pdp11

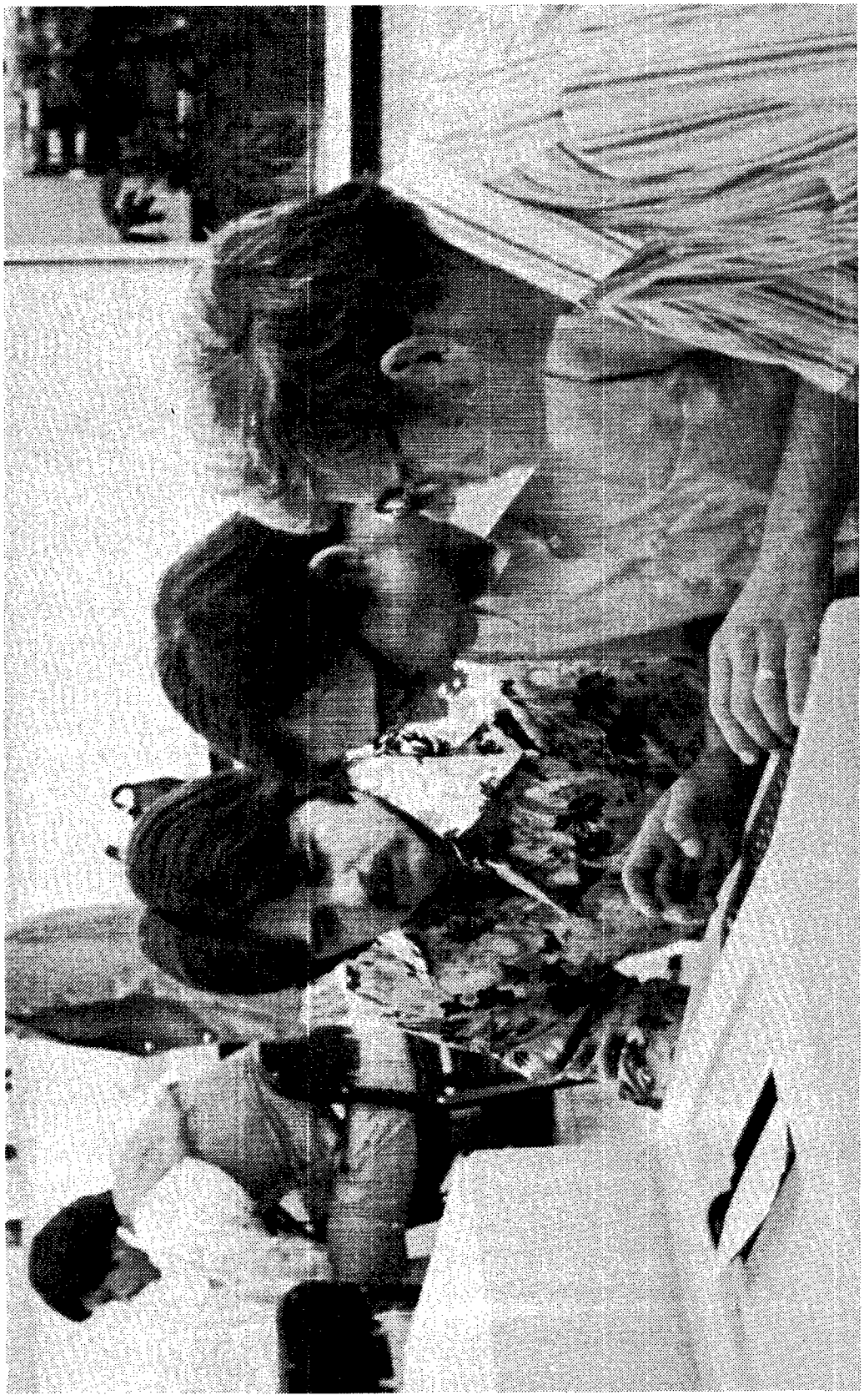
processor handbook

pdp11/04/34a/44/60/70

digital

This handbook was designed, produced and typeset
by DIGITAL's Sales Support Literature Group
using an In-house text-processing system
operating on a DECSYSTEM-20.

CONTENTS



CHAPTER 1
INTRODUCTION

DIGITAL's 11 family of interactive computers ranges in size from the single-board LSI-11 through the extensive PDP-11 group. Development efforts are constantly expanding both ends of the spectrum, as well as creating enhanced products in the PDP-11 price-versus-performance matrix.

The processors specifically discussed in this handbook are:

- PDP-11/04
- PDP-11/34A
- PDP-11/44
- PDP-11/60
- PDP-11/70

PDP-11 processors are a family based on common architecture. Compatibility is inherent in design, and is reflected in the software and in the peripheral options. It is possible, for example, to develop programs on the smallest PDP-11 family member, the PDP-11/03, and, with only slight modifications, run them on any other PDP-11 system. Peripherals such as video terminals and line printers are equally upward and downward compatible in their ability to interface with PDP-11 family members.

The processors which are discussed specifically in this book have one outstanding characteristic in common: they all process data on a data bus called the UNIBUS.

The UNIBUS, (discussed in detail in Chapter 2), was first announced by DIGITAL in 1970, in conjunction with the announcement of the first PDP-11, the PDP-11/20. The UNIBUS and its unique capabilities have provided the flexibility and growth options for the PDP-11 family members discussed in this handbook. Figure 1-1 illustrates the major categories of PDP-11 processors. Figure 1-2 illustrates the block structure of the PDP-11.

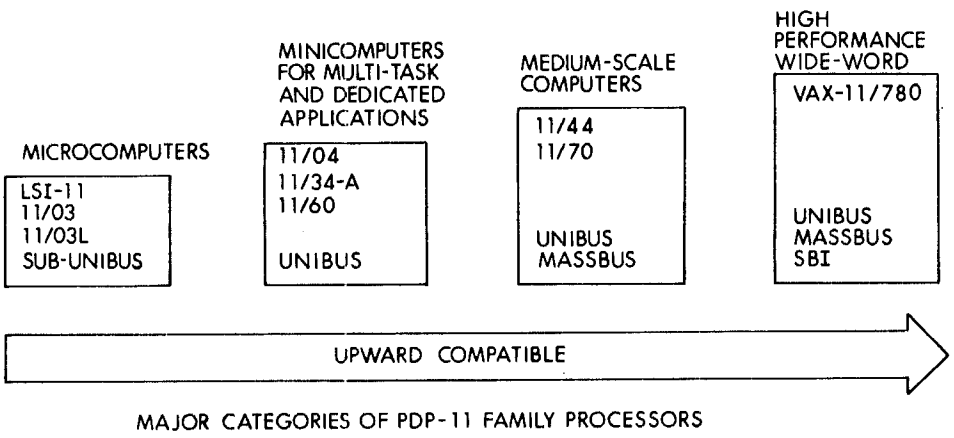


Figure 1-1 Major Categories of PDP-11 Processors

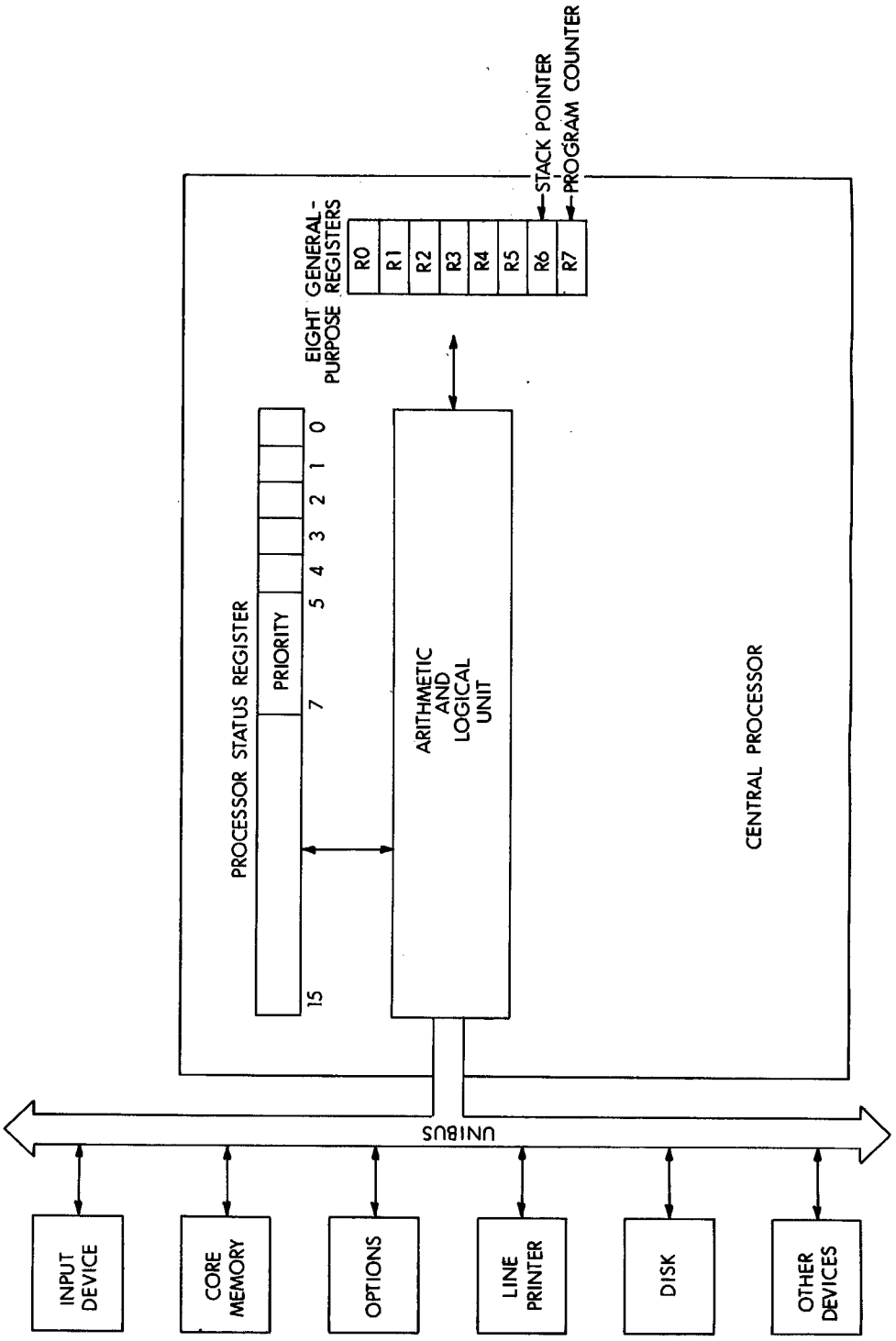


Figure 1-2 PDP-11 Block Structure

Beyond the UNIBUS commonality, each PDP-11 processor has features and capabilities uniquely suited for various applications. Some functionally similar features have been accomplished with different implementations. Therefore, there is some repetition of information in the chapters describing the individual processor members of the PDP-11 family. It is often necessary to discuss each separately because what may appear to be very subtle differences in operations may actually be key to a certain processor's uniqueness.

PROGRAMMING THE PDP-11

Information is provided in this handbook about the assembly language parameters, processes, and techniques involved in programming the PDP-11. DIGITAL publishes tutorial software documentation that provides detailed information about using the PDP-11 instruction set to develop programs. There are also well-developed courses for customers given by DIGITAL's Education Services group.

The material presented on the PDP-11 instruction set, addressing modes, and on programming techniques is intended, with the examples included, to illustrate the range of and possibilities for program development. A companion book, the PDP-11 Software Handbook, clearly explains the operating systems and associated software which run on the PDP-11 family of processors. Table 1 illustrates these software products.

Table 1 PDP-11 Operating Systems

Name	Description
RT-11	Real-Time Operating System for PDP-11 Processors. A small, single-user foreground/background system that can support a real-time application job's execution in the foreground and an interactive or batch program development job in the background.
DSM-11	DIGITAL Standard MUMPS Operating System for PDP-11 Processors. A small- to large-size timesharing system that offers a unique fast access data storage and retrieval system for large data base processing.
RSTS/E	Resource-Sharing Timesharing System/Extended Operating System for PDP-11 Processors.

Name	Description
	A moderate- to large-size timesharing system that can support up to 63 concurrent jobs, including interactive terminal user jobs, detached jobs, and batch processing.
RSX-11M and RSX-11M-PLUS	Real-Time Multiprogramming Executive Operating System for PDP-11 Processors. A small- to moderate-sized real-time multiprogramming system that can be generated for a wide range of application environments—from small, dedicated systems to large, multipurpose real-time application and program development systems.
RSX-11S	Real-Time Multiprogramming Executive Operating System for PDP-11 Processors. A small, execute-only member of the RSX-11 family for dedicated real-time multiprogramming applications (requires a host RSX-11M, RSX-11M-PLUS, VAX/VMS system).
IAS	Interactive Application System for PDP-11 Processors. A large multi-user operating system, allowing real-time applications execution concurrent with time-shared interactive and batch processing.
TRAX	A dedicated high-volume transaction processing system offering real-time and batch in a multi-user commercial environment.

