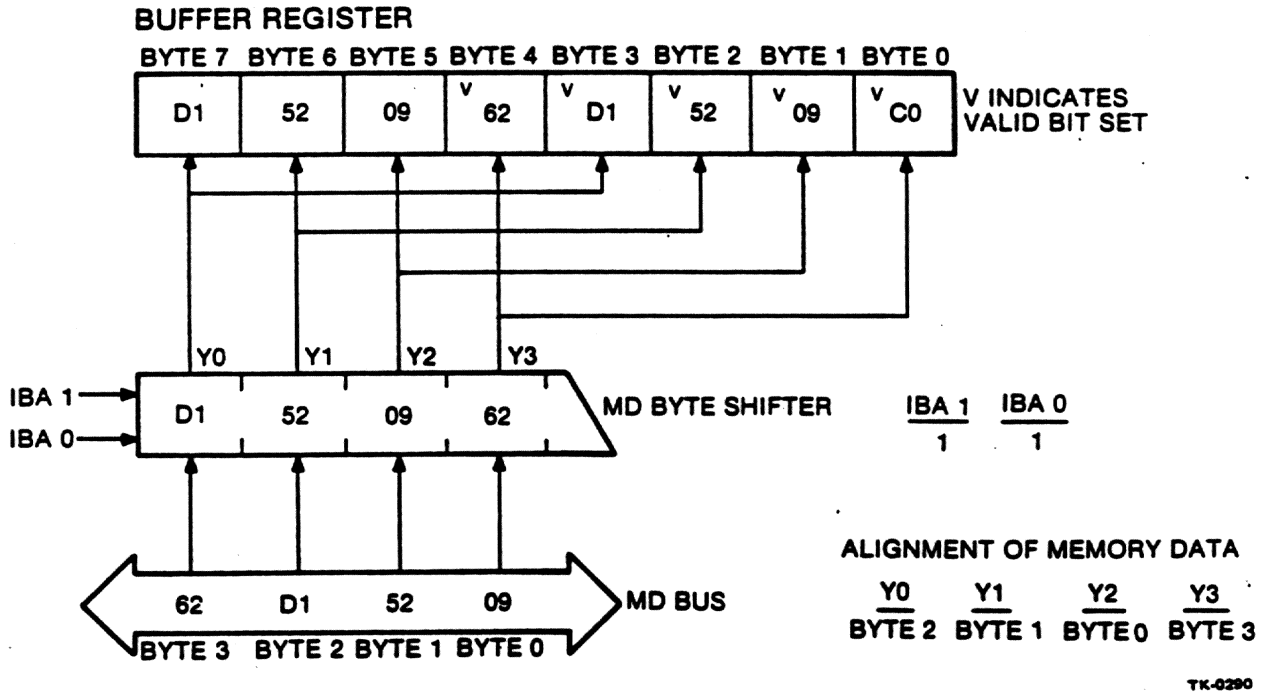


Figure 2-24 Result of Second Memory Fetch

Byte Ø was not written with memory data because its valid bit was set as a result of the first fetch. The IMX for byte Ø (refer to Figure 2-15) selects data from the SHF MUX rather than from the MD byte shifter. The SHF MUX data will be the same as the current contents of byte Ø since a shifter by zero (UIBC field = Ø) was performed. The number of valid bytes loaded from the first fetch depends on the instruction buffer byte address. The second fetch will always load four bytes of valid data.

After the second fetch, the instruction buffer has all the data required for the execution of the instruction ADDL #Ø9,R2.

At instruction decode time (microcode IRD state), the literal data (Ø9) in register byte 1 is selected by the DMX and transferred to the Q register of the data path (via the ID bus). This transfer leaves byte 1 vacant. At the end of IRD state, a shift by 1 will be performed and valid data will be stored in the buffer register as shown in Figure 2-25.

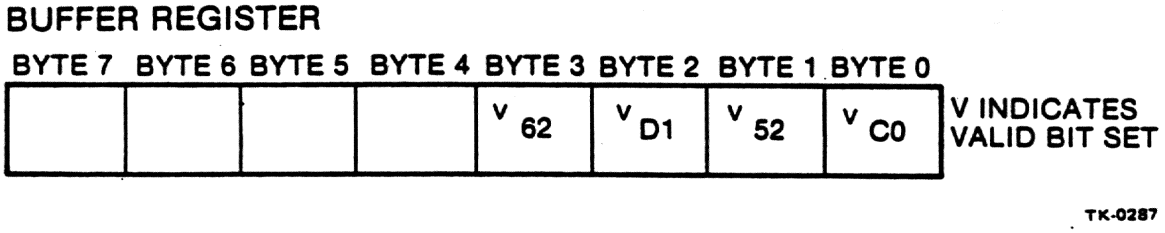


Figure 2-25 Buffer Register After Shift by 1 Byte

Byte Ø (op code) remains the same since its valid bit is still set. Bytes 1, 2, and 3 are loaded with data from the shift multiplexers. The SHF MUX inputs contain data and valid bits from the next higher byte location. Note that the valid bits are shifted with the data from the higher byte locations. Bytes 4 through 7 contain invalid data.

The low two address bits (IBA 1 and IBA Ø) are incremented by one since the data in the buffer register has moved by one byte position. The low two address bits keep track of the end of the buffer and are incremented independently of address bits 31:Ø2. The instruction buffer longword address (31:Ø2) was also incremented after the last memory fetch, resulting in the address equal to 2Ø8 (2Ø4 plus 4). The next memory fetch will begin at longword boundary 2Ø8. The data will not be rotated because IBA bits 1 and Ø are both equal to zero. Figure 2-26 illustrates the result of the third memory fetch.

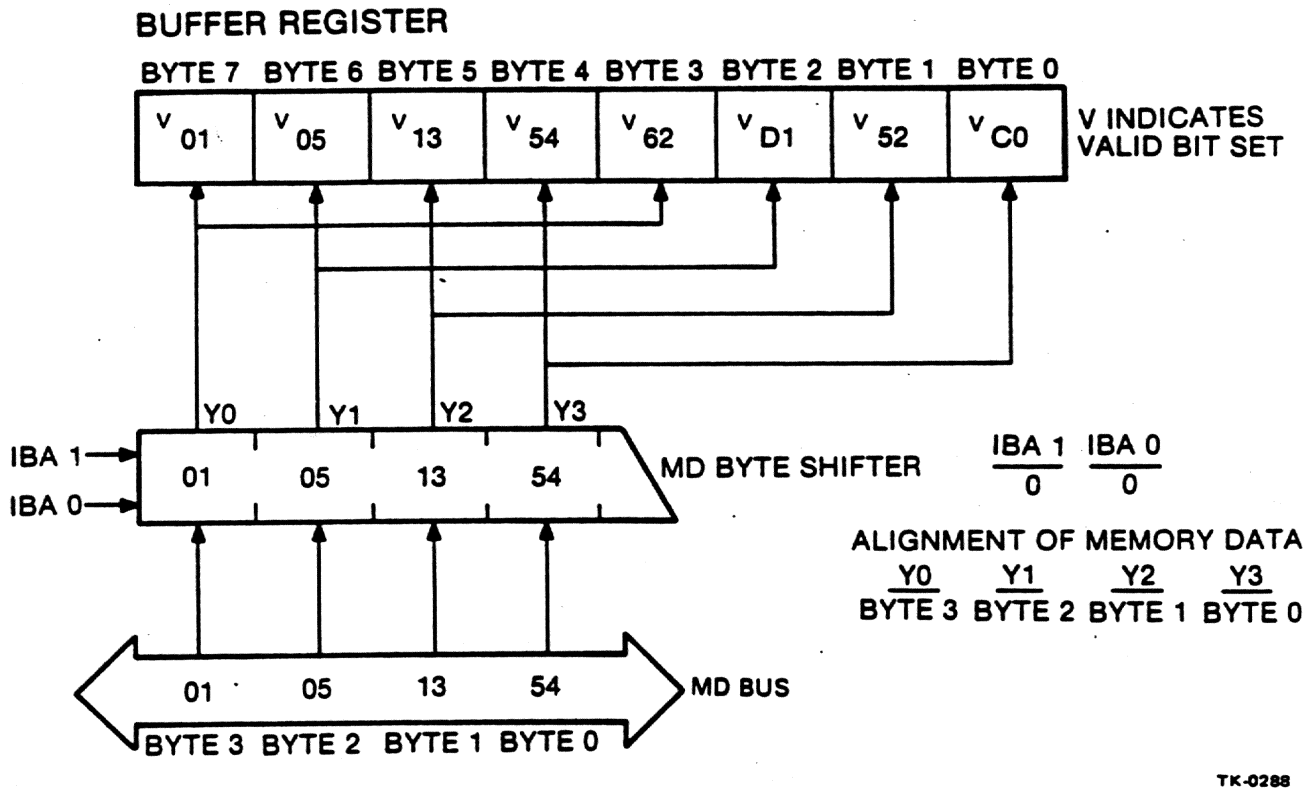


Figure 2-26 Result of Third Memory Fetch

The valid bit in byte 7 is set as a result of the third memory fetch. This condition inhibits the instruction buffer address from being incremented by 4. The address is not incremented until data is shifted out and the valid bit in byte 7 is cleared.

At the next fork entry in the microcode (A FORK), byte 0 (op code) and byte 1 (operand specifier) of the buffer register are removed. The microcode will specify a shift by 2 bytes, resulting in valid data being stored as shown in Figure 2-27.

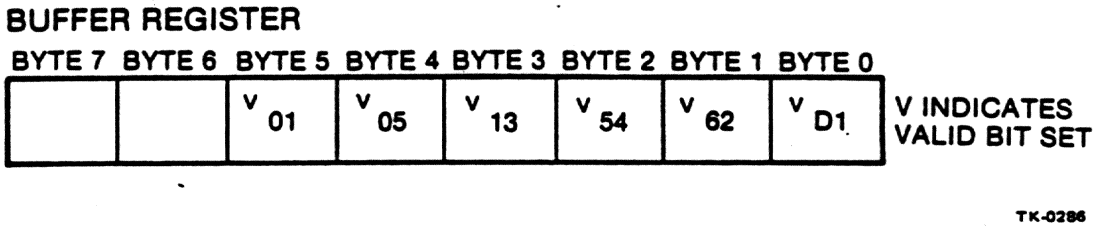


Figure 2-27 Buffer Register After Shift by 2 Bytes

Bytes 0 through 5 are loaded with data from the shift multiplexers and the valid bits in byte 6 and 7 are cleared when the data is shifted out.

The instruction buffer longword address (31:02) is incremented by 4 because the buffer register successfully fetched data during the previous cycle and the valid bit in byte 7 is not clear. Two is also added to the byte address (01:00) since two bytes of data were cleared from the buffer register.

The fourth memory fetch will load data from longword address 20C. The data will be rotated since the low two address bits now equal two. Figure 2-28 illustrates the result of the fourth fetch.

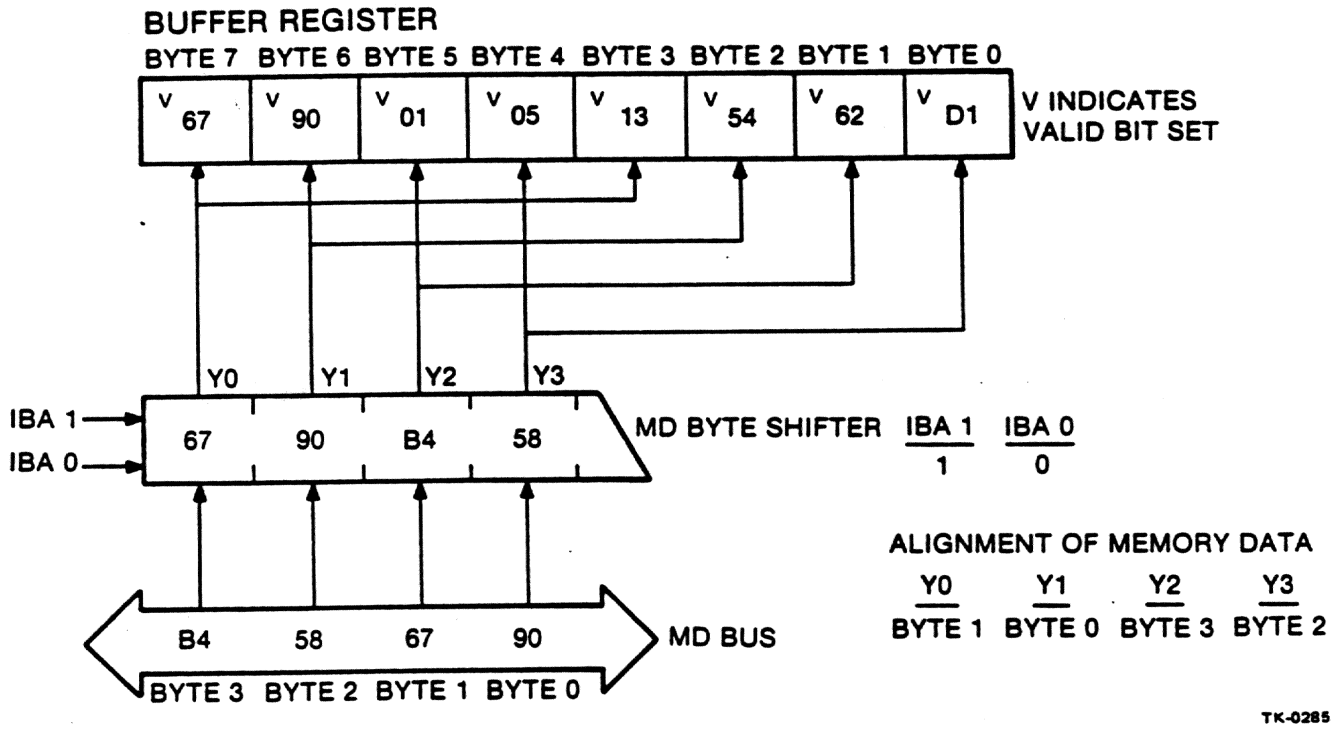


Figure 2-28 Result of Fourth Memory Fetch

Note that the instruction CMPL (R2), R4 could have been executed without the fourth memory fetch. The instruction buffer contained all the required information when the data was shifted by two bytes (refer to Figure 2-27).

2.4.6 Register Addresses

During specifier evaluations, the instruction buffer provides the address source for the scratch pad register sets in the data path (refer to Figure 2-29). The address lines generated by the instruction buffer logic are:

a. SP1 ADR 03:00 (Specifier 1 Register)
b. PRN ADR 03:00 (Previous Register Number)
c. SP2 ADR 03:00 (Specifier 2 Register)

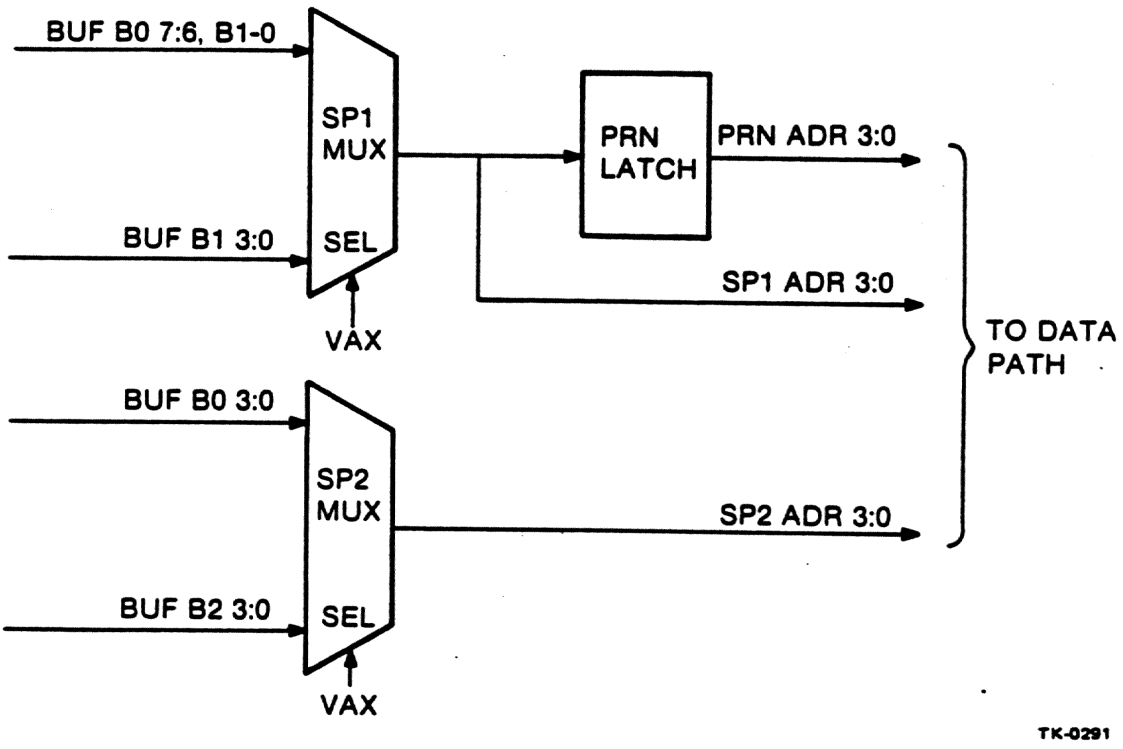


Figure 2-29 Register Addresses

The input selection of the specifier multiplexers depends on the mode of operation, native (VAX) or compatibility (PDP-11).

In native mode, SP1 is the register number of the operand specifier (source register) currently being evaluated in byte 1 of the buffer register. If the operation is a short literal to register or register to register transfer, SP2 will be the register number of the operand specifier (destination register) in byte 2 of the buffer register. Refer to Figure 2-30. The specifier 1 register number can also be held in the PRN (Previous Register Number) latch to be used as the scratch pad address source.

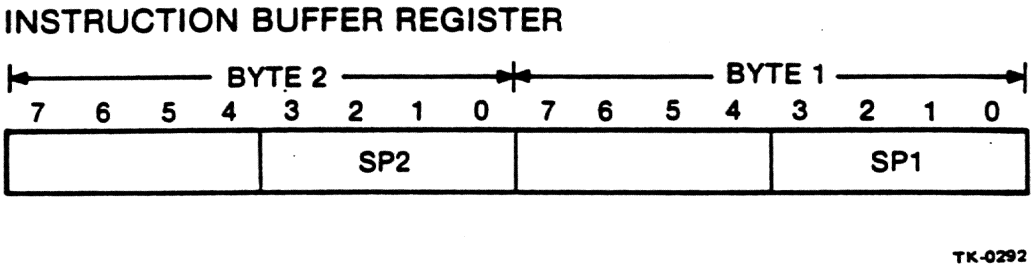


Figure 2-30 Register Fields in VAX Instructions

In compatibility mode, the value of SP1 is determined by the source register field of the instruction and SP2 is determined by the destination register field. Refer to Figure 2-31.

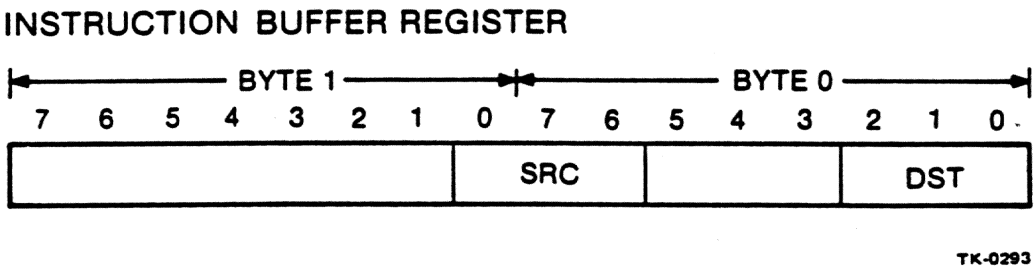


Figure 2-31 Register Fields in PDP-11 Instructions

2.4.7 Program Counter (PC) Updates

The program counter in the data path holds the address of the instruction's op code each time a new execution sequence is started. As operand specifiers are evaluated, the instruction buffer logic generates a three bit number (DELTA PC Ø2:ØØ) which is added to the low order bits of the PC register. This value effectively causes the PC to point beyond the instruction byte evaluated.

In native mode, the PC update value will reflect the specifier and any additional bytes of literal or displacement data associated with the specifier. The following lists the addressing modes which require additional data and the length number which is added to the PC.

Addressing Mode	Length Number
Autoincrement (R = PC)	1, 2, or 4 bytes depending on context
Autoincrement deferred (R = PC)	4 bytes
Byte displacement	1 byte
Byte displacement deferred	1 byte
Word displacement	2 bytes
Word displacement deferred	2 bytes
Longword displacement	4 bytes
Longword displacement deferred	4 bytes

The complete number added to the PC will include the length number for the addressing mode plus one for the specifier. The hardware capability of providing the PC update value eliminates an extra microinstruction in the flow. Note that the updates for the op code must be handled separately by the microcode.

In compatibility mode, the hardware determines the addressing mode and whether or not an address calculation is required. If the data following the current instruction is required for execution, the number two is added to the PC.

The PC update number is zero for the following conditions:

a. A fault is detected (e.g., error, TB miss, or stall)
b. Instruction consists of a single byte op code
c. Decision point entry is to an execution flow
d. First part done flag is set

2.5 INSTRUCTION DECODE

Instruction stream data stored in buffer register bytes 1 and 0 is decoded by the use of ROMS and combinational logic. Refer to Figure 2-32. The instruction decode logic provides the source for the lower eight bits (BUS UPC 07:00) of the next microaddress when the microprogram reaches a decision point fork. A decision point fork is specified if the Subroutine (USUB) field of the current microword equal 3 (CALL 3). The microsequencer (refer to Paragraph 2.3) disables the normal source of the lower address bits and enables the UPC ADRS MUX of the instruction decode logic. Therefore, each time a decision point fork is reached, the decoded instruction provides an entry point in the microprogram. The microprogram will either enter a flow which evaluates an operand specifier or enter an execution flow unique to the current instruction. If an error or service condition is present during a decision point fork, the UPC ADRS MUX selects the service input. The entry point in the microprogram will be to a specific routine which handles the current problem.

2.5.1 VAX Control Word

The output of the VAX Decode ROM and Context ROM form a 12 bit control word which determines the execution point entries generated for VAX instructions. The control word is divided into fields as shown in Figure 2-33. The Mode field enables the mode multiplexer (Paragraph 2.5.2) to select either the specifier decode logic or the VAX decode ROM as the source for the microaddress bits.

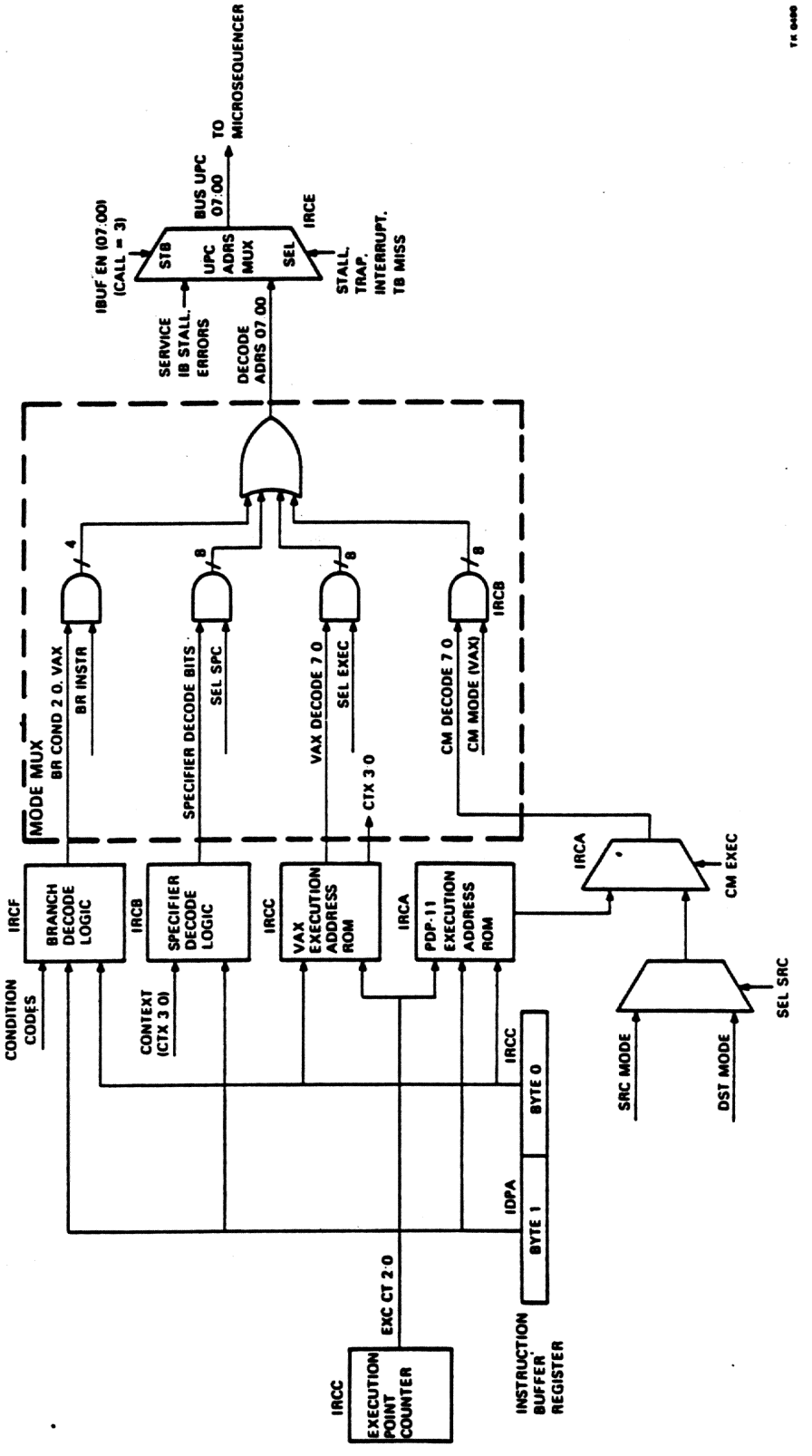


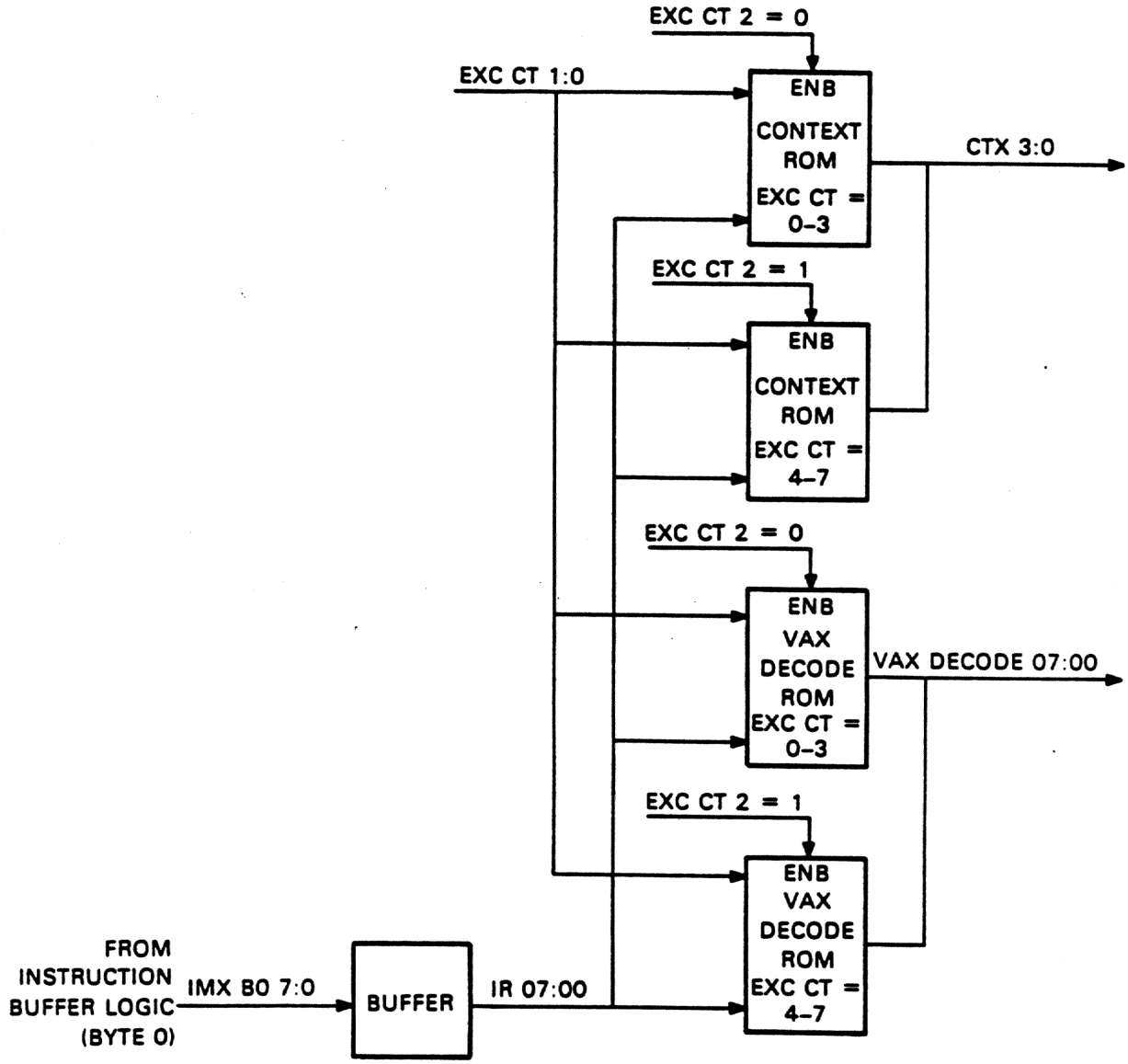
Figure 2-32 Instruction Decode Logic

Mode Field

VAX Decode bits 05	04	Operation
0	0	Select Specifier -- enables the specifier decode logic. The address bits generated will depend on the addressing mode of the specifier. Paragraph 2.5.3 provides a list of addresses generated by the specifier decode logic.
0	1	Execute if R Mode -- enables either the specifier decode logic or the VAX Decode ROM. If the specifier in byte 1 of the instruction buffer is in register mode, the one's complement of VAX Decode bits 07:00 are selected as the source for the microaddress bits. If the specifier in byte 1 is not in register mode, the specifier decode logic is selected.
1	0	Optimized -- enables the specifier decode logic and modifies the specifier address if a short literal to register or register to register operation is being performed. If the optimized conditions are not met, the specifier address is not modified.
1	1	Select Execute -- enables the one's complement of VAX Decode bits 07:00 as the source for the microaddress bits.

The address field of the VAX Control Word is used only when the mode field specifies Execute if R Mode or Select Execute operation. If an execute address is being generated, all eight bits of the VAX Decode ROM (VAX DECODE 07:00) are one's complement and selected as the source of the low eight microaddress bits. The four bits of the address field provide a 16-way branch for each mode and access combination specified.

The access field of the control word is used to select the branch decode logic or to indicate the type of transfer (read, write, or modify) specified by the instruction at each execution point. If the mode field equals 1 or 3 (Execute), the access field (VAX DECODE 07:06) is used to form the execution address.



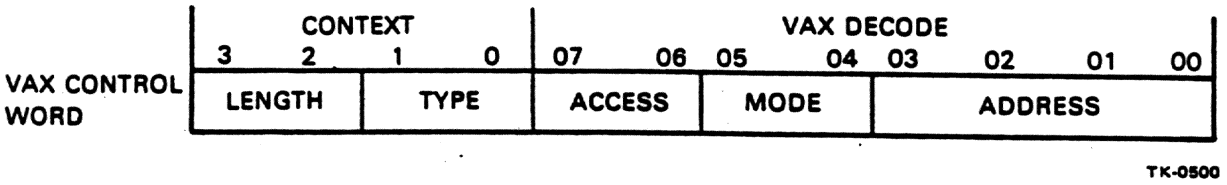


Figure 2-33 VAX Control Word

Access Field

VAX Decode Bits 07	06	Operation
0	0	Branch -- enables the Branch decode logic at execution point zero. At any other execution point, this code can only be used to form the execution address.
0	1	Read -- indicates the performance of a write operation from cache to the D register of the data path.
1	0	Write -- indicates the performance of a write operation from the D register to cache.
1	1	Modify -- indicates a read-modify-write operation. This code informs the translation buffer that the operation is a read with a write check.

The Context ROM provides information as to the length and type of operand, shown as follows:

Context Bits 03	02	Length	01	Context Bits 00	Type
0	0	Byte	0	0	Integer
0	1	Word	0	1	Float
1	0	Long	1	0	VSRC
1	1	Quad	1	1	ASRC

The ASRC code is specified for instructions that require the calculated effective address to be used as the operand. Since the operand is of address access type, register mode may not be designated.

The VSRC code is used only with field instructions and is specified when the calculated effective address is used as the operand. In field instructions, register mode may be designated in operand specifiers of address access type.

The following shows the length and type combinations used to specify operand data types.

Length	Type	Use
Byte	Integer	Byte data type
Word	Integer	Word data type
Long	Integer	Longword data type
Quad	Integer	Quadword data type
Byte	Float	Undefined
Word	Float	Undefined
Long	Float	Floating data type
Quad	Float	Double-Floating data type

2.5.1.1 Execution Point Counter -- The execution point counter (Figure 2-34) determines the number of decision point forks that have been entered for each instruction. This 3-bit counter and the op code of the instruction provide the address of the VAX control word ROMs and compatibility mode ROMs. Incrementing the counter changes the output of the ROMs and therefore changes the entry point address generated.

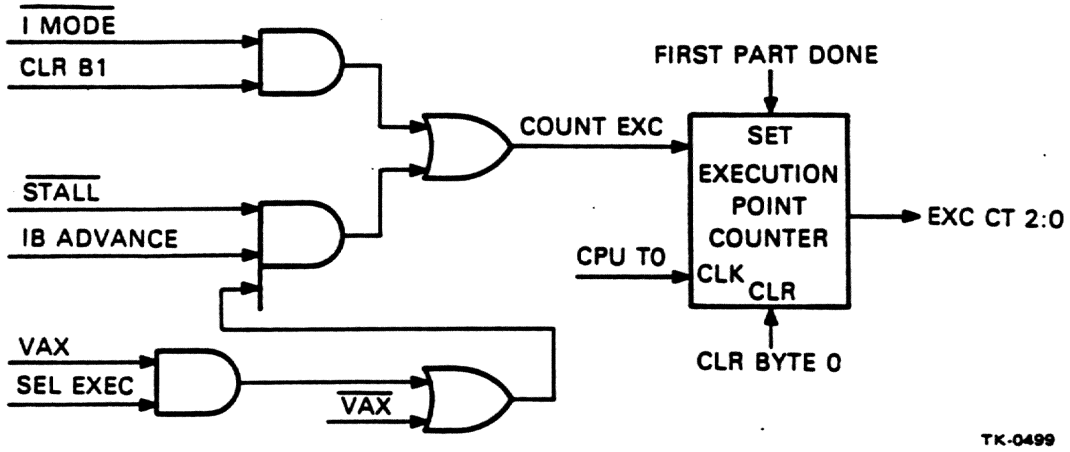


Figure 2-34 Execution Point Counter

The execution point counter is reset to zero when the op code (byte 0) of the instruction is removed from the instruction buffer register or as a result of a buffer flush. Removing the op code indicates that the current instruction has been executed and the next instruction can begin.

The counter is incremented at the beginning of each CPU cycle (T0) providing that one of the following two conditions have been met:

a. the operand specifier has been cleared from the buffer register and the specifier was not in I (Index) mode.

b. the USUB field of the microword indicates the microaddress is to be determined by the instruction decode (IB ADVANCE) and a STALL condition is not present.

The signal FIRST PART DONE (FPD) sets the counter to 7. The FPD flag is set by the microcode (UMSC field = 9) and indicates that an interrupt was received during the middle of an interruptable instruction. When the interrupt service routine is completed, the instruction is fetched a second time. However, rather than evaluating specifiers again, the microcode enters an execution flow.. The FPD flag must be cleared (UMSC field = 8) before evaluation of the next instruction can begin.

2.5.2 Mode Multiplexer (Mode Mux)

The mode multiplexer (Figure 2-35) selects the source for the decode address lines (DECODE ADRS 07:00) which are used to form the microaddress. The address source selected will depend on the operating mode (native or compatibility), the instruction being executed and the state of the execution point counter. Note that in either native or compatibility mode, the BRANCH INSTR line can be generated only at execution point zero.

In native mode, the VAX control word determines which of the following enable lines are generated:

Enable Line	Comment
SEL SPECIFIER	The specifier decode lines are selected for specifier evaluations and double operand optimizations. Refer to Paragraph 2.5.3.
SEL EXECUTE	SEL EXECUTE is generated once the necessary specifier evaluations have been performed and the instruction can be executed. The one's complement of VAX DECODE bits 07 through 00 is used to form the execute address.
BRANCH INSTR	BRANCH INSTR causes the branch condition bits to be ORed with the one's complement of VAX DECODE bits 03:00 to form the low four bits of the microaddress. The high four bits are generated by the one's complement of VAX DECODE bits 07:04.

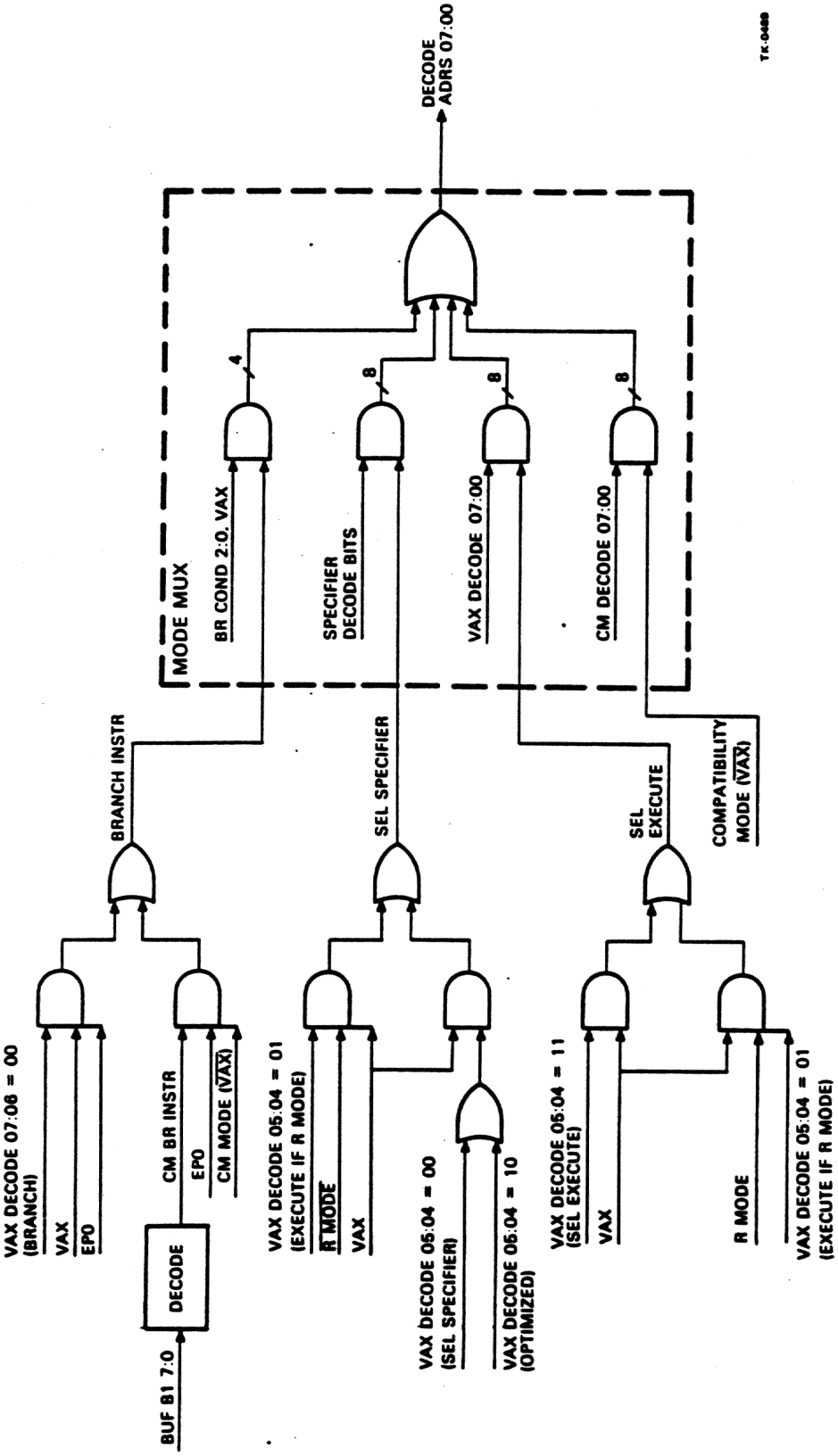


Figure 2-35 Mode Multiplexer Selection

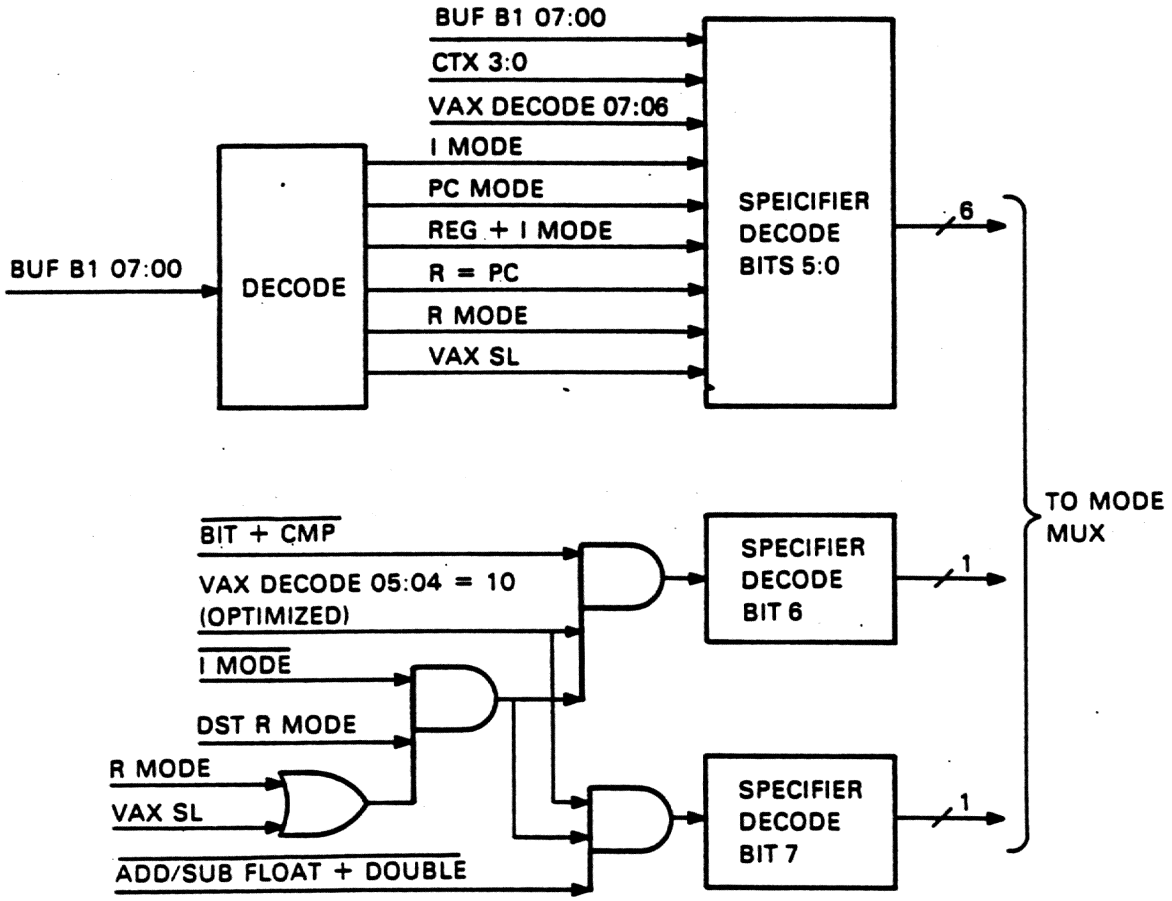
When the system is operating in compatibility mode ($\overline{VAX}$), the compatibility mode decode bits (Paragraph 2.5.9) are selected to generate the microaddress. If the PDP-11 instruction is decoded as a branch instrution, the branch condition bits are ORed with CM DECODE bits 03:00 to form the low four microaddress bits. The high four address bits are generated from the CM DECODE bits 07:04.