In each chapter describing the operating systems, the PDP-11 Software Handbook includes: a general description of the requirements for the system, the monitor/executive characteristics, the file structures and data handling facilities, the user interfaces, the programmed monitor services, the system utilities, and the language processors supported.

PERIPHERALS

DIGITAL manufactures a full range of peripheral equipment designed to meet specific needs as well as to maintain PDP-11 family compatibility. I/O and storage devices range from cassette tape devices

through high volume disk packs, and from the DECwriter to the intelligent terminals which provide both hard copy and video display. There is a complete spectrum of peripheral devices available to complement the software, and to provide the complete answer to customer needs in all market areas—business, education, industry, laboratory, and engineering.

The Peripherals Handbook and the Terminals and Communications Handbook describe in detail the optional equipment available for use with the PDP-11 family members.

SPECIALIZED SYSTEMS

DIGITAL's Computer Special Systems (CSS) and OEM (Original Equipment Manufacturers) groups can provide the exact hardware and software combination to fill any customer need. Software Services provides software consultation services for customers who have specialized application software needs.

PACKAGE SYSTEMS

DIGITAL's Package Systems program offers you the opportunity to purchase a well-defined, pretested, hardware/software system, rather than purchasing the options separately. Package systems are fully equipped PDP-11 configurations including operating system, disk storage and loading device. Entry level systems consist of the correct minimum set of options defined in the *Software Product Description (SPD)* as necessary to run the operating system. Medium and high performance systems have expanded configurations that in some cases substantially exceed minimum SPD requirements. Package systems are available for all of DIGITAL's major operating systems. The introductory family of systems represents the combined effort of the product lines and of central engineering to offer the best set of systems to meet customer application needs. Package systems are priced less than the sum of the individual options. Figure 1-3 illustrates the combinations (shaded portions) of options currently available under the Packaged Systems program.

PDP-11 FAMILY PACKAGE SYSTEMS

CPU / OS (SPD VERSION)	04	34A	44	60	70
RT-11 (VERSION 03B)			▨		▨
DSM-11 (VERSION 1.0)	▨		▨		
RSX-11M (VERSION 3.2)					
RSX-11M PLUS (VERSION 1.0)	▨	▨		▨	
IAS (VERSION 3.0)	▨		▨		
RSTS-E (VERSION 7.0)	▨				
TRAX (VERSION 1)	▨		▨	▨	

□ OPTIONS CURRENTLY AVAILABLE UNDER THE PACKAGE SYSTEMS PROGRAM

▨ OPTIONS NOT AVAILABLE UNDER PACKAGE SYSTEMS PROGRAM

FUTURE PDP-11/44 PACKAGE SYSTEMS WILL BE CONFIGURED WITH THE NEXT VERSION OF THE DSM-11 AND TRAX OPERATING SYSTEMS.

Figure 1-3 Package Systems

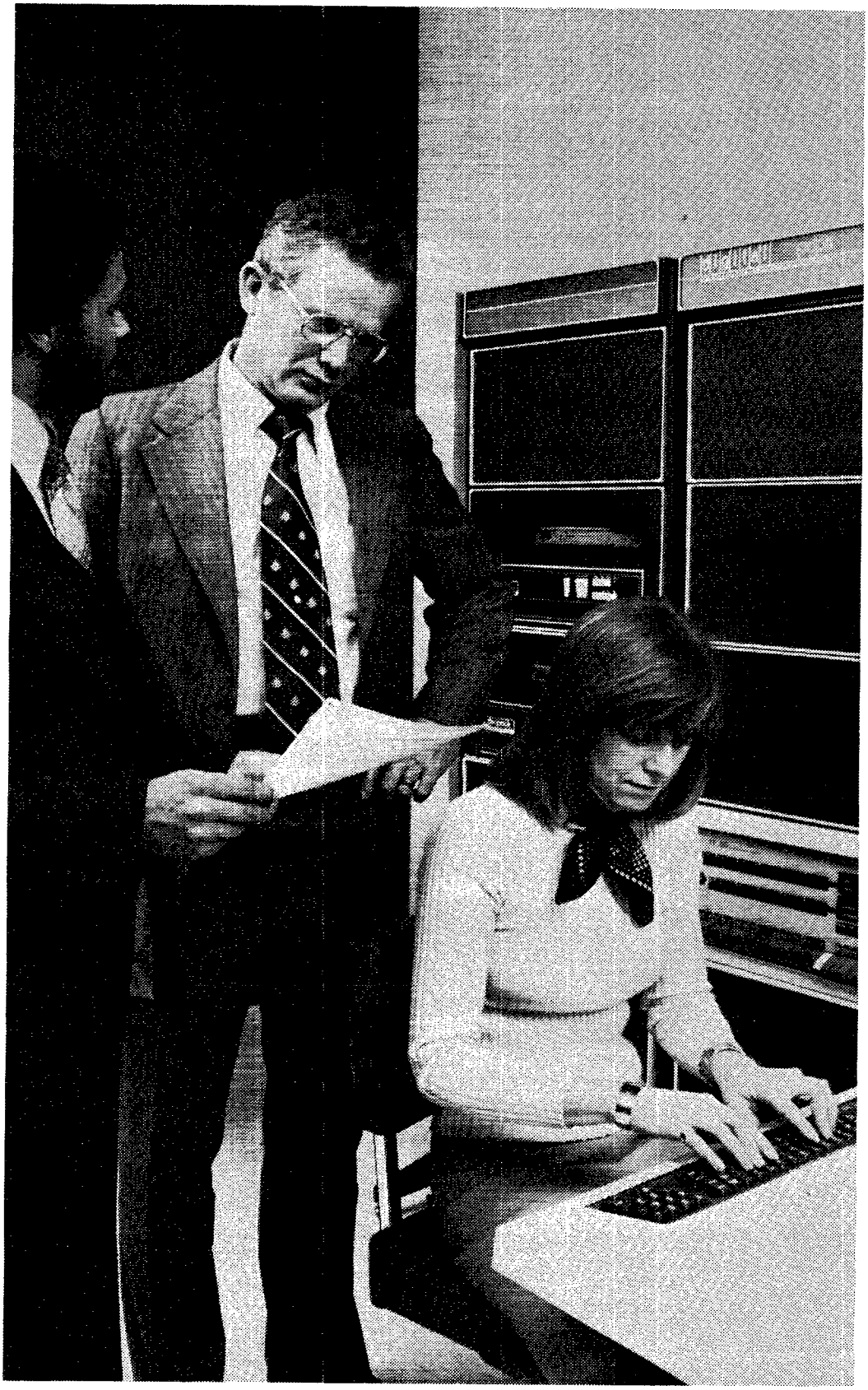
DOCUMENTATION

DIGITAL offers several levels of documentation describing PDP-11 software and hardware. The PDP-11 Handbook series, which includes the Peripherals Handbook, the Terminals and Communications Handbook, and the Software Handbook, presents an introductory technical level of PDP-11 family information. The hardware user documentation and software tutorial documentation which accompany the delivery of

a PDP-11 computer system offer the most detailed levels of information. There are also several good books published by commercial publishers which discuss the PDP-11 family. Specific topics such as microprogramming are also well-covered in commercially available books. If you have a specific documentation need, discuss the issue with a DIGITAL sales representative, who will guide you to the appropriate literature.

NUMERICAL NOTATION

Three number systems are used in this handbook: octal, base eight; binary, base two; and decimal, base ten. **Octal** is used for address locations, contents of addresses, and instruction operation codes. **Binary** is used for descriptions of words and **decimal** for normal quantitative references. Refer to Appendix C for a conversion table including these three number systems.



CHAPTER 2
UNIBUS

The UNIBUS is the outstanding design feature that makes possible the strengths and flexibility of the PDP-11 family members discussed in this book. DIGITAL's unique data bus, the UNIBUS, provides the hardware and software backbone of the PDP-11/04, 34A, 44, 60, and 70 processors. The UNIBUS was the first data bus in the history of the minicomputer industry to enable devices to send, receive, or exchange data without processor intervention and without intermediate buffering in memory.

PDP-11 ARCHITECTURE AND THE UNIBUS

PDP-11 architecture takes advantage of the UNIBUS in its method of addressing peripheral devices. Memory elements, such as the main core memory, or any read-only or solid state memories, have ascending addresses starting at zero, while registers that store I/O data or the status of individual peripheral devices have addresses in the highest 8K bytes of addressing space.

There are tens of thousands of memory addresses, but only two—one for data, one for control—for some peripheral devices, and up to half a dozen for more complicated equipment like magnetic tapes or disks.

The PDP-11 UNIBUS consists of 56 signal lines, to which all devices, including the processor, are connected in parallel.

51 lines are bidirectional and **5** are unidirectional.

Communication between any two devices on the bus is in a master/slave relationship. During any bus operation, one device, the bus master, controls the bus when communicating with another device on the bus, called the slave. For example, the processor, as master, can fetch an instruction from the memory, which is always a slave; or the disk, as master, can transfer data to the memory, as slave. Master/slave relationships are dynamic: the processor, for example, may pass bus control to a disk, then the disk may become master and communicate with slave memory.

When two or more devices try to obtain control of the bus at once, priority circuits decide among them. Devices have unique priority levels, fixed at system installation. A unit with a high priority level obviously always takes precedence over one with a low priority level; in the case of units with equal priority levels, the one electrically closest to the processor on the bus takes precedence over those further away.

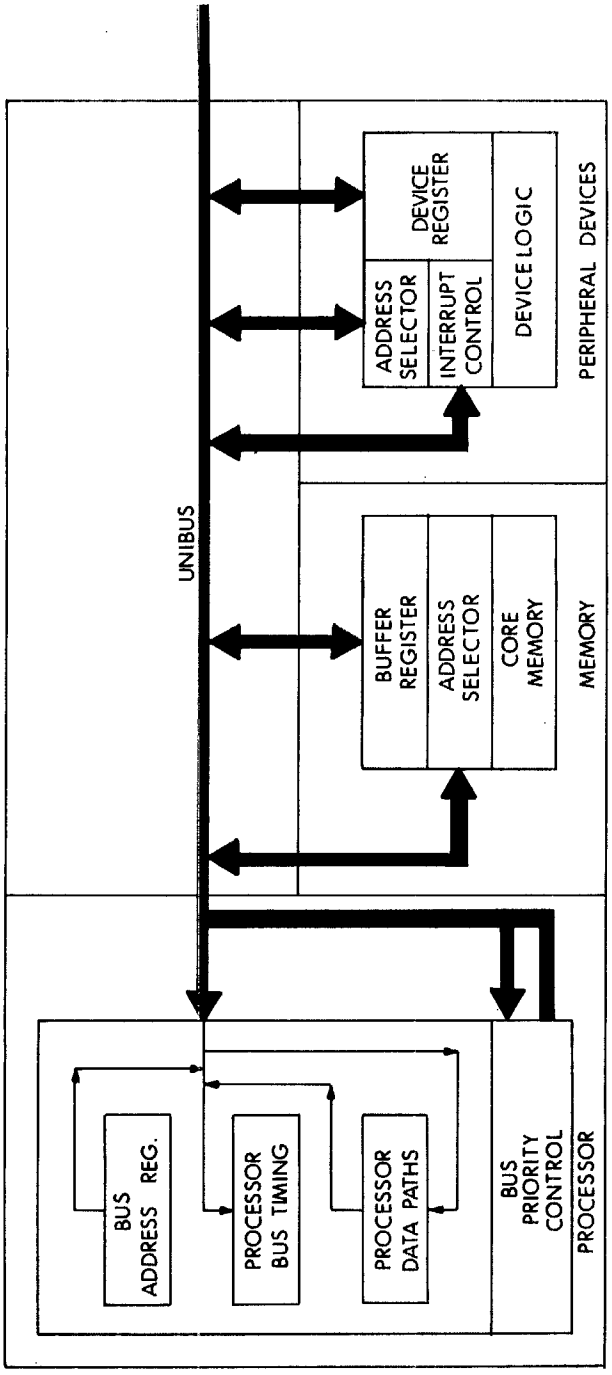


Figure 2-1 UNIBUS

Suppose the processor has control of the bus when three devices, all of higher priority than the processor, request bus control. If the requesting devices are of different priority, the processor will grant use of the bus to the one with the highest priority. If they are all of the same priority, all three signals come to the processor along the same bus line, so that it sees only one request signal. Its reply granting priority travels down the bus to the nearest requesting device, passing through any intervening non-requesting devices. The requesting device takes control of the bus, executes a single bus cycle of a few hundred nanoseconds, and relinquishes the bus. (Some devices will take the bus for more than one bus cycle.) Then the request grant sequence occurs again, this time going to the second device down the line, which has been waiting its turn. When all higher-priority requests has been granted, control of the bus returns to the lowest-priority device, usually the processor.