The DECODE ADRS lines (07:00) of the mode mux are transferred to the microsequencer to generate bits 07:00 of the next microaddress. Bits 12:08 of the microaddress are generated from the UJMP field of the current microword. The lines selected by the mode mux direct the microprogram to flows which are required to evaluate and perform the current instruction.

2.5.3 Specifier Decode

The mode multiplexer selects the specifier decode logic (Figure 2-36) as the microaddress source for specifier evaluations and double operand optimizations (Paragraph 2.5.3.1). The operand specifier in byte 1 of the instruction buffer register is decoded to provide the correct entry point in the microcode. At each execution point, the mode field (Paragraph 2.5.1) determines if the specifier logic is selected and under what circumstances it can be selected.

Bits 7 of 6 of the specifier logic are equal to zero unless the optimized conditions are met (refer to Paragraph 2.5.2.1). Bits 5 through 0 are generated as a function of the addressing mode, context, and use of the PC. Table 2-7 shows the possible addresses generated by specifier bits 5 through 0.



OPTIMIZATIONS

SPECIFIER DECODE

BIT 7	BIT 6	COMMENT
0	0	OPTIMIZED CONDITIONS NOT MET
0	1	F1 CLASS (ADDX2, SUBX3, ETC.)
1	0	R CLASS (BIT X, CMPX)
1	1	M CLASS (ADDXX, SUBXX, ETC.)

TK-0491

Figure 2-36 Specifier Decode Logic

Table 2-7 Specifier Decode

Addressing Mode		Address Formed by Bits 05:00			
Hex	Notation	R≠PC	R=PC	QUAD	ABORT
0	S_#	00	00	02	01/03
1	S_#	00	00	02	01/03
2	S_#	00	00	02	01/03
3	S_#	00	00	02	01/03
4	I	0C	1C	--	1D
5	R	04	14	6/16	7/17, 5/15
6	(R)	08	18	--	--
7	-(R)	0A	1A	--	--
8	(R)+	09	19	1F	--
9	@ (R)+	0B	1B	--	--
A	D8	0D	0D	--	--
B	@ D8	0F	0F	--	--
C	D16	0D	0D	--	--
D	@ D16	0F	0F	--	--
E	D32	0D	0D	--	--
F	@ D32	0F	0F	--	--

The following is a list of abort addresses and the conditions which cause them to be generated:

Abort Address	Conditions
01	a. Writing into a short literal b. I mode followed by a short literal c. Using a short literal as a VSRC or ASRC
03	Quad context and a. Writing into a short literal b. I mode followed by a short literal c. Using a short literal as a VSRC or ASRC
05	a. Using register mode as an ASRC b. I mode followed by register mode
07	Quad context and a. Using register mode as an ASRC b. I mode followed by register mode
14	Register mode and Rn equals PC
15	Rn equals PC and a. Using register mode as an ASRC b. I mode followed by register mode

16	Rn equals PC with quad content
17	Quad context, Rn equals PC, and a. Using register mode as an ASRC b. I mode followed by register mode
18	Register deferred mode and Rn equals PC
1A	Autodecrement mode and Rn equals PC
1C	I mode and Rn equals PC
1D	I mode followed by I mode

2.5.3.1 Optimizations -- The execution of certain instructions can be optimized by eliminating specifier evaluations, if particular addressing modes are used in the operand specifiers.

Double operand optimizations are implemented for instructions such as ADDW2, BISW2, etc. if the first operand specifier is in short literal or register mode. The second operand specifier in byte 2 of the instruction buffer register is decoded and the signal DST R MODE is generated if it is in register mode.

For double operand instruction which can be optimized, the mode field of the VAX control word will equal 2 (Optimized) at execution point zero. If the specifiers are in an optimized form (i.e., short literal to register or register to register), the address generated will be modified by changing the value of specifier decode bits 7 and 6. The class of the optimized instruction will determine the value of bits 7 and 6, as illustrated in Figure 2-36. If the optimized conditions are not met, bits 7 and 6 are zeros and the specifier address is not modified.

If the first specifier in double operand instructions is not in short literal or register mode, the operand specifier is evaluated at execution point zero. At execution point one, the second specifier is checked to determine if it is in register mode and if it is, the instruction can be executed. If it is not in register mode, a second specifier evaluation must be performed before execution. Figure 2-37 shows the general flow of double operand instructions which can be optimized.

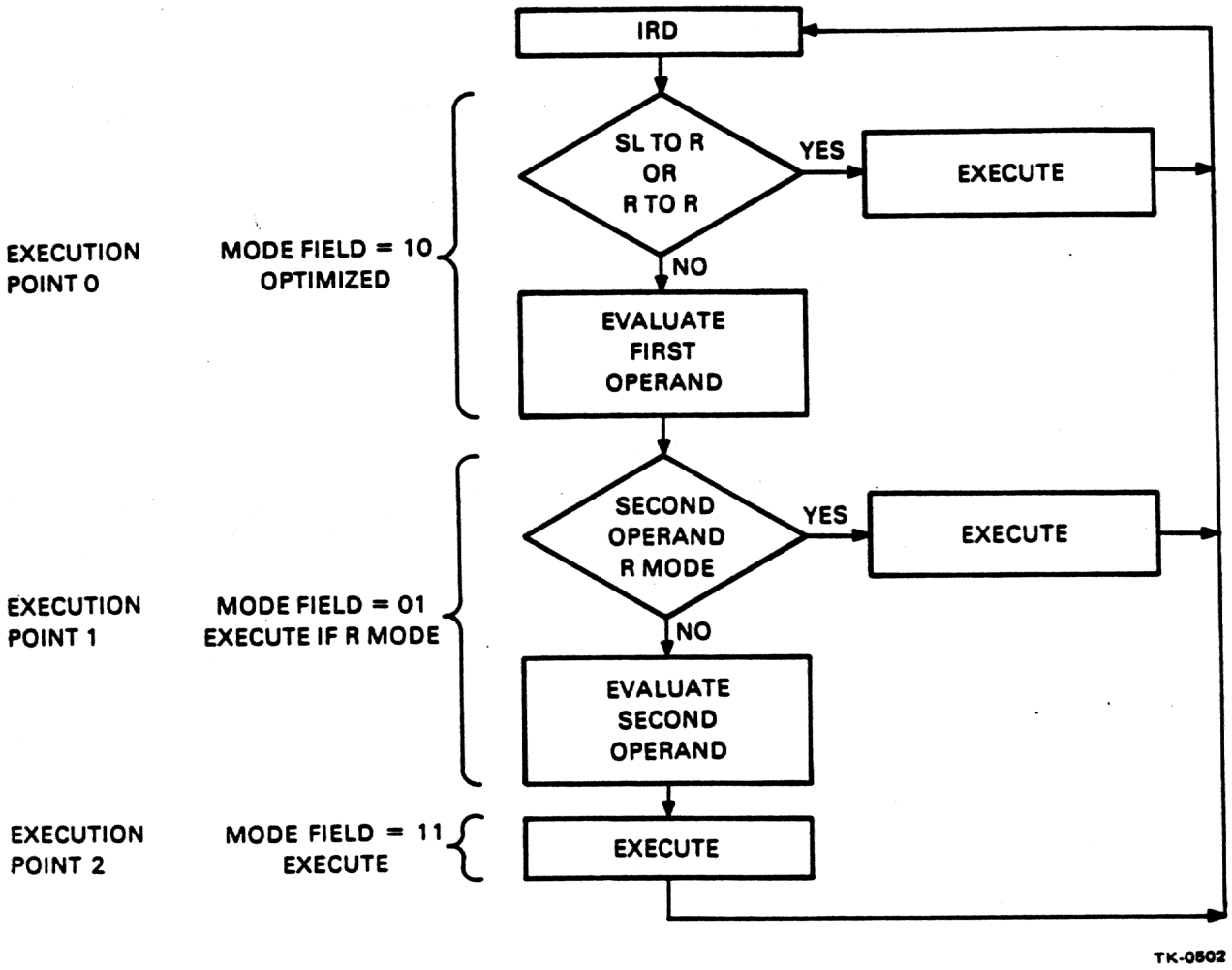


Figure 2-37 Double Operand Optimizations

Single operand optimizations are implemented for instructions such as INCB, TSTB, etc. if the operand specifier is in register mode. For single operand instructions which can be optimized, the mode field of the VAX control word will equal 1 (Execute if R Mode) at execution point zero. If the operand specifier is in register mode, the ones complement of VAX DECODE bits 07:00 will be used to form the microaddress. If the specifier is not in register mode, the specifier decode logic is selected as the address source. Figure 2-38 shows the general flow of single operand instructions which can be optimized.

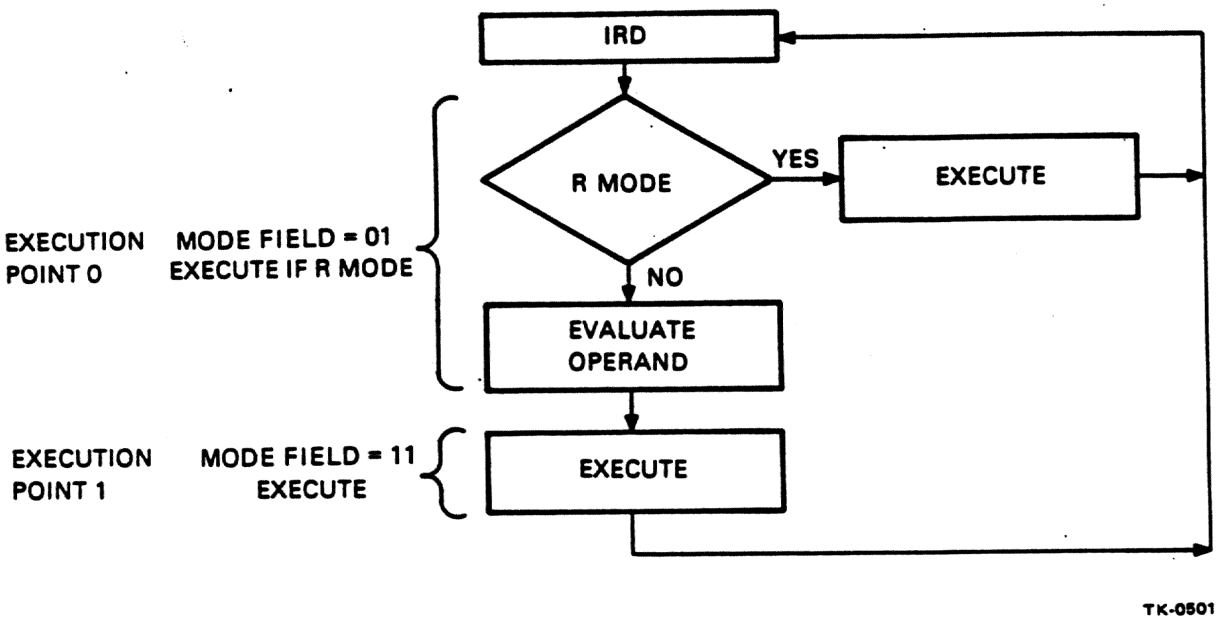


Figure 2-38 Single Operand Optimizations

2.5.4 Branch Decode

If either a VAX or PDP-11 branch instruction is decoded, the branch condition bits (BR COND 02:00) are ORed with the VAX DECODE or CM DECODE bits that form the microaddress (refer to Paragraph 2.5.2). In native mode, the low four bits of the op code (IR03:00) are input to the branch decode logic (Figure 2-39) to determine the particular branch instruction being executed. In compatibility mode, bits from the upper half of the PDP-11 instruction (BUF B1 7, 02:00) are decoded to identify the particular instruction. The condition code bits (N, Z, V, C) of the processor status longword (PSL) are input to the branch decode logic and combined with the selected instruction lines to form the branch condition bits. The microaddress generated will be a function of the operating mode (native or compatibility), the branch instruction being executed, and the status of the PSL condition code bits.

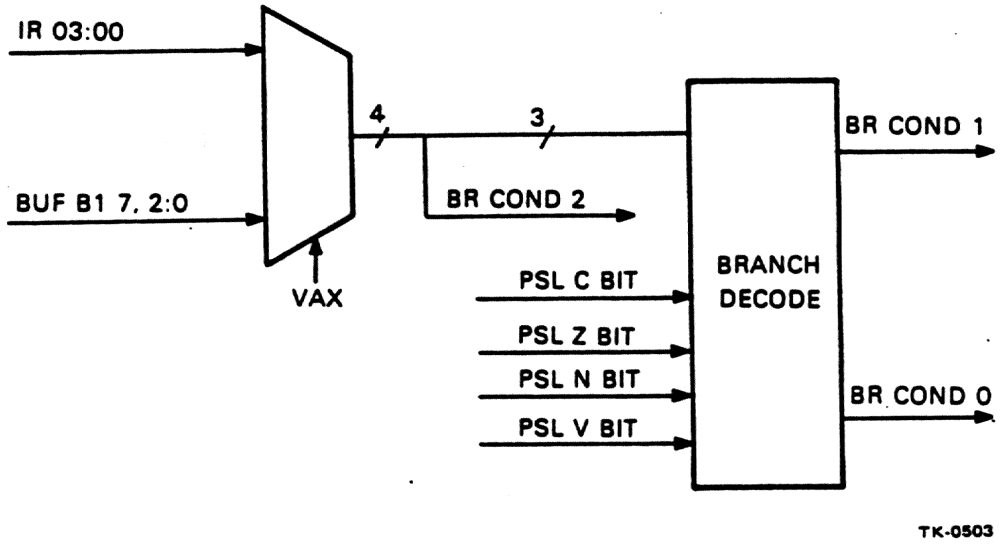


Figure 2-39 Branch Decode Logic

2.5.5 Size Select

The size select logic (Figure 2-40) is used to generate the PC update value (Paragraph 2.4.7) and the STALL condition.

In certain addressing modes, the operand specifier requires additional data to generate the operand address. When these modes are encountered, the PC is incremented to reflect the length of additional data and will point beyond the operand specifier and its extension. The STALL conditions will also be generated if these addressing modes are used and the number of bytes required are not valid. The STALL condition inhibits the execution point counter from being incremented and forces the microcode to try another call. This will continue until all necessary bytes are valid in the buffer. This prevents the microcode from moving to the next decision point before the specifier is evaluated.

In both native and compatibility mode, the specifiers are decoded to check for the addressing modes which require additional data. The following shows the number of additional bytes required for each associated mode.

Native Addressing Modes	Additional Bytes Required
Displacement or Displacement Deferred	
Byte	1
Word	2
Longword	4
Immediate	1, 2, or 4 bytes depending on context
Absolute	4
Branch Displacement	
Byte	1
Word	2

Compatibility Addressing Modes	Additional Bytes Required
Index	2
Index Deferred	2
Immediate	2
Absolute	2

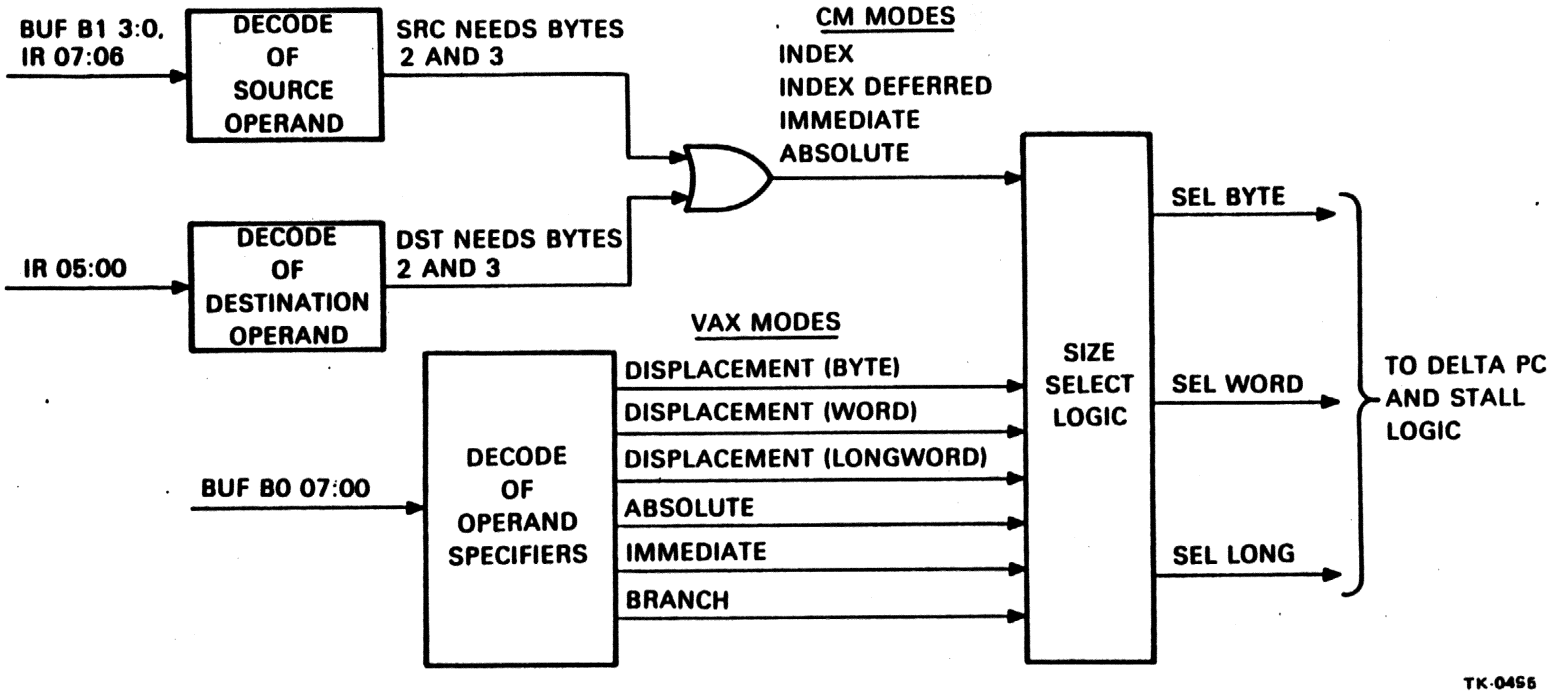


Figure 2-40 Size Select Logic

2.5.6 I Mode Flag

In index (I) mode, the operand specifier consists of two bytes -- a primary operand specifier and a base operand specifier (Figure 2-41).

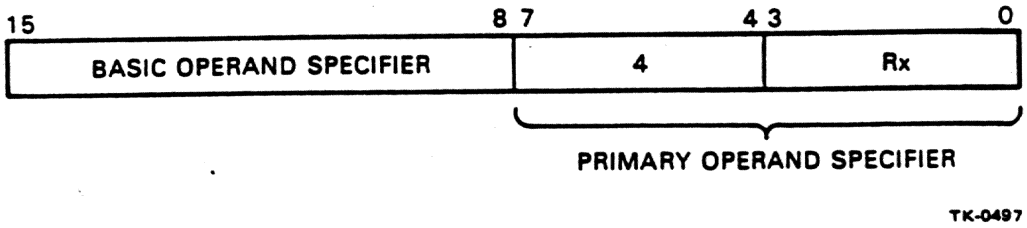


Figure 2-41 Index (I) Mode Operand Specifiers

Both specifiers are required to generate the operand address. When the addressing mode is decoded as I mode, incrementation of the execution point counter is inhibited when the primary operand specifier is removed from byte 1 of the buffer register. This prevents the microcode from moving to the next decision point before both specifiers are evaluated and the operand address is generated.

The I mode flag (Figure 2-42) is set when the primary operand specifier is in I mode and it has been cleared from the buffer register. This flag enables the hardware to evaluate the base operand specifier and still retain the mode of the primary specifier. Abort addresses are generated when the base operand specifier is in register, literal, or index mode or when the PC is used as the index register in the primary specifier. Refer to Paragraph 2.5.3 for a list of abort addresses generated as a result of I mode specifiers. The I mode flag is cleared when the op code is cleared from byte 0 of the buffer register or when the base operand specifier is cleared from the buffer.

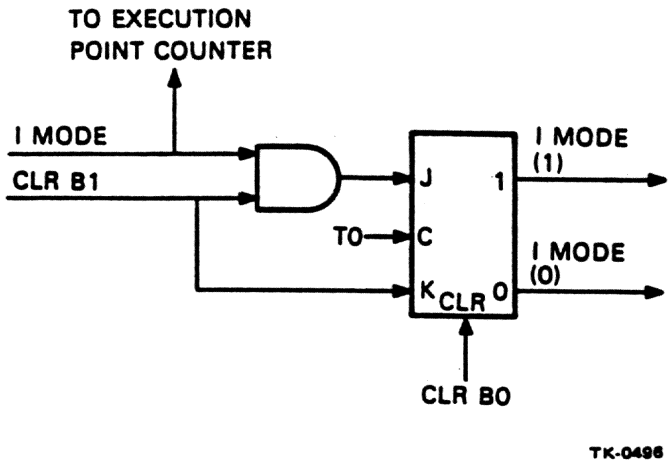


Figure 2-42 I Mode Flag

2.5.7 Specifier Constants

The specifier constants (Figure 2-43), generated by the instruction decode logic, are input to the Fast Constant multiplexer of the data path (Paragraph 2.6.3.5). These constants are generally implemented in the evaluation of autoincrement and autodecrement modes.

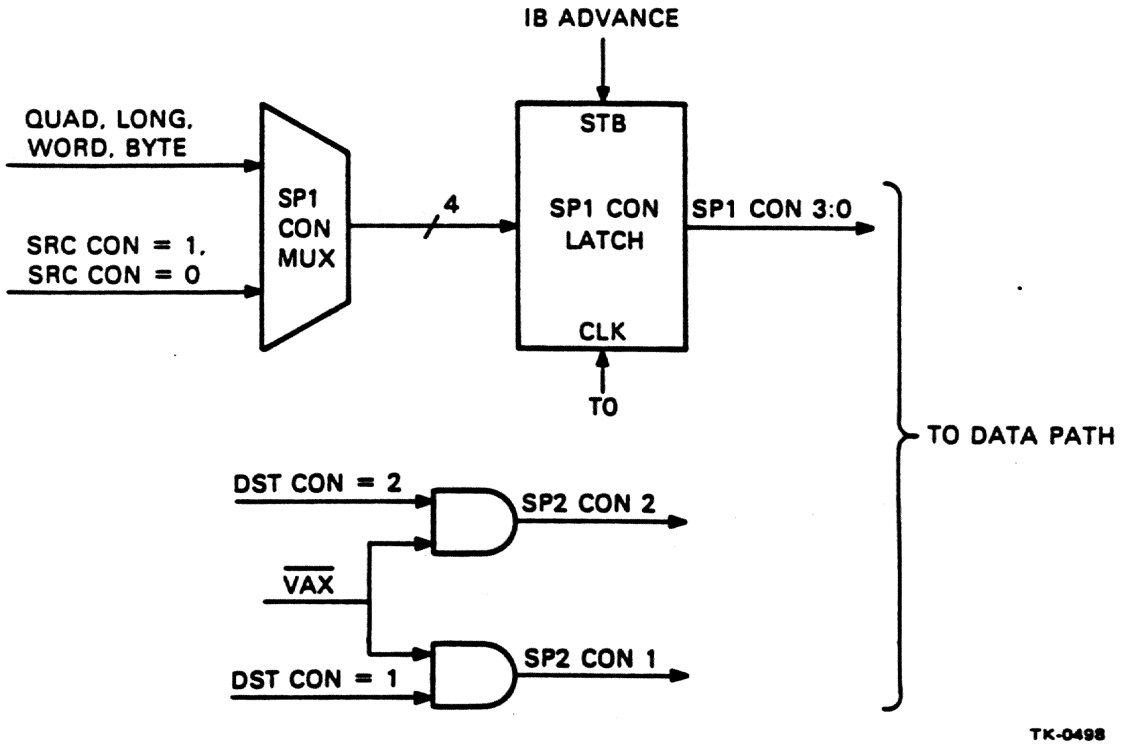


Figure 2-43 Specifier Constants

In native operating mode, the Specifier 1 Constant (SP1, CON Ø3:ØØ) is a value determined by the data type of the operand specifier being evaluated; 8 (quadword), 4 (longword), 2 (word), and 1 (byte). Specifier 2 Constant (SP2 CON Ø2:Ø1) is Ø.

In compatibility mode, Specifier 1 Constant is the number 1 or 2, determined by the data type (byte or word) of the instruction and the number of the source register. Specifier 2 Constant is the number 1 or 2, determined by the instruction data type and the number of the destination register.

2.5.8 Microsequencer Branch Conditions

The instruction decode logic provides two branch condition bits (BRC Ø1:ØØ) which are used in the microsequencer to form the microbranch addresses (Paragraph 2.3.2.1). The condition bits are transferred to a branch multiplexer (physically located on the Interrupt Control board). When the UBEN field of the current microword equals 8, 9, A, or B, the BRC bits are selected to generate the low two bits of the next microaddress.

The BRC multiplexers (refer to Figure 2-44) select inputs generated from a decode of both VAX and PDP-11 instructions. Table 2-8 shows the inputs selected for the BEN field values in each mode.

Table 2-8 Instruction Decode Microbranch Conditions

BEN Field Value	Native Mode		Compatibility Mode	
	BRC 1	BRC Ø	BRC 1	BRC Ø
8	ASRC or VSRC	ASRC or QUAD	Ø Class	J class or DM27
	Ø = Normal, 1 = Quad 2 = Field Src, 3 = Address Src			
9	IRØ2	IRØ1	SM or DM = 47 or 57	DST R = PC
A			Ø	SRC R = PC
B	IB CHK 1	IB CHK Ø	IB CHK 1	IB CHK Ø
	Ø = TB Miss, 1 = Error 2 = STALL, 3 = Data OK		Ø = TB Miss, 1 = Error 2 = STALL, 3 = Data OK	

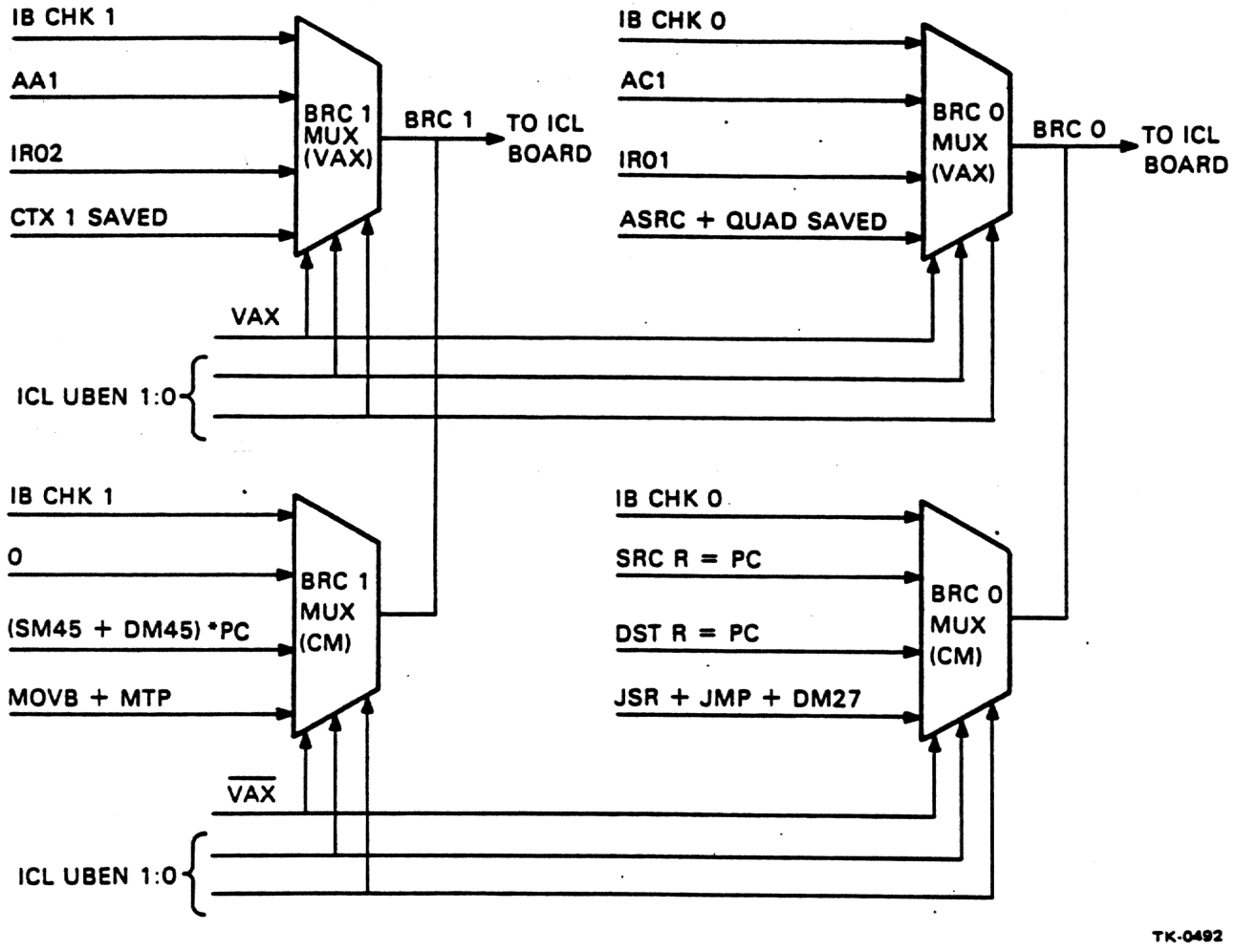


Figure 2-44 Microbranch Condition Multiplexers

2.5.9 Compatibility Mode Decode

Compatibility mode is specified when bit 31 (CM bit) of the Processor Status Longword is set. This mode enables the execution of PDP-11 instructions stored in the instruction buffer register. The compatibility mode decode logic generates address lines (CM DECODE 07:00) which are selected by the mode multiplexer to form the next microaddress. The CM decode logic can direct the microcode to a flow which either evaluates the source or destination mode of the operand or exeuctes the instruction. The PDP-11 instructions are stored in bytes 0 and 1 of the buffer register as shown in Figure 2-45.

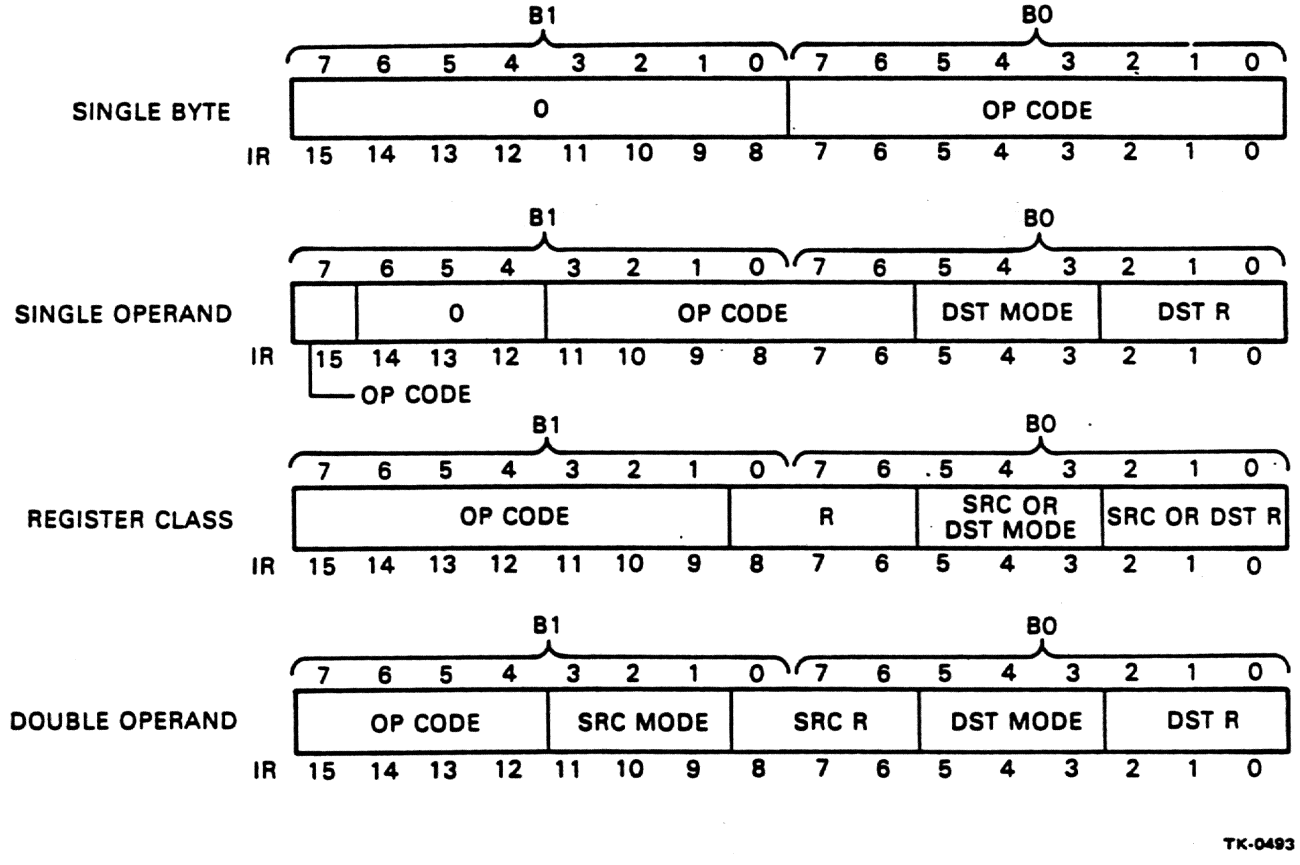


Figure 2-45 Format of PDP-11 Instructions in Buffer Register Bytes 0 and 1

2.5.9.1 CM Execution Address ROM -- The compatibility mode decode bits are generated from execution ROMs or from a decode of the SRC/DST mode field of the instruction. Refer to Figure 2-46. The execution ROMs are divided between double operand and register class instructions, single operand instructions, and single byte instructions. Enabling of the ROMs depends on the format of the instruction stored in buffer register bytes Ø and 1. For example, the single byte ROM is enabled if byte 1 is all zeros. The ORed output of the execution ROMs is transferred to the CM DECODE multiplexer. The execution lines are selected if CM EXEC is enabled. Generation of the compatibility mode execute line is dependent on the instruction class and state of the execution point counter (EXC CT Ø1:ØØ). The execution ROMs are used to generate the microaddress once all necessary source and destination operand evaluations have been completed. The following shows the conditions under which CM EXEC is generated for each value of the execution point counter:

Execution Point Counter	Conditions which enable CM EXEC
EXC CT Ø1:ØØ = ØØ	Single byte OR DM = Ø AND SM = Ø OR DM = Ø AND register class OR DM = Ø AND single operand
Execution Point Counter	**Conditions which enable CM EXEC**
ESC CT Ø1:ØØ = Ø1	SM = Ø OR DM = Ø OR register class OR single operand
EXC CT Ø1:ØØ = 1Ø or 11	Any combination of instruction class and operand mode

A maximum of three execution points are required to evaluate and execute any PDP-11 instruction. In double operand instructions (with neither the source or destination mode equal to zero), the source operand is evaluated at execution point Ø, the destination operand is evaluated at execution point 1, and the instruction is executed at execution point 2. The instruction can be executed at execution point Ø if, for example, the instruction is a type with no operands, or a double operand instruction with both operands in register mode.

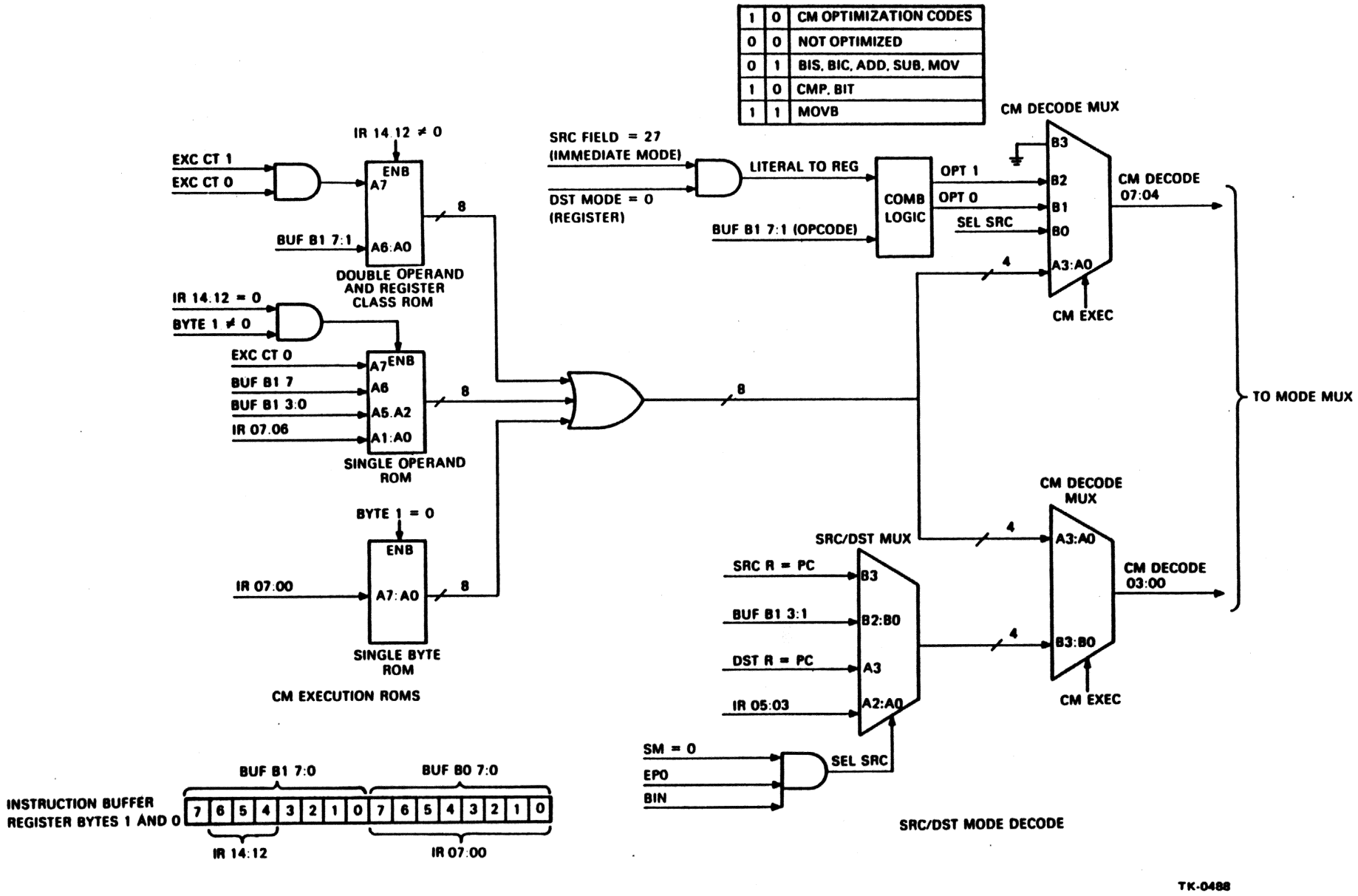


Figure 2-46 Compatibility Mode Decode

2.5.9.2 SRC/DST Mode Decode -- If source destination operand evaluations are required, the CM EXEC line is not generated and the CM DECODE multiplexers select inputs from a decode of instruction source or destination fields. The lower four bits (CM DECODE Ø3:ØØ) are generated from the SRC/DST multiplexer. The source inputs to the mux are enabled only under one set of circumstances; if the instruction is of double operand (binary) class, the execution point counter equals Ø, and the source mode is not equal to zero. The destination inputs are enabled to evaluate the source or destination mode in register class instructions and the destination mode in single or double operand instructions.