The processor usually has lowest priority because in general it can stop whatever it is doing without serious consequences. Peripheral devices may be involved with some kind of mechanical motion, or may be connected to a real-time process, either of which requires immediate attention to a request, to avoid data loss.

The priority arbitration takes place asynchronously in parallel with data transfer. Every device on the bus except memory is capable of becoming a bus master.

BUS COMMUNICATION

Communication is interlocked, so that each control signal issued by the master must be acknowledged by a response from the slave to complete the transfer. This simplifies the device interface because timing is no longer critical. The maximum transfer rate on the UNIBUS is one 16-bit word every 400 ns, or about 2.5 million 16-bit words per second. However, the typical transfer rate including average bus delays, is 1 million 16-bit words per second.

USING THE BUS

A device uses the bus if it needs to:

- Request the processor. As a result, the processor stops what it is doing, enters an interrupt service routine, and services the device.
- Transfer a word or byte of data to or from another device, (usually memory), without involving the processor, an NPR (non-processor request) transfer. Such functions are performed by direct memory access devices such as disks or tape units.

Whenever two devices communicate, it is called a bus cycle. Only one word or byte can be transferred per bus cycle. An instruction cycle

involves one or more bus cycles. Fetching an instruction involves a bus cycle; storing a result in memory or a device register involves another bus cycle.

BUS CONTROL

There are two ways of requesting bus control: non-processor requests (NPRs) or bus requests (BRs).

An NPR is issued when a device wishes to perform a data transaction. An NPR device does not use the CPU once the running program has set up parameters of buffer address, disk sector selection and byte count; therefore, the CPU can relinquish bus control while an instruction is being executed.

A BR is issued when a device needs to interrupt the CPU for service. An interrupt is not serviced until the processor has finished executing its current instruction.

BUS REQUESTS

- DEVICE makes a bus request by asserting a BR.
- BUS ARBITRATOR recognizes the request by issuing a Bus Grant (BG). This bus grant is issued only if the priority of the device is greater than the priority currently assigned to the processor.
- DEVICE acknowledges the bus grant and inhibits further grants by asserting Selection Acknowledge (SACK). The device also clears BR.
- BUS ARBITRATOR receives SACK and clears BG.
- DEVICE asserts Bus Busy (BBSY) and clears SACK.
- DEVICE asserts Bus Interrupt (INTR) and its vector address.
- CPU responds

NON-PROCESSOR REQUESTS

- DEVICE makes a non-processor request by asserting NPR.
- BUS ARBITRATOR recognizes the request by issuing a non-processor grant or NPG.
- DEVICE acknowledges the grant and inhibits further grants by asserting SACK; device also clears NPR.
- BUS ARBITRATOR receives SACK and clears NPG.
- DEVICE asserts Bus Busy (BBSY) and clears SACK.
- DEVICE starts its data transfer.

BUS BUSY SIGNAL

Once a device's bus request has been honored, it becomes bus master as soon as the current bus master relinquishes control.

- Current bus master relinquishes bus control by clearing bus busy (BBSY).
- New device assumes bus control by setting BBSY.

INTERRUPTS

Interrupt handling is automatic in the PDP-11. No device **polling** is required to determine which service routine to execute. A device can interrupt the CPU only if it has gained bus control via a BR. The DEVICE requests an interrupt by asserting INTR along with an interrupt vector. The vector directs the CPU to a memory location previously loaded by the running program with the starting address of an interrupt service routine (ISR). ("I need to interrupt.") The CPU accepts the interrupt vector an asserts SSYN (Slave SYNC) to indicate the vector has been accepted. ("I have your interrupt.") The DEVICE releases the bus to the CPU by clearing INTR, removing the vector, and clearing BBSY. ("I'm giving control of the bus back to you.") The CPU acknowledges by clearing SSYN (Slave SYNC), stores the information it needs to return to the interrupted program (a hardware stack located in memory is used for this purpose), and enters the interrupt handling sequence. ("Thank you, I'm starting to service your interrupt.") When the interrupt operation is completed, the CPU removes the information that was stored on the stack and resumes the program at the point where it was interrupted. A more detailed description of the operations required to service an interrupt follows:

1. Processor relinquishes control of the bus, priorities permitting.
2. When a master gains control, it sends the processor an interrupt request and a unique memory address which contains the address of the device's service routine, called the interrupt vector address. Immediately following this pointer address is a word (located at vector address +2) which is to be used as the new processor status (PS) word.
3. The new PC and PS (interrupt vector) are taken from the specified address. The old PS and PC are pushed onto the current stack. The service routine is then entered when the contents of the vector address are moved to the PC and program execution resumes—at the address of the interrupt service routine (ISR) loaded previously as a vector by the running program
4. The device service routine can cause the processor to resume the interrupted process by executing the return from interrupt instruction, described in Chapter 4, which pops the two top words from the current processor stack and uses them to load the PC and PS registers.

A device routine can be interrupted by a higher priority bus request any time after the new PC and PS have been loaded. If such an interrupt occurs, the PC and PS of the service routine are automatically stored in the temporary registers and then pushed onto the new current stack, and the new device routine is entered. This is known as "nesting."

Interrupt Servicing

Every hardware device capable of interrupting the processor has a unique pair of locations (2 words) reserved for its interrupt vector in low memory. The first word contains the location of the device's service routine, and the second, the processor status word that is to be used by the service routine. The program is responsible for loading the address of the ISR into this low memory address before interrupt time occurs. Through proper use of the PS, the programmer can switch the operational mode of the processor, and modify the processor's priority level to mask out lower level interrupts.

PRIORITY CONTROL

The PDP-11 priority system determines which device obtains the bus. Each PDP-11 device is assigned a specific location in the priority structure. Priority arbitration logic determines which device obtains the bus according to its position in the priority structure. The priority structure is 2-dimensional; i.e., there are vertical priority levels and horizontal priorities at each level. There are five vertical priority levels.

Devices that gain bus control with one of the bus request lines (BR7, BR6, BR5, BR4) can take full advantage of the power of the processor by requesting an interrupt. The entire instruction set is then available for manipulating data and status registers. When a device servicing program is being run, the task being performed by the processor is interrupted, and the device service routine is initiated. After the device request has been satisfied, the processor returns to its former task. Note that interrupt requests can be made only if bus control has been gained through a BR priority level.

Bus Request Level

There are two lines associated with each BR level. The bus request is made on a BR line (BR7, BR6, BR5, or BR4). The bus grant is made on the corresponding grant line (BG7, BG6, BG5, or BG4). BR levels BR3 through BR0 are used only by the software; devices are not assigned to these BR levels. Unlike NPRs, a BR can be handled only between instruction cycles. The BR levels are used for interrupts so that the device can obtain service from the CPU. A request made at any BR level requires processor intervention.

Priority Levels

Because there are only five vertical priority levels, NPR, BR7, BR6, BR5 and BR4, it is often necessary to connect more than one device to a single level. When a number of devices are connected to the same level, the situation is referred to a horizontal priority. If more than one device makes a request at the same level, then the device electrically closest to the CPU has the highest priority.

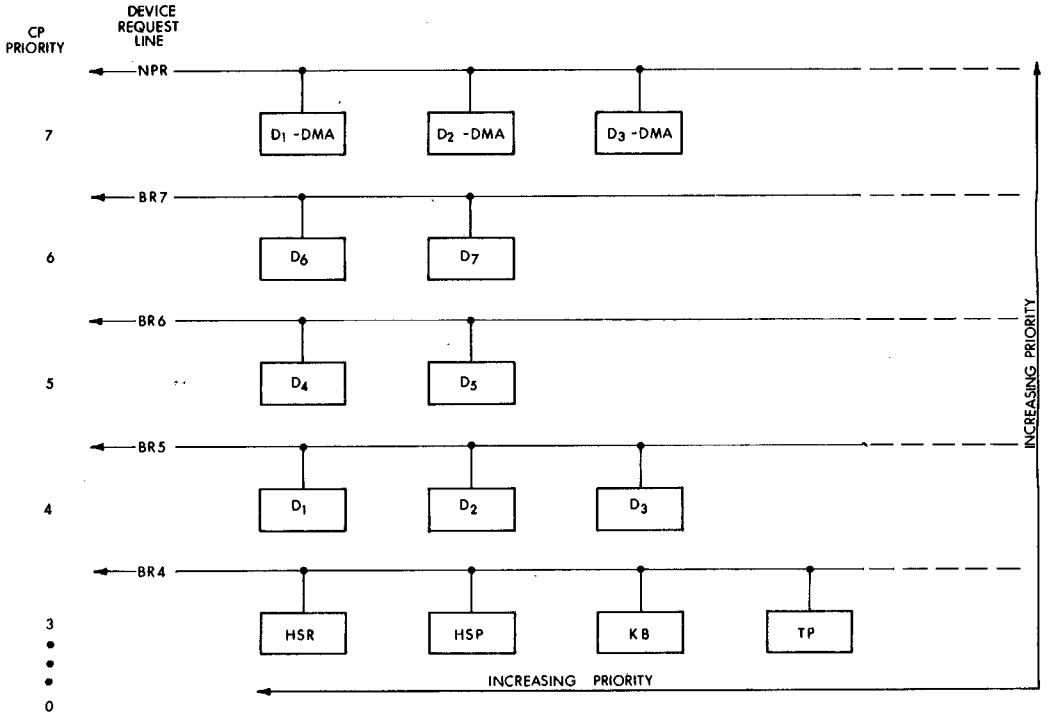


Figure 2-2 Priority Control

The grant line for the NPR level is connected to all devices on that level in a "daisy chain" arrangement. When an NPG is issued, it first goes to the device electrically closest to the CPU. If that device did not make the request, it permits the NPG to travel to the next device. Whenever the NPG reaches a device that has made a request, that device captures the grant, and prevents it from passing to any subsequent device in the chain.

BR chaining is identical to NPR chaining in function. However, each BR level has its own BG chain. Thus, the grant chain for BR7 is the BG7 line which is chained through all devices at the BR7 level.

PRIORITY ASSIGNMENTS

When assigning priorities to a device, three factors must be considered: operating speed, ease of data recovery, and service requirements.

Data from a fast device may be available for only a short time period. Therefore, highest priorities are usually assigned to fast devices to prevent loss of data and to prevent the bus from being tied up by slower devices.

If data from a device is lost, recovery may be automatic, may require manual intervention, or may be impossible. Therefore, highest priorities are assigned to devices whose data cannot be recovered, while lowest priorities are reserved for devices with automatic data recovery features.

CPU Priority Level

In addition to device priority levels, the CPU has a programmable priority. The CPU can be set to any one of eight priority levels. Priority is not fixed; it can be raised or lowered by software. The CPU priority is elevated from level 4 to level 6 when the CPU stops servicing a BR4 device and starts servicing a BR6 device. This programmable priority feature (the second vector word) permits masking of bus requests. The CPU can hold off servicing lower priority devices until more critical functions are completed. For example, when CPU priority is set to level 6, all bus requests on the same and lower levels are ignored (in this case, all requests appearing on BR4, BR5, and BR6).

DATA TRANSACTIONS

There are four types of data transactions:

- DATO—a data *word* is transferred out of the master and into its slave.
- DATOB—a data *byte* is transferred out of the master and into its slave.
- DATI—a data word is transferred from the slave to the master. The master may select the low or high byte if only a data byte is desired.
- DATIP—used with destructive readout devices such as core memory. It is similar to a DATI except that data is not rewritten (restored) into the addressed memory location (data is restored during a DATI) unless followed by DATO or DATOB to the same location.

EXECUTION OF DATA TRANSACTIONS

Before a device can perform a data transaction, it must:

- Obtain control of the bus via an NPR.

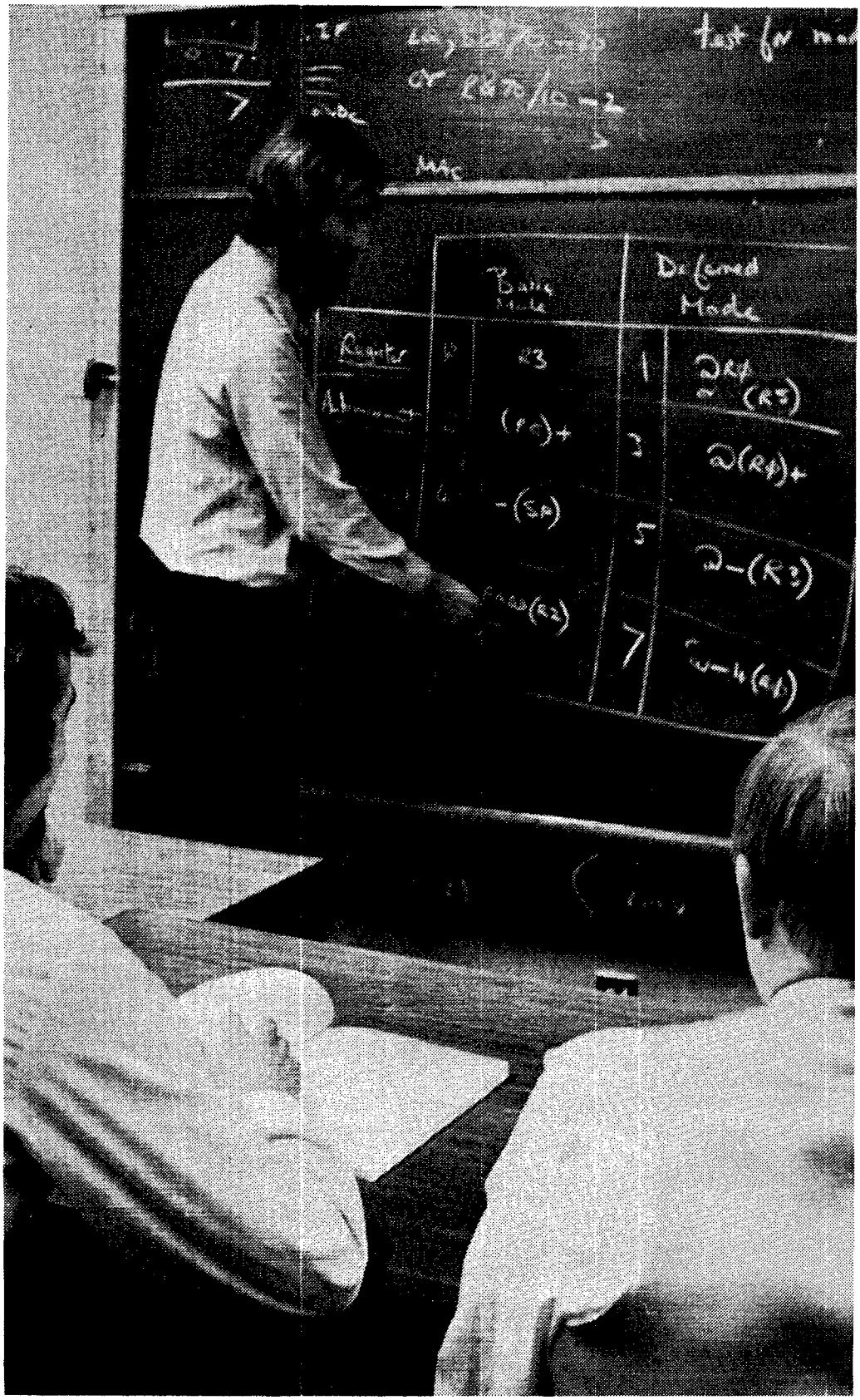
- Select (address) the slave device it wishes to communicate with. Each device on the bus has a unique address.
- Tell the slave what type of data transaction is to be performed.
- Wait for a response from the slave indicating the slave is present and ready.

Data transactions between a master and a slave device are synchronized by master sync (MSYNC) and slave sync (SSYN) signals. Below is an example of how these signals are used during a typical DATI transaction:

1. Master selects the slave by addressing it, specifies the type of data transaction, and requests data by asserting MSYN. ("Give me data.")
2. Slave gathers the data and asserts SSYN when the data is available. ("Here it is.")
3. Master drops MSYN after it accepts the data. ("Thank you, I have the data.")
4. Slave removes data from the lines and acknowledges the master by dropping SSYN. ("You're welcome.")

Table 2 Bus Control

SIGNAL	NAME	SOURCE	DEST.	TIMING	FUNCTION
NPR	Non-processor Request	Any DMA device	UNIBUS Control Logic	Asynchronous	Highest priority bus request
NPG	Non-processor Grant	CPU	Next bus master	Asynchronous	Transfers bus control
BR7 through BR4	Bus Request	Any device	UNIBUS Control Logic	Asynchronous	Requests bus control
BG7 through BG4	Bus Grant	Memory	Next bus master	After instruction	Transfers bus control
SACK	Selection Acknowledge	Next bus master	UNIBUS Control Logic	Response to NPG or BG	Acknowledges grant and inhibits further grants
BBSY	Bus Busy	Master	All devices	Asserted by bus master	Asserts control of the bus
INTR	Interrupt	Master	UNIBUS Control Logic	If control has been gained by a BR (not NPR), INTR asserted after BBSY	Transfers bus control to handling routine in the processor


Register
Direct Mode

CHAPTER 3
ADDRESSING MODES

In the PDP-11 family, all memory reference addressing is accomplished using the eight general purpose registers. In specifying an address of the data (operand address), one of the eight registers is selected with one of several addressing modes. Each memory reference instruction specifies the:

- function to be performed (operation code)
- general purpose register to be used when locating the destination or source and destination operand(s)
- addressing mode, which specifies how the selected registers are to be used

The instruction format and addressing techniques available to the programmer are of particular importance. It is the combination of addressing modes with the instruction set that provides the PDP-11 family a unique number of capabilities. The PDP-11 is designed to handle structured data efficiently and with flexibility. The general purpose registers implement these functions in the following ways, by acting:

- as accumulators: holding the data to be manipulated
- as pointers: the contents of the register are the address of the operand, rather than the operand itself.
- as index registers: the contents of the register are added to the second word of the instruction to produce the address of the operand. This capabilty allows easy access to variable entries in a list.

Using registers for both data manipulation and address calculation results in a variable length instruction format. If registers alone are used to specify the data source, only one memory word is required to hold the instruction. In certain modes, two or three words may be utilized to hold the basic instruction components. Special addressing mode combinations enable temporary data storage for convenient dynamic handling of frequently accessed data. This is known as **stack addressing**. Programming techniques utilizing the stack are discussed in Chapter 5. Register 6 is always used as the hardware stack pointer, or SP. Register 7 is used by the processor as its program counter (PC). Thus, the register arrangement to be considered in conjunction with instructions and with addressing modes is: registers 0-5 are general purpose registers, register 6 is the hardware stack pointer, and register 7 is the program counter. The full instruction set and instruction formats are explained in Chapter 4.

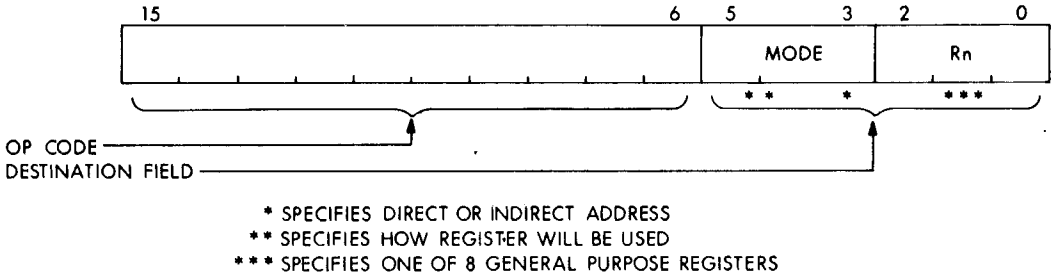
For the purpose of clearly illustrating the use of the various addressing modes, the following instructions are used in this chapter:

Mnemonic	Description	Octal Code
CLR	Clear (Zero the specified destination.)	0050DD
CLRB	Clear Byte (Zero the byte in the specified destination.)	1050DD
INC	Increment (Add 1 to contents of destination.)	0052DD
INCB	Increment Byte (Add 1 to the contents of destination byte.)	1052DD
COM	Complement (Replace the contents of the destination by their logical 1's complements; each 0 bit is set and each 1 bit is cleared.)	0051DD
COMB	Complement Byte (Replace the contents of the destination byte by their logical 1's complements; each 0 bit is set and each 1 bit is cleared.)	1051DD
ADD	Add (Add source operand to destination operand and store the result at destination address.)	06SSDD

DD = destination field (6 bits)
SS = source field (6 bits)
() = contents of

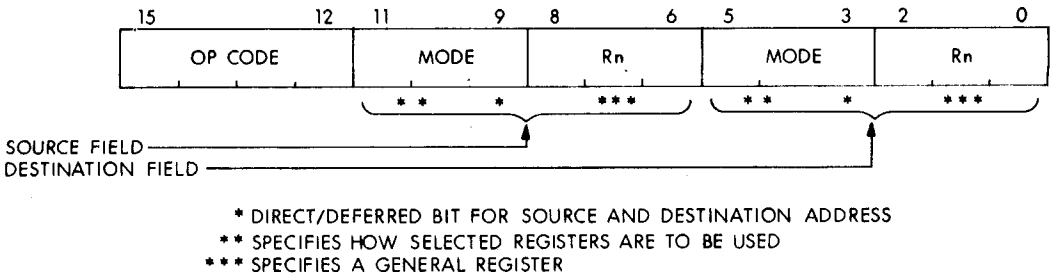
Single and double operand instructions use the following format.

The instruction format for the first word of all single operand instructions (such as clear, increment, test) is:



Single Operand Instruction Format

The instruction format for the first word of the double operand instruction is:



Double Operand Instruction Format

Bits 5-3 of the source or destination fields specify the binary code of the addressing mode chosen. Bits 2-0 specify the general register to be used.

The four direct addressing modes are:

- register
- autoincrement
- autodecrement
- index

In a register mode, the content of the selected register is taken as the operand. In autodecrement mode, after the register has been modified, it contains the address of the operand. In autoincrement mode, at the start of the instruction execution, the register contains the address of the operand, and after the instruction is executed, the address of the next higher word or byte memory location. In index mode, the register is added to the displacement, X, to produce the address of the operand.

When bit 3 of the source/destination field is set, indirect addressing is specified and the four basic modes become deferred modes.

Prefacing the register operand(s) with an "@" sign or placing the register in parentheses indicates to the MACRO-11 assembler that deferred addressing mode is being used.

The indirect addressing modes are:

- register deferred
- autoincrement deferred
- autodecrement deferred
- index deferred

Program counter (register 7) addressing modes are:

- immediate
- absolute
- relative
- relative deferred

The addressing modes are explained and shown in examples in the following pages. They are summarized, in text and in graphic representation, at the end of the chapter.

REGISTER MODE **MODE 0** **Rn**

Register mode provides faster instruction execution. There is no need to reference memory to retrieve an operand. Any of the general registers can be used as simple accumulators. The operand is contained in the selected register (low order byte for byte operations). Assembler syntax requires that a general register be defined as follows:

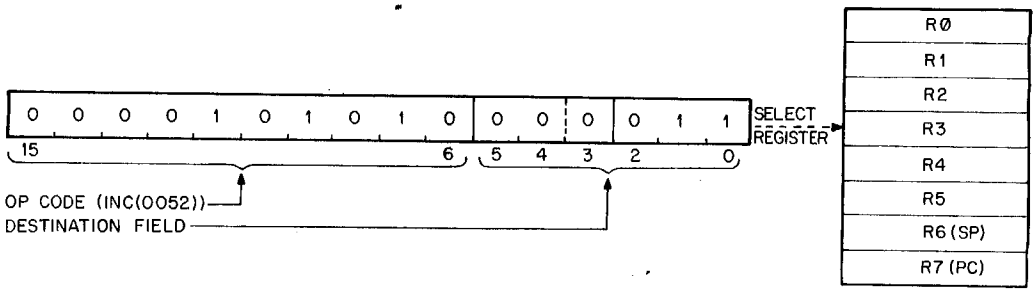
R0 = %0
R1 = %1
R2 = %2

% indicates register definition.

Register Mode Example

Symbolic	Instruction Octal Code	Description
INC R3	005203	Add 1 to the contents of R3.

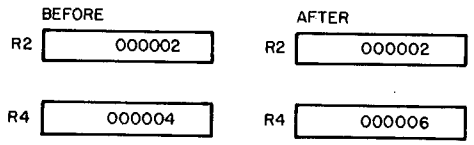
Represented as:



Register Mode Example

Symbolic	Instruction Octal Code	Description
ADD R2,R4	060204	Add the contents of R2 to the contents of R4, replacing the original contents of R4 with the sum.

Represented as:



REGISTER DEFERRED MODE MODE 1 (Rn)

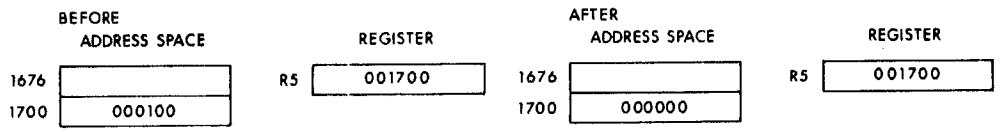
In register deferred mode, the address of the operand is stored in a general purpose register. The address contained in the general purpose register directs the CPU to the operand. The operand is located outside the CPU, either in memory, or in an I/O register.