The upper four address bits (CM DECODE Ø7:Ø4) are generated from a decode of the SRC/DST mode fields and the op code. The execution of certain double operand instructions can be optimized if the operation is a literal to register transfer. Optimizations require that the source operand be in immediate mode (27) and the destination operand be in register mode. The constant or literal data will be located in the second word of the instruction (buffer register bytes 2 and 3). The microaddress generated will depend on the op code of the instruction and the optimization code generated. Refer to Figure 2-46.

2.6 DATA PATH DESCRIPTION

This section provides a functional description of each area of the data path with relation to microcode control and instruction execution. A discussion of logic unit operation is provided in areas which require further clarification.

2.6.1 General Data Path Organization

The data path is divided into four major areas: Address, Arithmetic, Data, and Exponent section. Refer to Figure 2-48. Each section operates as an independent unit, capable of processing data or addresses in parallel with operations being performed in another section.

2.6.2 Data Path Control

The execution of each instruction requires a given number of sequential operations to be performed in the data path. The steps needed for instruction execution are defined by microinstructions or microwords, each of which consists of 96 bits and is organized into various length control fields. Each section of the data path derives its control from an associated microword field. The fields which are designed for data path control are illustrated in Figure 2-47.

The Control Store bus (BUS CS) provides the path for the transfer of each microword field to various areas of the central processor.

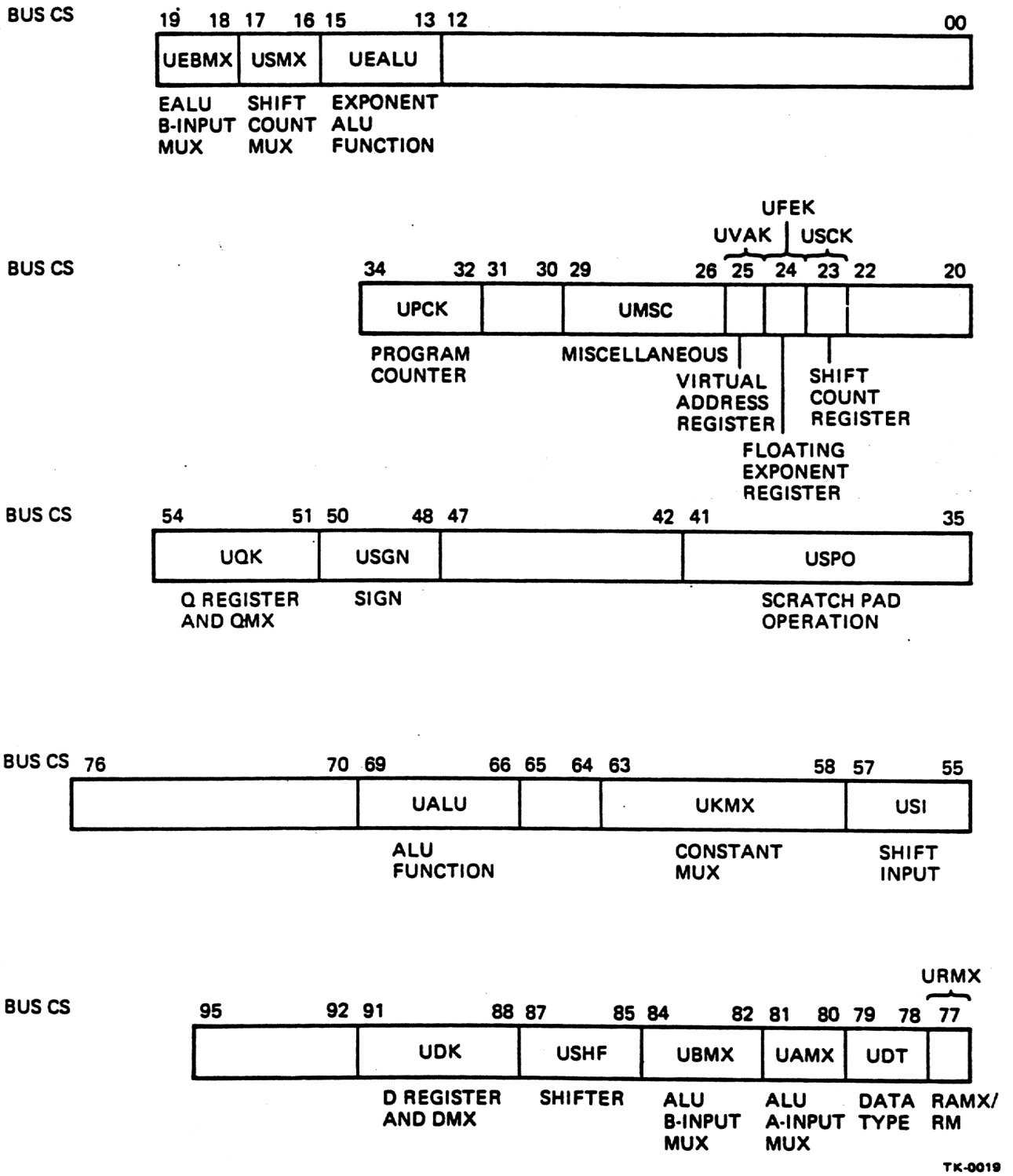


Figure 2-47 Microword Fields for Data Path Control

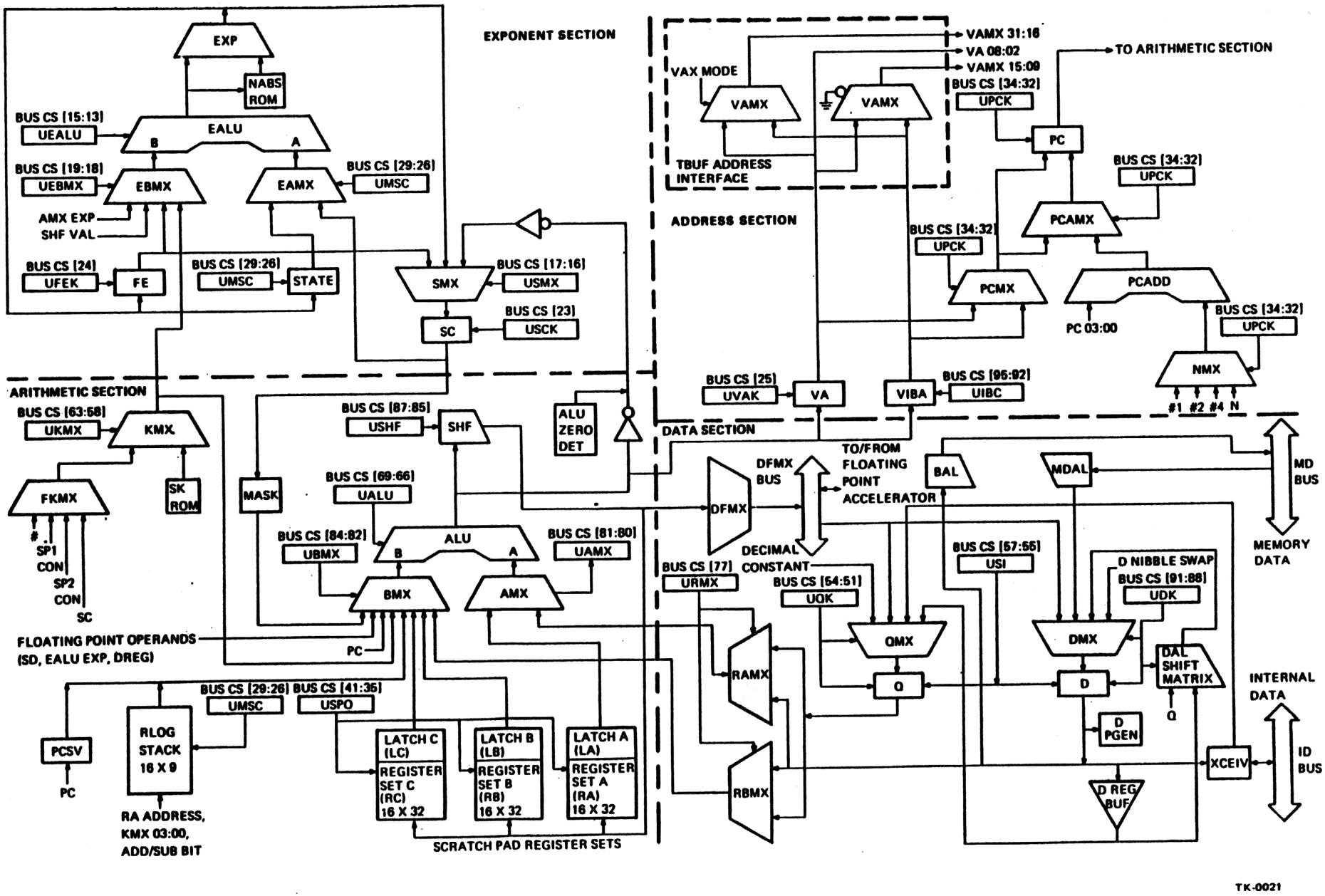


Figure 2-48 Data Path Block Diagram

2.6.3 Arithmetic Section

The Arithmetic section of the data path consists of the arithmetic logic unit (ALU), general purpose registers, bit mask generator, constant generator, shifter, and temporary storage registers. Contents from registers in the Address and Data sections are input to the arithmetic section to allow the required arithmetic and logic operations to be performed on data and addresses.

Three data types are processed by the Arithmetic section: 8-bit bytes, 16-bit words, and 32-bit longwords. The data type is controlled by the UDT field of the microinstruction.

	UDT Field		
Hex	BUS CS 79	BUS CS 78	Context
Ø	Ø	Ø	Longword (Long, Quad, Floating, or Double floating)
1	Ø	1	Word
2	1	Ø	Byte
3	1	1	Instruction Dependent

If the UDT field equals 3, the context is determined by the instruction decode logic.

2.6.3.1 Arithmetic/Logic Unit (ALU)

The main processing unit of the arithmetic section is the ALU which performs 32-bit arithmetic operations (with fast carry look-ahead logic) 32-bit logic operations. The ALU functions provided are used during instruction execution for data modification and address generation. The ALU also provides the focal point for the transfer of information betweeen other sections of the data path. Register contents from the address, data, and exponent sections are routed to the ALU through the A and B input multiplexers (AMX and BMX). These inputs are used for a number of functions depending on the instruction being executed and the ALU operation selected. The operation select inputs of the ALU are controlled by the UALU field of the microinstruction. The operation to be performed may be defined explicitly or if the UALU field equals 3, the function is determined by the instruction decode logic.

Table 2-9 shows the correspondence between the UALU field value and the operation performed.

As previously mentioned, if the UALU field equals 3, the function selected is a result of the instruction being executed. The instruction dependent ROMs are addressed by the instruction register op code bits and the ROM outputs provide the necessary select lines. In Instruction Dependent mode, the full logic and arithmetic functions of the ALU are available (refer to Table 2-1Ø).

When the UALU field equals 1 or 6, the RLOG stack is updated with the general register (RA) address, the lower four bits of the KMX

Table 2-9 UALU Function Select

UALU(Ø3:ØØ) Hex	BUS CS 69	68	67	66	ALU Select S3	S2	S1	SØ	ALU Mode Input Logic M=1=H(FLn) Arith M=Ø=L(FLn)	ALU C Input with C=1=L(carry)	ALU Function
0	0	0	0	0	0	1	1	0	0	1	A minus B
1	0	0	0	1	0	1	1	0	0	1	A minus B(R LOG)
2	0	0	1	0	0	1	1	0	0	1	A minus B minus1
3	0	0	1	1	-	-	-	-	-	-	Instruction Dependent
4	0	1	0	0	1	0	0	1	0	1	A plus B plus 1
5	0	1	0	1	1	0	0	1	0	D	A plus B
6	0	1	1	0	1	0	0	1	0	Ø	A plus B(R LOG)
7	0	1	1	1	1	1	0	1	1	1	$A+\overline{B}$
8	1	0	0	0	0	1	1	0	1	1	$A \oplus B$
9	1	0	0	1	0	1	1	1	1	1	$A\overline{B}$
A	1	0	1	0	0	0	0	0	1	1	$\overline{A}$
B	1	0	1	1	1	0	0	1	0	COND with PSL CBit	A plus B plus C
C	1	1	0	0	1	1	1	0	1	1	A + B
D	1	1	0	1	1	0	1	1	1	1	AB
E	1	1	1	0	1	0	1	0	1	1	B
F	1	1	1	1	1	1	1	1	1	1	A

Table 2-10 Available ALU Functions in Instruction Dependent Mode

SELECTION				ACTIVE-HIGH DATA		
				M = H	M = L: ARITHMETIC OPERATIONS	
				LOGIC	C_n = 0 = H	C_n = 1 = L
S_3	S_2	S_1	S_0	FUNCTIONS	(no carry)	(with carry)
L	L	L	L	$F = \overline{A}$	F = A	F = A plus 1
L	L	L	H	$F = \overline{A + B}$	F = A + B	F = (A + B) plus 1
L	L	H	L	$F = \overline{A}B$	$F = A + \overline{B}$	$F = (A + \overline{B})$ plus 1
L	L	H	H	F = 0	F = minus 1 (2's complement)	F = zero
L	H	L	L	$F = \overline{AB}$	$F = A$ plus $A\overline{B}$	$F = A$ plus $A\overline{B}$ plus 1
L	H	L	H	$F = \overline{B}$	$F = (A + B)$ plus $A\overline{B}$	$F = (A + B)$ plus $A\overline{B}$ plus 1
L	H	H	L	$F = A \oplus B$	F = A minus B minus 1	F = A minus B
L	H	H	H	$F = A\overline{B}$	$F = A\overline{B}$ minus 1	$F = A\overline{B}$
H	L	L	L	$F = \overline{A} + B$	F = A plus AB	F = A plus AB plus 1
H	L	L	H	$F = \overline{A \oplus B}$	F = A plus B	F = A plus B plus 1
H	L	H	L	F = B	$F = (A + \overline{B})$ plus AB	$F = (A + \overline{B})$ plus AB plus 1
H	L	H	H	F = AB	F = AB minus 1	F = AB
H	H	L	L	F = 1	F = A plus A*	F = A plus A plus 1
H	H	L	H	$F = A + \overline{B}$	F = (A + B) plus A	F = (A + B) plus A plus 1
H	H	H	L	F = A + B	$F = (A + \overline{B})$ plus A	$F = (A + \overline{B})$ plus A plus 1
H	H	H	H	F = A	F = A minus 1	F = A

*Each bit is shifted to the next more significant position.

and a bit which determines if an add or subtract is requested. The RLOG stack contains 16 locations which are used to keep a record of changes made to the scratch pad register set (refer to Paragraph 2.6.3.8).

2.6.3.2 ALU A-Input Multiplexer (AMX)

The AMX provides the means for transferring information from the data section or from the scratch pad register set A to the A input of the ALU (refer to Figure 2-49). Data which will be used during instruction execution can be stored in the Q or D register (in the data section) or in the scratch pad register sets. When the contents of these registers must be manipulated or used in an operation, the AMX selects the correct source as the ALU input.

If the required data is contained in the scratch pad register set, the AMX will select the latch A (LA) output of the register set. The contents of the correct register location must be stored in the latch before the AMX selection is made.

If the data required for the operation is contained in the D or Q registers, the AMX will select the RAMX of the data section as the ALU input. Both the D and Q register are input to the RAMX and the proper source is selected by the microinstruction (refer to Section 2.6.5.1). If the contents of the D or Q register is less than 32 bits, the data must be sign or zero extended to 32 bits by the RAMX. The data input to the AMX must be in the correct format because the ALU operates on 32 bit (longword) data types only.

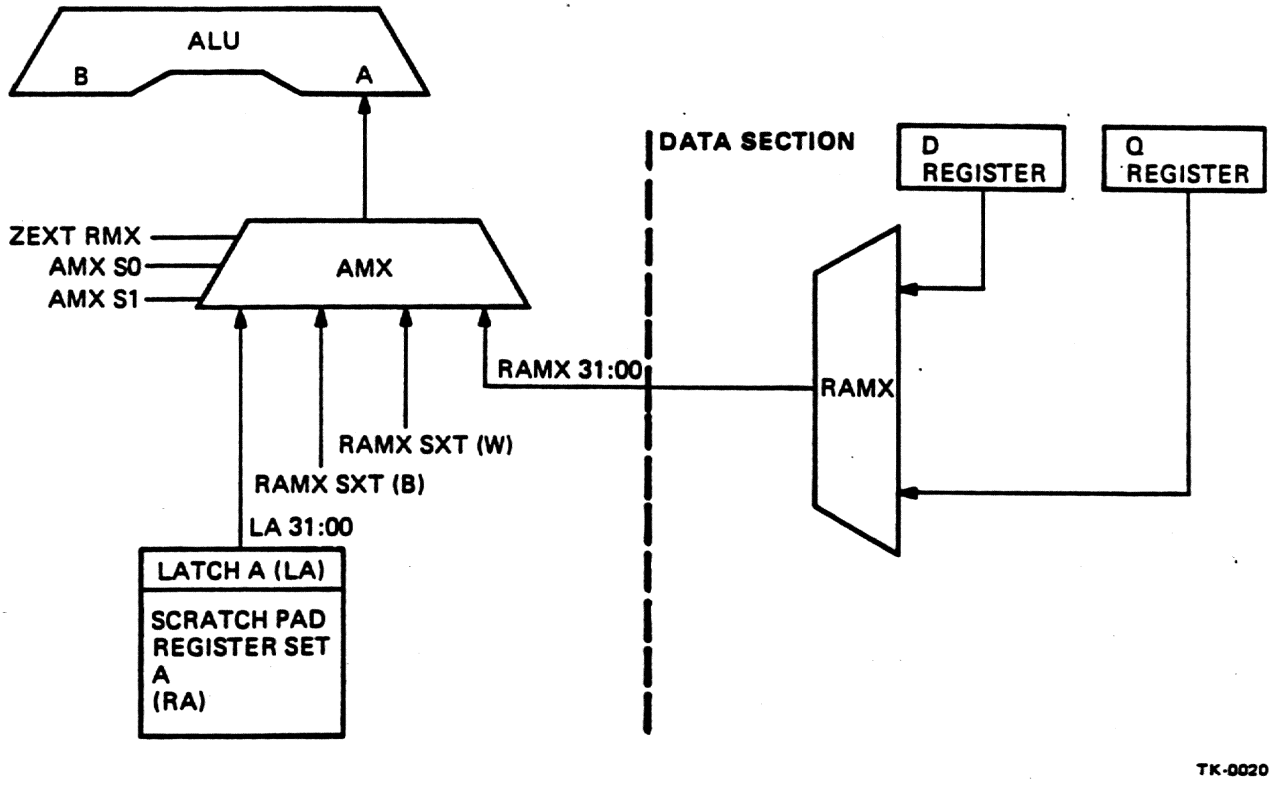


Figure 2-49 ALU A Input Multiplexer (AMX)

The AMX is controlled by the UAMX field of the microword as follows:

Hex	UAMX Field *BUS CS 81	BUS CS 80	Data Selected
0	0	0	LA
1	0	1	RAMX
2	1	0	RAMX SXT (Sign extension determined by UDT field)
3	1	1	RAMX 0XT (Zero extension determined by UDT field)

The data type and required sign or zero extension of the RAMX is determined by the value of the UDT field as follows:

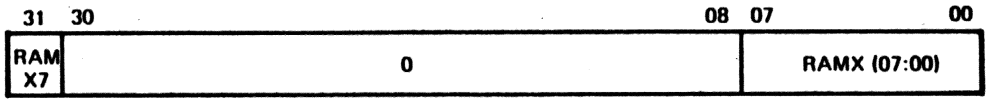
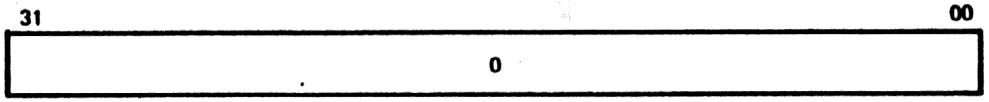
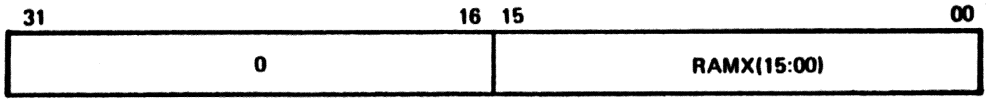
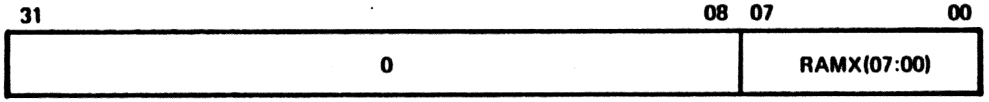
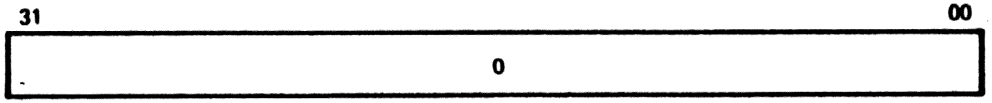
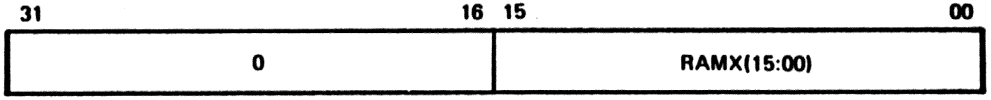
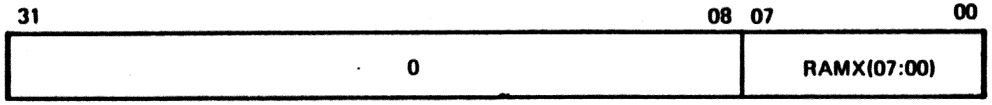
Hex	UDT Field BUS CS 79	BUS CS 78	Data Type
0	0	0	Longword: SXT(L) or All zeros
1	0	1	Word: SXT(W) or 0XT(W)
2	1	0	Byte: SXT(B) or 0XT(B)
3	1	1	Instruction Dependent

Note that the zero extension of a longword format results in all zeros at the AMX output.

When the UDT field equals 3, the data type is determined by the instruction decode logic which provides information for instruction execution and operand specifier evaluation. Instructions requesting quad, floating, or double floating context will result in a longword data type.

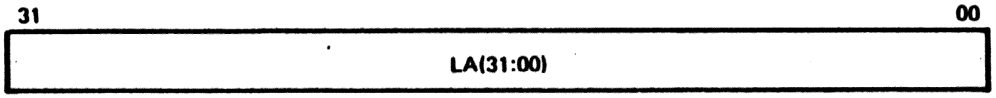
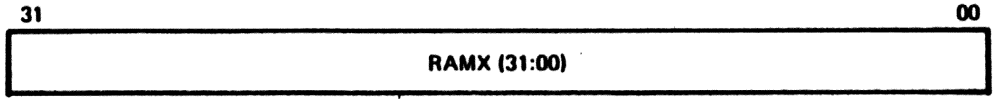
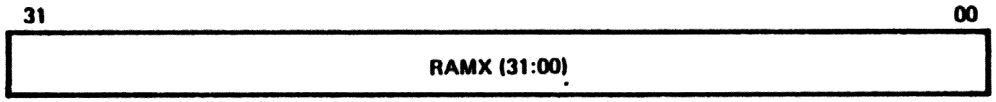
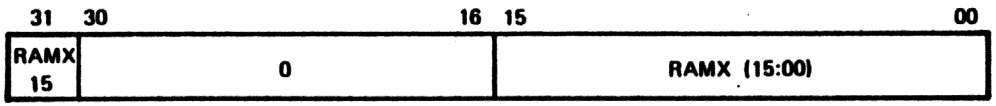
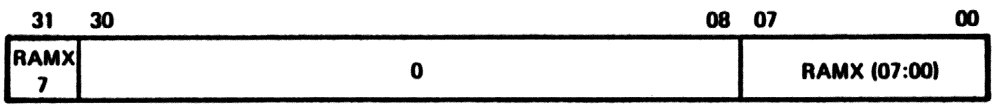
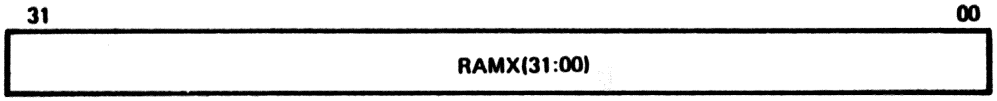
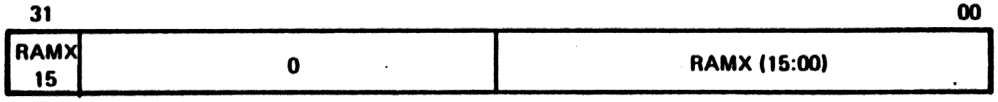
Table 2-11 shows the relationship between the control field values and the data format of the AMX output.

Table 2-11 AMX Control

UAMX 1 0	UDT 1 0	INST DEP DATA TYPE L W B	AMX DATA FORMAT	SIGN OR ZERO EXTENSION
H L	H H	L L H		SXT BYTE
H H	L L	X X X		0XT LONG
H H	L H	X X X		0XT WORD
H H	H L	X X X		0XT BYTE
H H	H H	H L L		0XT LONG
H H	H H	L H L		0XT WORD
H H	H H	L L H		0XT BYTE

X=IRRELEVANT

Table 2-11 AMX Control (Cont)

UAMX 1 0	UDT 1 0	INST DEP DATA TYPE L W B	AMX DATA FORMAT	SIGN OR ZERO EXTENSION
L L	X X	X X X		
L H	X X	X X X		
H L	L L	X X X		SXT LONG
H L	L H	X X X		SXT WORD
H L	H L	X X X		SXT BYTE
H L	H H	H L L		
H L	H H	L H L		SXT WORD

X= IRRELEVANT

2.6.3.3 ALU B-Input Multiplexer (BMX)

The BMX (Figure 2-50) allows information from the data, exponent, address, and arithmetic sections to be input to the ALU. These inputs are used separately or in combination for the following operations:

Address generation -- Modification of the Program Counter (PC) can be accomplished by routing the PC from the address section to the ALU through the BMX. For example, in displacement addressing modes utilizing the PC, the new PC value is calculated with the BMX selecting PC and the AMX selecting the sign extended displacement value.

Manipulation of stored data -- In certain operations the BMX is used for the same purposes as the AMX. Information stored in the data section (D or Q registers) and in the scratch pad register set B are input to the ALU through the BMX. The contents of register set B are an exact copy of register set A contents. The feature of having the same information available at both ALU inputs enables fast access to both the source and destination register during the execution of register to register mode instructions. Also, this feature allows instructions which require the B MINUS A function to be executed without swapping the operand from one ALU input to the other. The function B MINUS A is not provided by the ALU. Therefore, set-up of the operands at the proper inputs would be required if the contents of the register set was not provided at both the ALU inputs.

Data required for instruction execution is also stored in scratch pad register set C. The contents of these temporary storage locations are latched (in LC) before being input to the BMX.

Instruction Restart -- The RLOG and PCSV inputs to the BMX are used specifically to restart an instruction. If a fault occurs, the entire 32 bit PC can be reconstructed from the contents of the PC Save (PCSV) register and the general registers can be restored from the RLOG stack (refer to Paragraph 2.6.3.8). The PCSV register is used to hold the lower 8 bits of the PC at the beginning of an instruction. The RLOG stack is comprised of 16 locations which contain a record of changes made to the scratch pad register set. The RLOG and PCSV inputs are selected when the BMX microword field equals zero and the signal READ RLOG is present.

Mask and Constant Generation -- The MASK input to the BMX routes the output of the bit mask generator (Paragraph 2.6.3.6) to the ALU for logic operations. The KMX input supplies constants (Paragraph 2.6.3.5) required for the execution of instructions and evaluation of operand specifiers.

Assembly of floating-point data formats -- During the execution of floating-point instructions, inputs from the data, exponent, and control section are assembled by the BMX to form a packed floating-point data type.

The fraction position is taken from the DREG, the sign (SD) from the control logic, and the exponent from the EALU (refer to Table 2-14 for the format of the packed floating-point data type). The SD bit contains the sign of the destination fraction in floating-point operations. The SD bit is loaded and controlled by the USGN field of the microword during execution of a floating-point instruction. Table 2-12 shows the relationship between the USGN field value and the source sign (SS) and destination sign (SD) selected. Due to the timing delays in routing the data, both the EALU and ALU must be selected for logic mode to ensure that the data is available in the arithmetic section when required.

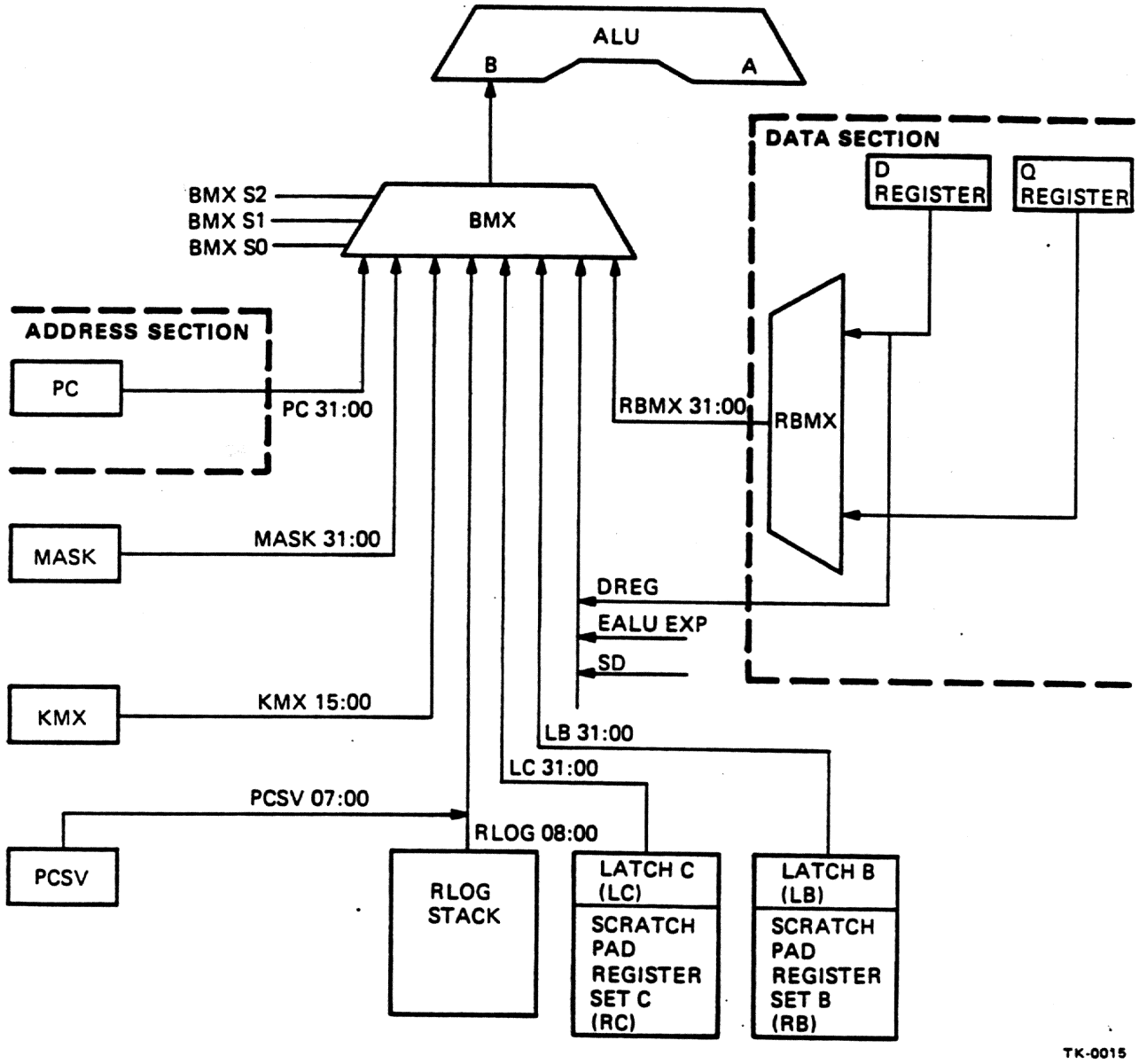


Figure 2-50 ALU B-Input Multiplexer (BMX)

Table 2-12 Source and Destination Sign Selection

USGN Field				Value Selected	
Hex	BUS CS 50	BUS CS 49	BUS CS 48	Source Sign (SS)	Destination Sign (SD)
0	L	L	L	SS	SD
1	L	L	H	ALU15	SD
2	L	H	L	SD	SD
3	L	H	H	SS	SD
4	H	L	L	SS	SS
5	H	L	H	ALU15 XOR SS XOR IR1	ALU15
6	H	H	L	ALU15 XOR SS	ALU15
7	H	H	H	0	0

Selection of the BMX input is controlled by the UBMX field of the microword and two enabling conditions (R=PC and READ RLOG). Table 2-13 shows the relationship between the UBMX field and the input selected.

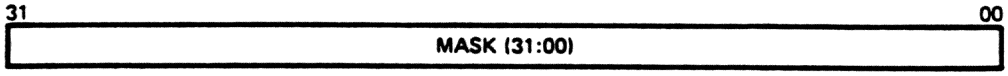
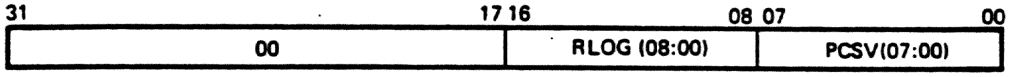
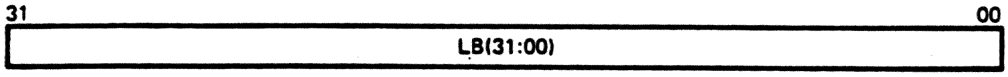
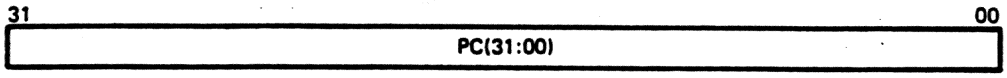
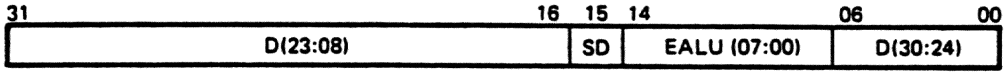
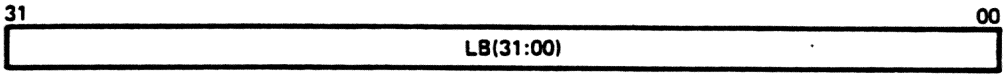
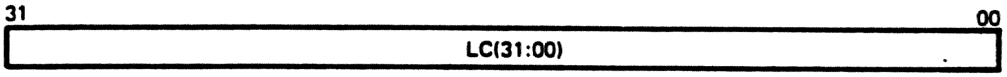
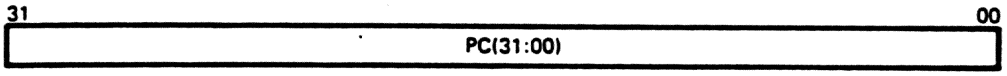
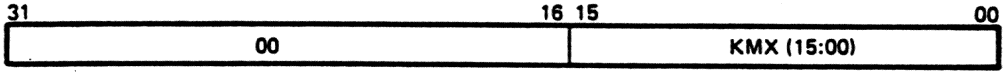
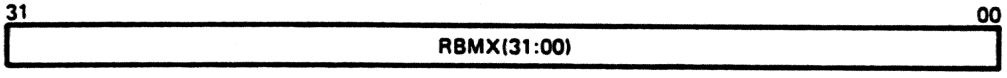
Table 2-13 BMX Input Selection

Hex	UBMX Field BUS CS 84	BUS CS 83	BUS CS 82	RLOG READ	R=PC	BMX SELECT S2	S1	SØ	BMX DATA
Ø	Ø	Ø	Ø	Ø	X	Ø	Ø	Ø	MASK
Ø	Ø	Ø	Ø	1	X	Ø	Ø	1	RLOG and PCSV
1	Ø	Ø	1	Ø	Ø	Ø	1	1	LB
1	Ø	Ø	1	Ø	1	1	Ø	1	PC
2	Ø	1	Ø	Ø	X	Ø	1	Ø	Packed Floating-point
3	Ø	1	1	Ø	X	Ø	1	1	LB
4	1	Ø	Ø	Ø	X	1	Ø	Ø	LC
5	1	Ø	1	Ø	X	1	Ø	1	PC
6	1	1	Ø	Ø	X	1	1	Ø	KMX
7	1	1	1	Ø	X	1	1	1	RBMX

X=irrelevant

Table 2-14 illustrates the BMX data format for each input selected.

Table 2-14 BMX Data Format

UBMX (HEX)	READ RLOG	R=PC	BMX DATA FORMAT
0	0	X	
0	1	X	
1	0	0	
1	0	1	
2	0	X	
3	0	X	
4	0	X	
5	0	X	
6	0	X	
7	0	X	

X=IRRELEVANT

2.6.3.4 ALU Shifter (SHF)

The Shifter (Figure 2-51) provides the data source for the scratch pad register sets and allows the transfer of information between the arithmetic and data sections. The Shifter is used in the arithmetic section to multiply (left shift) or divide (right shift) the ALU output.

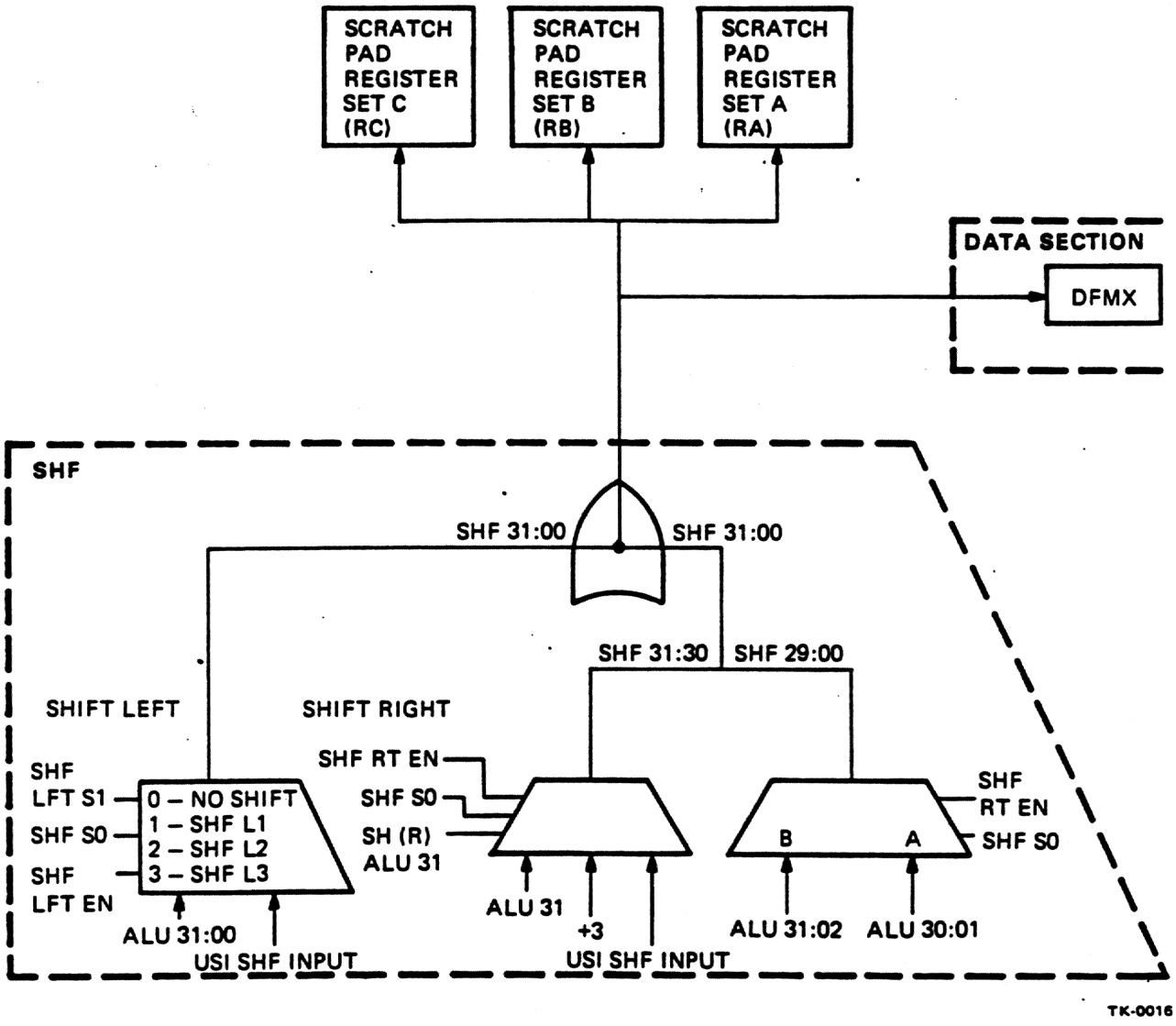


Figure 2-51 ALU Shifter (SHF)

In index mode specifier evaluations, the SHF is used as a multiplier to create the correct index values for address calculations. Index mode addressing permits the access of data arrays which can be byte, word, longword, or quadword organized.