This mode is used for sequential lists, indirect pointers in data structures, top of stack manipulations, and jump tables.

Register Deferred Mode Example

Symbolic	Instruction Octal Code	Description
CLR (R5)	005015	The contents of the location specified in R5 are cleared.

Represented as:



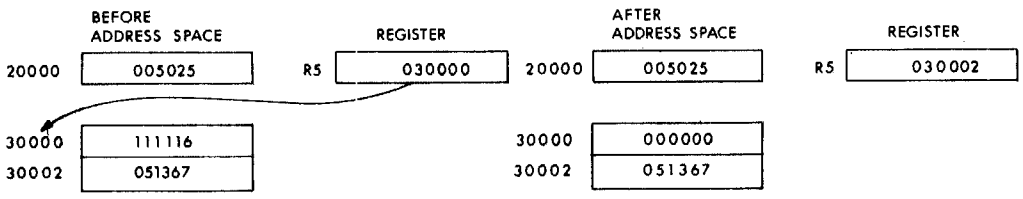
AUTOINCREMENT MODE MODE 2 (Rn)+

In autoincrement mode, the register contains the address of the operand; the address is automatically incremented after the operand is retrieved. The address then references the next sequential operand. This mode allows automatic stepping through a list or series of operands stored in consecutive locations. When an instruction calls for mode 2, the address stored in the register is incremented each time the instruction is executed. It is incremented by 1 if you are using byte instructions, by 2 if you are using word instructions. However, R6 and R7 are always incremented by 2.

Autoincrement Mode Example

Symbolic	Instruction Octal Code	Description
CLR (R5)+	005025	Contents of R5 are used as the address of the operand. Clear selected operand and then increment the contents of R5 by 2.

Represented as:



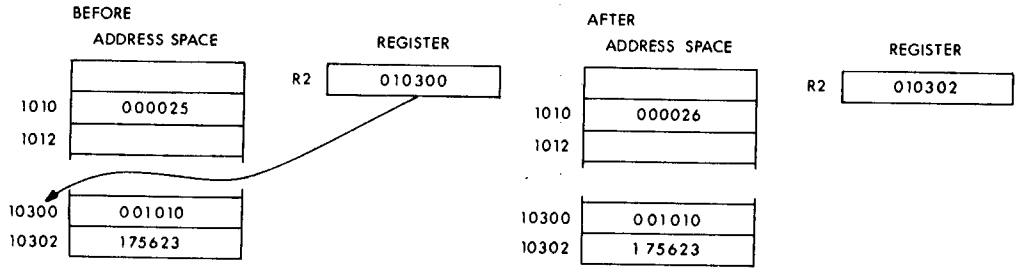
AUTOINCREMENT DEFERRED MODE MODE 3 @(Rn)+

In autoincrement deferred mode, the register contains a pointer to an address. The "+" indicates that the pointer in Rn is incremented by 2 (for both word and byte operations) after the address is located. Mode 2, autoincrement, is used only to access operands that are stored in consecutive locations. Mode 3, autoincrement deferred, is used to access lists of operands stored anywhere in the system; i.e., the operands do not have to reside in adjoining locations. Mode 2 is used to step through a table of operands, mode 3 is used to step through a table of addresses.

Autoincrement Deferred Example

Symbolic	Instruction Octal Code	Description
INC @(R2)+	005232	Contents of R2 are used as the address of the address of the operand. The operand is increased by 1, contents of R2 are incremented by 2.

Represented as:



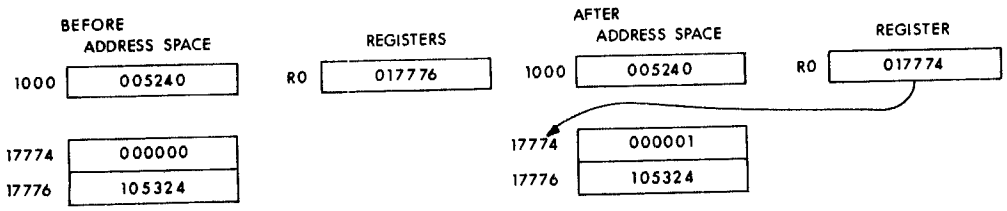
AUTODECREMENT MODE MODE 4 −(Rn)

In autodecrement mode, the register contains an address that is automatically decremented; the decremented address is used to locate an operand. This mode is similar to autoincrement mode, but allows stepping through a list of words or bytes in reverse order. The address is decremented by 1 for bytes, by 2 for words. However, R6 and R7 are always decremented by 2.

Autodecrement Mode Example

Symbolic	Instruction Octal Code	Description
INCB −(R0)	105240	The contents of R0 are decremented by 1, then used as the address of the operand. The operand byte is increased by 1.

Represented as:



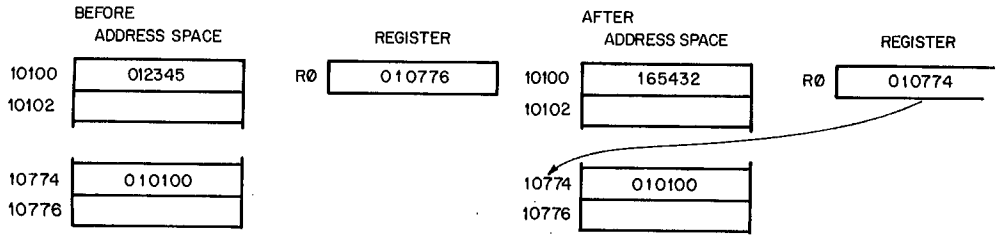
AUTODECREMENT DEFERRED MODE MODE 5 @−(Rn)

In autodecrement deferred mode, the register contains a pointer. The pointer is first decremented by 2 (for both word and byte operations), then the new pointer is used to retrieve an address stored outside the CPU. This mode is similar to autoincrement deferred, but allows stepping through a table of addresses in reverse order. Each address then redirects the CPU to an operand. Note that the operands do not have to reside in consecutive locations.

Autodecrement Deferred Mode Example

Symbolic	Instruction Octal Code	Description
COM @−(R0)	005150	The contents of R0 are decremented by 2 and then used as the address of the address of the operand. The operand is 1's complemented.

Represented as:



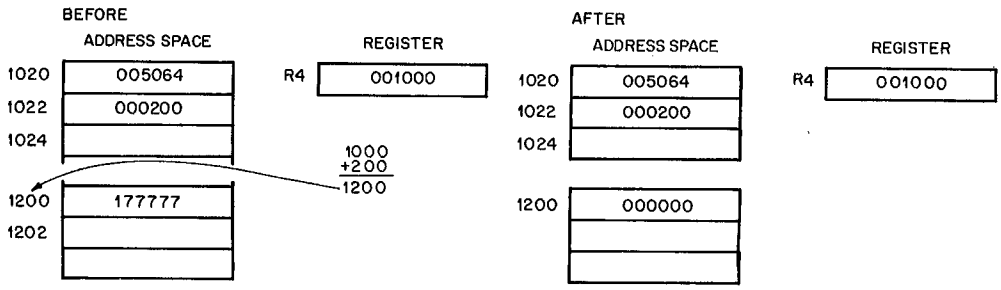
INDEX MODE MODE 6 X(Rn)

In index mode, a base address is added to an index word to produce the effective address of an operand; the base address specifies the starting location of table or list. The index word then represents the address of an entry in the table or list relative to the starting (base) address. The base address may be stored in a register. In this case, the index word follows the current instruction. Or the locations of the base address and index word may be reversed (index word in the register, base address following the current instruction).

Index Mode Example

Symbolic	Instruction Octal Code	Description
CLR 200(R4)	005064 000200	The address of the operand is determined by adding 200 to the contents of R4. The location is then cleared.

Represented as:



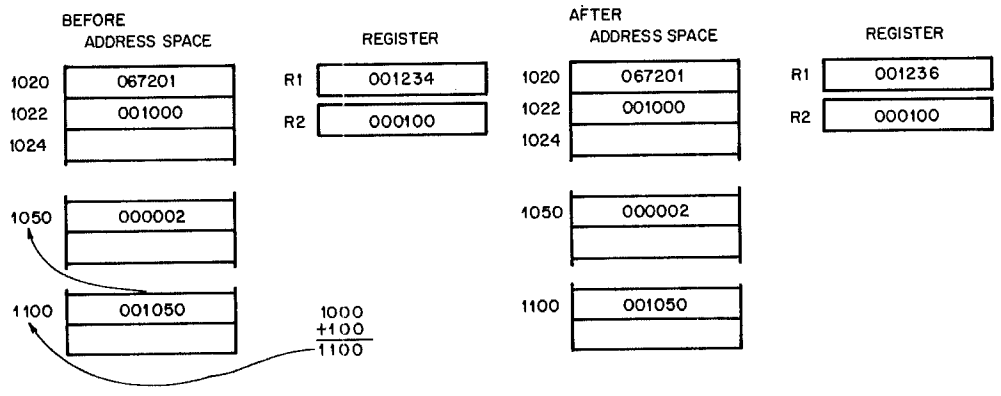
INDEX DEFERRED MODE MODE 7 @X(Rn)

In index deferred mode, a base address is added to an index word. The result is a pointer to an address, rather than the actual address. This mode is similar to mode 6, except that it produces a pointer to an address. The content of that address then redirects the CPU to the desired operand. Mode 7 provides for the random access of operands using a table of operand addresses.

Index Deferred Mode Example

Symbolic	Instruction Octal Code	Description
ADD @1000(R2),R1	067201 001000	1000 and the contents of R2 are summed to produce the address of the address of the source operand, the contents of which are added to the contents of R1. The result is stored in R1.

Represented as:



USE OF THE PC AS A GENERAL REGISTER

Register 7 is both a general purpose register and the program counter on the PDP-11. When the CPU uses the PC to access a word from memory, the PC is automatically incremented by two to contain the address of the next word of the instruction being executed or the address of the next instruction to be executed. When the program uses the PC to access byte data, the PC is still incremented by two.

The PC can be used with all the 7 addressing modes if you use machine language only. There is no symbol in MACRO-11 for 7 PC addressing modes so it will not accept all modes. There are four modes in which the PC can provide advantages for handling position-independent code and for handling unstructured data. These modes refer to the PC and are termed immediate, absolute (or immediate deferred), relative, and relative deferred.

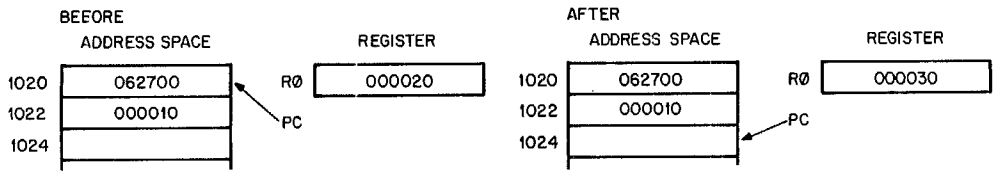
PC IMMEDIATE MODE **MODE 2** **#n**

Immediate mode is equivalent to using the autoincrement mode with the PC. It provides time improvements for accessing constant operands by including the constant in the memory location immediately following the instruction word.

PC Immediate Mode Example

Symbolic	**Instruction Octal Code**	**Description**
ADD #10,R0	062700 000010	The value 10 is located in the second word of the instruction and is added to the contents of R0. Just before this instruction is fetched and executed, the PC points to the first word of the instruction. The processor fetches the first word and increments the PC by two. The source operand mode is 27 (autoincrement the PC). Thus, the PC is used as a pointer to fetch the operand (the second word of the instruction) before being incremented by two to point to the next instruction.

Represented as:



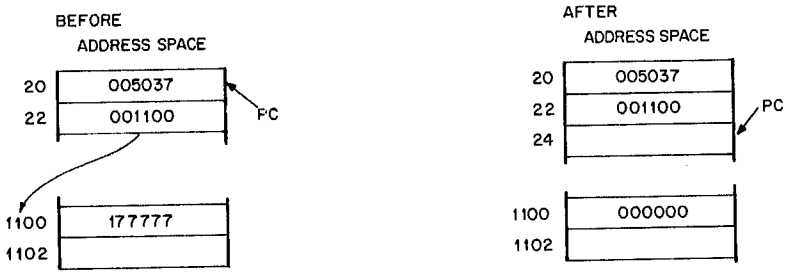
PC ABSOLUTE MODE MODE 3 @#A

This mode is the equivalent of immediate deferred or autoincrement deferred mode using the PC. The contents of the location following the instruction are taken as the address of the operand. Immediate data is interpreted as an absolute address (i.e., an address that remains constant no matter where in memory the assembled instruction is executed).

PC Absolute Mode Example

Symbolic	Instruction Octal Code	Description
CLR @#1100	005037 001100	Clears the contents of location 1100.

Represented as:



PC RELATIVE MODE MODE 6 A

This mode is index mode 6 using the PC. The operand's address is calculated by adding the word that follows the instruction (called an "offset") to the updated contents of the PC.

PC+2 directs the CPU to the offset that follows the instruction. PC+4 is summed with this offset to produce the effective address of the operand. PC+4 also represents the address of the next instruction in the program.

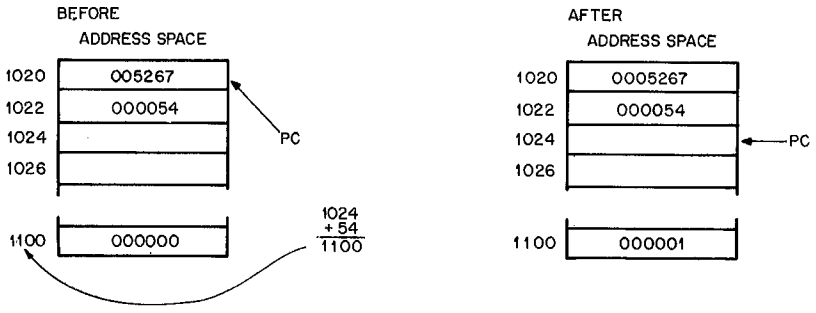
With the relative addressing mode, the address of the operand is always determined with respect to the updated PC. Therefore, when the instruction is relocated, the operand remains the same relative distance away.

The distance between the updated PC and the operand is called an **offset**. After a program is assembled, this offset appears in the first word location that follows the instruction. This mode is useful for writing position-independent code.,

PC Relative Mode Example

Symbolic	Instruction Octal Code	Description
INC A	005267 000054	To increment location A, contents of memory location in the second word of the instruction are added to PC to produce address A. Contents of A are increased by 1.

Represented as:



PC RELATIVE DEFERRED MODE **MODE 7** **@A**

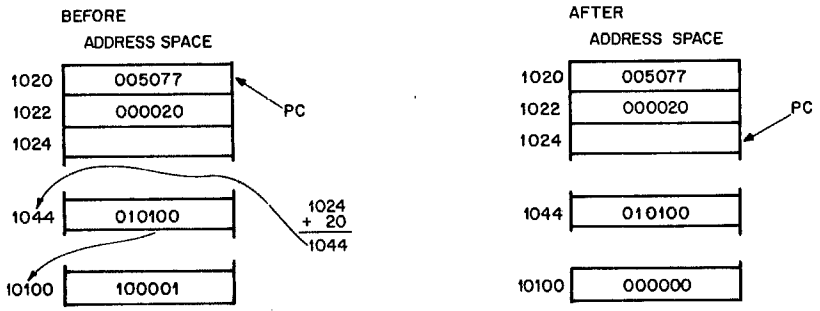
This mode is index deferred (mode 7), using the PC. A pointer to an operand's address is calculated by adding an offset (that follows the instruction) to the updated PC.

This mode is similar to the relative mode, except that it involves one additional level of addressing to obtain the operand. The sum of the offset and updated PC (PC+4) serves as a pointer to an address. When the address is retrieved, it can be used to locate the operand.

PC Relative Deferred Mode Example

Symbolic	Instruction Octal Code	Description
CLR @A	005077 000020	Adds the second word of the instruction to PC to produce the address of the address of the operand. Clears operand.

Represented as:



Direct Addressing Modes

Binary Code	Mode	Name	Symbolic	Function
000	0	Register	Rn	Register contains operand.
010	2	Autoincrement	(Rn)+	Register is used as a pointer to sequential data, then incremented. R0-R5 are incremented by 1 for byte and 2 for word instruction. R6-R7 are always incremented by 2.

Binary Code	Mode	Name	Symbolic	Function
100	4	Autodecrement	−(Rn)	Register is decremented and then used as a pointer to sequential data. R0-R5 are decremented by 1 for byte and by 2 for word instructions. R6-R7 are always decremented by 2.
110	6	Index	X(Rn)	Value X is added to (Rn) to produce address of operand. Neither X nor (Rn) is modified. X, the index value, is always found in the next memory location and increments the PC.

Indirect Addressing Modes

Binary Code	Mode	Name	Symbolic	Function
001	1	Register Deferred	@Rn or (Rn)	Register contains the address of the operand.

Binary Code	Mode	Name	Symbolic	Function
011	3	Autoincrement Deferred	@(Rn)+	Register is first used as a pointer to a word containing the address of the operand, then incremented (always by 2, even for byte instructions).
101	5	Autodecrement Deferred	@−(Rn)	Register is decremented (always by 2, even for byte instructions) and then used as a pointer to a word containing the address of the operand.
111	7	Index Deferred	@X(Rn)	Value X (the index is always found in the next memory location and increments the PC by 2) and (Rn) are added and the sum is used as a pointer to a word containing the address of the operand. Neither X nor (Rn) is modified.

When used with the PC, these modes are termed immediate, absolute (or immediate deferred), relative, and relative deferred.

PC Register Addressing Modes

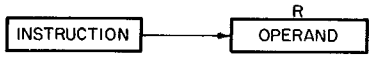
Binary Code	Mode	Name	Symbolic	Function
010	2	Immediate	#n	Operand is contained in the instruction.
011	3	Absolute	@#A	Absolute address is contained in the instruction.
110	6	Relative	A	Address of A, relative to the instruction, is contained in the instruction.
111	7	Relative Deferred	@A	Address of location containing address of A, relative to the instruction, is contained in the instruction.

GRAPHIC SUMMARY OF PDP-11 ADDRESSING MODES

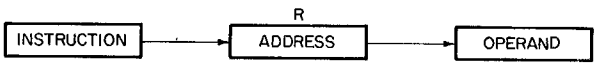
General Register Addressing Modes

R is a general register, 0 to 7.
(R) is the contents of that register.

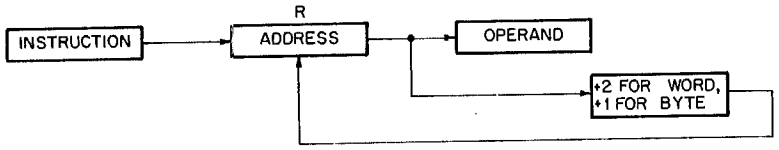
Mode 0 **Register** OPR R R contains operand.



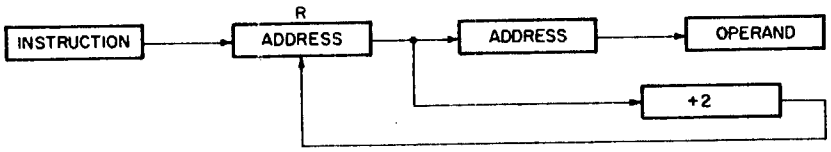
Mode 1 **Register deferred** OPR (R) R contains address.



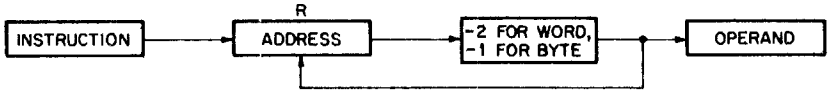
Mode 2	**Autoincrement**	OPR (R)+	R contains address, then increment (R). Note that R6 and R7 are always incremented by 2.



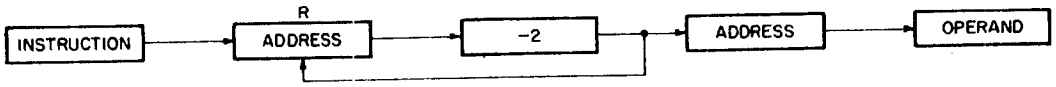
Mode 3	**Autoincrement deferred**	OPR @(R)+	R contains address of address, then increment (R) by 2.



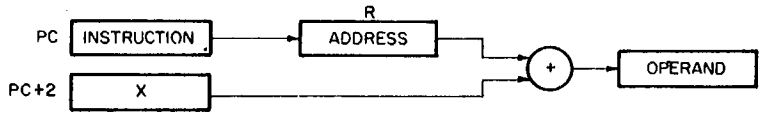
Mode 4	**Autodecrement**	OPR –(R)	Decrement (R), then R contains address. Note that R6 and R7 are always decremented by 2.



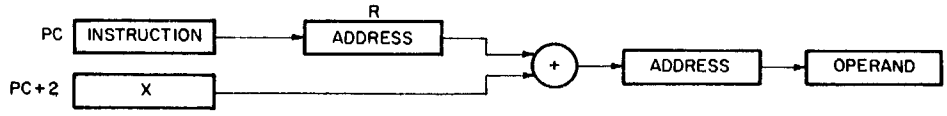
Mode 5	**Autodecrement deferred**	OPR @–(R)	Decrement (R) by 2, then R contains address of address.



Mode 6 **Index** OPR X(R) (R)+X is address.

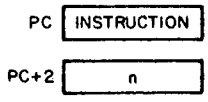


Mode 7 **Index deferred** OPR @X(R) (R)+X is address of address.

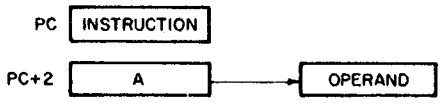


Program Counter Addressing Modes
Register = 7

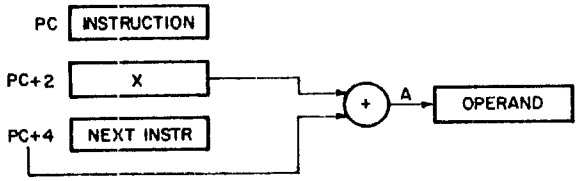
Mode 2 **Immediate** OPR #n Literal operand n is contained in the second instruction word.



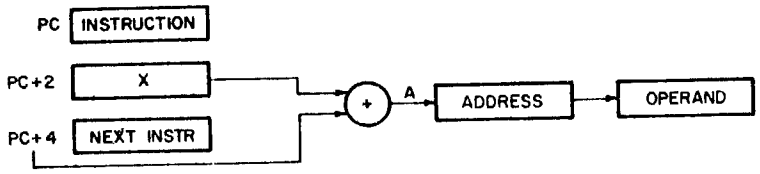
Mode 3 **Absolute** OPR @#A Address A is contained in the second instruction word.

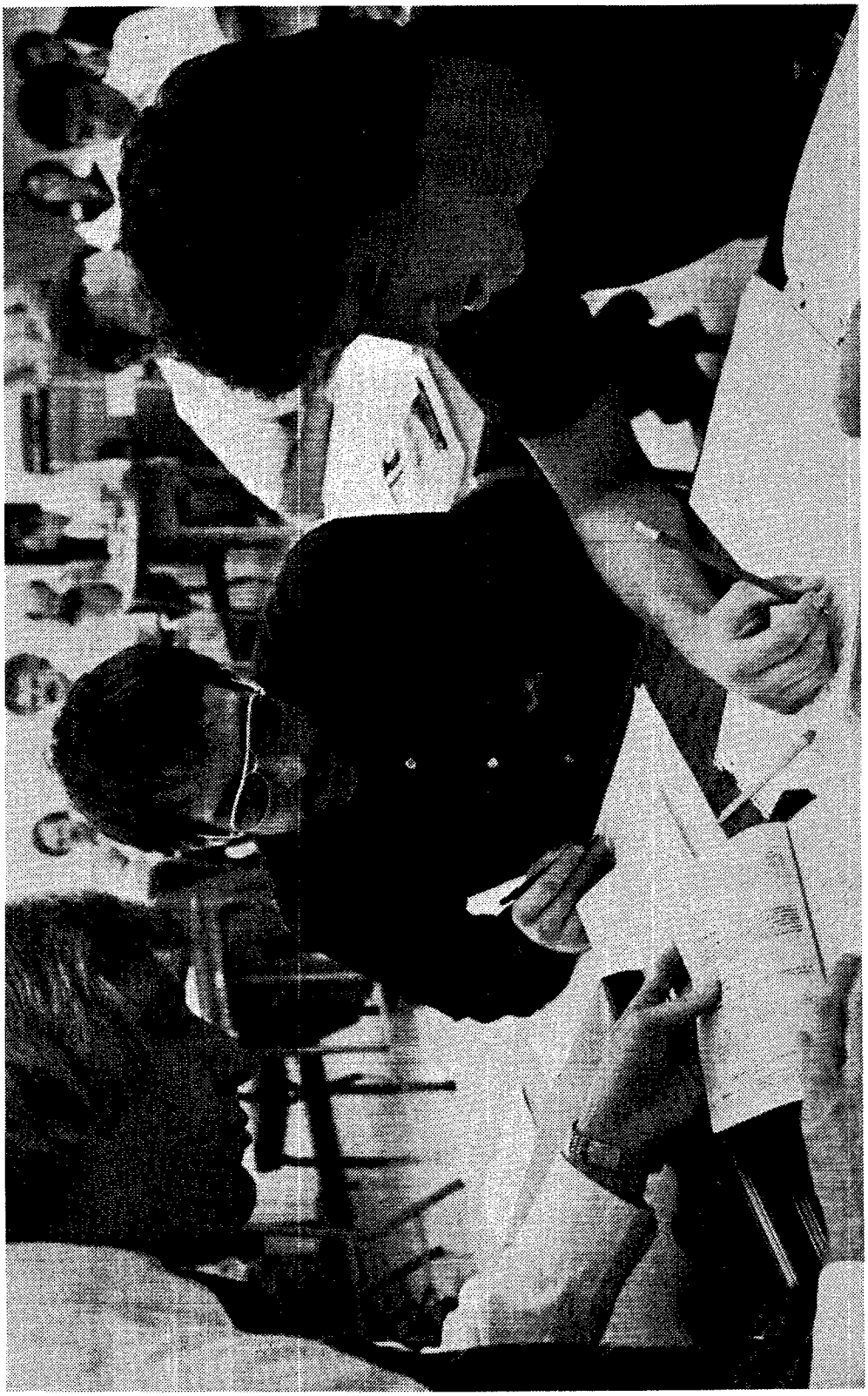


Mode 6 **Relative** OPR A PC+4 + X is address. PC+4 is updated PC.