Access to an array requires the contents of an index register to be multiplied by the size of the array's primary operand in bytes (1 for byte, 2 for word, 4 for longword or floating, and 8 for quadword or double floating). The SHF provides the required multiplication by shifting the data an appropriate number of bits. Multiplication by 1 requires no shift (L0), multiplication by 2 requires a left shift of 1 (L1), by 4 requires a left shift of 2 (L2), by 8 requires a left shift of 3 (L3). The shift requirement, which is a function of the data type, can be defined explicitly by the USHF field or can be controlled by the UDT field and determined by the instruction decode logic.

The SHF is also implemented in the execution of multiply, divide compatibility mode rotate, and shift instructions. These instructions can force a left shift by 1 (L1), right shift by 1 (R1) or right shift by 2 (R2). Input to bit positions which are left empty by the shift is controlled by the microword shift input control (USI) field.

		USI Field		
Hex	BUS CS 57	BUS CS 56	BUS CS 55	Shift input
0	0	0	0	PSL (N bit)
1	0	0	1	ALU (Bit 31)
2	0	1	0	0
3	0	1	1	0
4	1	0	0	0
5	1	0	1	Q Register (Bit 31)
6	1	1	0	0
7	1	1	1	1

Selection of the SHF is determined by the value of the USHF field of the microword. For USHF field values 1, 2, or 4, input to the vacated SHF bit positions is determined by the USI field. For USHF field values 3 or 5, the vacated SHF positions are zero.

		USHF Field		
Hex	BUS CS 87	BUS CS 86	BUS CS 85	SHF Output
0	0	0	0	ALU with no shift (L0)
1	0	0	1	ALU left shifted by 1 (L1)
2	0	1	0	ALU right shifted by 1 (R1)
3	0	1	1	ALU shift determined by UDT field
4	1	0	0	ALU right shifted by 2 (R2)
5	1	0	1	ALU left shifted by 3 (L3)
6	1	1	0	Reserved
7	1	1	1	Reserved

If the USHF field equals 3, the SHF output is controlled by the value of the UDT field.

Hex	UDT Field BUS CS 79	BUS CS 78	SHF Output
Ø	Ø	Ø	Longword; ALU left shifted by 2 (L2)
1	Ø	1	Word; ALU left shifted by 1 (L1)
2	1	Ø	Byte; ALU with no shift (LØ)
3	1	1	Instruction dependent: any of the above or Quadword; ALU left shifted by 3 (L3)

If the UDT field equals 3, the SHF output is determined by the instruction decode logic (Specifier 1 Constant Ø2:ØØ).

Table 2-15 shows the relationship between the microcode field values and the data format of the SHF output.

USI indicates that the bit value is determined by the USI field.

2.6.3.5 Constant Multiplexer (KMX)

The KMX provides the arithmetic section with constants required for the performance of various functions. The KMX constants are input to the ALU via the BMX. Selection of the KMX allows the source of constants to be either the Fast Constant Multiplexer (FKMX) or the Slow Constant (SK) ROM (refer to Figure 2-52).

Table 2-15 SHF Data Format

USHF (HEX)	UDT (HEX)	SHF DATA FORMAT
0	X	31 … 00: ALU 31:00
1	X	31 … 01 00: ALU 30:00 | USI
2	X	31 30 … 00: USI | ALU 31:01
3	0	31 … 02 01 00: ALU 29:00 | 0 | 0
3	1	31 … 01 00: ALU 30:00 | 0
3	2	31 … 00: ALU 31:00
3	3	31 … 00: INSTRUCTION DEPENDENT
4	X	31 30 29 … 00: USI | USI | ALU 31:02
5	X	31 … 03 02 01 00: ALU 28:00 | 0 | 0 | 0

X=IRRELEVANT

USI INDICATES THAT THE BIT VALUE IS DETERMINED BY THE USI FIELD.

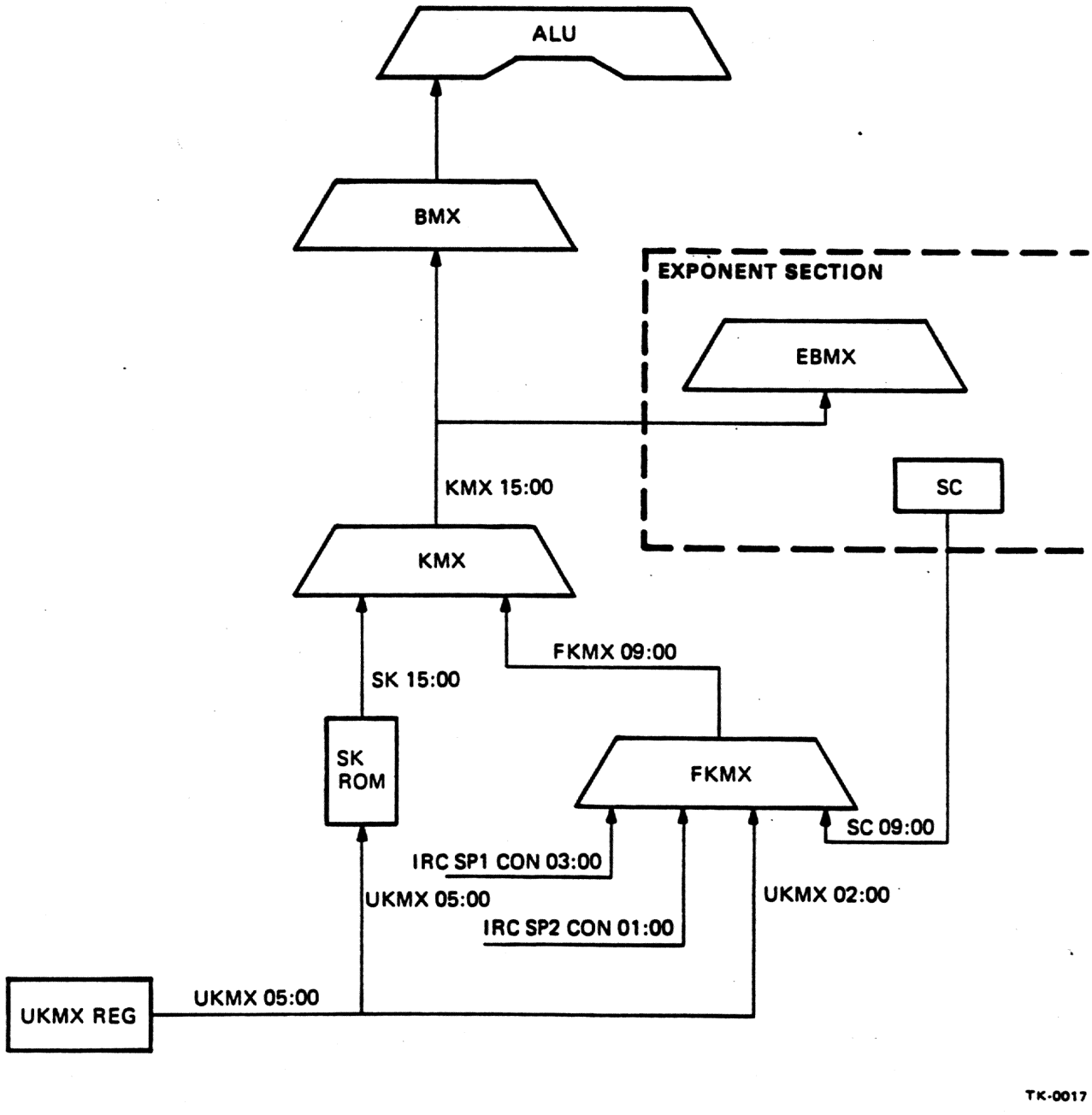


Figure 2-52 Constant Multiplexer

Fast Constant Multiplexer (FKMX) -- The fast constant inputs are provided by the instruction decode logic, the UKMX field of the microinstruction, or the Shift Count (SC) register of the exponent section.

The instruction decode logic generates SP1CON (Specifier 1 Constant) and SP2CON (Specifier 2 Constant). In VAX operating mode, SP1CON is the number 1, 2, 4, or 8, determined by the data type of the operand specifier being evaluated. SP2CON is the number 0.

In 11 Compatibility mode, SP1CON is the number 1 or 2, determined by data type and register number of the instruction source register. SP2CON is the number 1 or 2, determined by the data type and register number of the instruction destination register.

A constant (#1, 2, 3, 4, or 8) may also be explicitly defined by the UKMX field of the microinstruction.

The Shift Count (SC) input to the FKMX provides a data path between the exponent and arithmetic sections. The Shift Count register may also be used to store constants for arithmetic operations.

The fast constants are generally used to increment or decrement data. These constants are also implemented in the evaluation of auto-increment and auto-decrement modes.

Slow Constant (SK) ROM -- The remainder of microprogram constants are supplied by the SK ROM. These constants are used to execute instructions, isolate bits or bit fields, provide exponent biasing and select shift constants.

The KMX selection is determined by the value of UKMX field of the microinstruction.

	UKMX Field						
			BUS	CS			
Hex	63	62	61	60	59	58	KMX Output
0	0	0	0	0	0	0	#8
1	0	0	0	0	0	1	#1
2	0	0	0	0	1	0	#2
3	0	0	0	0	1	1	#3
4	0	0	0	1	0	0	#4
5	0	0	0	1	0	1	SP1CON
6	0	0	0	1	1	0	SP2CON
7	0	0	0	1	1	1	SC09:00
8 through 3F							SK ROM Constant (SK15:00)

2.6.3.6 Bit Mask Generator (MASK)

The MASK is used in the data section to generate a bit pattern within a 32-bit word. This bit pattern can be used to isolate fields of bits through the use of the ALU logic functions. This process occurs in the execution of bit field instructions and in the memory management process of translating virtual addresses to physical when the addresses are not already translated in the Translation Buffer.

The MASK generator output is a single zero bit in a field of ones. The Shift Count (SC) input addresses a single bit in a longword. The MASK generator decodes the address and inserts a zero in the desired position with the remaining output bits equalling ones (refer to Figure 2-53).

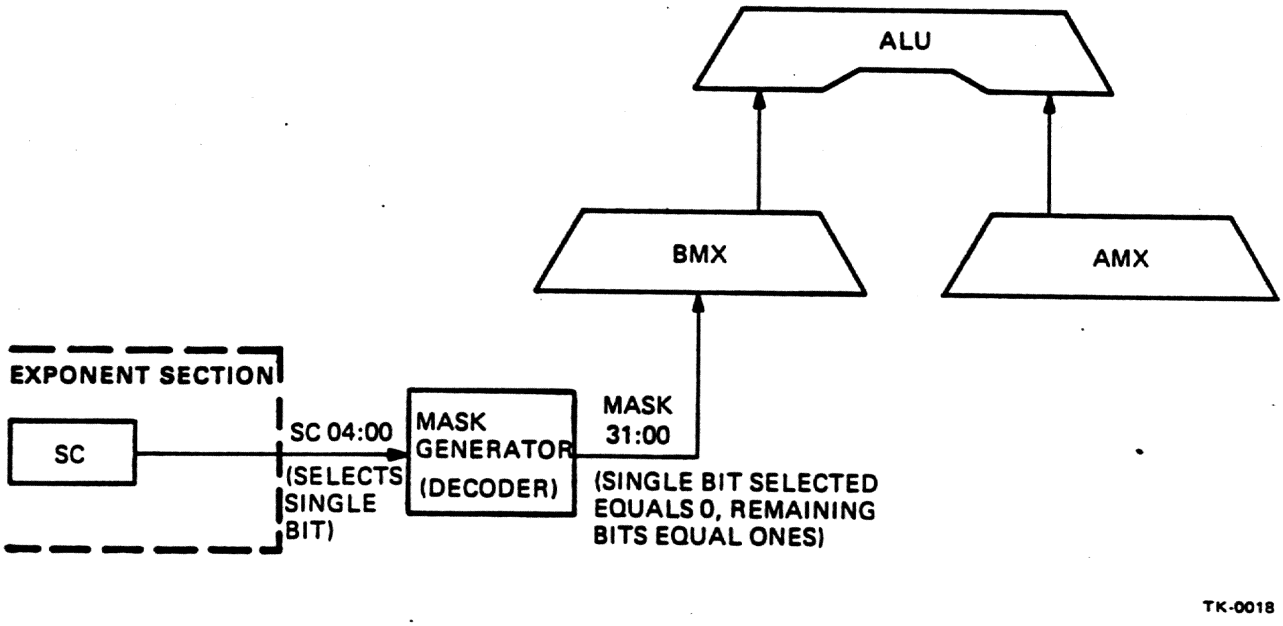


Figure 2-53 Mask Generator

The MASK output is used to generate a bit pattern in a 32 bit word.

Example: To generate a pattern of all ones from bit position 5 and above, the procedure would require: AMX=Ø, BMX=MASK, SC=5, and ALU Operation=A PLUS B PLUS 1 (Figure 2-54).

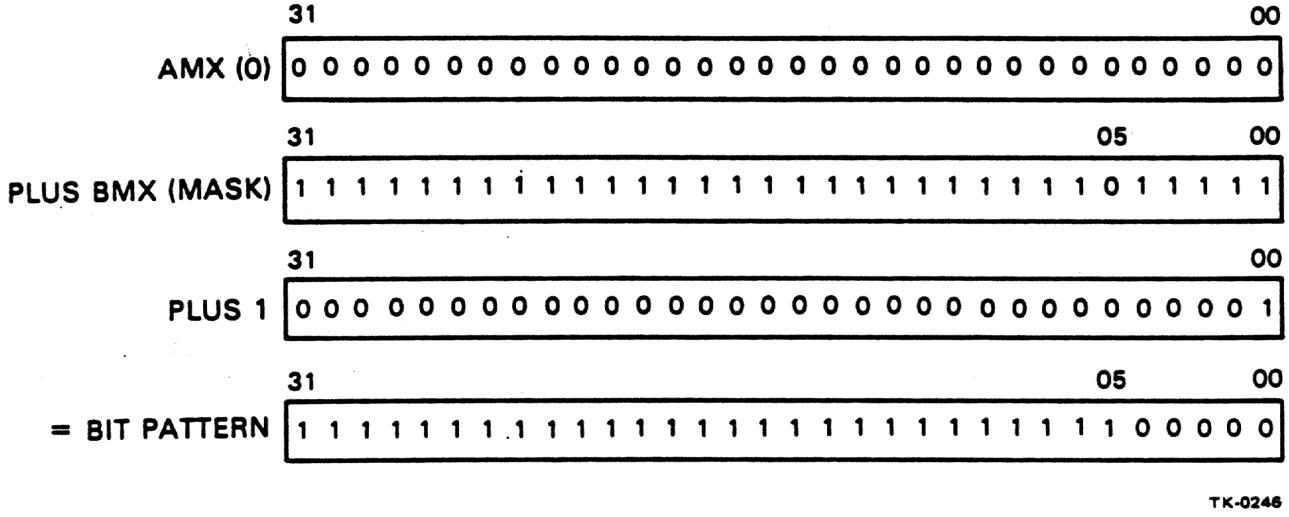


Figure 2-54 Mask Example

The following example demonstrates the use of the MASK generator to extract a field of information from a 32 bit longword.

Suppose that the field begins at bit P and the field is of length S (Figure 2-55).

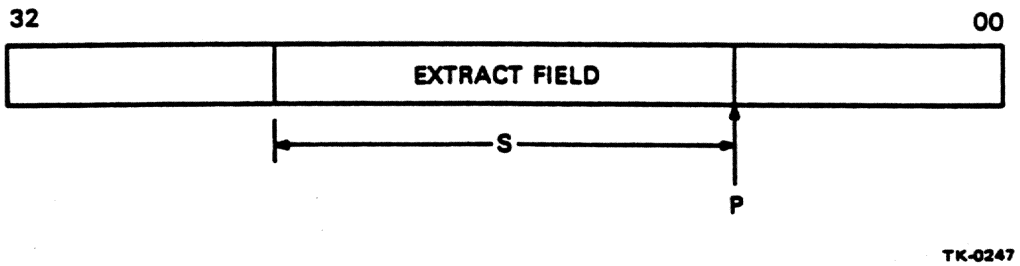


Figure 2-55 Extract Field

Let P = 9 and S = 8 for this example.

Two masks must be generated to extract a field.

To generate the first mask, the procedure would require: AMX = Ø, BMX = MASK, SC = 9, and ALU Operation = A PLUS B PLUS 1. This operation, shown in Figure 2-56, results in a bit pattern A.

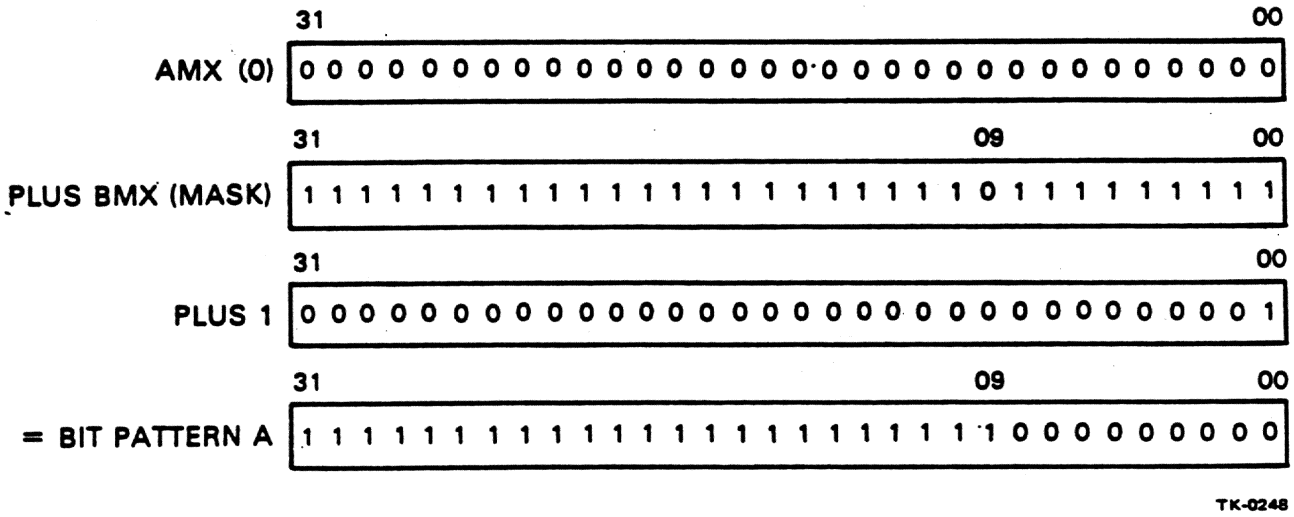


Figure 2-56 Bit Pattern A

Bit pattern A is stored in a temporary register for later use.

The second mask would require the procedure: AMX = Ø, BMX = MASK, SC = 17, (P plus S) and ALU Operation = A PLUS B PLUS 1. This operation will result in bit pattern B, shown in Figure 2-57.

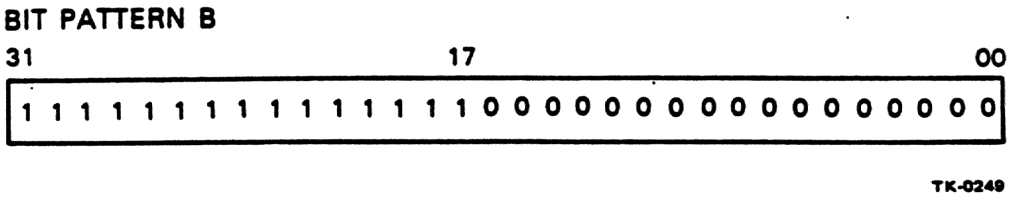


Figure 2-57 Bit Pattern B

The final procedure requires the ALU Logic function A AND NOT B. This ALU operation will result in the bit pattern shown in Figure 2-58.

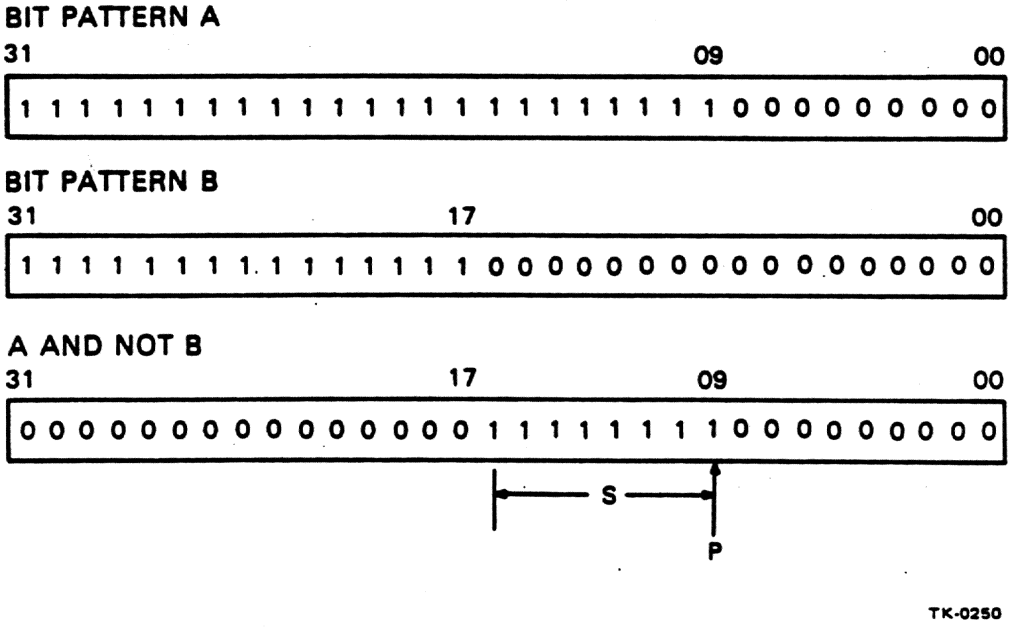


Figure 2-58 Extract Field Pattern

The resulting bit pattern shown in Figure 2-58 can be ANDed with the longword of data to extract 8-bit field of information beginning at bit Ø9.

2.6.3.7 Scratch Pad Register Sets -- Three 16 word by 32 bit register sets are provided as a fast memory storage area (refer to Figure 2-59).

Register Set C (RC) -- RC is used as temporary storage for addresses and operands generated during the execution of the microprogram. A latch (LC) holds the contents of a temporary register which is fetched for use in the Arithmetic Section.

Register Set A (RA) and Register Set B (RB) -- RA and RB provide a two-port storage area for the 16 processor general registers. These temporary registers are implemented during addressing mode evaluations and used as fast memory storage during instruction execution. The two-port feature allows fast access to both the source register (from RB to the BMX) and the destination register (from RA to the AMX) during the execution of register to register mode instructions. The associated latches (LA and LB) hold the contents of the temporary registers for use in the Arithmetic Section.

The three register sets (RC, RB, RA) and associated latches are controlled by the value of the USPO field of the microword. Table 2-16 shows the relationship between the USPO field value and the function selected.

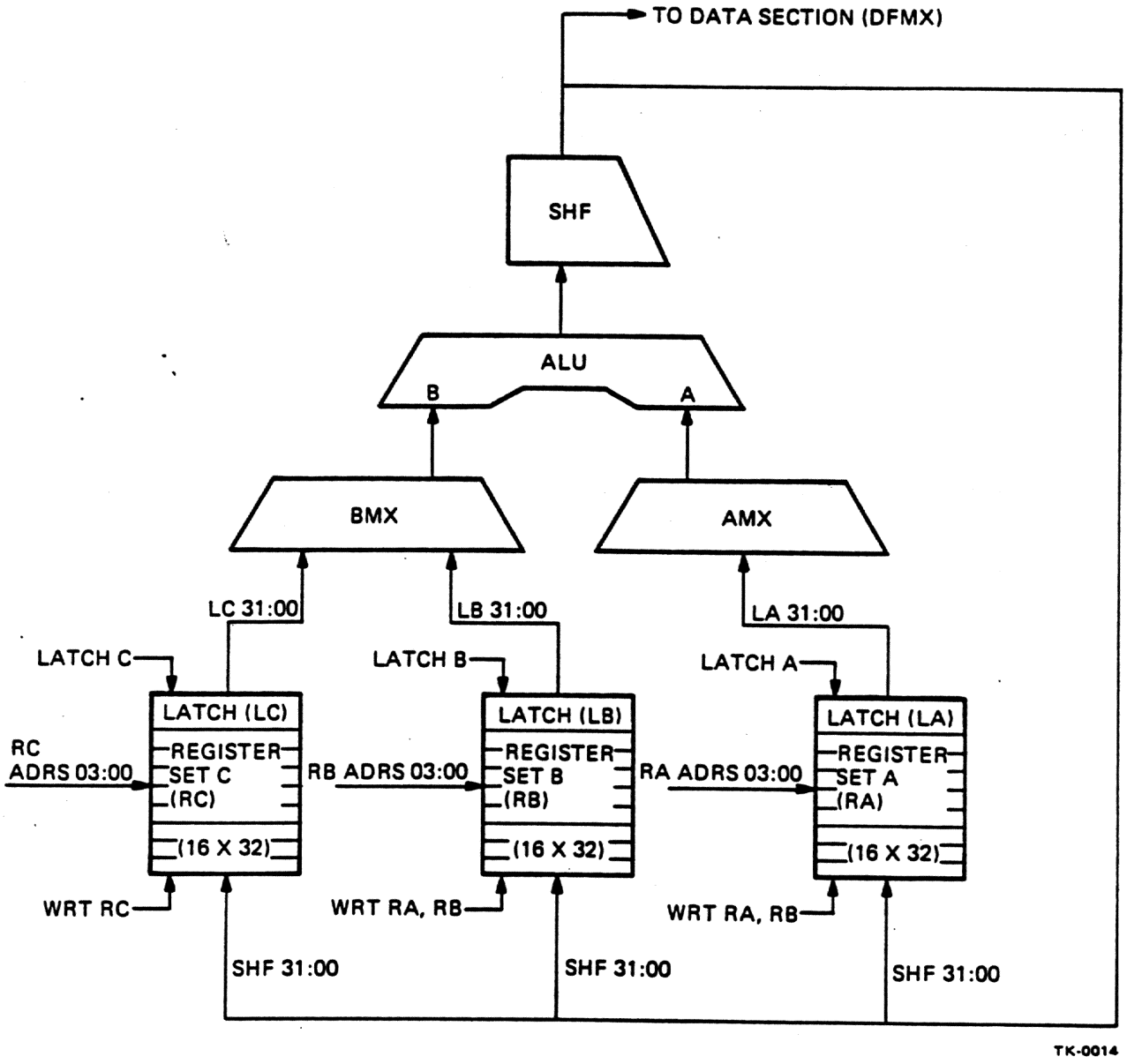


Figure 2-59 Scratch Pad Register Sets

Table 2-16 Scratch Pad Operation

Hex Value	USPO Field BUS CS 41	40	39	38	37	36	35	Scratch Pad Operation
00-05	0	0	0	0	X	X	X	No Operation
06	0	0	0	0	1	1	0	Load LC ;Address=SC03:00
07	0	0	0	0	1	1	1	Write RC, RB ;Address=SC03:00
08-0F	0	0	0	1	A	C	N	Load LA, LB ; Address determined by ACN value
10-17	0	0	1	0	R	N		Load LA ;Address=General Register (R0-R7)
18-1F	0	0	1	1	A	C	N	Write RA, RB ; Address determined by ACN value
20-2F	0	1	0		R	N		Load LC ;Address=Temporary Register (T0-TF)
30-3F	0	1	1		R	N		Write RC ;Address=Temporary Register (T0-TF)
40-4F	1	0	0		R	N		Load LA, LB ;Address=General Register (R0-RF)
50-5F	1	0	1		R	N		Write RA, RB ;Address=General Register (R0-RF)
60-6F	1	1	0		R	N		Load LA, LB ;Address=General Register (R1) and Write RC ;Address=Temporary Register (T0-TF)
70-7F	1	1	1		R	N		Load LC ;Address=Temporary Register (T0-TF) and Write RA, RB ;Address=General Register (R1)

Register Set C (RC) can be addressed explicitly as a register number (T0-TF) or the address can be defined by the SC register bits 03:00

Register Sets A and B (RA and RB) can be addressed explicitly as a register number (R0-RF) or the address can be determined by an Address Code Number (ACN).

The address defined by the ACN value is dependent on the operating mode (VAX or 11 Compatibility). Table 2-17 shows the relationship between the ACN value and the register address selected.

Table 2-17 Scratch Pad Address Code Number (ACN)

ACN Hex Value	VAX Mode		11 Compatibility Mode	
	RA Address	RB Address	RA Address	RB Address
00	SP1 R	SP1 R	SRC R	SRC R
01	SP1 R	SP2 R	DST R	DST R
02	SP2 R	SP1 R	DST R	SCR R
03	PRN	PRN	SRC R	SRC R
04	PRN PLUS 1	PRN PLUS 1	SRC R OR 1	SRC R OR 1
05	SC (03:00)	SC (03:00)	SC (03:00)	SC (03:00)
06	SP1 R PLUS 1	SP1 R PLUS 1	SRC R PLUS 1	SRC R PLUS 1
07	0	0	0	0

In VAX mode, the three register values are determined by SP1 R (Specifier 1 Register), SP2 R (Specifier 2 Register), and PRN (Previous Register Number). SP1 R is the register number of the operand specifier currently being evaluated by the Instruction Buffer control logic. SP2 R is the register number of the operand specifier which follows the specifier currently being evaluated by the Instruction Buffer control logic.

In 11 Compatibility mode, the two register values are determined by SRC R (Source Register) and DST R (Destination Register). The SRC R and DST R numbers are defined by the register field of the instruction.

Note that in both modes, the ACN value may also select SC 03:00 as the address source for the RA and RB sets. This provision allows the general registers to be sequentially indexed.

The RC register write operations are always longword data types. The RA and RB register write operations are context dependent and controlled by the UDT field of the microinstruction as follows:

UDT Field (Hex)	Context
0	Longword (Long, Quad, Floating, or Double Floating)
1	Word
2	Byte
3	Instruction Dependent (Any of the above)

When the UDT field equals 3, the instruction decode logic determines the data type to be used.

Certain USPO values (60:7F) provide individual control of the RC and RA/RB register sets within the same microinstruction. These USPO values allow the RC register to be written while the RA/RB registers are read or vice versa. However, the contents of one register set cannot be interchanged with the contents of the other set in the same microinstruction.

2.6.3.8 Register Log Stack (RLOG) and Program Counter Save Register (PCSV) -- The RLOG stack and PCSV register are used to hold all the information required to restart an instruction.

The PCSV register is loaded with the lower 8 bits of the Program Counter (PC 07:00) at the time an instruction is fetched. From this information, the entire 32-bit starting PC can be reconstructed if a fault occurs. The remaining high order bits are saved in the Instruction Physical Address (IPA) register of the translation buffer. Only the lower 8 bits of the PC need to be held by the PCSV register because no instruction is longer than 256 bytes.

The RLOG stack contains a record of changes made to the Scratch Pad Register Set during the instruction sequence (e.g., autoincrementation or autodecrementation of registers). If an instruction causes a memory management fault requiring a macro level trap, it is necessary to restore the general registers to their original state. The information stored in the RLOG stack allows reconstruction of the register contents so that the instruction can be restored. Each RLOG entry contains an ADD/SUBTRACT bit, the constant value used in the operation, and the address of the register modified. Figure 2-60 shows the data format of each RLOG entry.

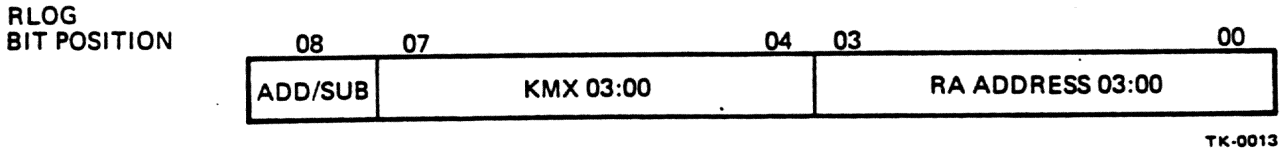


Figure 2-60 RLOG Entry Format

The RLOG stack contains 16 locations. At each instruction fetch, a pointer into the stack is initiated and an RLOG empty flag is asserted. When the microcode fault routine reads the RLOG stack, the pointer is decremented and the next entry in the stack becomes available. The RLOG stack is written when the UALU field of the microinstruction specifies an RLOG update. If the ULAU field equals 6, the operation selected is A PLUS B (RLOG UPDATE) and RLOG bit 8 is set to a 1. If the UALU field equals 1, the operation selected is A MINUS B (RLOG UPDATE) and RLOG bit 8 is set to a Ø.

The RLOG stack is read when the UMSC field of the microinstruction equals 7 (READ RLOG). The current value is read from the stack and the pointer is decremented at the end of the microinstruction.

2.6.4 Address Section

The Address Section of the data path consists primarily of a virtual address register for memory data references, an instruction buffer address register, and the program counter. The address registers can be counted, thereby allowing the addresses to be incremented without implementation of the arithmetic section.

2.6.4.1 Instruction Buffer Address Register (VIBA) -- The VIBA (Figure 2-61) holds the address of the instruction stream data being fetched by the instruction buffer control logic. The lower two bits of the address (VIBA Ø1:ØØ) are retained by the instruction buffer logic. These two bits control the byte rotation of data loaded from memory.

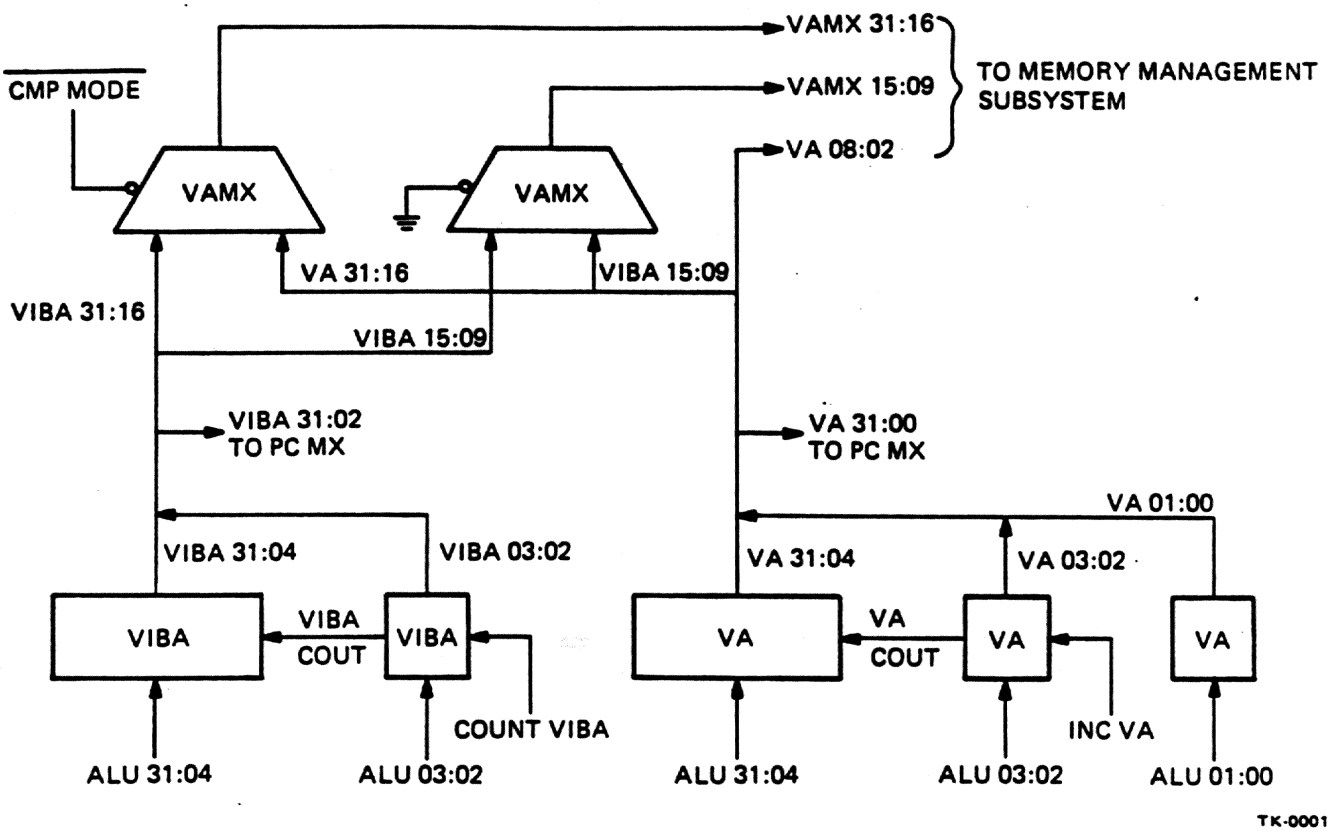


Figure 2-61 Instruction Buffer Address and Virtual Address Registers

The VIBA holds a virtual address which must be converted to a physical memory address by the translation buffer.

The VIBA is controlled by the UIBC field of the microinstruction. When the UIBC field equals 2, the VIBA is loaded with data from the arithmetic section (ALU 31:Ø2). The VIBA is loaded with a new address whenever the instruction execution changes sequence. Sequence changes occur in JUMP or successful BRANCH instructions or in the initialization of a trap or interrupt routine. The instruction buffer control logic will increment the VIBA by four whenever instruction data has been successfully fetched.

2.6.4.2 Virtual Address Register (VA) -- The VA (Figure 2-61) holds the address generated by the microprogram to write or read memory data to or from the data path. The VA will generally contain a virtual address which must be converted to a physical memory address by the translation buffer. However, the VA may hold a physical address which the microprogram generated during the translation process or when the memory management mechanisms have been disabled.

The VA may also be used as an index to the translation buffer when the translation buffer is being updated or invalidated by the microprogram. During the execution of the PROBE instruction, the indexing function is used to determine if an access violation would occur if a memory reference was actually mode to that particular virtual address.

The VA is controlled by the UVAK field of the microinstruction as follows:

UVAK Field (BUS CS 25)	Function
Ø	HOLD
1	LOAD with ALU 31:ØØ (from arithmetic section)

The VA is incremented by four when the UPCK field of the microinstruction equals three. The load function will override the incrementation if both functions are selected simultaneously.

2.6.4.3 Virtual Address Multiplexer (VAMX) -- The VAMX provides an interface to the memory management subsystem (translation buffer and cache). The VAMX is selected to provide the correct format of the virtual address for VAX mode or 11 compatibility mode. The VA or VIBA can be selected as the source for address bits 31:09. Address bits 08:02 are always derived from the VA and are not input to the VAMX. Address bits Ø1:ØØ are not sent to the memory management subsystem since all memory references are made on longword boundaries and the lower two address bits specify only the byte location within the longword.

The state of the Compatibility mode bit in the Processor Status Longword (PSL) determines which address format is selected in the VMAX.

The address source selection is provided by a signal from the translation buffer. The VA is selected as the address source when the microprogram requests a memory data reference. The VIBA is selected as the address source when the microprogram requests instruction stream data and the Instruction Buffer Control Logic is allowed to use the cycle to request a memory transfer. Table 2-18 shows the address format for each mode and source selected.

2.6.4.4 Program Counter (PC) -- The PC holds the address of the instruction op code each time a new execution sequence is started. The PC is incremented by an appropriate value as the operand specifiers and the instruction are evaluated. The data source of PC bits 31:04 is either the VIBA or VA (via the PCMX). The data source of PC bits 03:00 is either the VIBA, VA, or PC ADDER (via the PCAMX).

Refer to Figure 2-62. The PC ADDER allows the PC to be incremented by 1, 2, 4, or N. The instruction buffer control logic generates the value N, incrementing the PC to point beyond instruction stream bytes being evaluated.

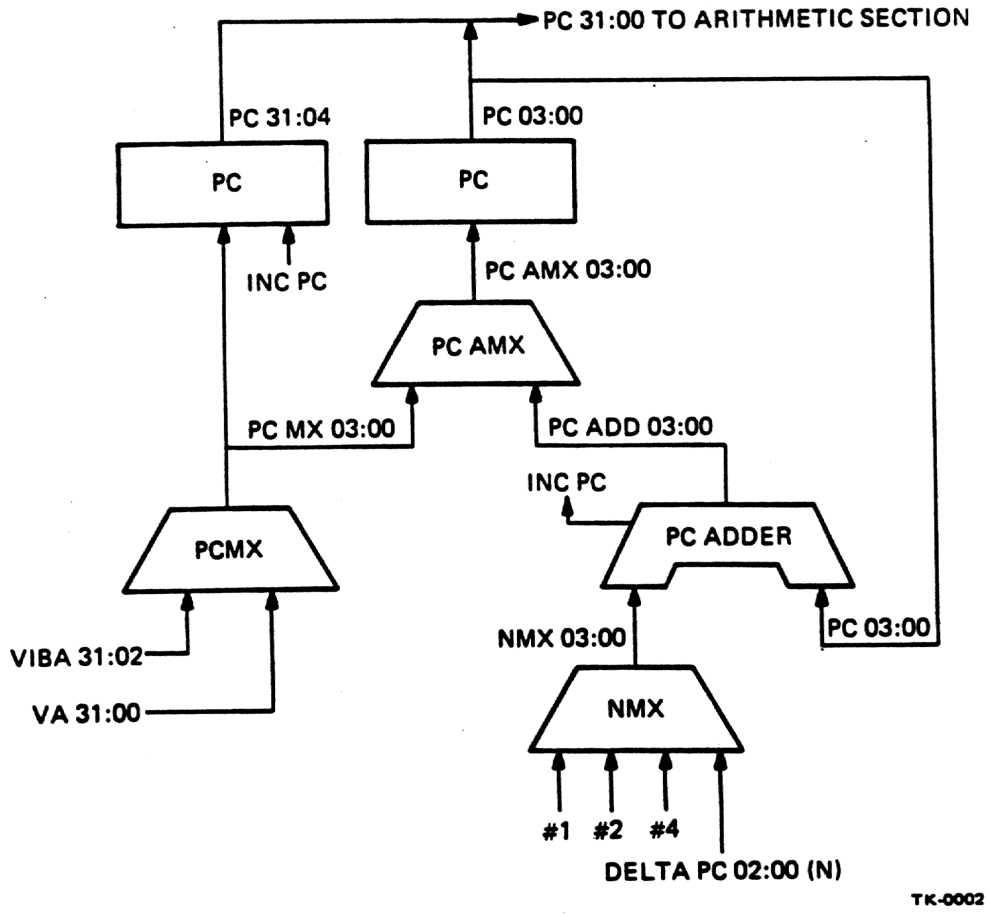


Figure 2-62 Program Counter (PC) Register

Table 2-18 VAMX Data Format

MODE	ADDRESS SOURCE	VIRTUAL ADDRESS FORMAT	
		VAMX OUTPUT	ADDRESS BITS 08:02
		31 09	08 02
VAX	VA	VA 31:09	VA 08:02
		31 09	08 02
VAX	VIBA	VIBA 31:09	VA 08:02
		31 16 15 09	08 02
11 COMPATIBILITY	VA	00 VA 16:09	VA 08:02
		31 16 15 09	08 02
11 COMPATIBILITY	VIBA	00 VA 16:09	VA 08:02

The PC input selection is controlled by the UPCK field of the microinstruction.

Hex	UPCK Field BUS CS 34	BUS CS 33	BUS CS 32	Function Selected
Ø	Ø	Ø	Ø	NO-OP
1	Ø	Ø	1	VA input to PC
2	Ø	1	Ø	VIBA input to PC
3	Ø	1	1	Increment VA by 4
4	1	Ø	Ø	Increment PC by 1
5	1	Ø	1	Increment PC by 2
6	1	1	Ø	Increment PC by 4
7	1	1	1	Increment PC by N

2.6.4.5 Program Counter Adder (PCADD) and Number Multiplexer (NMX) -- The Program Counter Adder performs the function PC Ø3:ØØ PLUS NMX Ø3:ØØ. The numbers 1, 2, 4, or N are selected by the NMX to provide the proper increment value. The value N, generated by the instruction buffer decode logic, may be 1, 2, 3, or 5. The output bits of the PC ADDER (PCADD Ø3:ØØ) are multiplexed by the PCAMX which provides the input to PC bits Ø3:ØØ. If the PC ADD function results in a carry, the upper 28 bits of the program counter are incremented. The increment value selected by the NMX is controlled by the UPCK field of the microinstruction (refer to Paragraph 2.6.4.4).

2.6.4.6 PC Multiplexer (PCMX) and PC Adder Multiplexer (PCAMX) -- The PCMX and PCAMX provide the data input to the Program Counter. When the PC is initially loaded, the PCMX selects the VA or IBA as the input source and the PCAMX selects PCMX Ø3:ØØ. Therefore, 32 bits of the PCMX (VA 31:ØØ or IBA 31:Ø2) are input to the PC at the beginning of the instruction sequence.

When the PC is incremented, the PCAMX selects PCADD Ø3:ØØ as the data source for the lower four bits of the PC and the input to the upper 28 bits is disabled. The value of the upper 28 bits remains unchanged unless the PC ADD function results in a carry.

2.6.5 Data Section

The data section (Figures 2-64 and 2-72) provides the interface for the transfer of data to and from the Memory Data (MD) bus and Internal Data (ID) bus and also performs the shifting, byte alignment and upacking of floating-point data types.

2.6.5.1 Data/Arithmetic Section Interface -- Data transfer between the arithmetic and data sections is performed through the DFMX, RAMX, and RBMX. Refer to Figure 2-64.

Data Format Multiplexer (DFMX) -- The DFMX allows data from the Shifter (SHF) to be transferred in either integer or unpacked floating-point format. If a floating-point instruction is being executed, the Shifter data will be in the packed floating-point format, as shown in Figure 2-63.

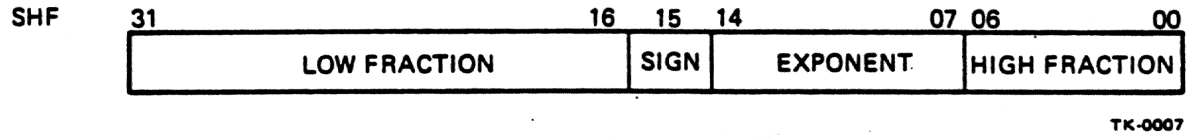


Figure 2-63 Shifter Data in Packed Floating Point Format

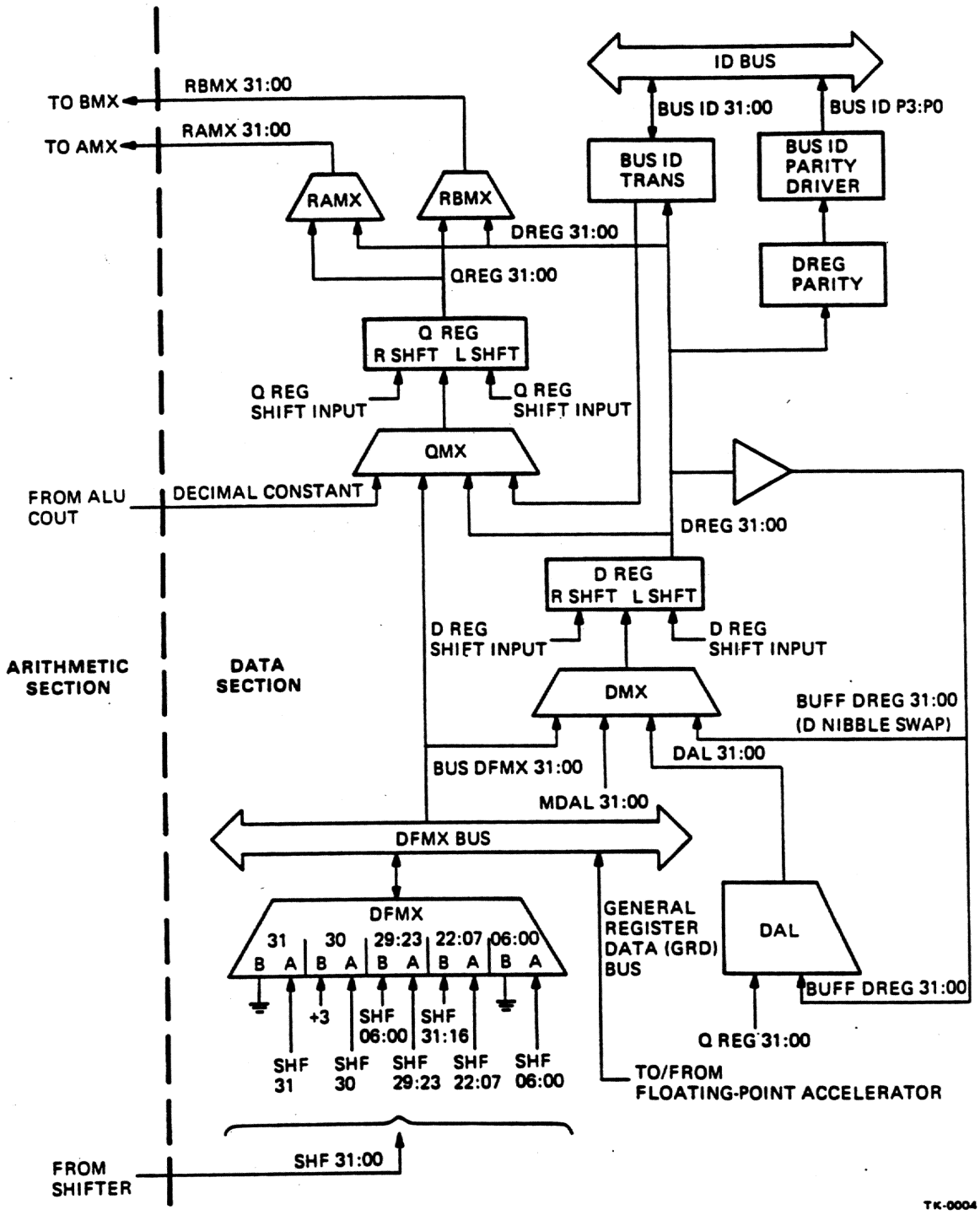


Figure 2-64 Data Section (Arithmetic Section Interface, Q and D Registers, Data Aligner)

The DFMX unpacks the floating-point format by reassembling the fraction and removing the sign and exponent bits (Figure 2-65).

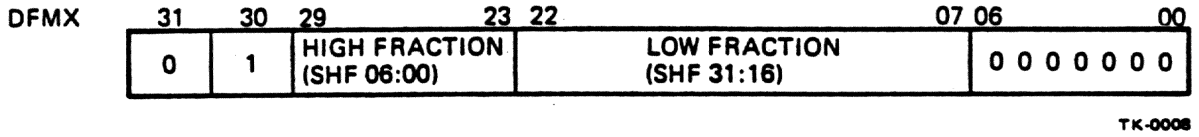


Figure 2-65 Unpacked Floating-point Format

If the integer format is selected, the Shifter data (SHF 31:00) is transferred through the DFMX unmodified. Selection of the DFMX' format is controlled by the UDK and UQK fields of the microinstruction. If either field equals 8, integer format is selected. If either field equals 9, unpacked floating-point format is selected.

If the UDK field equals A or the UQK field equals B, data from the Floating-point Accelerator (FPA) is transferred to the DMX or QMX via the DFMX bus.

Register AMX (RAMX) and Register BMX (RBMX) Multiplexers -- The RAMX allows selection of either the D register or the Q register to be transferred to the A input multiplexer of the arithmetic section. The RBMX allows selection of either the D or Q register to be transferred to the B input multiplexer.

2.6.5.2 **Holding Register and Data Aligner** -- The holding registers provide temporary storage for data generated in other sections and the data aligner performs the required shifting for certain operations. Refer to Figure 2-64.

Q Register -- The Q register is used for the following purposes:

During the execution of field and double floating-point instruction, the Q register is used in conjunction with the D register to hold data types larger than 32 bits.

In the execution of multiply and divide instructions, the Q register stores the multiplier and quotient bits.

During the evaluation of instruction operands, the Q register stores the first operand while the subsequent operand is being evaluated.

The Q register is capable of a right or left shift by one bit. The UQK field of the microinstruction determines if the Q register is to be loaded with the contents of the Q register multiplexer (QMX) or if the current contents of the register are to be shifted. The UQK field also allows the register to be shifted twice to the right or left. Table 2-19 shows the relationship between the UQK field value and register control.

If the function selected is a register shift, the value shifted into the vacant bit positions is determined by the Shift Input (USI) field of the microinstruction.

USI Field	Q Register Shift Input
Ø	ALU CARRY 31
1	Q Register (Bit 31)
2	D Register (Bit 31)
3	Ø
4	Ø
5	ALU CARRY 31
6	Ø
7	1

If the UQK field equals 8-F, the Q register is loaded with the output of the QMX. The data selected by the QMX is determined by the UQK value.

Q Register Multiplexer (QMX) -- The QMX provides the data source for the Q register. The QMX allows selection of either the DFMX, D register, Internal Data (ID) bus, or a decimal constant (6) in each nibble of the multiplexer. The UQK field of the microinstruction controls the QMX selection. Refer to Table 2-20.

Table 2-19 Q Register Control

UQK FIELD					Q REGISTER DATA FORMAT	FUNCTION SELECTED
HEX	BUS CS S4	S3	S2	S1		
0	0	0	0	0	31 … 00: Q 31:00	HOLD
1	0	0	0	1	31 … 02: Q 29:00; 01: USI; 00: USI	DOUBLE SHIFT LEFT
2	0	0	1	0	31: USI; 30: USI; 29 … 00: Q 31:02	DOUBLE SHIFT RIGHT
3,4						RESERVED
5	0	1	0	1	31 … 01: Q 30:00; 00: USI	SINGLE SHIFT LEFT
6	0	1	1	0	31: USI; 30 … 00: Q 31:01	SINGLE SHIFT RIGHT
7	0	1	1	1		RESERVED
8-F					31 … 00: QMX 31:00	LOAD QMX DATA

Table 2-20 QMX Selection

UQK FIELD					QMX DATA SELECTED
HEX	BUS CS 54	53	52	51	
8	1	0	0	0	31 … 00 — DFMX 31:00 (INTEGER FORMAT)
9	1	0	0	1	31 … 00 — DFMX 31:00 (UNPACKED FORMAT)
A*	1	0	1	0	31 28 27 24 23 20 19 16 15 12 11 08 07 04 03 00 — 0 1 1 0 | 0 1 1 0 | 0 1 1 0 | 0 1 1 0 | 0 1 1 0 | 0 1 1 0 | 0 1 1 0 | 0 1 1 0
B	1	0	1	1	31 … 00 — FLOATING-POINT ACCELERATOR DATA
C	1	1	0	0	31 … 00 — D 31:00
D	1	1	0	1	31 … 00 — RESERVED
E	1	1	1	0	31 … 00 — BUS ID D31:00
F	1	1	1	1	31 … 00 — ZEROES

*DECIMAL 6 IS INPUT TO EACH QMX BYTE ONLY IF AN ALU CARRY IS GENERATED

The constant input to the QMX (6_{10}) is required when decimal arithmetic is being performed. The following example demonstrates the need for the constant input.

Example 1

	Decimal		Binary
	9_{10}		1001
plus	1_{10}	plus	0001
	10_{10}		1010

1010 is not the correct BCD equivalent of 10_{10}. Two 4 bit nibbles are required to represent a two digit decimal number. To accomplish this, the first binary number is adjusted (by adding a decimal 6 or binary 0110) before being added to the second binary number, shown as follows:

	9_{10}		1001	
plus	1_{10}	plus	0110	(decimal 6)
	10_{10}		1111	
		plus	0001	
			0001 0000	(decimal 10)

The constant 6 is used to provide the necessary carry in decimal addition and the necessary borrow in decimal subtraction, demonstrated in the following examples.

Example 2 Decimal Addition

Decimal		Binary Representation		
14_{10}		0001	0100	(14_{10})
plus 7_{10}	plus	0110	0110	(decimal 6 added to both nibbles)
21_{10}		0111	1010	(adjusted binary values)
	plus	0000	0111	(07_{10})
	nc	1000 c	0001	(c indicates carry generated, nc indicates no carry)
		1000	0001	To obtain the proper
	minus	0110	0000	binary representation,
21_{10}		0010	0001	decimal 6 is subtracted from a nibble if there was no carry generated in that nibble addition. If a carry was generated, zero is subtracted.

Example 3 Decimal Subtraction

	Decimal		Binary Representation		
	14_{10}		0001	0100	(14_{10})
minus	7_{10}	minus	0000	0111	(07_{10})
	$\underline{\quad 10}$				
	7_{10}	nb	00001 b	1101	(b indicates borrow required, nb indicates no borrow)
			0000	1101	To obtain the proper
		minus	0000	0110	binary representation,
					decimal 6 is subtracted
	7_{10}		0000	0111	from each nibble that required a borrow. If a borrow was not required, zero is subtracted.

D Register -- The D register is used for the following purposes:

Provide temporary storage for data received from the Memory Data (MD) bus.

Provide temporary storage for data to be transmitted to the Internal Data (ID) bus. When the D register is used for ID bus write operations, odd parity is generated for each byte of data. Parity is not checked on data received from the ID bus.

When used in conjuction with the Q register, the D register holds data types larger than 32 bits.

The UDK field of the microinstruction determines if the D register is to be shifted to the right or left (by one or two bits) or if the D register is to be loaded with the output of the D Register Multiplexer (DMX). Table 2-21 shows the relationship between the UDK field value and D register control.

If the UDK field specifies a D register shift, the value shifted into the vacant bit positions is determined by the Shift Input (USI) field of the microinstruction.

USI Field	D Register Shift Input
0	Q Register (Bit 31)
1	Q Register (Bit 00)
2	0
3	0
4	0
5	Q Register (Bit 31)
6	ALU 01/ALU 00
7	ALU 01/ALU 00

If the USI field equals 6 or 7 and the D register is selected for a double shift, ALU bit ØØ is input to the D register on the first shift and ALU bit Ø1 is input on the second shift. If a single shift has been selected and USI equals 6 or 7, ALU bit Ø1 will be shifted in.

The D register is loaded with the output of the DMX when the USK field equals 8-F. The data selected by the DMX is determined by the UDK value.

D Register Multiplexer (DMX) -- The DMX provides the data source for the D register. The DMX allows selection of either memory data through the memory data aligner (MDAL), the DFMX, buffered D register for the D nibble swap function, or the data aligner (DAL). The selection of the source is determined by the value of the UDK field of the microinstruction. Refer to Table 2-22.

Table 2-21 D Register Control

UDK FIELD					D REGISTER DATA FORMAT	FUNCTION SELECTED
HEX	BUS CS 91	90	89	88		
0	0	0	0	0	31 … 00: D 31:00	HOLD
1	0	0	0	1	31 … 02: D 29:00; 01: USI; 00: USI	DOUBLE SHIFT LEFT
2	0	0	1	0	31: USI; 30: USI; 29 … 00: D 31:02	DOUBLE SHIFT RIGHT
3	0	0	1	1		RESERVED
4	0	1	0	0	31 … 01: D 30:00; 00: USI	SINGLE SHIFT LEFT*
5	0	1	0	1	31 … 01: D 30:00; 00: USI	SINGLE SHIFT LEFT
6	0	1	1	0	31: USI; 30 … 00: D 31:01	SINGLE SHIFT RIGHT
7	0	1	1	1		RESERVED
8-F					31 … 00: DMX 31:00	LOAD DMX DATA

*SINGLE SHIFT LEFT IS THE FUNCTION SELECTED IF ALU CARRY 31 IS GENERATED. IF THERE IS NO ALU CARRY, THE DMX OUTPUT (DFMX DATA IN INTEGER FORM) IS LOADED.

Table 2-22 DMX Selection

UDK FIELD HEX	UDK FIELD BUS CS 91	90	89	88	DMX DATA SELECTED
0	0	0	0	0	31 … 00: MDAL 31:00 (NOTE 1)
8	1	0	0	0	31 … 00: DFMX 31:00 (INTEGER FORMAT)
9	1	0	0	1	31 … 00: DFMX 31:00 (UNPACKED FORMAT)
A	1	0	1	0	31 … 00: FLOATING-POINT ACCELERATOR DATA
B	1	0	1	1	31 … 24: BUFF DREG 07:00; 23 … 16: BUFF DREG 15:03; 15 … 08: BUFF DREG 23:16; 07 … 00: BUFF DREG 31:24
C	1	1	0	0	31 … 00: DAL 31:00 (Q REG 31:00; NOTE 2)
D	1	1	0	1	31 … 00: DAL 31:00 (SHIFTED BY SC 09, 04:00; NOTE 2)
E	1	1	1	0	31 … 00: DAL 31:00 (SHIFTED BY SHF VAL; NOTE 2)
F	1	1	1	1	31 … 00: ZEROES

Note 1: When the UDK field equals 0 and the signal MD TO D is generated by the SBI control board, the DMX selects the Memory Data aligner as the data source for the D register.

Note 2: When the UDK field equals C, the DAL performs a data shift of 32 bits which results in Q register bits 31:00 on its output.

When the UDK field equals D, the DAL shift is determined by Shift Count (SC) register bits 09 and 04:00.

When the UDK field equals E, the DAL shift is determined by a number (NORM SHF VAL) generated to normalize the fraction in floating-point instructions.

If the UDK field equals B, the DMX selects D register data which has been buffered and reformatted. This function is referred to as decimal nibbles swap and is required to perform decimal arithmetic. The bytes of data are swapped as shown in Figure 2-66.

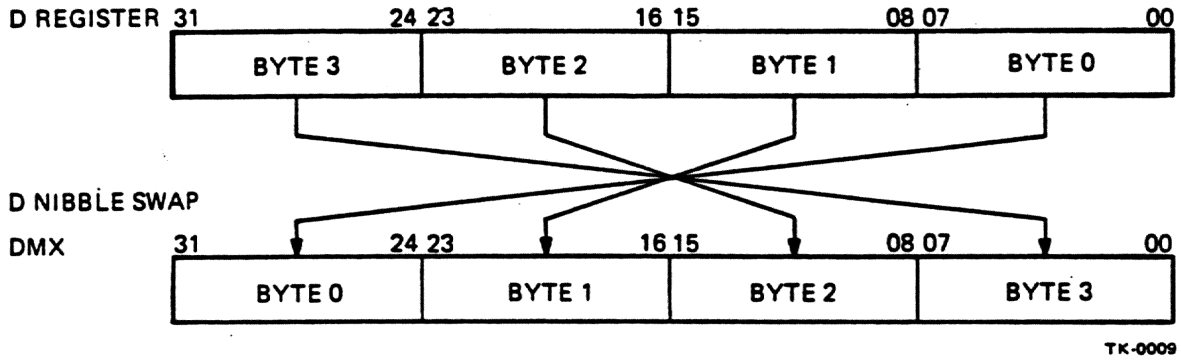


Figure 2-66 Decimal Nibble Swap

A decimal nibbles swap is required prior to performing decimal arithmetic to compensate for the format in which decimal numbers are stored in memory. For example, the decimal number +12345678 would be in consecutive memory byte locations as shown in Figure 2-67.

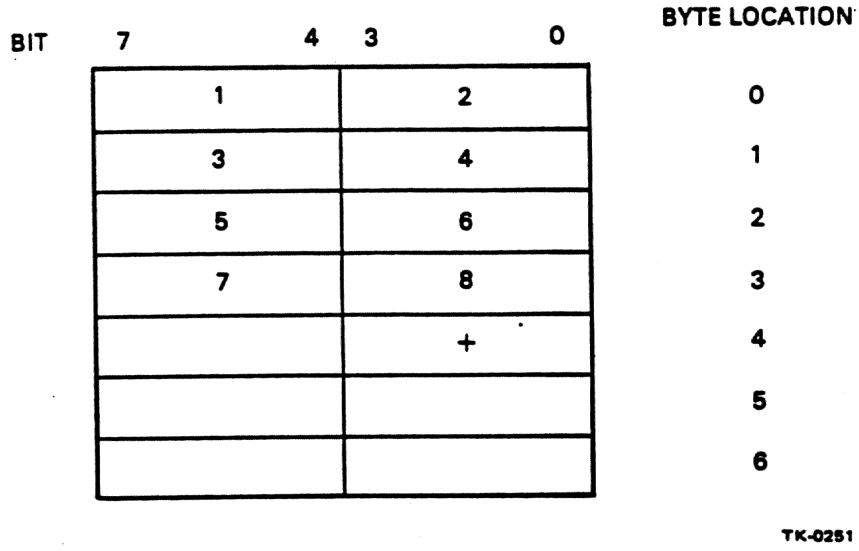


Figure 2-67 Memory Storage of a Decimal Number

If the longword containing this decimal number was accessed from memory, it would be loaded into the D register in the format shown in Figure 2-68.

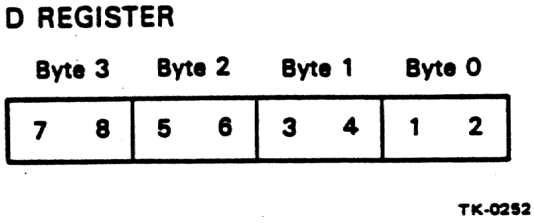


Figure 2-68 Format of a Decimal Number Loaded from Memory

The data must be swapped and loaded into the DMX in the format shown in Figure 2-69 which correctly represents the decimal number.

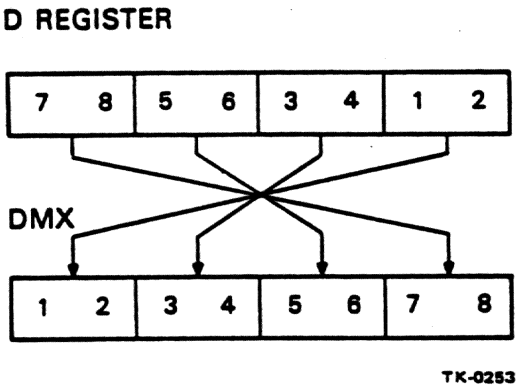


Figure 2-69 Format of Swapped Decimal Number

Data aligner (DAL) -- The DAL performs the shifting of D register contents to the right or left by a maximum of 32 bits in each direction. The contents of the Q register are shifted into the vacant bit positions as shown in Figure 2-70.

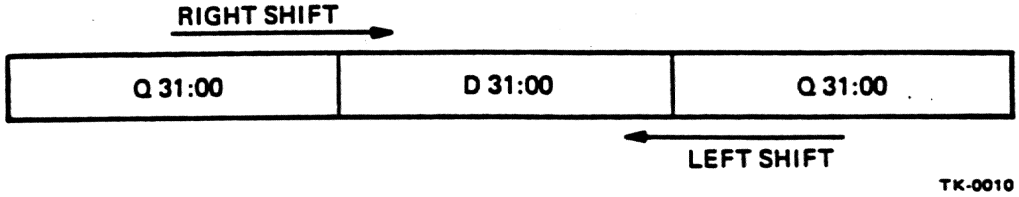


Figure 2-70 DAL Shift Format

The DAL is used during the execution of bit field, multiply, divide, shift, decimal arithmetic, and floating-point instructions. The shift operation is also used to isolate bit fields in the virtual to physical address translation.

The DAL actually consists of three levels of shifters, with the output of the lower levels providing the source for the inputs of the next higher level (refer to Table 2-23 and Figure 2-71). The result is an adding effect of the shift performed at each level. Level 1 of the DAL allows left shifting by 0, 16, 32, or 48 bits. A left shift by 32 has the same effect as shifting right by 32 bits and a left shift by 48 has the same effect as a right shift by 16. Level 2 performs a left shift by 0, 4, 8, or 12 bits and the final DAL level performs a left shift by 0, 1, 2, or 3 bits.

Table 2-23 Data Aligner Operation

DAL Level 1 (DALL 1) Outputs

DATA ALIGNER Selection			DREG 31:16	DREG 15:00	QREG 31:16	Level 1 Function Selected
DALK 5	DALK 4					
0	0		D 31:16	D 15:00	Q 31:16	Shift by 0
0	1	Data Selected	D 15:00	Q 31:16	Q 15:00	Left Shift by 16
1	0		Q 31:16	Q 15:00	D 31:16	Left Shift by 32 (effectively Right Shift by 3
1	1		Q 15:00	D 31:16	D 15:00	Left Shift by 48 (effectively Right Shift by 1

DAL Level 2 (DALL 2) Outputs

			DREG				DREG				QREG	
		31:28	27:24	23:20	19:16	15:12	11:08	07:04	03:00	31:29	Level 2 Function Selected
DALK 3	DALK 2										
0	0	D 31:28	D 27:24	D 23:20	D 19:16	D 15:12	D 11:08	D 07:04	D 03:00	Q 31:29	Shift by 0
0	1	D 27:24	D 23:20	D 19:16	D 15:12	D 11:08	D 07:04	D 03:00	Q 31:28	Q 27:25	Left Shift by 4
1	0	D 23:20	D 19:16	D 15:12	D 11:08	D 07:04	D 03:00	Q 31:28	Q 27:24	Q 23:21	Left Shift by 8
1	1	D 19:16	D 15:12	D 11:08	D 07:04	D 03:00	Q 31:28	Q 27:24	Q 23:20	Q 19:20	Left Shift by 12

Data Selected

DAL Outputs

			DAL				DAL						DAL Function Selected
		31:28	27:24	23:20	19:16	15:12	11:08	07:04	03	02	01	00	
DALK 1	DALK 0												
0	0	D 31:28	D 27:24	D 23:20	D 19:16	D 15:12	D 11:08	D 07:04	D 03	D 02	D 01	D 00	Shift by 0
0	1	D 30:27	D 26:23	D 22:19	D 18:15	D 14:11	D 10:07	D 06:03	D 02	D 01	D 00	Q 31	Left Shift by 1
1	0	D 29:26	D 25:22	D 21:18	D 17:14	D 13:10	D 09:06	D 05:02	D 01	D 00	Q 31	Q 30	Left Shift by 2
1	1	D 28:25	D 24:21	D 20:17	D 16:13	D 12:09	D 08:05	D 04:01	D 00	Q 31	Q 30	Q 29	Left Shift by 3

Data Selected

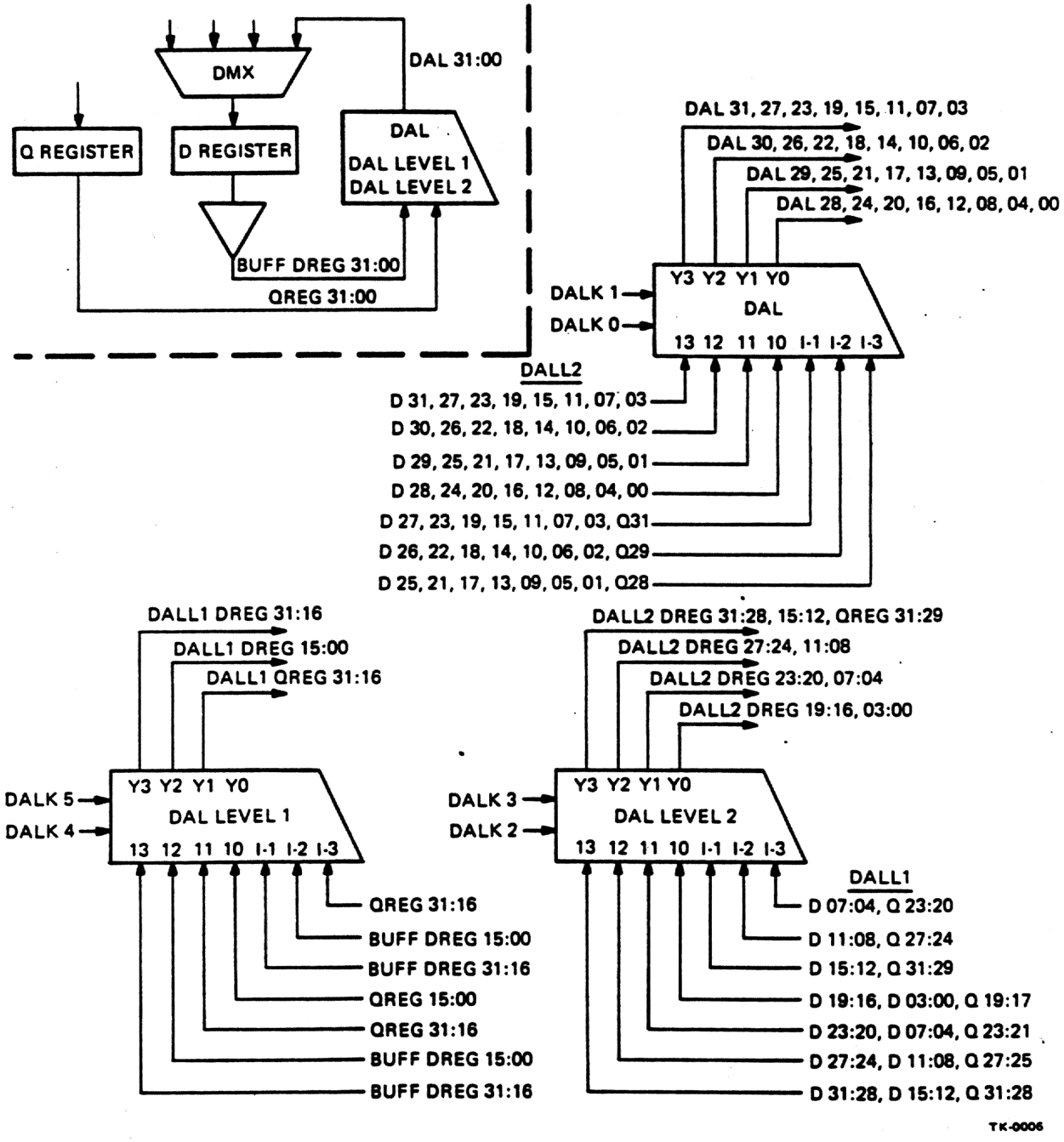


Figure 2-71 Data Aligner

The adding effect of the shift levels of the DAL is demonstrated as follows:

Example 1

Level 1		Left Shift by 16
Level 2	plus	Left Shift by 12
DAL	plus	Left Shift by 3
DAL Output	=	Left Shift by 31

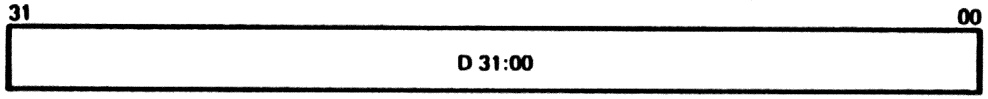
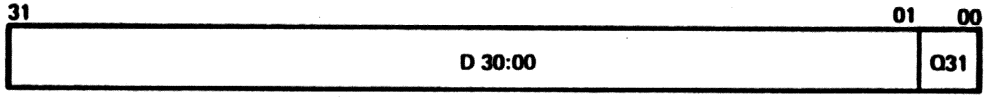
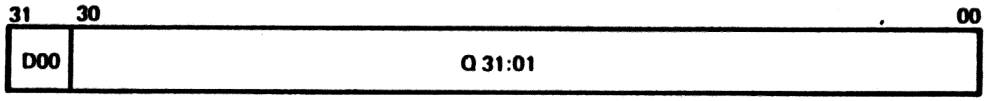
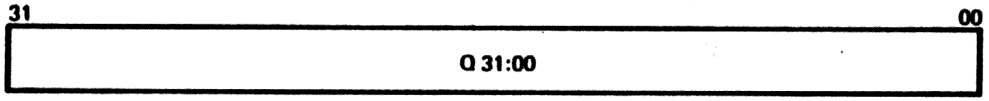
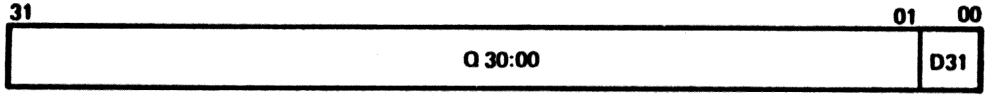
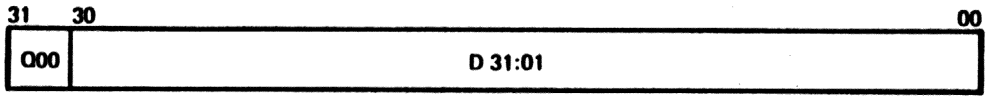
Example 2

Level 1		Right Shift by 16
Level 2	plus	Left Shift by 12
DAL	plus	Left Shift by 3
DAL Output	=	Right Shift by 1

The shift value selected by the DAL is determined by the Shift Count register (SC09, 04:00) or by a number (NORM SHF VAL) generated to normalize the fraction in floating-point instructions. The UDK field of the microinstruction (refer to Table 2-22) selects the source of the DAL control.

Table 2-24 shows the relationship of the level selection and DAL output for the range of shift values available.

Table 2-24 DAL Shift Range

DALK SELECT 5 4 3 2 1 0	DAL OUTPUT	SHIFT VALUE SELECTED
0 0 0 0 0 0		SHIFT BY 0
0 0 0 0 0 1		LEFT SHIFT BY 1
0 1 1 1 1 1		LEFT SHIFT BY 31
1 0 0 0 0 0		LEFT (OR RIGHT) SHIFT BY 32
1 0 0 0 0 1		RIGHT SHIFT BY 31
1 1 1 1 1 1		RIGHT SHIFT BY 1

2.6.5.3 Memory Data Interface -- Data from memory is transferred to and from the data section via the Memory Data (MD) bus. Data in memory must be read or written on longword boundaries. However, references to memory locations by the data section are made on byte boundaries. Therefore, transfer of information between the data section and memory requires that the bytes of data be properly aligned. The required alignment is provided by the Memory Data Aligner (MDAL), Byte Aligner (BAL), BUS MD Parity Aligner, and BUS MD Mask generator. Refer to Figure 2-72.

Memory Data Aligner (MDAL) -- If a memory read operation is specified, the D register is loaded with the data on the MD bus via the MDAL and D register multiplexer (DMX). The MDAL reformats the data from the MD bus before it is transferred to the DMX. Data on the MD bus is longword aligned and must be shifted according to the byte address on which the memory reference was made. The shifting of MD data types is controlled by the value of the low two bits (VA01, VA00) of the memory reference address. These two bits specify a byte location in the longword. The MD data is shifted by the MDAL as shown:

VA01	VA00	Memory Data
0	0	Byte 3, Byte 2, Byte 1, Byte 0
0	1	Byte 0, Byte 3, Byte 2, Byte 1
1	0	Byte 1, Byte 0, Byte 3, Byte 2
1	1	Byte 2, Byte 1, Byte 0, Byte 3

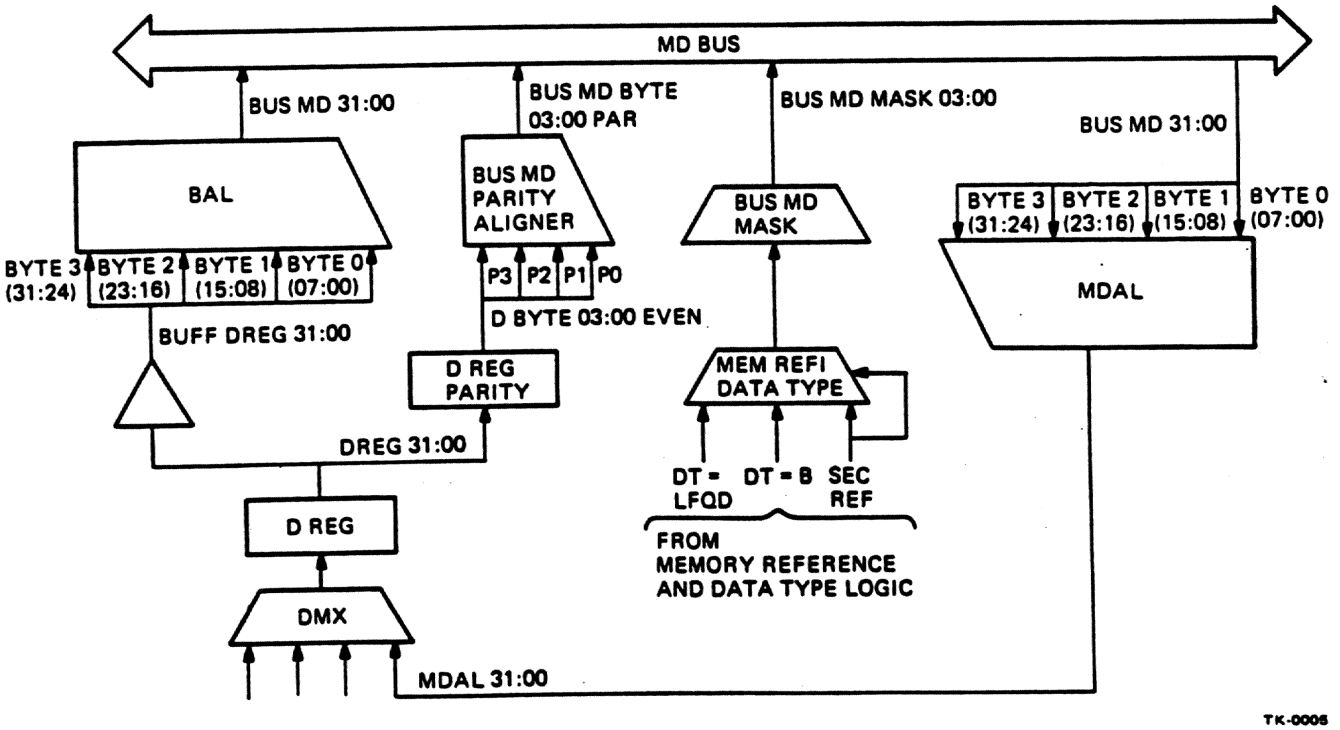


Figure 2-72 Data Section (Memory Data Bus Interface)

As an example of memory data alignment, assume the central processor references memory address 202. The data transferred over the MD bus will begin on longword boundary 200.