Mode 7 **Relative deferred** OPR @A PC+4 + X is address of address. PC+4 is updated PC.





CHAPTER 4
INSTRUCTION SET

The PDP-11 instruction set and addressing modes produce over 400 unique instructions. The instruction set offers a wide choice of operations, so that a single instruction will frequently accomplish a task that would require several in a traditional computer. PDP-11 instructions allow byte and word addressing in both single and double operand formats. This saves memory space and simplifies the implementation of control and communications applications. The PDP-11's use of double operand instructions allows you to perform several operations with a single instruction. For example, ADD A,B adds the contents of location A to location B, storing the result in location B. Traditional computers would implement this instruction in the following way:

```
LDA A
ADD B
STR B
```

The PDP-11 instruction set also contains a full set of conditional branches, eliminating excessive use of jump instructions. PDP-11 instructions fall into one of seven categories:

- *Single Operand* — one part of the word specifies the operation, referred to as "op code," the second part provides information for locating the operand.
- *Double Operand* — the first part of the word specifies the operation to be performed, the remaining two parts provide information for locating two operands.
- *Branch* — the first part of the word specifies the operation to be performed, the second part indicates where the action is to take place in the program.
- *Jump and Subroutine* — these instructions have an opcode and address part, and in the case of JSR, a register for linkage.
- *Trap* — these instructions contain an opcode only. In TRAP and EMT, the low order byte may be used for function dispatching.
- *Miscellaneous* — HALT, WAIT, and Memory Management
- *Condition Code*

SINGLE OPERAND INSTRUCTIONS

	Mnemonic	Instruction
General		
	CLR(B)	clear
	COM(B)	1's complement
	INC(B)	increment
	DEC(B)	decrement
	NEG(B)	2's complement (negate)
	TST(B)	test
Shift & Rotate		
	ASR(B)	arithmetic shift right
	ASL(B)	arithmetic shift left
	ROR(B)	rotate right
	ROL(B)	rotate left
	SWAB	swap bytes
Multiple Precision		
	ADC(B)	add carry
	SBC(B)	subtract carry
	SXT	sign extend

Instruction Format

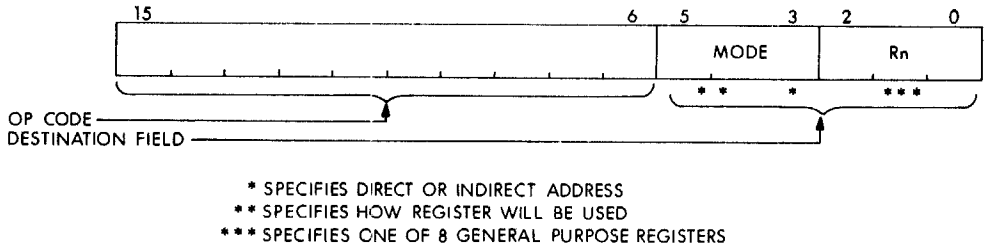


Figure 4-1 Single Operand Instruction Format

The instruction format for single operand instructions is:

- Bit 15 indicates word or byte operation.
- Bits 14-6 indicate the operation code, which specifies the operation to be performed.
- Bits 5-0 indicate the 3-bit addressing mode field and the 3-bit general register field. These two fields are referred to as the destination field.

DOUBLE OPERAND INSTRUCTIONS

	Mnemonic	Instruction
General		
	MOV(B)	move source to destination
	ADD	add source to destination
	SUB	subtract source from destination
	CMP(B)	compare source to destination
	ASH	shift arithmetically
	ASHC	arithmetic shift combined
	MUL	multiply
	DIV	divide
Logical		
	BIT(B)	bit test
	BIC(B)	bit clear
	BIS(B)	bit set
	XOR	exclusive OR

Instruction Format

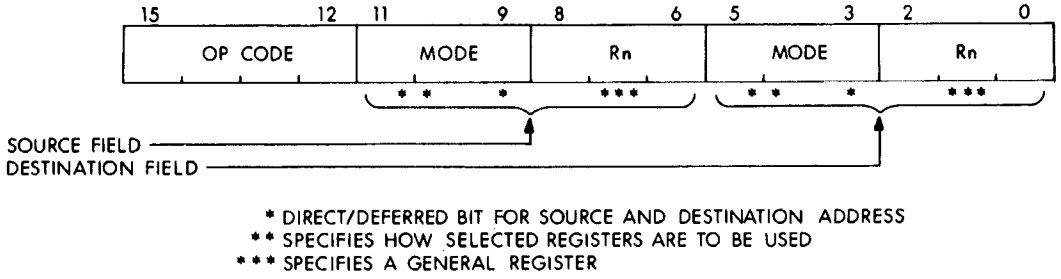


Figure 4-2 Double Operand Instruction Format

The format of most double operand instructions is similar to that of single operand instructions except that they have *two* fields for locating operands. One field is called the source field, the other is called the destination field. Each field is further divided into addressing mode and selected register. Each field is completely independent. The mode and register used by one field may be completely different than the mode and register used by another field.

- Bit 15 indicates word or byte operation *except* when used with op-code 6_8. Then it indicates an ADD or SUBtract instruction.
- Bits 14-12 indicate the op code, which specifies the operation to be done.

- Bits 11-6 indicate the 3-bit addressing mode field and the 3-bit general register field. These two fields are referred to as the **source** field.
- Bits 5-0 indicate the 3-bit addressing mode field and the 3-bit general register field. These two fields are referred to as the **destination** field.
- Some double operand instructions (ASH, ASHC, MUL, DIV) must have the destination operand only in a register. Bits 15-9 specify the opcode. Bits 8-6 specify the destination register. Bits 5-0 contain the source field. XOR has a similar format, except that the source is in a register specified by bits 8-6, and the destination field is specified by bits 5-0.

Byte Instructions

Byte instructions are specified by setting bit 15. Thus, in the case of the MOV instruction, bit 15 is 0; when bit 15 is set, the mnemonic is MOVB. There are no byte operations for ADD and SUB, i.e., no ADDB or SUBB.

BRANCH INSTRUCTIONS

	Mnemonic	Instruction
Branch		
	BR	branch (unconditional)
	BNE	branch if not equal (to zero)
	BEQ	branch if equal (to zero)
	BPL	branch if plus
	BMI	branch if minus
	BVC	branch if overflow is clear
	BVS	branch if overflow is set
	BCC	branch if carry is clear
	BCS	branch if carry is set
Signed Conditional Branch		
	BGE	branch if greater than or equal (to zero)
	BLT	branch if less than (zero)
	BGT	branch if greater than (zero)
	BLE	branch if less than or equal (to zero)
	SOB	subtract one and branch (if not = 0)
Unsigned Conditional Branch		
	BHI	branch if higher
	BLOS	branch if lower or same
	BHIS	branch if higher or same
	BLO	branch if lower

Instruction Format

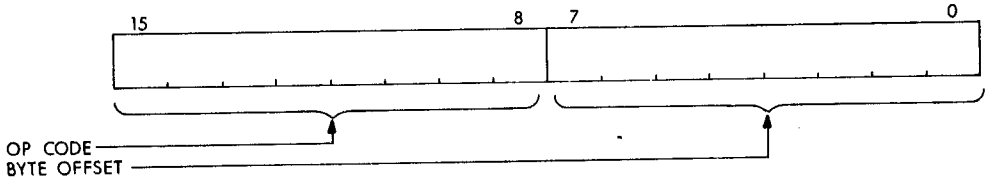


Figure 4-3 Branch Instruction Format

- The high byte (bits 15-8) of the instruction is an op code specifying the conditions to be tested.
- The low byte (bits 7-0) of the instruction is the signed offset value in words that determines the new program location if the branch is taken. Thus, program control can be transferred within a range of −128 to +127 words from the updated PC.

JUMP AND SUBROUTINE INSTRUCTIONS

Mnemonic	Instruction
JMP	jump
JSR	jump to subroutine
RTS	return from subroutine
MARK	facilitates stack clean-up procedures

Instruction Format

JSR Format

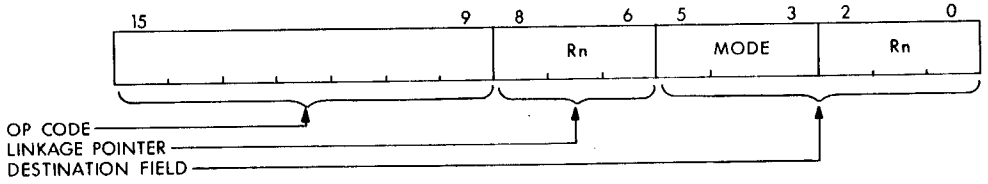


Figure 4-4 JSR Instruction Format

- Bits 15-9 are always octal 004 indicating the op code for JSR.
- Bits 8-6 specify the link register. Any general purpose register may be used in the link, except R6 (SP).

- Bits 5-0 designate the destination field that consists of addressing mode and general register fields. This specifies the starting address of the subroutine.
- Register R7 (the Program Counter) is frequently used for both the link and the destination. For example, you may use JSR R7, SUBR, which is coded 004767. R7 is the *only* register that can be used for both the link and destination, the other GPRs cannot. Thus, if the link is R5, any register except R5 can be used for one destination field.

RTS Format

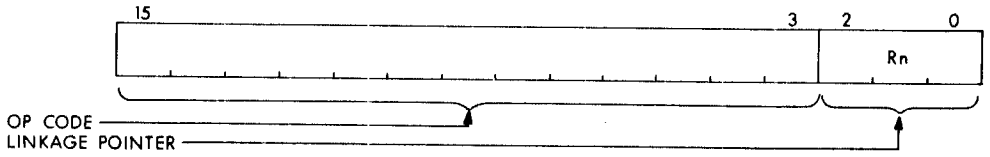


Figure 4-5 RTS Instruction Format

The RTS (return from subroutine) instruction uses the link to return control to the main program once the subroutine is finished.

- Bits 15-3 always contain octal 00020, which is the op code for RTS.
- Bits 2-0 specify any one of the general purpose registers.
- The register specified by bits 2-0 must be the same register used as the link between the JSR causing the jump and the RTS returning control.

TRAPS AND INTERRUPTS

Mnemonic	Instruction
EMT	emulator trap
TRAP	trap
BPT	breakpoint trap
IOT	input/output trap
CSM	call to supervisor mode
RTI	return from interrupt
RTT	return from interrupt

There are three ways of leaving a main program:

- *software exit* — the program specifies a jump to some subroutine
- *trap exit* — internal hardware on a special instruction forces a jump to an error handling routine
- *interrupt exit* — external hardware forces a jump to an interrupt service routine

In all of the above cases, there is a jump to another program. Once that program has been executed, control is returned to the proper point in the main program.

MISCELLANEOUS INSTRUCTIONS

Mnemonic	Instruction
HALT	halt
WAIT	wait for interrupt
RESET	reset UNIBUS
MTPD	move to previous data space
MTPI	move to previous instruction space
MFPD	move from previous data space
MFPI	move from previous instruction space
MTPS	move byte to processor status word
MFPS	move byte from processor status word

Note that on the PDP-11/70, the four instructions for referencing the previous address space (MTPD, MTPI, MFPD, MFPI) use the General Register set indicated by PSW<11> when they are executed.

CONDITION CODE OPERATION

Mnemonic	Instruction
CLC,CLV,CLZ,CLN,CCC	clear
SEC,SEV,SEZ,SEN,SCC	set

There are four condition code bits:

- N, indicating a negative condition when set to 1
- Z, indicating a zero condition when set to 1
- V, indicating an overflow condition when set to 1
- C, indicating a carry condition when set to 1

These four bits are part of the processor status word (PS). The result of any single operand or double operand instruction affects one or more of the four condition code bits. A new set of condition codes is usually created after execution of each instruction. Some condition codes are not affected by the execution of certain instructions. The CPU may be asked to check the condition codes after execution of an instruction. The condition codes are used by the various instructions to check software conditions.