	Memory Data			
	Byte 3	Byte 2	Byte 1	Byte 0
Addresses of data transferred over the MD bus	203	202	201	200

The low two bits of the memory reference address (202) are 1, 0. The MDAL shifts the bytes into a format which will load data from memory location 202 into the first byte of the D register.

	D Register			
	Byte 3	Byte 2	Byte 1	Byte 0
Addresses of data loaded into the D register	201	200	203	202

If the instruction specifies a byte data type, only byte 0 of the D register would contain useful data after the memory read. The RAMX or RBMX would zero or sign extend the D register before it is transferred to the arithmetic section. If a word data type was specified, bytes 0 and 1 would contain valid data.

A second memory reference is required if a longword data type was specified in this example. Since the referenced longword begins on byte boundary 202, only half of the data could be loaded on the first fetch. Memory location 206 (202 plus 4) is referenced on the second fetch. The data transferred over the MD bus on the second reference would begin on longword boundary 204.

	Memory Data			
	Byte 3	Byte 2	Byte 1	Byte 0
Addresses of data transferred over the MD bus	207	206	205	204

The low two bits of the memory reference address (206) are still 1, 0. The MDAL will shift the data so that memory byte 204 will be loaded into byte 2 of the D register.

	D Register			
	Byte 3	Byte 2	Byte 1	Byte 0
Addresses of data loaded into the D register	205	204	207	206

The D Register Write Enable logic prevents bytes 1 and 0 of the D register from being written over on the second read operation. Only bytes 2 and 3 will be enabled. The two memory references would result in data being loaded into the D register as shown.

	D Register			
	Byte 3	Byte 2	Byte 1	Byte Ø
Addresses of data loaded into the D register	205	204	203	202

The D Register Write Enable logic controls the loading of the D register on a per byte basis. All bytes are written on the first memory reference. If a second reference is required, the enabling of the D register bytes is dependent on the data type of the reference and the low two bits of the byte address (VAØ1, VAØØ). Table 2-25 shows which operations require a second reference and the D register bytes which are enabled.

Table 2-25 D Register Write Enable

Data Type	VAØ1	VAØØ	Second Reference Required	D Register Byte Enable			
				Byte 3	Byte 2	Byte 1	Byte Ø
Byte	Ø	Ø	no	1	1	1	1
	Ø	1	no	1	1	1	1
	1	Ø	no	1	1	1	1
	1	1	no	1	1	1	1
Word	Ø	Ø	no	1	1	1	1
	Ø	1	no	1	1	1	1
	1	Ø	no	1	1	1	1
	1	1	yes	Ø	Ø	1	Ø
Long-word	Ø	Ø	no	1	1	1	1
	Ø	1	yes	1	Ø	Ø	Ø
	1	Ø	yes	1	1	Ø	Ø
	1	1	yes	1	1	1	Ø

Byte Aligner (BAL) -- If a memory write operation is specified, data from the D register is transferred to the MD bus through the BAL. The BAL functions identically to MDAL but in the reverse direction. The D register contents are reformatted by the BAL so that the bytes of data are loaded into the correct memory byte address. The shifting of D register data is controlled by the value of the lower two bits of the memory reference address (VAØ1, VAØØ)

The D register data is shifted by the BAL as shown:

VAØ1	VAØØ	D Register Data
Ø	Ø	Byte 3, Byte 2, Byte 1, Byte Ø
Ø	1	Byte 2, Byte 1, Byte Ø, Byte 3
1	Ø	Byte 1, Byte Ø, Byte 3, Byte 2
1	1	Byte Ø, Byte 3, Byte 2, Byte 1

BUS MD Parity Aligner -- The D register parity generator provides one parity bit for each byte of data. Odd parity is generated. The parity bits are transmitted with the D register data over the ID bus and MD bus. When parity is transmitted over the MD bus, each parity bit must be aligned with its associated data byte. The BUS MD Parity Aligner provides the required rotation of the parity bits. The lower two bits of the memory reference address (VA01, VA00) determine the bit rotation as shown:

VA01	VA00	D Register Parity
0	0	P3, P2, P1, P0
0	1	P2, P1, P0, P3
1	0	P1, P0, P3, P2
1	1	P0, P3, P2, P1

BUS MD Mask Generator -- During memory read or write operations, a byte mask field is generated by the data section and transmitted over the MD bus. In a read operation, the mask field specifies which byte or bytes of a longword should be checked by the memory controller for data integrity. In a write operation, the Mask field allows the writing of an individual byte or bytes of a memory longword. If a second memory reference is required due to the starting memory address and data type, a second mask field is generated. Table 2-26 shows the BUS MD Mask generated for each memory address and data type. If a second reference is required, the associated mask field is also shown.

Table 2-26 BUS MD Byte Mask

Data Type	VA01	VA00	Second Reference Required	Mask #1/ Mask #2	BUS 3	MD 2	BYTE 1	MASK 0
Byte	0	0	no	1	0	0	0	1
	0	1	no	1	0	0	1	0
	1	0	no	1	0	1	0	0
	1	1	no	1	1	0	0	0
Word	0	0	no	1	0	0	1	1
	0	1	no	1	0	1	1	0
	1	0	no	1	1	1	0	0
	1	1	yes	1	1	0	0	0
				2	0	0	0	1
Long-word	0	0	no	1	1	1	1	1
	0	1	yes	1	1	1	1	0
				2	0	0	0	1
	1	0	yes	1	1	1	0	0
				2	0	0	1	1
	1	1	yes	1	1	0	0	0
				2	0	1	1	1

2.6.6 Exponent Section

The exponent section of the data path processes the exponent value of floating-point numbers. Exponent processing is performed in parallel with the fraction processing performed in the arithmetic and data sections. The 10-bit data path of this section consists of an 8-bit exponent and 2-bit overflow/underflow code.

A packed floating-point number is formed in the arithmetic section via the BMX (ALU B-Input MUX). The exponent is taken from the EALU of the exponent section, the fraction is taken from the D register of the data section, and the sign of the destination fraction (SD) is determined by the USGN field of the microinstruction. The BMX formats the floating-point number as shown in Figure 2-73.

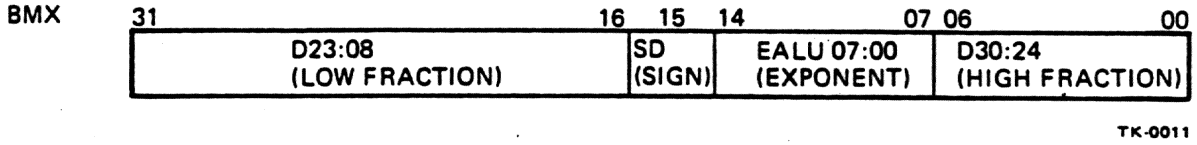


Figure 2-73 BMX Data in Packed Floating-Point Format

The exponent is transferred back to the Shift Count Mux (SMX) of the exponent section via the ALU (bits 14:07). The entire floating-point number is transferred through the ALU and Shifter (SHF) of the arithmetic section and into the Data Format Mux (DFMX) of the data section. The DFMX unpacks the floating-point number and reassembles the fraction for processing.

The exponent section of the data path is also used to generate values for the Shift Count (SC) register. The SC register is implemented in both the arithmetic and data sections for various functions.

2.6.6.1 Exponent Arithmetic Logic Unit (EALU) -- The EALU is the processing unit of the exponent section. The function performed by the EALU is selected by the UEALU field of the microinstruction (refer to Table 2-27).

The EALU output is fed into a negative absolute value (NABSV) ROM and an exponent multiplexer (EXP MUX). Refer to Figure 2-74. The EXP MUX selects the NABSV ROM input when the UALU field equals 7. The ROM provides the required shift value for floating-point arithmetic alignment.

Table 2-27 EALU Function Selection

UALU Field (Hex)	BUS CS 15	14	13	EALU Select S3	S2	S1	S0	MODE 1=H=Logic 0=L=Arith.	C Input 1=L= with carry	ELAU Function
0	0	0	0	1	1	1	1	1	1	A
1	0	0	1	1	1	1	0	1	1	A + B
2	0	1	0	1	0	1	1	1	1	AB
3	0	1	1	1	0	1	0	1	1	B
4	1	0	0	1	0	0	1	0	0	A plus B
5	1	0	1	0	1	1	0	0	1	A minus B
6	1	1	0	0	0	0	0	0	1	A plus 1
7	1	1	1	0	1	1	0	0	1	A minus B (EXPMUX selects NABSV ROM)

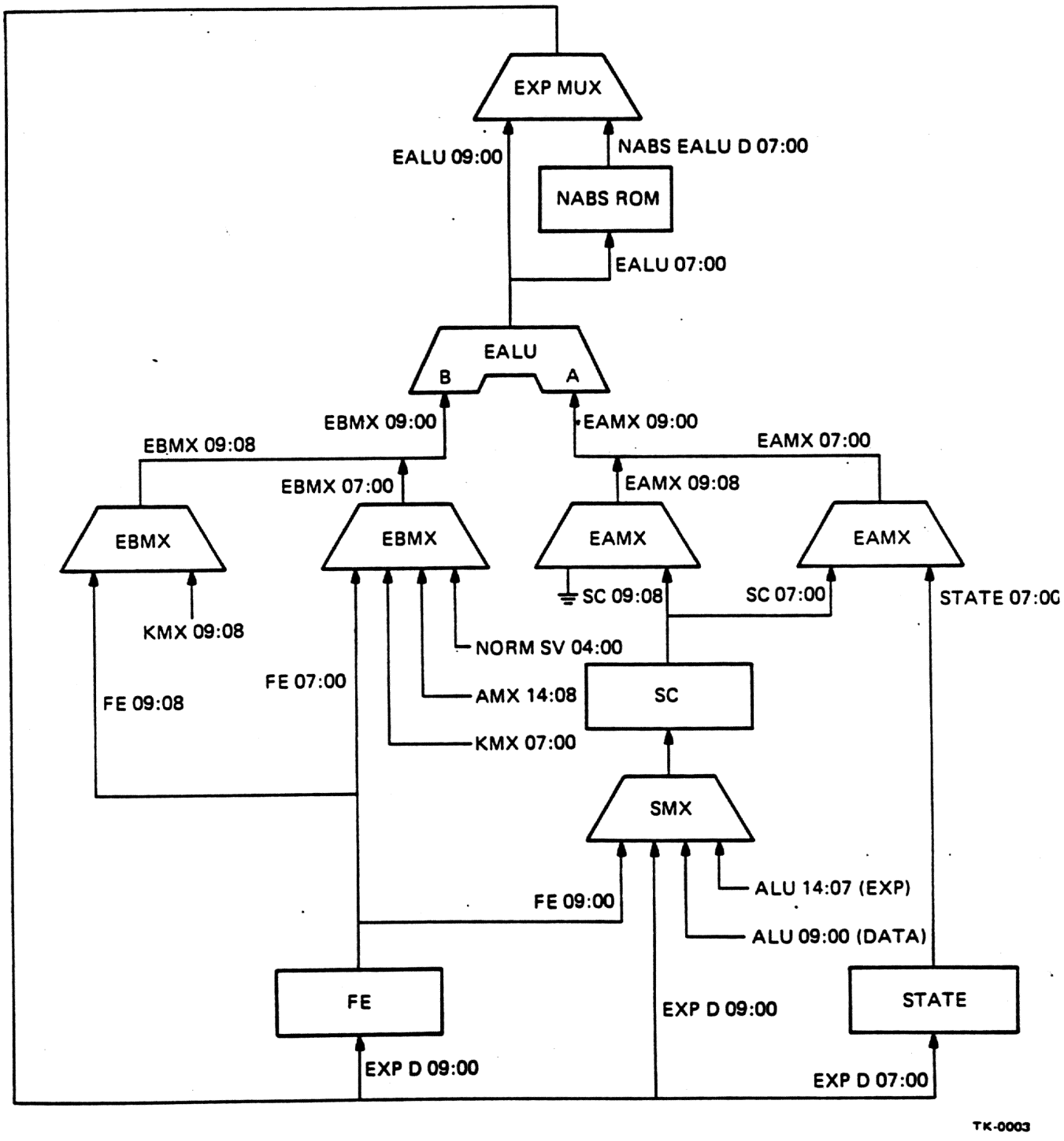


Figure 2-74 Exponent Section

The requirement for the NABSV ROM can be demonstrated by the following examples:

ADD 7_{10} plus 12_{10}

The two operands represented as binary normalized floating-point numbers are:

$7_{10} = .111 \times 2^3$ Exponent = 3 Fraction = .111
$12_{10} = .1100 \times 2^4$ Exponent = 4 Fraction = .11

To perform the floating-point addition, the exponents of the two operands must be equal or aligned. This requires shifting of the fraction associated with the smaller exponent.

To begin execution, the operands are stored in the following registers:

(a) Source exponent (3) is stored in the FE register.
(b) Destination exponent (4) is stored in the SC register.
(c) Source fraction (.111) is stored in the D register.
(d) Destination fraction (.11) is stored in the Q register.
(e) Both fractions are stored in temporary registers.

The selection of EALU function SC-FE (4--3) yields a result of positive 1. This result indicates that the fraction associated with the smaller exponent must be shifted by one bit position so that it is aligned with the fraction associated with the larger exponent. In order to align the fractions, a right shift or negative shift value is required. Therefore, if the result of the exponent subtraction is positive, the NABS ROM output is selected by the EXP MUX.

In this example, the output of the EALU is a positive 1 which results in the NABS ROM being selected. The output of the NABS ROM is a negative 1 represented in 2's complement form (FF). The output of the EXP MUX is 3FF because the two most significant bits are hardwired ones.

As previously mentioned, the fraction associated with the smaller exponent must be shifted to align the operands. However, only the D register contents can be shifted and in this example the smaller fraction was stored in the Q register. The contents of the Q register must first be loaded into the D register and then the D register is shifted by the value resulting from SC-FE. In this example, the D register contents are right shifted by 1 bit. The fraction associated with the larger exponent is loaded into the Q register from the temporary storage register and the two fractions can then be added.

2.6.6.2 EALU A-Input Multiplexer (EAMX) -- The EAMX provides the data source for the A input of the EALU. The EAMX allows selection of either the State register or Shift Count (SC) register. The State register is selected with the UMSC field of the microinstruction equals 5 (LOAD STATE REGISTER). Otherwise, the Shift Count register is selected by the EAMX.

2.6.6.3 EALU B-Input Multiplexer (EBMX) -- The EBMX provides the data source for the B input of the EALU. The EBMX allows selection of either the Floating Exponent (FE) register, AMX from the arithmetic section, Normalized Shift Value (NORM SV) from the data section, or the Constant Multiplexer (KMX) from the arithmetic section.

When exponents are being processed, the EBMX receives the exponent from the FE register or from the exponent field of the AMX (bits 14:07).

The constant input (KMX 07:00) or shift value input (NORM SV 04:00) is selected to allow the Shift Count (SC) to be updated. The constant input may also be used to set or clear flags in the State register.

The shift value (NORM SV), generated in the data section, is the number of left shifts necessary to normalize the contents of the D register (i.e., fraction part of the floating-point number). The normalized fraction is generated by left shifting the D register contents until the most significant 1 of D register data is in bit position 31.

Input selection of the EBMX is controlled by the UEBMX field of the microinstruction.

	UEBMX Field		
Hex	BUS CS 19	BUS CS 18	EBMX Data Selected
0	0	0	FE 09:00
1	0	1	KMX 09:00
2	1	0	AMX 14:07 (exponent field)
3	1	1	NORM SV 04:00

2.6.6.4 Floating Exponent Register (FE) -- The FE register is used to hold exponent or temporary values to be processed in the exponent section. The FE register is loaded with the output of the Exponent Multiplexer (EXP D 09:00) when the UFEK field (BUS CS 24) of the microinstruction equals 1.

2.6.6.5 State Register -- The State register consists of 8 flag bits, generated by the microprogram to control program flow. A 16 way branch condition in the microsequencer is created by each 4-bit group of the State register. A microinstruction may set or clear individual bits in the State register through the use of the logic operations of the EALU and constants from the KMX.

The State register is loaded with the output of the Exponent multiplexer (EXP D 07:00) when the UMSC field of the microinstruction equals 5.

2.6.6.6 Shift Count Multiplexer (SMX)
The SMX allows data to be transferred from the arithmetic section to the exponent section.

The SMX provides the path for a 10-bit data field (ALU 09:00) or an 8-bit exponent (ALU 14:07) from the ALU of the arithmetic section to the Shift Count register of the exponent section.

The SMX also allows selection of the Exponent multiplexer (EXP D 09:00) or Floating Exponent register (FE 09:00) as a source for the Shift Count (SC) register. The Exponent Multiplexer source allows the contents of the SC register to be incremented or decremented using the EALU. The FE register source is provided to allow the contents of FE and SC to be swapped in a single microinstruction.

The data type selected by the SMX is controlled by the USMX field of the microinstruction.

Hex	USMX Field BUS CS 17	BUS CS 16	SMX Data Selected
0	0	0	EXP D 09:00
1	0	1	FE 09:00
2	1	0	ALU 09:00 (data field)
3	1	1	ALU 14:07 (exponent field)

2.6.6.7 Shift Count Register (SC) -- The SC register is used for various functions within the CPU.

Area of CPU	Function of SC
ID Bus Control	SC 05:00 address an internal processor register word
Data Section	SC09 and SC04:00 control the shift amount in DAL
Arithmetic Section	SC04:00 control the bit mask generator and SC03:00 address the scratch pad register sets
Exponent Section	SC register used to store exponents or data.

The SC register is loaded with data from the Shift count Multiplexer (SMX09:00) when the USCK of the microinstruction (BUS CS 23) equals 1.

2.7 INTERRUPTS AND EXCEPTIONS

Interrupts and exceptions are the notification of events within the system which require the execution of software outside the current flow of control. These events cause the processor to transfer control from that of the currently executing process to a routine which can handle the interrupt or exception condition.

Exceptions are the notification of events which are relevant to the currently executing process and are normally serviced by software in the context of the current process.

Interrupts are the notification of events which are generally independent of the currently executing process and are serviced in a system wide context.

Certain interrupts and exceptions require high priority service while others must be synchronized with independent events. The priority logic in the processor determines the order in which events will be serviced. The priority associated with an interrupt is termed its interrupt priority level (IPL). Most exception service routines execute at the lowest interrupt priority level (IPL 0). However, exceptions which represent serious system failures raise the IPL to the highest level (IPL IF hex). This minimizes the processor interruption until the problem has been completely serviced. Paragraph 2.7.1.1 provides an explanation of the processor priority logic and the interrupt priority levels assigned.

Generally exceptions and interrupts are very similar. When either is initiated, the processor status longword (PSL) and the program counter (PC) are pushed onto the stack. However, there are a number of differences between exceptions and interrupts, listed as follows;

Exception	Interrupt
a. Caused by the execution of the current instruction.	a. Caused by an activity in the system that may be independent of the current instruction.
b. Usually serviced in the context of the process that produced the exception condition.	b. Serviced independently from the currently running process.
c. IPL of the processor is usually not changed when an exception is initiated.	c. IPL is always changed when an interrupt is initiated.
d. Service routines normally execute on a per-process stack (usually KSP).	d. Service routines normally execute on a per-CPU stack (ISP).

e. Enabled exceptions are initiated immediately, regardless of the current processor IPL.	e. Interrupts are not serviced until the processor IPL drops below the IPL of the requesting interrupt.
f. Most exceptions cannot be disabled. However, if an event causes an exception that is presently disabled, the exception will not be initiated even if it is subsequently enabled.	f. If an interrupt condition occurs while the interrupt is disabled (i.e., the processor is at the same or higher IPL), the interrupt will eventually be initiated when the enabling conditions are met.
g. The previous mode field in the PSL indicates the mode of the exception.	g. The previous mode field in the PSL is always set to Kernel.

2.7.1 Interrupts

The processor services interrupt requests at the end of instructions or at defined points during the execution of long, iterative instructions (e.g., string instructions). Each interrupt condition is assigned a request level. During the execution of each instruction, the interrupt requests are sampled and prioritized by the processor. When the priority of the interrupt requests is higher than the current IPL (bits 20:16 of the processor status longword), the processor will raise the IPL and service the interrupt request. The interrupt will cause the processor status longword (PSL) and the program counter (PC) of the next instruction to be pushed on the kernel or interrupt stack.

2.7.1.1 Interrupt Priority Level (IPL) -- Interrupt requests can be received from devices, controllers, or the processor itself. As previously mentioned, each request is assigned a level which determines the order in which multiple interrupts will be serviced. The processor has 31 interrupt priority levels (IPL), divided into 15 software levels (01 to 0F, hex) and 16 hardware levels (10 to 1F, hex). User programs and most exception service routines are run at process level, which can be thought of as IPL 0.

The interrupt requests are sampled during the execution of each instruction. The software and hardware requests are clocked into the priority logic from the hardware interrupt (HIR) and software interrupt (SIR) registers. Refer to Figure 2-75.

The priority encoder logic selects the highest interrupt level requested and will generate an interrupt level active code (IPL ACT 04:00). The IPL active bits are compared with the interrupt priority level (IPL) of the current process contained in bits 20:16 in the processor status longword (PSL) register. If the priority level of the interrupt request is greater than the current interrupt priority level and no exceptions occurred during the instruction, INTR REQ is generated and a branch at A fork in the microcode flow is taken to the interrupt service routine. The IPL ACT code is also used to generate the interrupt vector as described in Paragraph 2.7.1.3. Table 2-27 shows the interrupt conditions and the priority level and vector assigned to each.

2.7.1.2 Vectors -- Vectors are longword addresses in memory whose contents point to interrupt or exception service routines and determine how the event is to be serviced. The low two bits of the vector contents define the manner in which the event causing the interrupt or exception will be serviced. Bits 01:00 of the vector contents are interpreted as follows:

Bit 1	Bit Ø	Operation
Ø	Ø	This event is to be serviced on the kernel stack unless already on the interrupt stack.
Ø	1	This event is to be serviced on the interrupt stack. If this event is an exception, the IPL is raised to 1F (hex).
1	Ø	This event is to be serviced by microcode in the writable diagnostic control store. Bits 15:Ø2 of the vector contents are used as a parameter by code in WCS.
1	1	The operation is a halt.

When bits Ø1:ØØ specify codes Ø or 1, bits 31:Ø2 of the vector contents contain the virtual address of the service routine.

Separate vectors are defined for each interrupt and class of exceptions. All vectors are contained in a page of memory named the system control block (SCB). The system control block base register (SCBB) is an ID bus register (ID bus address = 3B, hex) which contains the physical page address of the system control block.

Specific longword addresses (interrupt vectors) are formed by adding hardware generated vector bits Ø8:Ø2 to the system control block base register. Bits Ø1:ØØ are zero since the vector generated is the physical longword address of a specific location in the SCB. The contents of each vector point to the service routine which handles the event causing the interrupt. Figure 2-76 illustrates the manner in which interrupt vectors are formed.

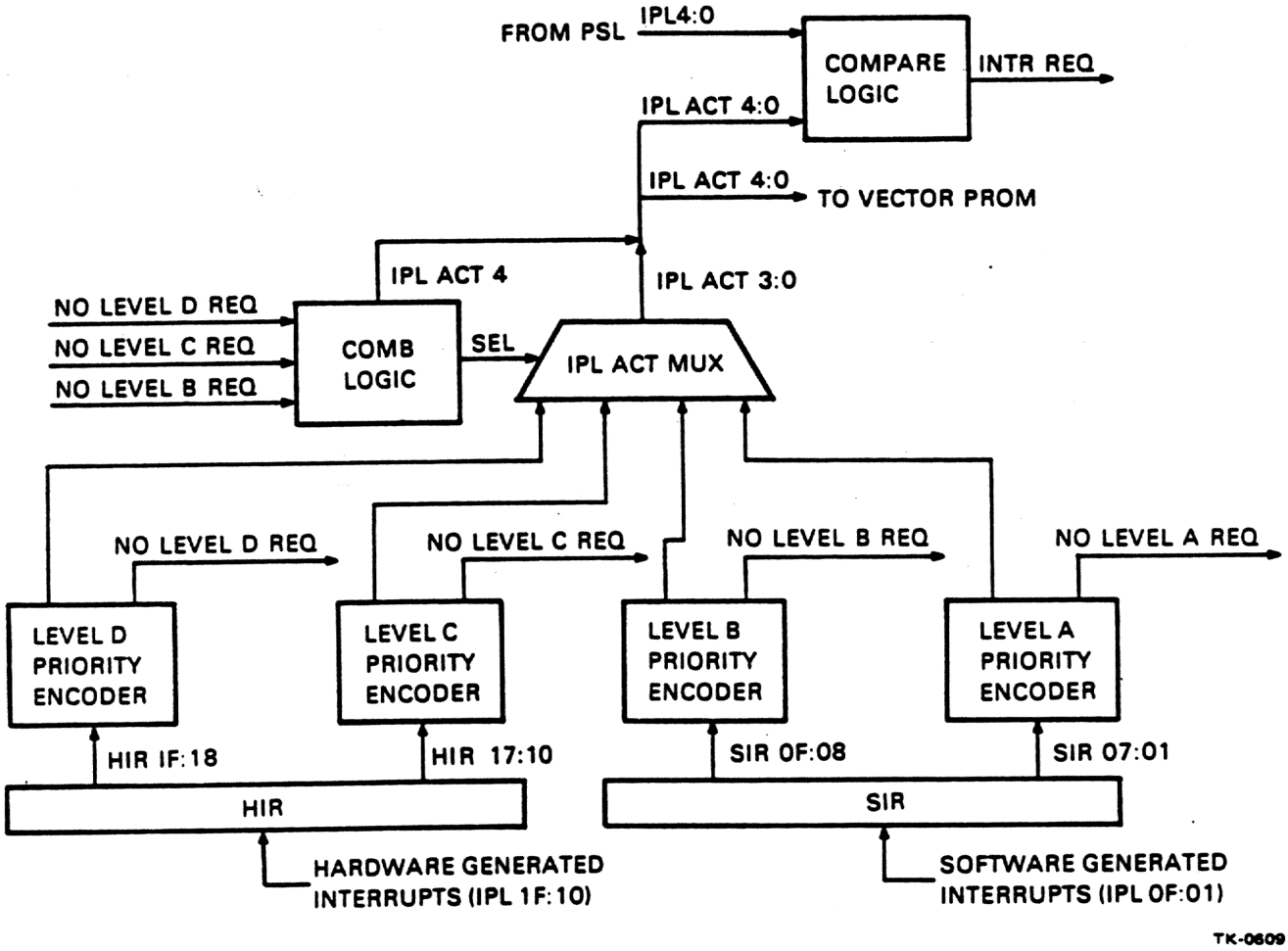


Figure 2-75 Interrupt Request Arbitration

Table 2-28 lists the priority level of each interrupt and the vectors assigned.

Table 2-28 Interrupt Priority Levels and Vector Assignments

IPL ACT	INTERRUPT CONDITION	VECTOR	PRIORITY
1F	NONE ASSIGNED	X	HIGHEST
1E	CPU POWER FAIL	0C	
1D	CPU TIMEOUT	60	
1C	SBI FAULT	5C	
1B	SBI ALERT	58	
1A	CRD/RDS	54	
19	SBI SILO COMPARE	50	
18	INTERVAL TIMER	C0	
17	SBI REQ 7	1C0 TO 1FC	
16	SBI REQ 6	180 TO 1BC	
15	SBI REQ 5	140 TO 17C	
14	SBI REQ 4	100 TO 13C	
	CNSL RECEIVE INTR	F8	
	CNSL TRANSMIT INTR	FC	
13	NONE ASSIGNED	X	
12	NONE ASSIGNED	X	
11	NONE ASSIGNED	X	
10	NONE ASSIGNED	X	
0F	SIR0F	BC	
0E	SIR0E	B8	
0D	SIR0D	B4	
0C	SIR0C	B0	
0B	SIR0B	AC	
0A	SIR0A	A8	
09	SIR09	A4	
08	SIR08	A0	
07	SIR07	9C	
06	SIR06	98	
05	SIR05	94	
04	SIR04	90	
03	SIR03	8C	
02	SIR02 OR AST DEL	88	
01	SIR01	84	LOWEST
00	NO INTERRUPT	X	X

Exception vectors are generated by microservice routines (refer to Paragraph 2.7.2). Table 2-29 lists the vectors assigned to each class of exceptions.

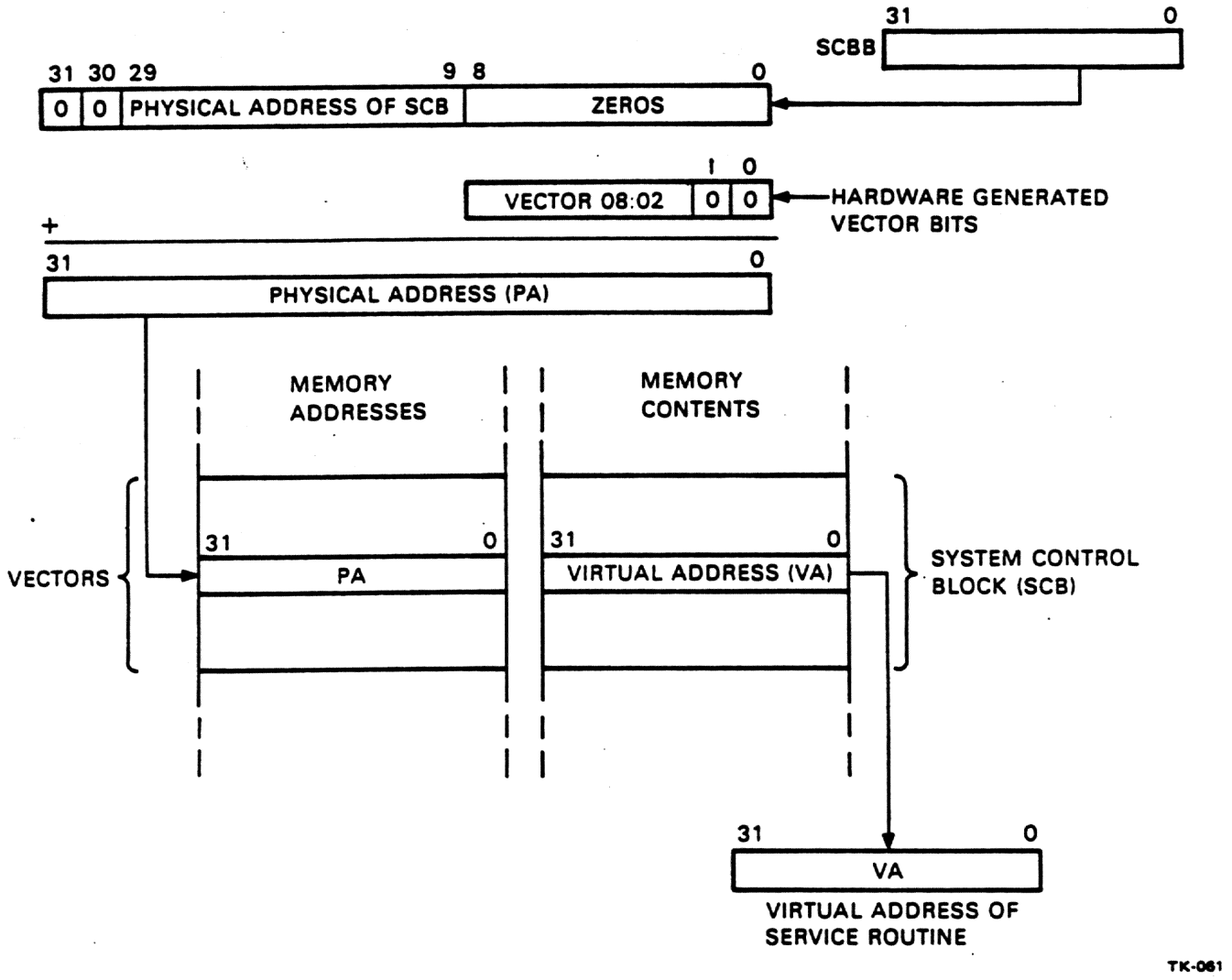


Figure 2-76 Interrupt Vector Formation

2.7.1.3 Hardware Generated Interrupt Vector -- Bits 08:02 of the vector are generated by the interrupt control logic. Vector generation for internal interrupts is straightforward since there is only one vector assigned to each internal interrupt and corresponding interrupt priority level (IPL). However, multiple external interrupts can occur on each of interrupt priority levels 17:14. These priority levels correspond the SBI request levels 07:04. Several nexus (e.g., Unibus adapter, Massbus adapter) can simultaneously request service at the same SBI level. Therefore, the IPL at which an external interrupt occurs will not in itself identify the nexus that caused the interrupt. A number of vectors are assigned to interrupts occuring at each of the interrupt priority levels 17:14.

The following paragraphs describe how vector bits 08:00 are generated for both internal interrupts and external SBI interrupts.

Internal interrupts -- The hardware generates vector bits 08:02 as a function of the interrupt priority active level and the status of the console receive and transmit lines. Refer to Figure 2-77. IPL ACT bits 04:00 and the console lines are input to a VECTOR PROM and combinational logic. The VECTOR PROM generates vector lines 08:02. The combinational logic is used to generate the external interrupt request line when the IPL is 17:14. However, the console receive and transmit interrupts are also requested on IPL 14. Therefore, when the IPL ACT is 14 and the console interrupt lines are asserted without an external SBI REQ 4, the generation of EXT INT REQ is inhibited. The EXT INTR REQ line will be disabled for all internal interrupts and the value of vector bits 08:02 will correspond directly to the output of the VECTOR PROM.

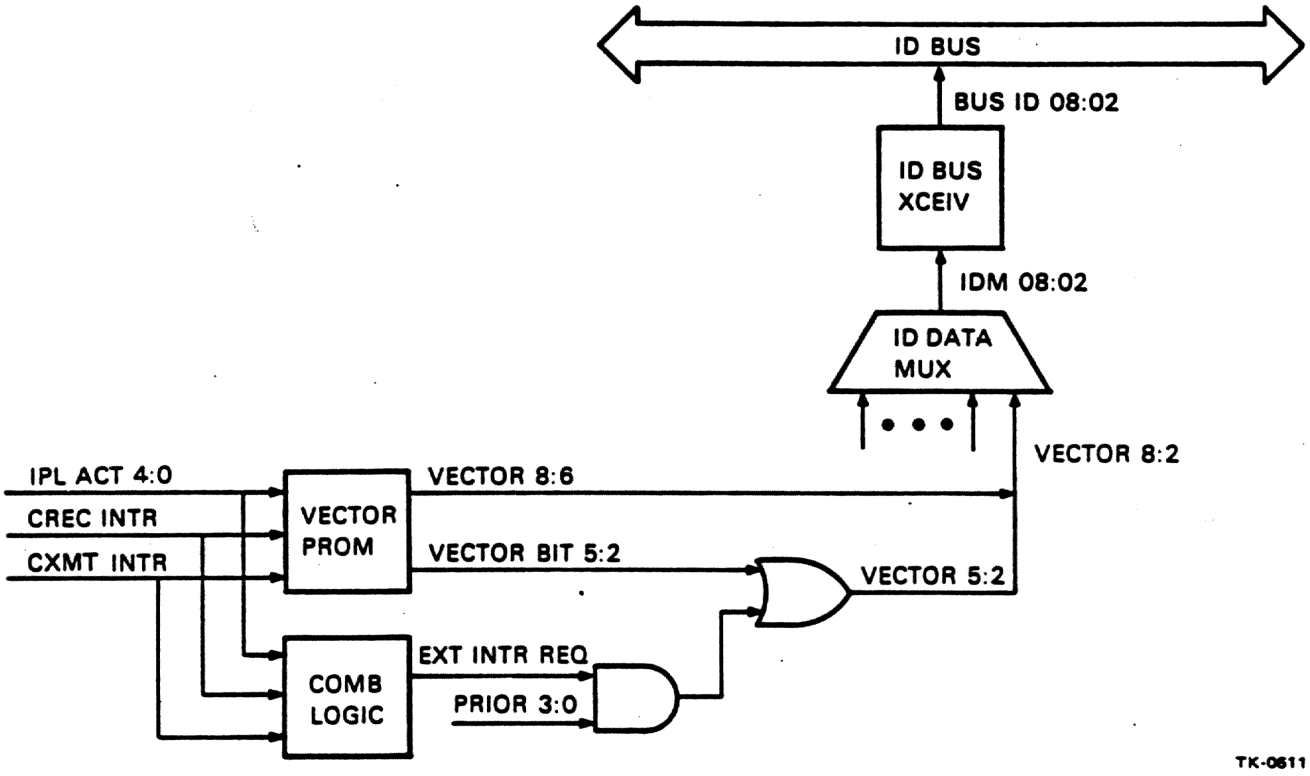


Figure 2-77 Generation of Interrupt Vector Bits 08:02

External interrupts -- external interrupts occurring on SBI request levels 07:04 are serviced on interrupt priority levels 17:14 respectively. The nexus (UBA, MBA, etc.) on the SBI can interrupt the processor on each of the four request levels. Unlike internal interrupts, the IPL activated will not specifically identify the event which causes the external interrupt. In order to identify the interrupting nexus, the processor must issue an interrupt summary read on the SBI. Refer to Figure 2-78.

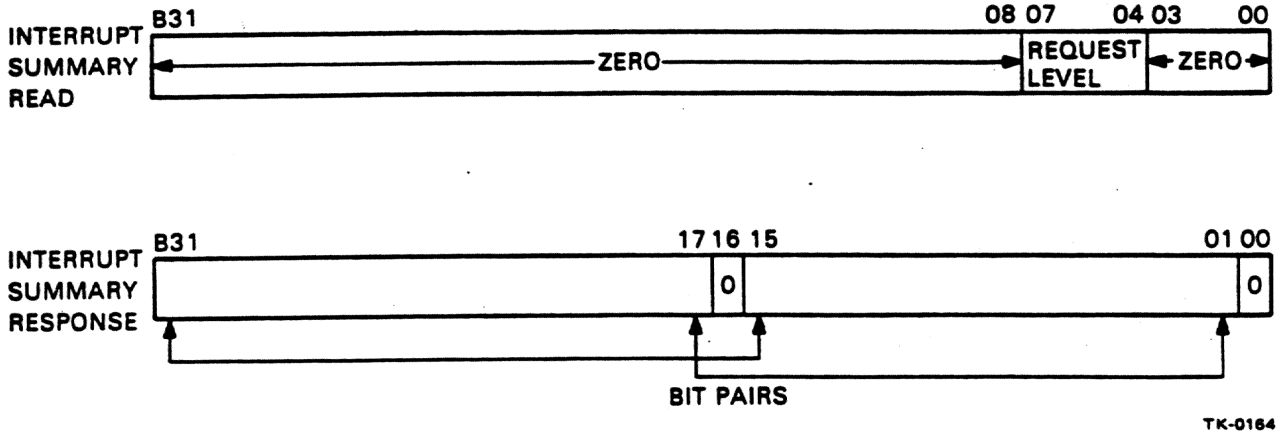


Figure 2-78 Interrupt Summary Read and Response Formats

The processor will specify the request level at which it is polling interrupts. Nexus receiving the interrupt summary read command, and asserting the request line specified in the interrupt level mask (B07:04), will generate an interrupt summary response by asserting a bit pair in the information field (B31:00). The bits are asserted in corresponding positions in the upper and lower half of the information field (refer to Figure 2-78). Bit pairs are asserted to maintain correct parity. The two bits asserted by the requesting nexus are equal to the nexus TR number and the nexus TR number plus 16. A transfer request (TR) number is assigned to each nexus to establish the fixed priority access. Assertion of a given TR line in the interrupt summary response will identify the nexus which is interrupting at the specific request level.

The upper half of the information field (B31:16) is transferred to the interrupt logic via the Internal Data (ID) bus. Refer to Figure 2-79.

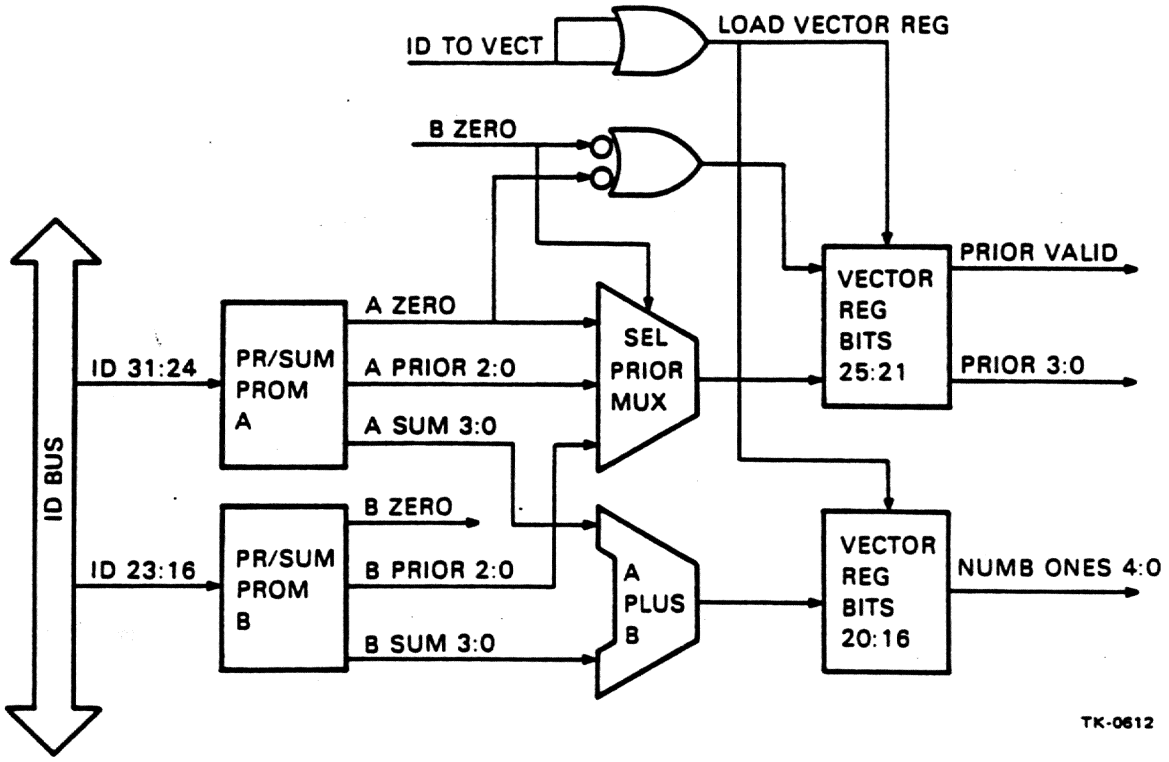


Figure 2-79 Vector Register Bits 25:16

The ID bus data (ID 31:16) is input to priority/sum PROMs which encode the information for use in the vector register. The A or B priority bits specify the nexus with the lowest TR number (highest priority) which interrupted at the given request level. The output from PROM A is selected by the prior mux if ID 23:16 is all zeros.

The PRIOR VALID bit in the vector register, when set, indicates that ID 31:16 was not all zeros and that at least one nexus interrupted at the given request level. The sum bits of PROMs A and B are added to form the number of one bits in the ID 31:16 field and are not used in the interrupt process. Figure 2-80 illustrates the format of the entire vector register. The vector register is a read-only register addressable on the internal data bus (ID bus address = 0D, hex).

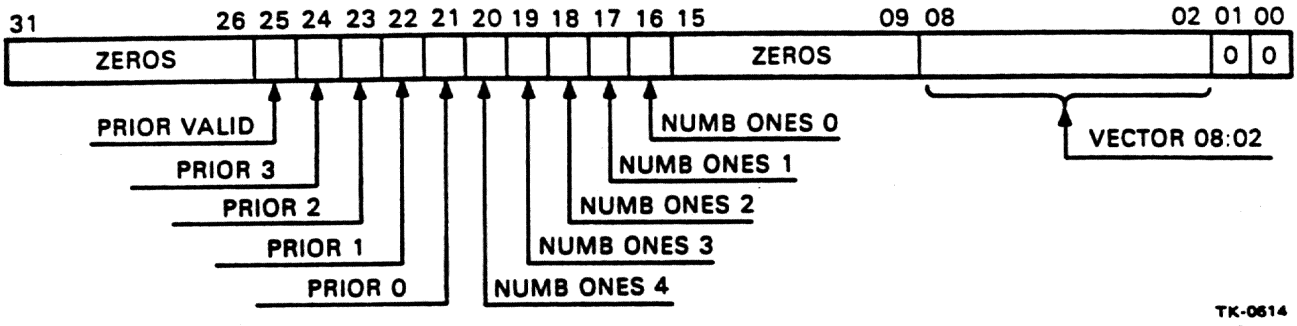


Figure 2-80 Vector Register Format

Vector register bits 24:21 (PRIOR 03:00) are used by hardware to generate VECTOR lines 05:02. Refer to Figure 2-77. If the interrupt is decoded as an external request, PRIOR 03:00 are ORed with VECTOR PROM output bits 05:02 to form VECTOR 05:02. These lines, in conjunction with VECTOR 08:06, are used to point to a service routine unique to the interrupting nexus. The nexus service routine will then initiate a read transfer on the SBI to further identify the particular kind of interrupt requested by the nexus (e.g., Unibus device interrupt). The information read is used to point to a subroutine which can service that particular interrupt.

If multiple nexus request interrupts on the same level, multiple interrupt summary read commands are issued by the processor until all nexus have been serviced.

2.7.1.4 Hardware Interrupt Conditions -- The following paragraphs provide a description of each of the hardware conditions (Figure 2-81) which cause interrupts on interrupt priority levels 1E:14.

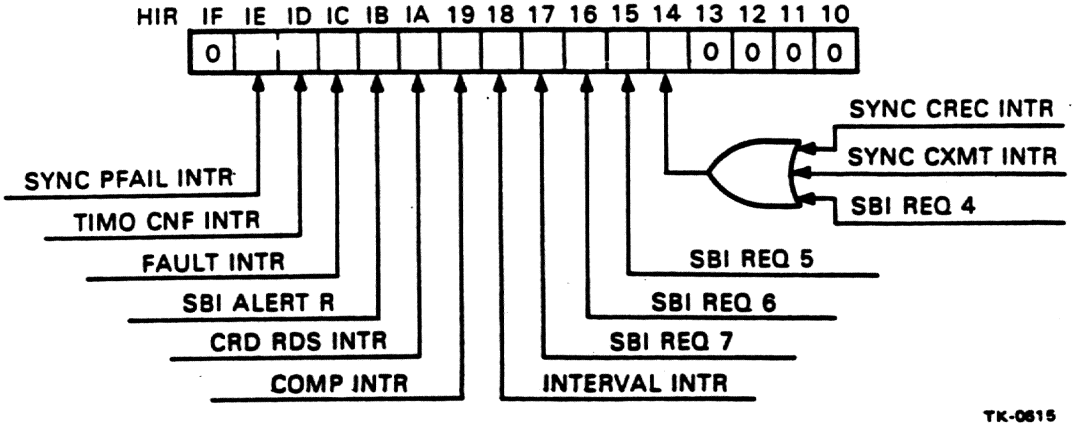


Figure 2-81 Hardware Interrupt Register (HIR)

SYNC PFAIL INTR (1E) -- The CPU power fail interrupt occurs if the processor receives a power fail warning (power supply AC LO) or if a critical system element receives a power fail warning (SBI FAIL, QBUS AC LO). Critical system elements are those which must be functioning before the power up routine is initiated by the processor. Critical elements include the 11/03 microprocessor, bootstrap or main memories, SBI terminators, clock circuitry, or CPU.

TIMO CNF INTR (ID) -- The CPU write timeout interrupt occurs if the processor initiates a write command and the destination does not respond with a confirmation within 512 SBI cycles. Note that write timeout interrupts do not necessarily occur during the instruction which initiated the write command. Write commands are stored in a buffer and the processor is allowed to continue while an SBI write cycle is pending.

FAULT INTR (1C) -- The SBI fault interrupt occurs if an SBI bus error was detected by any device on the bus including the processor. If the processor detects a fault condition which prevents the completion of a read cycle, an exception condition is also generated.

NOTE

This interrupt can occur only if the Fault Interrupt Enable bit is set in the SBI Fault/Status register.

SBI ALERT (1B) -- This interrupt occurs when a device which does not contain SBI interrupt request sequencing logic wishes to interrupt the processor. Events causing this interrupt may be device power failure or power up, or when environmental conditions such as overtemperature are detected. The SBI alert line is generated by the logic OR of alert status bits in the devices configuration register.

CRD/RDS INTR (1A) -- The corrected read data (CRD) interrupt is asserted if the processor received read data which has been corrected by main memory. The read data substitute (RDS) interrupt is asserted if the processor received uncorrected read data on a read cycle to the instruction buffer. These interrupts can occur only if the RDS/CRD interrupt enable bit is set in the SBI error register.

COMP INTR (19) -- The SBI silo compare interrupt occurs when particular fields of the SBI bus match signals specified in the SBI comparator register. This interrupt can occur only if the silo lock interrupt enable bit is set in the SBI comparator register.

INTERVAL INTR (18) -- This interrupt occurs when the interval count register overflows and can only be initiated if enabled in the clock control status register.

SBI REQ 07:04 (17:14) -- External interrupts occurring on SBI request levels 07:04 are servived on interrupt priority levels 17:14. These interrupts result from device completion, device errors, and important device status changes.

SYNC CREC INTR/SYNC CXMT INTR (14) -- The console terminal receive interrupt occurs when the Done bit is set in the console's receiver control/status register (RXCS). The receive interrupt enable bit in the RXCS register must be set for the interrupt to occur.

The console terminal transmit interrupt occurs when the Ready bit is set in the console's transmit control/status register (TXCS). This interrupt cannot occur unless the transmit interrupt enable bit in the TXCS register is set. The receiver interrupt has higher priority than the transmit interrupt.

2.7.1.5 Software Generated Interrupts -- Interrupt priority levels 0F through 01 are reserved exclusively for software. Software can force an interrupt by executing the instruction MTPR SRC, #SIRR. This move to processor register instruction will move the contents of the source register to the software interrupt request register (SIRR). The SIRR is accessible in processor register space and its register number is 14 (hex). It is a write-only, four bit register formatted as shown in Figure 2-82.

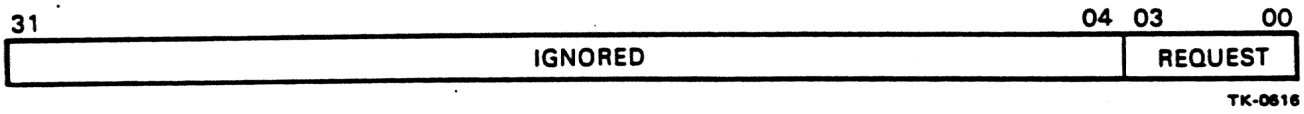


Figure 2-82 Software Interrupt Request Register (SIRR)

Executing MTPR SRC, #SIRR requests an interrupt at the level specified by the low four bits of the source register (SRC 03:00). Once a software interrupt request is made, it will be cleared by the hardware when the interrupt is taken. If SRC 03:00 is greater than the current IPL, the interrupt occurs before execution of the following instruction. If SRC 03:00 is less than or equal to the current IPL, the interrupt will be deferred until the IPL is lowered to less than SRC 03:00. If there are higher level interrupts pending, the higher interrupts will be taken first.

Pending software interrupts are held in the software interrupt summary register (SISR). The SISR is a read/write processor register (number 15, hex) which is formatted as shown in Figure 2-83. The SISR contains 1's in the bit positions corresponding to levels on which software interrupts are pending.

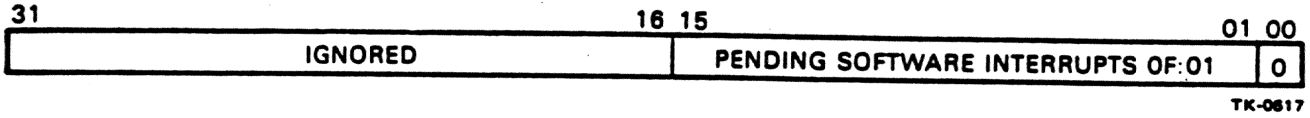


Figure 2-83 Software Interrupt Summary Register (SISR)

When a software request is made by writing into the software interrupt request register (SIRR), the microcode will interpret this write as a bit set operation to the SISR. The mask generator in the data paths is used to decode the request level in the SIRR into a single bit designation in the SISR. The software interrupt summary register can be written directly be executing the instruction MTPR SRC, #SISR. However, this is not the normal way of making software interrupt requests. This method is useful for clearing the software interrupt system and for reloading the system after power fail.

Bit Ø2 of the SISR is also set if an asynchronous system trap (AST) is delivered at interrupt priority level 2 (IPL Ø2). During the execution of the REI (return from exception or interrupt) instruction, the microcode compares the value in the current mode field of the PSL image (bits 25:24) with the asynchronous system trap level (ASTLVL) in the ASTR register. If the current mode is a greater value (lesser priority) than the 3-bit ASTLVL, an asynchronous system trap (AST) is delivered at interrupt priority level 2. The microcode will set bit Ø2 in the SISR before completing the execution of REI. The asynchronous system trap level register (ASTR) is a 3-bit, read/write processor register (number 13, hex) which is formatted as shown in Figure 2-84.

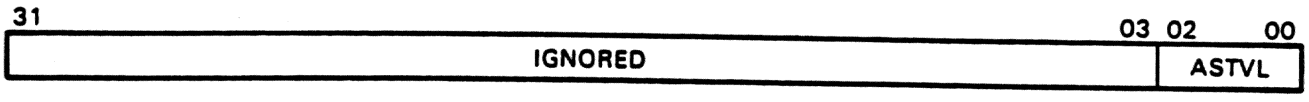


Figure 2-84 Asynchronous System Trap Level Register (ASTR)

Software Interrupt Register (SIR) -- The software interrupt register is an internal data bus register (ID bus address = ØE, hex) located on the Interrupt Control board. Refer to Figure 2-85.

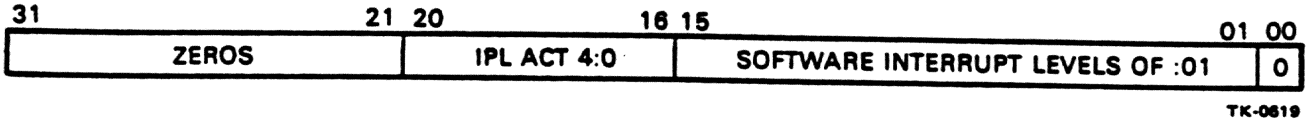


Figure 2-85 Software Interrupt Register

Bits 15:Ø1 of the SIR are read/write. They are in the same format as bits 15:Ø1 of the software interrupt summary register (SISR) in processor register space. When an MTPR instruction is executed and the processor register specified is the SISR or SIRR, the software interrupt register is clocked with data from the ID bus (ID 15:Ø1). The ID bus data specifies which software interrupt requests are being made. Bits 2Ø:14 (IPL ACT Ø4:ØØ) indicate the level of the highest priority interrupt pending.

2.7.2 Exceptions

Exceptions are the notification of events which are relevant primarily to the currently executing process and normally invoke software in the context of the current process. Exceptions occur in the middle or at the end of the instruction during which the exception conditions were detected. The PSL and PC of the instruction, or the PC of the next instruction, are pushed onto the kernel or interrupt stack. Also, up to 16 longwords of exception parameter information may be pushed onto the stack. The processor's IPL is generally not changed by exceptions. However, two exception conditions, Kernel Stack Not Valid and Machine Check Faults, do raise the IPL to the highest priority level (1F, hex).

Exceptions are classified into one of the following three categories, depending on when the exception condition occurs and how they leave the general registers and memory:

a. Trap — An exception condition that occurs at the end of the instruction that caused the exception. The PC saved on the stack is the address of the next instruction that would normally have been executed. Arithmetic traps are the only exceptions which can be disabled (via BISPSW and BICPSW instructions).

b. Fault — An exception condition that occurs in the middle of an instruction, and leaves the registers and memory in the state such that elimination of the fault conditons and restarting the instruction will give the correct results. The PC saved on the stack is the address of the instruction in which the fault was detected.

c. Abort — An exception condition that occurs in the middle of an instruction and potentially leaves the registers and memory such that the instruction cannot be correctly restarted or completed. The PC saved on the stack does not necessarily point to the beginning of the next instruction.

Exception conditions detected by the hardware modify microprogram flow by one of the following two methods:

a. Microbranches -- Some exception conditions can modify the next microaddress via the branch multiplexers in the microsequencer logic (refer to Paragraph 2.3.2.1). The signal lines representing the exception conditions are tested when the associated branch enable value is specified in the UBEN field of the microword. If the exception condition is present, the microaddress is modified to reflect it. Microbranches can also be performed via the A Fork service logic. The service bits are input to the instruction decode logic. When the subroutine field (USUB) of the current microword equals 3, the instruction decode logic generates the lower eight bits of the microaddress. If certain exception conditions are present, the service bits are selected by the instruction decode logic to generate the microaddress.

b. Microtraps -- Some exception conditions generate microtrap vector bits which are input to the branch multiplexers in the microsequencer logic (refer to Paragraph 2.3.2.3). The microtrap condition will force a particular branch enable to be selected (BEN 1Ø). This branch enable will select the vector bits (UTRAP VECT Ø3:ØØ) as the source for the low four bits of the microaddress. The upper bits of the microaddress are hardwired to form the rest of the vector. The exception conditions detected during microinstructions force the microcode flow to service routines pointed to by the vector. The microprogram counter is pushed onto the microstack so that the program can continue after the error is serviced.

2.7.2.1 **Exception Vectors** -- The microservice routines are pointed to by the microbranch address or microtrap vector. Each microservice routine will generate the correct exception vector and related exception codes using a set of constants specified in the UKMX field of the microword. The service routine will also use the information stored in the processor registers to assemble the necessary parameters to be saved on the stack. Table 2-29 lists each exception, class, vector assignment and method by which the exception conditions are detected.

Table 2-29 Exception Conditions and Assigned Vectors

Exception Condition	Vector	Class	Detection Function
Machine Check	04	fault\abort	ubranch/utrap
Kernel Stack Not Valid	08	abort	ubranch
Reserved DEC Op Codes and Privileged Instructions	10	fault	ubranch
Reserved Customer Op Codes	14	fault	ubranch
Reserved Operands	18	fault/abort	ubranch/utrap
Reserved Addressing Modes	1C	fault	ubranch
Access Control Violation	20	fault	utrap/ubranch
Translation Not Valid	24	fault	ubranch
Trace Pending (TP)	28	fault	ubranch
Breakpoint Instruction (BPT)	2C	fault	ubranch
Compatibility Mode Program Error	30	trap/abort	ubranch
Arithmetic Trap	34	trap	ubranch
CHMK OP CODE	40	trap	ubranch
CHME OP CODE	44	trap	ubranch
CHMS OP CODE	48	trap	ubranch
CHMU OP CODE	4C	trap	ubranch

2.7.2.2 Serious System Failures -- The following paragraphs provide a brief description of exceptions which are of such importance that the interrupt priority level is raised to 1F (hex) or the machine is halted.

2.7.2.2.1 Kernel Stack Not Valid Abort -- The kernel stack not valid abort is an exception that indicates the kernel stack was not valid while the processor was pushing information onto the kernel stack during the initiation of an exception or interrupt. This is usually an indication of a stack overflow or other executive software error. The attempted exception is changed into an abort that uses the interrupt stack. The interrupt priority level (IPL) is raised to 1F (hex). No additional parameters are pushed onto the interrupt stack.

2.7.2.2.2 Interrupt Stack Not Valid Halt -- An interrupt stack not valid halt is an exception that indicates that the interrupt stack was not valid or that a memory error occurred while the processor was pushing information onto the interrupt stack during the initiation of an exception or interrupt. No further interrupt requests are acknowledged on this processor.

2.7.2.2.3 Machine Check Exception -- A machine check exception indicates that an internal processor error was detected. The interrupt priority level is raised to 1F (hex). In addition to the the PC and PSL, parameters are pushed onto the stack as longwords. These parameters will depend on the type of machine check encountered. The last longword pushed will specify the number of additional bytes pushed, excluding the PC, PSL, and count longword. The information pushed will enable software to decide whether or not to abort the current process, and is logged for analysis by field service.

At any machine check, the error handling microcode attempts to logout the following information. Ordinarily, it appears on the stack as shown. However, if a double error halt occurs, the operator can find the same information in the ID bus temporary registers.

Data	Memory Location	ID Bus Location
Byte Count	(SP)	none
Summary Parameter	(SP)+4	T0(30)
CPU Error Status	(SP)+8	T1(31)
Trapped UPC	(SP)+12	T2(32)
VA/VIBA	(SP)+16	T3(33)
D Register	(SP)+20	T4(34)
TB ERR 0	(SP)+24	T5(35)
TB ERR 1	(SP)+28	T6(36)
Timeout Address	(SP)+32	T7(37)
Parity	(SP)+36	T8(38)
SBI Error	(SP)+40	T9(39)
PC	(SP)+44	none
PSL	(SP)+48	none

The summary parameter is a longword. Byte 1 of the longword is a flag. It is non-zero if a CP timeout or CP error confirmation interrupt was pending at the time the machine check occurred. Byte 0 identifies the type of machine check as follows:

Machine Check Code	Exception Condition
00	CP Read Timeout or Error Confirmation Fault
02	CP Translation Buffer Parity Error Fault
03	CP Cache Parity Error Fault
05	CP Read Data Substitute Fault
0A	IB Translation Buffer Parity Error Fault
0C	IB Read Data Substitute Fault
0D	IB Read Timeout or Error Confirmation Fault
0F	IB Cache Parity Error Fault
F1	Control Store Parity Error Abort
F2	CP Translation Buffer Parity Error Abort
F3	CP Cache Parity Error Abort
F4	CP Read Timeout or Error Confirmation Abort
F5	CP Read Data Substitute Abort
F6	Microcode "not supposed to get here" abort

The control store parity error is detected by a utrap. The remaining exception conditions are detected by ubranches during instruction buffer cycles and utraps for data path cycles.

2.7.2.3 Exceptions Detected During Operand Reference

2.7.2.3.1 Access Control Violation -- An access control violation fault is an exception that occurs when the process attempts a reference not allowed at the access mode in which the process was operating (protection violation). An access control violation fault is also taken if the virtual address referenced is beyond the end of the associated page table (length violation). The protection violation is detected by a utrap for data path cycles and a ubranch for instruction buffer cycles if the entry was in the TB. Otherwise, it is detected by a ubranch. The length violation is detected by the branch function.

2.7.2.3.2 Translation Not Valid -- A translation not valid fault is taken when a read or write reference is attempted through an invalid page table entry (PTE 31 = 0). This fault is detected by a ubranch.

2.7.2.3.3 Reserved Addressing Mode -- A reserved addressing mode fault is an exception which occurs when certain addressing modes are used in a prohibited situation. No additional parameters are pushed. The following lists the situations in which the use of certain addressing modes will cause a fault.

Addressing Mode	Situation
Short Literal	Modify, destination, address source, or within index mode.
Register	Address source or within index mode.
Index	Within index mode, or with PC as index.

2.7.2.3.4 Reserved Operand -- A reserved operand exception indicates that the operand accessed has a format reserved for future use by Digital. No additional parameters are pushed. The PC is always backed up to point to the op code. The service routine determines the type of operand by examining the op code using the stored PC. Only changes made as a result of the instruction fetch or operand specifier evaluation can be restored. Therefore, some instructions are not restartable and the associated exception is an abort rather than a fault. The PC is always properly restored unless the instruction attempted to modify it in a manner that yields unpredictable results. The PSL, other than the FPD and TP bits, is not changed except for the condition codes which are unpredictable.

The following lists the events which cause reserved operand exceptions and whether the event results in a fault or an abort:

a. A floating-point number that has the sign bit set and the exponent zero except in the POLY table (FAULT)

b. A floating-point number that has the sign bit set and the exponent zero in the POLY table (ABORT)

c. POLY degree too large (FAULT)

d. Decimal string too long (FAULT)

e. Invalid digit in CVTTP, CVTSP (FAULT)

f. Bit field too wide (FAULT)

g. Invalid combination of bits in PSL restored by REI (FAULT)

h. Reserved pattern operator in EDITPC (ABORT)

i. Incorrect source string length at completion of EDITPC (ABORT)

j. Invalid combination of bits in PSW/MASK longword during RET (FAULT)

k. Invalid combination of bits in BISPSW/BICPSW (FAULT)

l. Invalid CALLx entry mask (FAULT)

m. Invalid register number in MFPR or MTPR (FAULT)

n. Invalid combinations in PCB loaded by LDPCTX (ABORT)

o. Unaligned operand in ADAWI, INSQU, or REMQUE (FAULT)

p. Invalid register contents in some MTPR's (FAULT)

2.7.2.4 Exceptions Occurring as the Consequence of an Instruction

2.7.2.4.1 Op Code Reserved to Digital -- An op code reserved to Digital fault occurs when the processor encounters an op code that is not specifically defined or requires higher privileges than the current mode. No additional parameters are pushed. Op code FFFF (hex) will always fault.

2.7.2.4.2 Op Code Reserved to Customers and CSS -- This fault occurs if an op code reserved to customers or Digital's Computer Special Systems (CSS) group (xxFC) is executed. The operation is identical to the op code reserved to Digital fault except that the event is caused by a different set of op codes and faults through a different vector.

2.7.2.4.3 Compatibility Mode Exception -- This exception occurs when a reserved op code or an illegal instruction is encountered when executing instructions in compatibility mode. Also, a compatibility mode abort may occur if an odd address error is detected during the following memory references:

a. Any reference with VA00 = 0 and not a byte instruction.

b. An address fetch in the evaluation of addressing mode 3, 5, or 7.

c. An index word fetch in the evaluation of addressing modes 6 and 7.

d. Instruction fetch

An additional longword of information (trap code) is pushed onto the stack which indicates the event which caused the exception. The following lists the condition, trap code, and class of exception.

Exception Condition	Trap Code	Class
reserved op code	0	fault
BPT op code	1	fault
IOT op code	2	fault
EMT op code	3	fault
TRAP op code	4	fault
illegal instruction	5	fault
odd address error	6	abort

The special op codes and illegal instructions are detected by ubranches and the odd address error is detected by a utrap.

2.7.2.4.4 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 debuffer 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 is completed, a trace trap will occur. At this point, the tracing program can again re-insert the BPT instruction, restore T to its original state, and resume.

2.7.2.5 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 trap occurs before the execution of the subsequent instruction (except that a service routine invoked by CHMx and terminated by REI is considered a single instruction). The saved PC on a trace is the address of the next insturction that would normally be executed.

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). If only one bit were used then the occurrence of an interrupt at end of instruction would either produce zero or two traces, depending on the design. Instead, the PSL T bit is defined to produce a trap after any other traps or aborts. The trap effect is implemented by copying PSL T to a second bit (PSL TP) which is actually used to generate the exception. PSL TP generates a fault before any other processing at the start of the next instruction.

The rules of operation for trace are:

1. At the beginning of an instruction, if T is set then TP is set.

2. If the instruction faults or an interrupt is serviced, the pushed PSL TP is cleared. The pushed PC is set to the start of the faulting or interrupt instruction.

3. If the instruction aborts or takes an arithmetic trap, the pushed PSL TP is set or cleared as the result of step 1.

4. If an interrupt is serviced after instruction completion and arithmetic traps but before tracing is checked for at the start of the next instruction, then the pushed PSL TP is set or cleared as the result of step 1.

5. At the beginning of an instruction, if TP is set then a trace pending fault is taken.

There are two special cases. They are the CHMx and the REI instructions. These are special since they may change TP, something that no other instructions do. However, these also follow the rules given above.

The routine entered by a CHMx is not traced because CHMx clears T and TP in the new PSL. However, if T was set at the beginning of CHMx the saved PSL will have both T and TP set. REI will trap either if T was set when the REI was executed or if TP in the saved PSL is set. Because of this, the instruction sequence CHMx...REI acts as a single instruction. Note that the trace trap occurring after an REI that has TP set before being executed will be taken with the new PSL. Thus, special care must be taken if exception or interrupt routines are traced.

In addition, the CALLx instructions save a clear T, although T in the PSL is unchanged. This is done so that a debugger or trace program proceeding from a BPT fault does not get a spurious trace from the RET that matches the CALL.

The detection of interrupts and other exceptions occurs before the detection of a trace trap. However, this causes no difficulties since the entire PSL (including T and TP) is automatically saved on interrupt or exception initiation and is restored at the end with an REI. This makes interrupts and benign exceptions totally transparent to the executing program.

2.7.2.6 Change Mode Instruction Trap -- When execution of each of the change mode instructions (CHMK, CHME, CHMS, CHMU) is complete, a trap is performed. Additional parameters pushed include the sign extended operand contained in the D register. This exception condition is detected by a microbranch.

2.7.2.7 Arithmetic Traps -- Arithmetic traps occur as the result of performing arithmetic or conversion operations. The traps are mutually exclusive and all are assigned the same vector, 34 (hex). The arithmetic traps indicate that an exception had occurred during the last instruction and that the instruction has been completed. The exception conditions are detected by ubranches. The PC pushed on the stack is that of the next instruction to be executed. In addition to the PSL and PC, a longword is pushed onto the stack which identifies the exception condition. The following lists the trap code pushed on the stack and the associated condition.

Exception Condition	Trap Code
integer overflow	1
integer divide by zero	2
floating overflow	3
floating/decimal divide by zero	4
floating underflow	5
decimal overflow	6
subscript range	7

The following paragraphs provide a brief description of each arithmetic trap condition.

2.7.2.7.1 Integer Overflow Trap -- An integer overflow trap is an exception that indicates the last instruction executed had an integer overflow setting the V condition code and indicates 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, MOVTUC, and BISPSW do not cause overflow even if they set V. Also note that the EMODx, CVTFx, and CVTDx floating-point instructions can cause integer overflow.

2.7.2.7.2 Integer Divide By Zero Trap -- An integer divide by zero trap is an exception that indicates 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.

2.7.2.7.3 Floating Overflow Trap -- A floating overflow trap is an exception that indicates the last instruction executed resulted in an exponent greater than 127 (unbiased) after normalization and rounding. The result stored contains a one in the sign and zeros in the exponent and fraction fields. This is a reserved operand and will cause a reserved operand fault if used in a subsequent floating-point instruction. The N and V condition code bits are set and Z and C are cleared. The type code pushed on the stack is 3.

2.7.2.7.4 Floating/Decimal Divide By Zero Trap -- A floating divide by zero trap is an exception that indicates the last instruction executed had a floating zero divisor. The result stored is the reserved operand (as described above for floating overflow trap) and the condition codes are set as in floating overflow.

A decimal string divide by zero trap is an exception that indicates the last instruction executed had a decimal string zero divisor. The destination and condition codes are UNPREDICTABLE. The zero divisor can be either +Ø or -Ø.

The type code pushed on the stack for both types of divide by zero is 4.

2.7.2.7.5 Floating Underflow Trap -- The floating underflow trap is an exception that indicates the last instruction executed resulted in an exponent less than -127 (unbiased) after normalization and rounding and that floating underflow was enabled (FU set). The result stored is zero. Except for POLYx, the N, V, and C condition codes are cleared and Z is set. In POLYx, the trap occurs on completion of the instruction, which may be many operations after the underflow. The condition codes are set on the final result in POLYx. The type code pushed on the stack is 5.

2.7.2.7.6 **Decimal String Overflow Trap** -- The decimal string overflow trap is an exception that indicates the last instruction executed had a decimal string result too large for the destination string provided and that decimal overflow was enabled (DV set). The V condition code is always set. The type code pushed on the stack is 6.

2.7.2.7.7 **Subscript Range Trap** -- A subscript range trap is an exception that indicates the last instruction was an INDEX instruction with a subscript operation 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 was within range. The type code pushed on the stack is 7.

2.7.2.8 **Microtraps** -- Detection of certain unusual conditions by the hardware will cause the microprogram to trap to a service flow. Initiation of the trap will cause the micro PC to be pushed on the microstack so that the microprogram can be continued after the condition is serviced. Multiple utrap conditions can occur at the same time. Therefore, the conditions are input to priority encoders. Refer to Figure 2-86. If any conditions are present, the UTRAP signal will be generated. This signal will force selection of BEN 1Ø by the branch logic. This branch enable (BEN 1Ø) will select the utrap vector (UTRAP VECT Ø3:ØØ) to form the low four bits of the next microaddress. Refer to section 2.3.2.3. The remaining address bits are hardwired to form the entire microtrap vector. This vector will point to the microcode service routine which can handle the condition.

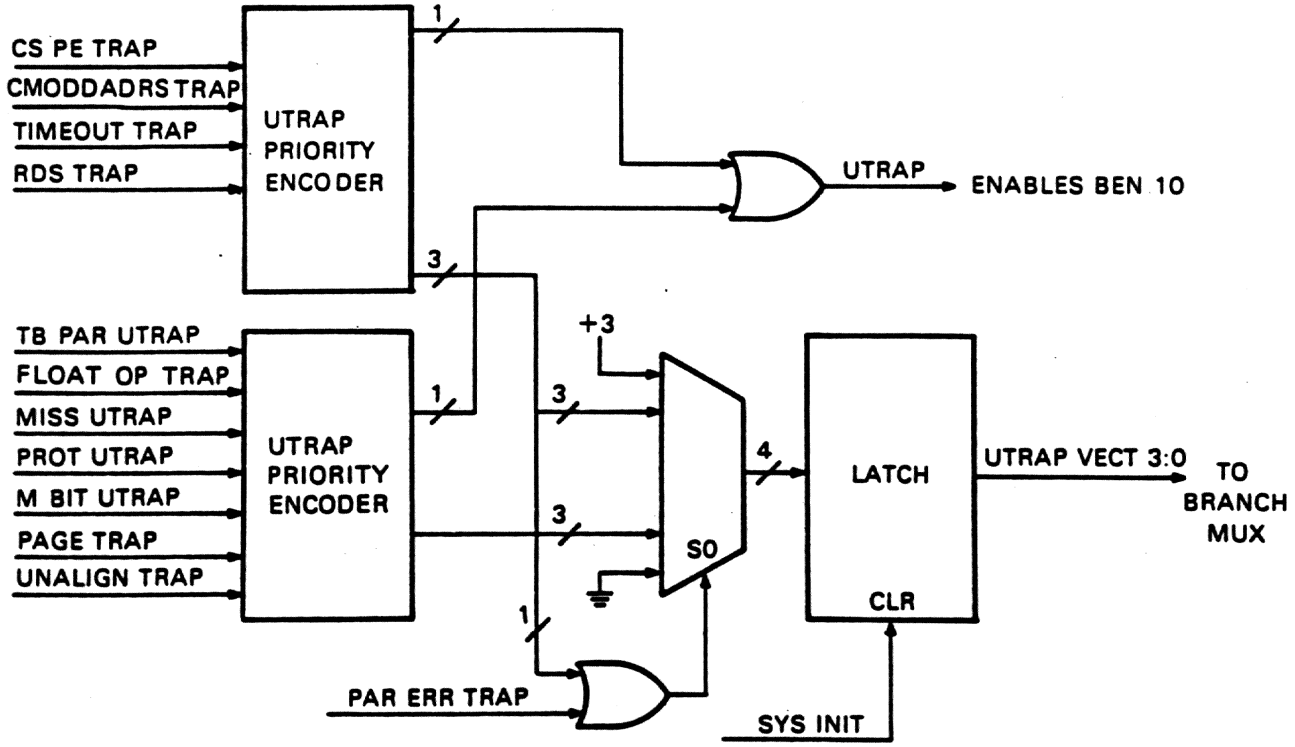


Figure 2-86 Microtrap Logic

TK-0610

The microtrap conditions and associated vectors are listed below in their relative priority.

	Condition	Vector
Highest	System Init	X1Ø0
	CS Parity Error	X1ØF
	Odd Address Error	X1ØE
	Read Timeout	X1ØD
	Read Data Substitute	X1ØC
	Cache Parity Error	X1Ø8
	Translation Buffer Parity Error	X1Ø7
	Reserved Floating Operand	X1Ø6
	Translation Buffer Miss	X1Ø5
	Protection Violation	X1Ø4
	Modify Bit (M bit)	X1Ø3
	Page Trap	X1Ø2
Lowest	Unaligned Data	X1Ø1
	If X = Ø, the vector is in PCS	
	If X = 1, the vector is in WDCS	

The following paragraphs provide a brief description of each of the microtrap error conditions:

SYSTEM INIT

The microtrap occurs as the result of DC LO being asserted in the processor or if DEAD is received from the SBI.

CS PARITY ERROR

This microtrap occurs when the microsequencer detects a parity error in the next microinstruction. This may cause a machine check abort at any time.

ODD ADDRESS

An Odd Address microtrap occurs when a 16-bit reference is made to an odd byte boundary in compatibility mode. The microtrap service routine performs an abort.

READ TIMEOUT

The Read Timeout microtrap occurs under two circumstances: (a) the bus control logic could not gain access to the SBI or the addressed location responded with BUSY for 512 cycles or (b) the bus control logic received a NO-RESPONSE confirmation indicating a non-existent address for 512 cycles.

READ DATA SUBSTITUTE	A Read Data Substitue microtrap occurs when the processor performs a read or interlock read on the SBI and memory has returned uncorrected read data.
CACHE PARITY ERROR	A Cache Parity Error microtrap occurs when a parity error is detected in Cache. The output of both groups of the address matrix and data matrix is parity checked as soon as a Cache reference is made.
TB PARITY ERROR	A TB Parity Error microtrap occurs when a parity error is detected in the TB. The information from both groups of the address matrix and data matrix is parity checked as soon as an address is sent to the TB matrices.
RESERVED FLOATING OPERAND	A Reserved Floating Operand microtrap occurs when the microword UMSC field equals 2 (CHK FLOAT OP) and ALU bit 15 equals 1 AND ALU bits 14:07 equal zeros.
TB MISS	A TB Miss microtrap occurs when a requested page table entry is not found in the TB. During the TB Miss microtrap service routine in the PTE is fetched from main memory and placed in the TB.
PROTECTION VIOLATION	A Protection Violation microtrap occurs if the current processor mode and/or intended page access violates the assigned protection for the page as dictated by the protection code of the PTE.
M BIT	A M Bit microtrap occurs when a write is attempted to a page whose PTE contains an unasserted M bit. During the microtrap service routine the M bit of the PTE is set in the TB and in memory. To accomplish this, the PTE in memory is fetched, modified, and rewritten.

PAGE BOUNDARY	A Page Boundary microtrap occurs when a cycle which crosses a page boundary is attempted. During the Page Boundary microtrap service routine, the intended access to the new page is checked before the cycle can be executed. This prevents the possibility of writing the first part of a data stream, after which the writing of the second part is prohibited (i.e., eliminates the possibility of half updated data).
UNALIGNED DATA	An Unaligned Data microtrap occurs when a reference is across a longword boundary. During the microtrap service routine, the microcode retrieves the portion of the data which was not part of the original longword.

2.7.3 Microword Control of Interrupts and Exception

Two fields of the microword, UIEK and UMSC, are used to control interrupts and exceptions. The following paragraphs provide a brief description of each field.

2.7.3.1 Interrupt and Exception Control (UIEK) Field -- The UIEK field (BUS CS 31:30) of the microword is used to monitor and acknowledge interrupt conditions. The field value specifies the following functions:

UIEK Field BUS CS 31	BUS CS 30	Function
0	0	NO-OP
0	1	Interrupt Strobe (ISTR)
1	0	Interrupt Acknowledge (IACK)
1	1	Exception Acknowledge (EACK)

The interrupt strobe (ISTR) function clocks the hardware interrupt register (HIR), thereby sampling the interrupt lines on levels 1E to 14. All interrupts that are detected at the time are prioritized. The interrupt strobe is usually enabled at the end of instructions prior to returning to the instruction decode state (IRD). If the priority of the interrupt is higher than the current IPL field in the PSL, an interrupt branch is performed at A fork in the microcode flow. During the execution of long iterative instructions, the ISTR function is used to periodically monitor the interrupt conditions and to allow a subsequent branch to be performed.

The interrupt acknowledge (IACK) function is used to clear the power fail, console terminal receive and console terminal interrupts when they are being serviced by the microcode routine.

The exception acknowledge (EACK) function is used to clear pending arithmetic traps after they have been serviced or when other exceptions occur.

The IACK function and the EACK function set the processor status longword to the following predetermined state:

PSL Bit Position	Name	IACK Function	EACK Function
31	CMP	Ø	Ø
30	TP	Ø	Ø
29:28	Ø	Ø	Ø
27	FPD	Ø	Ø
26	IS	IS	IS
25:24	CUR MOD	Ø	Ø
23:22	PREV MODE	Ø	CUR MODE
21	Ø	Ø	Ø
20:16	IPL	IPL ACT	IPL
15:08	Ø	Ø	Ø
07	DV	Ø	Ø
06	FU	Ø	Ø
05	IV	Ø	Ø
04	T	Ø	Ø
03	N	Ø	Ø
02	Z	Ø	Ø
01	V	Ø	Ø
00	C	Ø	Ø

2.7.3.2 Miscellaneous (UMSC) Field -- The UMSC field (BUS CS 29:26) of the microword provides a variety of functions, some of which affect the generation of exception conditions. The following provides a brief description of each function specified by the associated field value.

UMSC Field (Hex value)	Function	Description
Ø	NO-OP	
1	CHK CHMX INSTR	Loads the CUR MOD bits into the PRV MOD register. If the new data specified by the change mode instruction is greater than the CUR MOD in the PSL, this function prevents loading of the CUR MOD bits.
2	CHK FLOAT OP	Causes a utrap before execution of the next microinstruction if ALU bit 15 equals 1 AND ALU bits 14:Ø7 equal Ø.
3	CHK ODD ADDRS	Causes a utrap before execution of the next microinstruction if in compatibility mode AND VAØØ = 1 AND enabled by memory cycle.
4	unused	
5	LOAD STATE	Loads contents of state register into EAMX and enableS load function on state register.
6	LOAD ACC COND. CODES	Loads the PSL condition codes from the ACC condition codes, clocked on the previous microinstruction.
7	READ RLOG	Loads the RLOG and PCSV inputs into the BMX and decrements the RLOG pointer at the end of the microinstruction.

8	IRD STATE	Used to indicate that a new macroinstruction has begun to be evaluated. Causes the RLOG pointer to be initialized, TP to be loaded from PSL T, and HP to be loaded from the halt request signal.
9	unused	
A	CLR NESTED ERROR	Clears the nested error flag in the CPU Error/Status register.
B	SET NESTED ERROR	Sets the nested error flag in the CPU Error/Status register.
C	SEC REF, INH TRAPS, SAVED CTX	Used in conjunction with the UMCT and UADS fields to further specify a memory reference;
D	INH TRAPS, SAVED CTX	
E	SAVED CTX	SEC REF -- used to generate the second part of a byte or load mask when referencing unaligned data. INH TRAPS -- prevents a page and unalign utrap from occurring. SAVED CTX -- specifies the data type information saved by a previously specified UMCT code be used to generate the byte and load masks. Also used for the detection of odd address, page boundary, and unaligned data utraps.
F	INH COMPATIBILITY MODE	Prevents the PSL CMP bit from forcing VAMX 31:16 to zeros. Also inhibits the odd address utrap.

2.8 SYSTEM CLOCKS

The VAX-11/780 system contains three clocks: the processor clock, the time of year clock, and the interval time clock. Each of these clocks is described in the following paragraphs.

2.8.1 Processor Clock

The processor clock (Figure 2-87) provides the circuitry required for the generation of SBI timing signals, decoding of SBI signals and distribution to the processor modules, and power up/power fail sequencing.

The synchronous operation of the VAX-11/780 is based on a clock cycle of 200 ns. There are four 50 ns time states per cycle (T0, T1, T2, and T3). The CPU and SBI time states are derived from SBI signals called TP (timing pulse), PCLK (clock phase) and PDCLK (clock phase delayed). Refer to Paragraph 2.8.1.3 for the relationship between the SBI signals and the derived SBI and CPU time states.

2.8.1.1 Frequency Selection -- The clock frequency of the processor can be selected from three internal oscillators and one external oscillator. Frequency selection is dependent on the value of two bits in the machine control register (MCR bits 04 and 03) located on the console interface board. These two bits (FR1 and FR0) are set by LSI software to control the clock frequency as follows:

FR1 (bit 04)	FR0 (bit 03)	Frequency
0	0	10 Mhz (normal)
0	1	10.525 (5% short)
1	0	8.925 (12% long)
1	1	External Source

2.8.1.2 Start/Stop/Step Control Logic -- The console generates four signals which are used to control operation of the processor clock. The control signal synchronizer latches and synchronously clocks these signal lines prior to being input to the start/stop/step control logic. The following lists the four signals and their function:

Signal Name	Function
PROCEED L	Proceed
STS L	Single Time State
SBC L	Single Bus Cycle
SOMM L	Stop on Microbreak Match

The control logic uses the oscillator frequencies in conjunction with the console signals to generate GATEDCLK H and GATEDCLK L. These two major timing signals are then used by the sequence generator to produce the SBI timing signals.

As previously mentioned, four bits of the machine control register (MCR) are set or cleared by LSI-11 software to start, stop, or step the processor clock. The following paragraphs briefly describe each of these bits and their effect on clock operation.

MCR Bit 06, Stop on Microbreak Match (SOMM) -- In maintenance mode, the console can stop the processor clock at a specific microaddress by writing that address into the micro PC break register (via the ID bus). If the SOMM bit is set by the LSI-11, the clock will stop in CPT 0 of the cycle in which the contents of the micro PC break register matches the micro PC.

MCR Bit 02, Single Time State (STS) -- The STS bit, when set, will stop the clock in any one of the four time states. If the STS bit is set, the clock can be stepped one time state by writing a 1 to the proceed bit.

MCR Bit 01, Single Bus Cycle (SBC) -- If the LSI-11 sets the SBC bit and the STS bit is 0, the clock will stop at CPT 0. Also, if the SBC bit is set and the STS bit is 0, the clock can be stepped by one clock cycle (i.e., to the next CPT0) by writing a 1 to the proceed bit.

MCR BIT 00, Proceed (PROCEED) -- Writing a 1 into the proceed bit will affect the clock in any one of three ways depending on the states of the STS and SBC bits, shown as follows. Note that writing a 1 to the proceed bit while the clock is running will have no effect.

PROCEED	STS	SBC	Effect on Clock
	0	0	start clock running
	0	1	step clock one cycle
	1	0	step clock one time state
	1	1	step clock one time state

2.8.1.3 Processor Clock Timing Diagrams -- The following figures illustrate the SBI timing (Figure 2-88), CPU timing (Figure 2-89), SBI Clock Power Up Sequencer Timing (Figure 2-90), and CPU Power Fail Sequencer Timing (Figure 2-91).

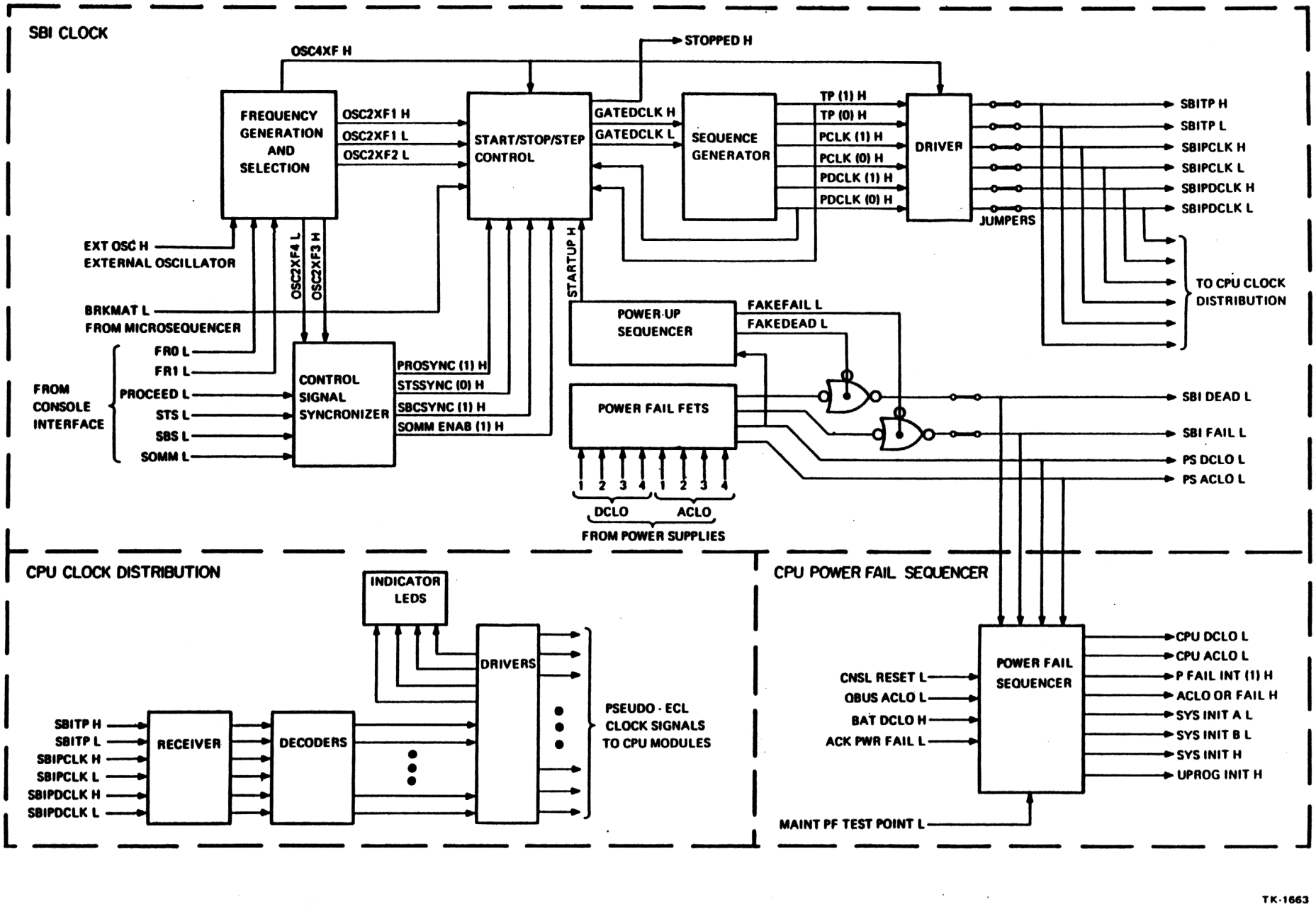


Figure 2-87 Processor
Clock Block Diagram

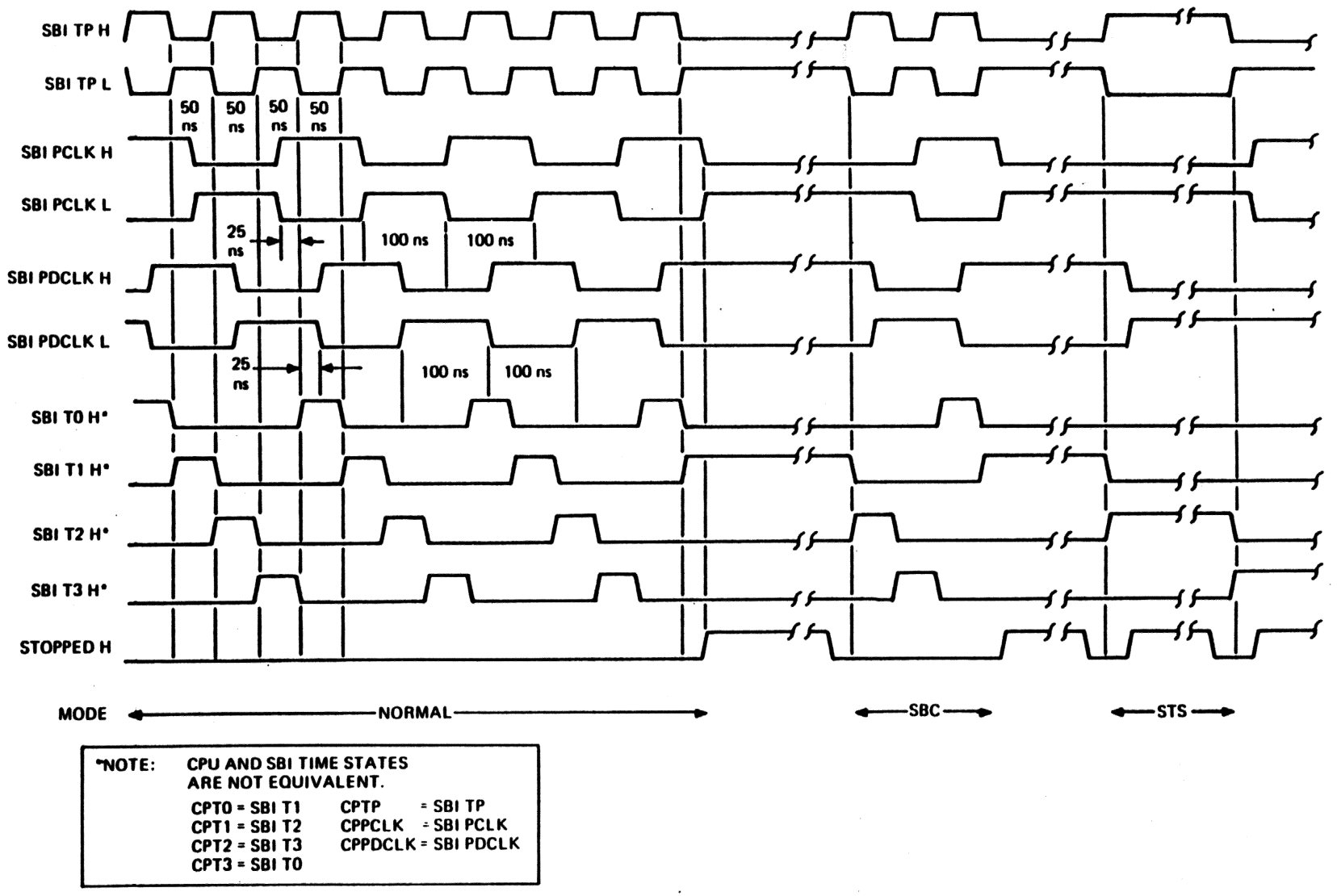


Figure 2-88 SBI Timing

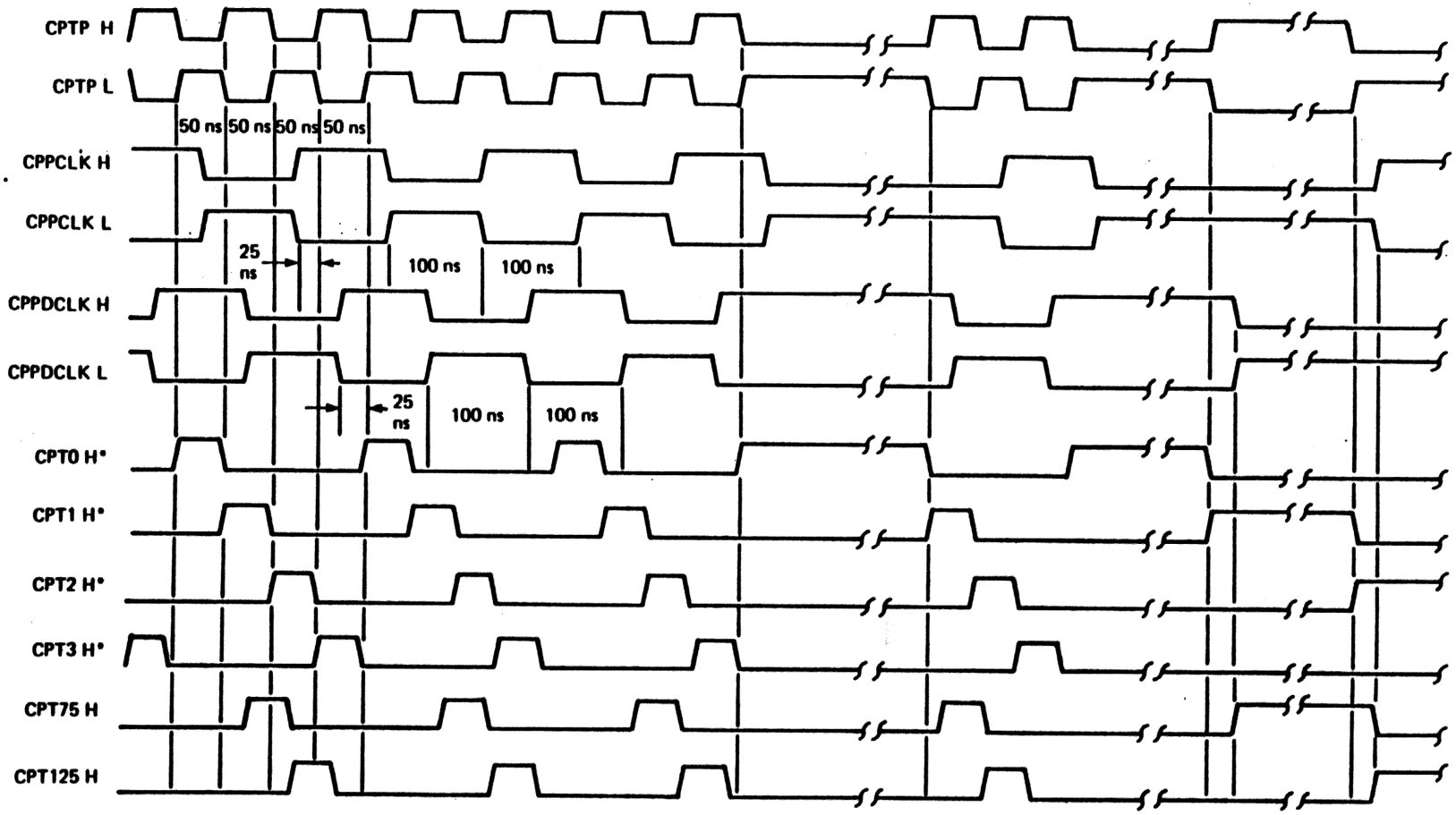


Figure 2-89 CPU Timing

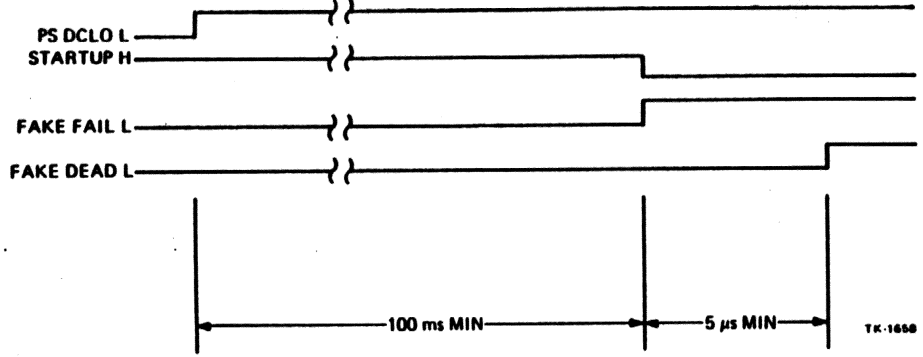
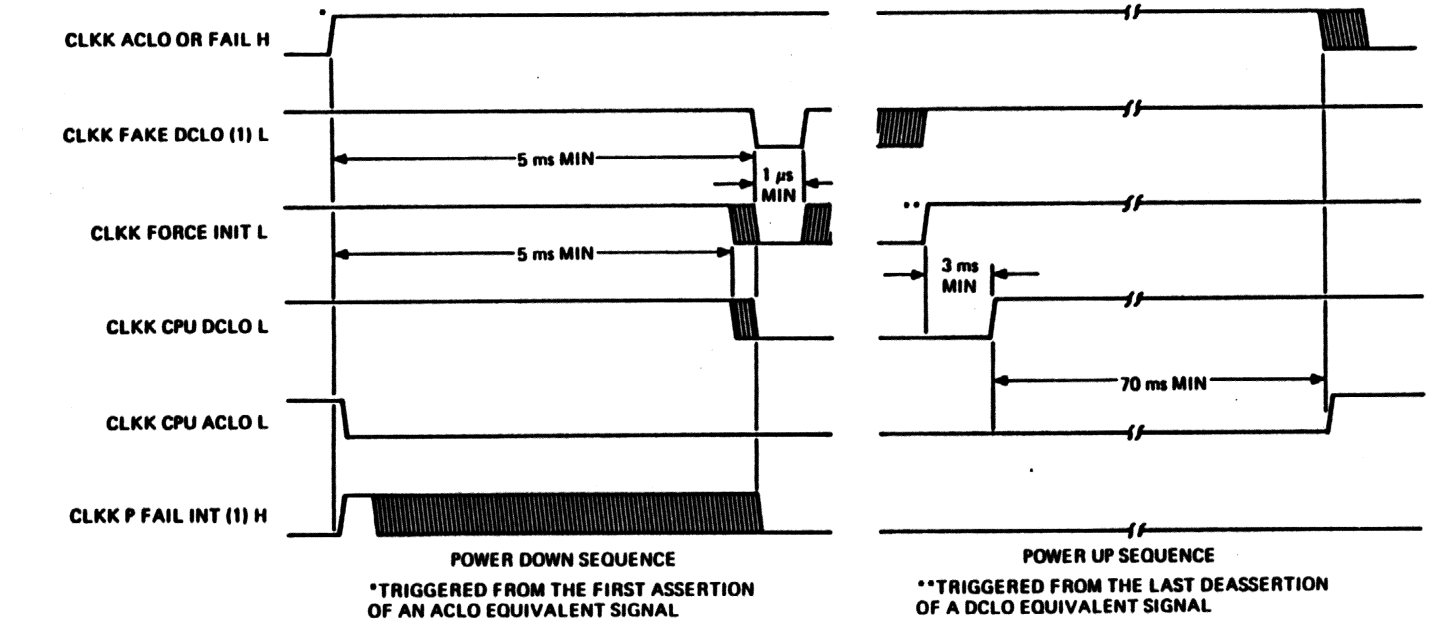


Figure 2-90 SBI Clock
Power Up Sequencer Timing



ACLO EQUIVALENT SIGNALS:
CLKR PS ACLO L
BUS SBI FAIL L
CIBV QBUS ACLO L
IRCM BAT DCLO H
CLKL MAINT PF (1) H

DCLO EQUIVALENT SIGNALS:
CLKR PS DCLO L
BUS SBI DEAD L
CLKK FAKE DCLO (1) L

TK-1659

Figure 2-91 CPU Power
Fail Sequencer Timing

2.8.2 Time of Year Clock

The time of year clock is used by software to perform various timekeeping functions. Its primary purpose is to provide the correct time to the system after power failures. This feature eliminates the need for an operator to enter the time at system restart.

The time of year clock (Figure 2-92) is physically located on the instruction decode board (M8224) and is accessible to software via the MTPR and MFPR instructions. The time of year register is a binary up counter that counts at a 100 Hz rate. This count frequency for the register gives a range of 497.1 days.

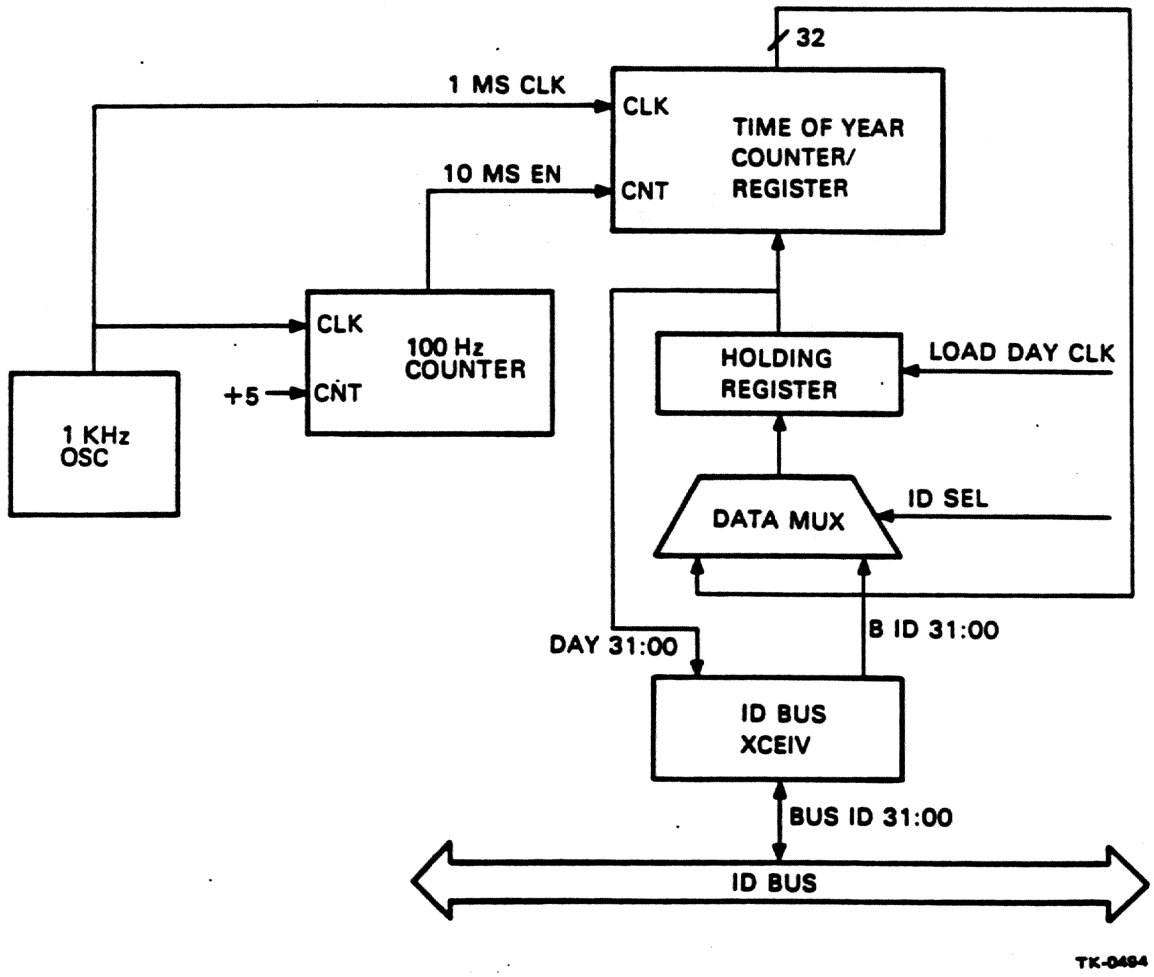


Figure 2-92 Time of Year Clock

The 32 bit time of year register can be read or written via the Internal Data (ID) bus. The data is actually read from or written to a holding register. In a write operation, the incoming data from the ID bus is latched into the holding register. The data is loaded into the time of year register after the 1 kHz clock has synchronized to the ID write of the holding register.

In a read operation, a special microprogram flow is required because of the very slow switching speed of CMOS. In order to solve the problem of synchronizing the 100 Hz counter to the CPU, the microprogram will read the time of register twice and compare the two values. If they are the same, the value is sent to the macro software. If they are different, the microprogram will continue testing until the values are the same.

The software will convert the time that is input by the operator into a binary number that represents that particular month, day, hour, etc. This number increases as time elapses. When the time of year register is read, the software will convert the current value to the appropriate form for output. At the end of each year, the software will reset the clock to the beginning of the year value.

2.8.3 Interval Time Clock

The interval time clock enables the accurate measurement of variable time intervals. The interval time clock notifies the processor of a completed time interval via an interrupt. This hardware feature enables software to perform time dependent events, accounting, and maintenance of software data and time.

There are three registers required for the operation of the interval timer. Each of these registers is accessible through the MTPR and MFPR instructions. Register data is transferred over the internal data (ID) bus under control of the microprogram. The following paragraphs provide a brief description of each of the registers:

Interval Count Register -- This is a 32-bit up counter that increments at the rate of 1 microsecond per count, when enabled by the RUN control bit. This register is read only.

Next Interval Register -- This is a 32-bit register that contains a value to be loaded into the interval count register each time the count register overflows. This register is write only.

Clock Control Status Register -- This register (Figure 2-93) contains six bits which include status information, control functions and maintenance functions.

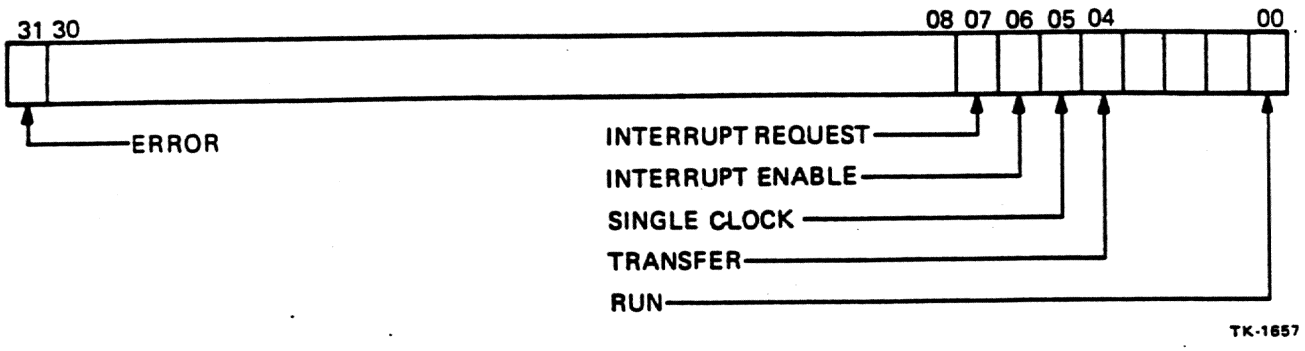


Figure 2-93 Interval Clock Control Status Register

The following provides a description of each of the bits in the control status register:

Bit	Name	Function
31	ERROR	This bit is set when a second overflow of the interval count register occurs before the first overflow has been serviced. It is cleared on power up and by writing a one to bit 31 of the control register.
07	INTERRUPT REQUEST	This bit is set when the interval count register overflows. It is cleared on power up or by writing a one to bit 07 of the control register.
06	INTERRUPT ENABLE	This bit, when set, allows an interrupt at IPL 24.
05	SINGLE CLOCK	This bit is used as a maintenance aid. Writing a one to bit 05 will advance the count register by one. This bit is always read as a zero.
04	TRANSFER	When a one is written to bit 04, it causes the contents on the next interval register to be transferred to the interval count register. This operation can be performed independent of the state of the run bit. This bit is always read as a zero.
00	RUN	This bit, when set, allows the counter to count. It is cleared on power up and can be read or written under program control.

2.8.3.1 Operation -- On power up, all the bits in the clock control status register are cleared. Since the run bit is cleared, the counter will not change. The next interval register is then loaded with a value corresponding to the 2's complement of the number of microseconds between interrupts. Next, the run, interrupt enable, and transfer bits are set in the control status register. This will cause the next interval register to be loaded into the interval count register and will cause the interval count register to start counting.

When the interval count register counts from all ones to the next state, an interrupt at IPL 24 is requested. At the same time, the count register is loaded from the next interval register and counting is continued from the next interval value.

If the interval count register overflows again before the previous interrupt is serviced, an error flag is set. The hardware will continue to request an interrupt at IPL 24 and software will clear the error flag when the interrupt is serviced.

APPENDIX A
OP CODE LISTING

OPCODE	MNEMONIC	INSTRUCTION
00	HALT	Halt
01	NOP	No operation
02	REI	Return from exception or interrupt
03	BPT	Break point fault
04	RET	Return from called procedure
05	RSB	Return from subroutine
06	LDPCTX	Load process context
07	SVPCTX	Save process context
08	CVTPS	Convert packed to leading separate numeric
09	CVTSP	Convert leading separate numeric to packed
OA	INDEX	Compute index
OB	CRC	Calculate cyclic redundancy check
OC	PROBER	Probe read access
OD	PROBEW	Probe write access
OE	INSQUE	Insert into queue
OF	REMQUE	Remove from queue
10	BSBB	Branch to subroutine with byte displacement
11	BRB	Branch with byte displacement
12	BNEQ, BNEQU	Branch on not equal unsigned, Branch on not equal
13	BEQL, BEQLU	Branch on equal, Branch on equal unsigned
14	BGTR	Branch on greater
15	BLEQ	Branch on less or equal
16	JSB	Jump to subroutine
17	JMP	Jump
18	BGEQ	Branch on greater or equal
19	BLSS	Branch on less
1A	BGTRU	Branch on greater unsigned
1B	BLEQU	Branch on less or equal unsigned
1C	BVC	Branch on overflow clear
1D	BVS	Branch on overflow set
1E	BGEQU, BCC	Branch on greater or equal unsigned, Branch on carry clear
1F	BLSSU, BCS	Branch on less unsigned, Branch on carry set
20	ADDP4	Add packed 4 operand
21	ADDP6	Add packed 6 operand
22	SUBP4	Subtract packed 4 operand
23	SUBP6	Subtract packed 6 operand
24	CVTPT	Convert packed to trailing numeric
25	MULP	Multiply packed

OPCODE	MNEMONIC	INSTRUCTION
26	CVTTP	Convert trailing numeric to packed
27	DIVP	Divide packed
28	MOVC3	Move character 3 operand
29	CMPC3	Compare character 3 operand
2A	SCANC	Scan for character
2B	SPANC	Span characters
2C	MOVC5	Move character 5 operand
2D	CMPC5	Compare character 5 operand
2E	MOVTC	Move translated characters
2F	MOVTUC	Move translated until character
30	BSBW	Branch to subroutine with word displacement
31	BRW	Branch with word displacement
32	CVTWL	Convert word to long
33	CVTWB	Convert word to byte
34	MOVP	Move packed
35	CMPP3	Compare packed 3 operand
36	CVTPL	Convert packed to long
37	CMPP4	Compare packed 4 operand
38	EDITPC	Edit packed to character
39	MATCHC	Match characters
3A	LOCC	Locate character
3B	SKPC	Skip character
3C	MOVZWL	Move zero-extended word to long
3D	ACBW	Add compare and branch word
3E	MOVAW	Move address of word
3F	PUSHAW	Push address of word
40	ADDF2	Add floating 2 operand
41	ADDF3	Add floating 3 operand
42	SUBF2	Subtract floating 2 operand
43	SUBF3	Subtract floating 3 operand
44	MULF2	Multiply floating 2 operand
45	MULF3	Multiply floating 3 operand
46	DIVF2	Divide floating 2 operand
47	DIVF3	Divide floating 3 operand
48	CVTFB	Convert float to byte
49	CVTFW	Convert float to word
4A	CVTFL	Convert float to long
4B	CVTRFL	Convert rounded float to long
4C	CVTBF	Convert byte to float
4D	CVTWF	Convert word to float
4E	CVTLF	Convert long to float
4F	ACBF	Add compare and branch floating
50	MOVF	Move float
51	CMPF	Compare floating
52	MNEGF	Move negated floating

OPCODE	MNEMONIC	INSTRUCTION
53	TSTF	Test float
54	EMODF	Extended modulus floating
55	POLYF	Evaluate polynomial floating
56	CVTFD	Convert float to double
57		RESERVED to DEC
58	ADAWI	Add aligned word interlocked
59		RESERVED to DEC
5A		RESERVED to DEC
5B		RESERVED to DEC
5C		RESERVED to DEC
5D		RESERVED to DEC
5E		RESERVED to DEC
5F		RESERVED to DEC
60	ADDD2	Add double 2 operand
61	ADDD3	Add double 3 operand
62	SUBD2	Subtract double 2 operand
63	SUBD3	Subtract double 3 operand
64	MULD2	Multiply double 2 operand
65	MULD3	Multiply double 3 operand
66	DIVD2	Divide double 2 operand
67	DIVD3	Divide double 3 operand
68	CVTDB	Convert double to byte
69	CVTDW	Convert double to word
6A	CVTDL	Convert double to long
6B	CVTRDL	Convert rounded double to long
6C	CVTBD	Convert byte to double
6D	CVTWD	Convert word to double
6E	CVTLD	Convert long to double
6F	ACBD	Add compare and branch double
70	MOVD	Move double
71	CMPD	Compare double
72	MNEGD	Move negated double
73	TSTD	Test double
74	EMODD	Extended modulus double
75	POLYD	Evaluate polynomial double
76	CVTDF	Convert double to float
77		RESERVED to DEC
78	ASHL	Arithmetic shift long
79	ASHQ	Arithmetic shift quad
7A	EMUL	Extended multiply
7B	EDIV	Extended divide
7C	CLRQ, CLRD	Clear quad, Clear double
7D	MOVQ	Move quad
7E	MOVAQ, MOVAD	Move address of quad, Move address of double
7F	PUSHAQ, PUSHAD	Push address of quad, Push address of double

OPCODE	MNEMONIC	INSTRUCTION
80	ADDB2	Add byte 2 operand
81	ADDB3	Add byte 3 operand
82	SUBB2	Subtract byte 2 operand
83	SUBB3	Subtract byte 3 operand
84	MULB2	Multiply byte 2 operand
85	MULB3	Multiply byte 3 operand
86	DIVB2	Divide byte 2 operand
87	DIVB3	Divide byte 3 operand
88	BISB2	Bit set byte 2 operand
89	BISB3	Bit set byte 3 operand
8A	BICB2	Bit clear byte 2 operand
8B	BICB3	Bit clear byte 3 operand
8C	XORB2	Exclusive OR byte 2 operand
8D	XORB3	Exclusive OR byte 3 operand
8E	MNEGB	Move negated byte
8F	CASEB	Case byte
90	MOVB	Move byte
91	CMPB	Compare byte
92	MCOMB	Move complemented byte
93	BITB	Bit test byte
94	CLRB	Clear byte
95	TSTB	Test byte
96	INCB	Increment byte
97	DECB	Decrement byte
98	CVTBL	Convert byte to long
99	CVTBW	Convert byte to word
9A	MOVZBL	Move zero-extended byte to long
9B	MOVZBW	Move zero-extended byte to word
9C	ROTL	Rotate long
9D	ACBB	Add compare and branch byte
9E	MOVAB	Move address of byte
9F	PUSHAB	Push address of byte
AO	ADDW2	Add word 2 operand
A1	ADDW3	Add word 3 operand
A2	SUBW2	Subtract word 2 operand
A3	SUBW3	Subtract word 3 operand
A4	MULW2	Multiply word 2 operand
A5	MULW3	Multiply word 3 operand
A6	DIVW2	Divide word 2 operand
A7	DIVW3	Divide word 3 operand
A8	BISW2	Bit set word 2 operand
A9	BISW3	Bit set word 3 operand
AA	BICW2	Bit clear word 2 operand
AB	BICW3	Bit clear word 3 operand
AC	XORW2	Exclusive OR word 2 operand
AD	XORW3	Exclusive OR word 3 operand

OPCODE	MNEMONIC	INSTRUCTION
AE	MNEGW	Move negated word
AF	CASEW	Case word
B0	MOVW	Move word
B1	CMPW	Compare word
B2	MCOMW	Move complemented word
B3	BITW	Bit test word
B4	CLRW	Clear word
B5	TSTW	Test word
B6	INCW	Increment word
B7	DECW	Decrement word
B8	BISPSW	Bit set processor status word
B9	BICPSW	Bit clear processor status word
BA	POPR	Pop registers
BB	PUSHR	Push register
BC	CHMK	Change mode to kernel
BD	CHME	Change mode to executive
BE	CHMS	Change mode to supervisor
BF	CHMU	Change mode to user
C0	ADDL2	Add long 2 operand
C1	ADDL3	Add long 3 operand
C2	SUBL2	Subtract long 2 operand
C3	SUBL3	Subtract long 3 operand
C4	MULL2	Multiply long 2 operand
C5	MULL3	Multiply long 3 operand
C6	DIVL2	Divide long 2 operand
C7	DIVL3	Divide long 3 operand
C8	BISL2	Bit set long 2 operand
C9	BISL3	Bit set long 3 operand
CA	BICL2	Bit clear long 2 operand
CB	BICL3	Bit clear long 3 operand
CC	XORL2	Exclusive OR long 2 operand
CD	XORL3	Exclusive OR long 3 operand
CE	MNEGL	Move negated long
CF	CASEL	Case long
D0	MOVL	Move long
D1	CMPL	Compare long
D2	MCOML	Move complemented long
D3	BITL	Bit test long
D4	CLRL, CLRF	Clear long, Clear float
D5	TSTL	Test long
D6	INCL	Increment long
D7	DECL	Decrement long
D8	ADWC	Add with carry
D9	SBWC	Subtract with carry
DA	MTPR	Move to processor register
DB	MFPR	Move from processor register

OPCODE	MNEMONIC	INSTRUCTION
DC	MOVPSL	Move processor status longword
DD	PUSHL	Push long
DE	MOVAL, MOVAF	Move address of long, Move address of float
DF	PUSHAL, PUSHAF	Push address of long, Push address of float
E0	BBS	Branch on bit set
E1	BBC	Branch on bit clear
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
E6	BBSSI	Branch on bit set and set interlocked
E7	BBCCI	Branch on bit clear and clear interlocked
E8	BLBS	Branch on low bit set
E9	BLBC	Branch on low bit clear
EA	FFS	Find first set bit
EB	FFC	Find first clear bit
EC	CMPV	Compare field
ED	CMPZV	Compare zero-extended field
EE	EXTV	Extract field
EF	EXTZV	Extract zero-extended field
F0	INSV	Insert field
F1	ACBL	Add compare and branch long
F2	AOBLSS	Add one and branch on less
F3	AOBLEQ	Add one and branch on less or equal
F4	SOBGEQ	Subtract one and branch on greater or equal
F5	SOBGTR	Subtract one and branch on greater
F6	CVTLB	Convert long to byte
F7	CVTLW	Convert long to word
F8	ASHP	Arithmetic shift and round packed
F9	CVTLP	Convert long to packed
FA	CALLG	Call with general argument list
FB	CALLS	Call with stack
FC	XFC	Extended function call
FD	ESCD to DEC	
FE	ESCE to DEC	
FF	ESCF to DEC	