Z bit — Whenever the CPU sees that the result of an instruction is zero, it sets the Z bit. If the result is not zero, it clears the Z bit. There are a number of ways of obtaining a zero result:

- adding two numbers equal in magnitude but different in sign
- comparing two numbers of equal value

- using the CLR or BIC instruction

N bit — The CPU looks only at the sign bit of the result. If the sign bit is set, indicating a negative value, the CPU sets the N bit. If the sign bit is clear, indicating a positive value, then the CPU clears the N bit.

C bit — The CPU sets the C bit automatically when the result of an instruction has caused a carry out of the most significant bit of the result. Otherwise, the C bit is cleared. During rotate instructions (ROL and ROR), the C bit forms a buffer between the most significant bit and the least significant bit of the word. A carry of 1 sets the C bit while a carry of 0 clears the C bit. However, there are exceptions. For example:

- SUB and CMP set the C bit when there is no carry.
- INC and DEC do not affect the C bit.
- COM always sets the C bit, TST always clears the C bit.

V bit — The V bit is set to indicate that an overflow condition exists. An overflow means that the result of an instruction is too large to be placed in the destination. There are two methods the hardware uses to check for an overflow condition.

One way is for the CPU to test for a change of sign.

- When using single operand instructions, such as INC, DEC, or NEG, a change of sign indicates an overflow condition.
- When using double operand instructions, such as ADD, SUB, or CMP, in which both the source and destination have like signs, a change of sign in the result indicates an overflow condition.

Another method used by the CPU is to test the N bit and C bit when dealing with shift and rotate instructions.

- If only the N bit is set, an overflow exists.
- If only the C bit is set, an overflow exists.
- If *both* the N and C bits are set, there is no overflow condition.

More than one condition code can be set by a particular instruction. For example, both a carry and an overflow condition may exist after instruction execution.

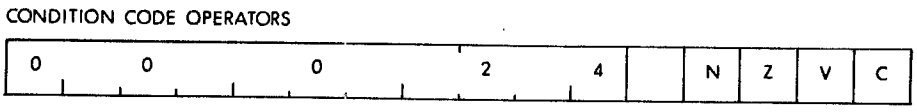


Figure 4-6 Condition Code Operators' Format

Instruction Format

The format of the condition code operators is as follows:

- Bits 15-5 — the opcode
- Bit 4 — the "operator" which indicates set or clear with the values 1 and 0 respectively. If set, any selected bit is set; if clear, any selected bit is cleared.
- Bits 3-0 — the **mask** field. Each of these bits corresponds to one of the four condition code bits. When one of these bits is set, then the corresponding condition code bit is set or cleared depending on the state of the "operator" (bit 4).

EXAMPLES

The following examples and explanations illustrate the use of the various types of instructions in a program.

Single Operand Instruction Example

This routine uses a tally to control a loop, which clears out a specific block of memory. The routine has been set up to clear 30_8 byte locations beginning at memory address 600.

```
INIT:   MOV #600,R0
        MOV # 30,R1
LOOP:   CLRB (R0)+
        DEC R1
        BNE LOOP
        HALT
```

Program Description

- The CLRB (R0)+ instruction clears the content of the location specified by R0 and increments R0.
- R0 is the pointer.
- Because the autoincrement addressing mode is used, the pointer automatically moves to the next memory location after execution of the CLRB instruction.
- Register R1 indicates the number of locations to be cleared and is, therefore, a counter. Counting is performed by the DEC R1 instruction. Each time a location is cleared, it is counted by decrementing R1.
- The Branch If Not Zero, BNE, instruction checks for done. If the counter is not zero, the program branches back to clear another location. If the counter is zero, indicating done, then the program halts.

Double Operand Instruction Example

The following routine moves characters to be printed from location 600 into a print buffer area in memory.

```
INIT:   MOV #600, R0        ;set up source address
        MOV #prtbuf, R1     ;set up destination address
        MOV #76, R2         ;set up loop count
START:  MOVB (R0)+, (R1)+   ;move one character
                            ;and increment
                            ;both source and
                            ;destination addresses
        DEC R2              ;decrement count by one
        BNE START           ;loop back if
        HALT                ;decremented counter is not equal
                            to zero
```

Program Description

- MOV is the instruction normally used to set up the initial conditions. Here, the first MOV places the starting address (600) into R0, which will be used as a *pointer*. The second MOV places the starting address of the print buffer into R1. The third MOV sets up R2 as a *counter* by loading the desired number of locations (76) to be printed.
- The MOVB instruction moves a byte of data to the printer buffer. The data comes from the location specified by R0. The pointers R0 and R1 are then incremented to point to the next sequential location.
- The counter (R2) is then decremented to indicate one byte has been transferred.
- The program then checks the loops for done with the BNE instruction. If the counter has not reached zero, indicating more transfers must take place, then the BNE causes a branch back to START and the program continues.
- When the counter (R2) reaches zero, indicating all data has been transferred, the branch does not occur and the program halts.

Branch Instruction Example

NOTE

Branch instruction offsets are limited to the range of $+177_8$ to -200_8 words.

A payroll program has set up a series of words to identify each employee by his badge number. The high byte of the word contains the employee's badge number, the low byte contains an octal number

ranging from 0 to 13 which represents his salary. These numbers represent steps within three wage classes to identify which employees get paid weekly, monthly, or quarterly. It is time to make out weekly paychecks. Unfortunately, employee information has been stored in a random order. The problem is to extract the names of only those employees who receive a weekly paycheck. Employee payroll numbers are assigned as follows: 0 to 3 — Wage Class I (weekly), 4 to 7 — Wage Class II (monthly), 10 to 13 — Wage Class III (quarterly).

600 is the starting address of memory block containing the employee payroll information. 1264 is the final address of this data area. The following program searches through the data area and finds all numbers representing wage class I, and, each time an appropriate number is found, stores the employee's badge number (just the high byte) on a "last-in/first-out" stack which begins at location 4000.

```
INIT:     MOV #600, R0
          MOV #4000, R1

START:    CMPB(R0)+,#3

          BHI CONT

STACK:    MOVB (R0),-(R1)

CONT:     INC R0

          CMP #1264,-R0

          BHIS START
```

Program Description

- R0 becomes the address pointer, R1 the stack pointer.
- Compare the contents of the first low byte with the number 3 and go to the first high byte.
- If the number is more than 3, branch to continue.
- If no branch occurs, it indicates that the number is 3 or less. Therefore, move the high byte containing the employee's number onto the stack as indicated by stack pointer R1.
- R0 is advanced to the next low byte.
- If the last address has not been examined (1264), this instruction produces a result equal to or greater than zero.

- If the result is equal to or greater than zero, examine the next memory location.

INSTRUCTION SET

The PDP-11 instruction set is presented in the following section. For ease of reference, the instructions are listed alphabetically.

SPECIAL SYMBOLS

You will find that a number of special symbols are used to describe certain features of individual instructions. The commonly used symbols are explained below.

SYMBOL	MEANING
MN	Maintenance Instruction
SO	Single Operand Instruction
DO	Double Operand Instruction
PC	Program Control Instruction
MS	Miscellaneous Instruction
CC	Condition Code
()	Indicates the contents of. For example, (R5) means "the contents of R5."
src	Source address
dst	Destination address
←	Becomes, or moves into. For example, (dst) ← (src) means that the source becomes the destination or that the source moves into the destination location.
(SP)+	Popped or removed from the hardware stack
−(SP)	Pushed or added to the hardware stack
Λ	Logical AND
v	Logical inclusive OR (either one or both)
∀	Logical exclusive OR (either one, but not both)
~	Logical NOT
Reg or R	Register
B	Byte

SYMBOL	MEANING
M.P.I.	Most Positive Integer—077777(word) or 177 (Byte)
M.N.I.	Most Negative Integer—100000(word) or 200 (Byte)

NOTE

Condition code bits are considered to be cleared unless they are specifically listed as set.

Table 4-1 PDP-11 Instruction Set

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
ADC ADCB Add Carry	SO	0055DD 1055DD	(dst) ← (dst)+C	N: set if result < 0 Z: set if result = 0 V: set if (dst) was M.P.I. and C was 1 C: set if (dst) was −1 and C was 1	Adds the contents of the C bit into the destination.
ADD Add	DO	06SSDD	(dst) ← (src) + (dst)	N: set if result < 0 Z: set if result = 0 V: set if there is arithmetic overflow as a result of the operation; that is, both operands were of the same sign and the result is of the opposite sign C: set if there is a carry from the most significant bit of the result	Adds the source operand to the destination operand and stores the result at the destination address. The original contents of the destination are lost. The contents of the source are not affected. 2's complement addition is performed.

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
ASH Arithmetic Shift	DO	072RSS	R ← R shifted arithmetically NN places to right or left where NN = (src) <5:0>	N: set if result < 0 Z: set if result = 0 V: set if sign of register changed during shift. Cleared if NN = 0. C: loaded from last bit shift out of register. Cleared if NN = 0.	The contents of the register are shifted right or left the number of times specified by the shift count (i.e., bits <5:0> of the source operand). The shift count is taken as the low order 6 bits of the source operand. This number ranges from −32 to +31. Negative is a right shift and positive is a left shift.
ASHC Arithmetic Shift Combined	DO	073RSS	tmp ← R, Rv1 tmp ← tmp shifted NN bits R ← tmp<31:16> Rv1 ← tmp<15:0> The double word R,Rv1 is shifted NN places to the	N: set if result < 0 Z: set if result = 0 V: set if sign bit changes during the shift C: loaded with high order bit when left shift; loaded with low order bit when right shift, (loaded	The contents of the specified register R, and the register Rv1 are treated as a single 32-bit operand and are shifted by the number of bits specified by the count field (bits <5:0> of the source operand) and the registers are replaced by the result. First, bits <31:16> of the result are stored in register R. Then, bits <15:0> of the result are

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
			right or left, where NN = (src) <5:0>	with the last bit shifted out of the 32-bit operand)	stored in register Rv1. The count ranges from −32 to +31. A negative count signifies a right shift. A positive count signifies a left shift. A zero count implies no shift, but condition codes are affected. Condition codes are always set on the 32-bit result. **Note:** 1) The sign bit of the register R is replicated in shifts to the right. The least significant bit is filled with zero in shifts to the left. The C bit stores the last bit shifted out. 2) Integer overflow occurs on a left shift if any bit shifted into the sign position differs from the initial sign of the register.
ASL ASLB	SO SO	0063DD 1063DD	(dst) ← (dst) shifted one place	N: set if high order bit of the result is set	Shifts all bits of the destination left one place. The low order bit is

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
Arithmetic Shift Left			to the left	(result < 0) Z: set if the result = 0 V: loaded with the exclusive OR of the N bit and C bit (as set by the completion of the shift operation) C: loaded with the high order bit of the destination	loaded with a 0. The C bit of the status word is loaded from the high order bit of the destination. ASL performs a signed multiplication of the destination by 2 with overflow indication. For example, −1 shifted left yields −2, +2 shifted left yields +4, and −3 shifted left yields −6.
ASR ASRB Arithmetic Shift Right	SO SO	0062DD 1062DD	(dst) ← (dst) shifted one place to the right	N: set if the high order bit of the result is set (result < 0) Z: set if the result = 0 V: loaded from the exclusive OR of the N bit and C bit (as set by the completion of the shift operation) C: loaded from low or-	Shifts all bits of the destination right one place. The high order bit is replicated. The C bit is loaded from the low order bit of the destination. ASR performs signed division of the destination by 2, rounded to minus infinity. −1 shifted right remains −1, +5 shifted right yields +2, −5 shifted right yields −3.

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
				der bit of the destination	**Note:** In the PDP-11/60, the ASRB does a DATI/DATIP/DATO bus sequence in the execution portion of the instruction. This allows an interlocking of memory addresses. If an I/O page reference is made, the ASRB does a DATIP/DATIP/DATO bus sequence during instruction execution.
BCC Branch if carry clear	PC	103000 PLUS 8-bit offset	PC ← PC + (2 × offset) if C = 0	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the C bit and causes a branch if C is clear.
BCS Branch if carry set	PC	103400 PLUS 8-bit offset	PC ← PC + (2 × offset) if C = 1	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the C bit and causes a branch if C is set. Used to test for a carry in the result of a previous operation.
BEQ	PC	001400	PC ← PC +	N: unaffected	Tests the state of the Z bit and

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
Branch if equal (to zero)		PLUS 8-bit offset	(2 × offset) if Z = 1	Z: unaffected V: unaffected C: unaffected	causes a branch if Z is set. As an example, it is used to test equality following a CMP operation, to test that no bits set in the destination were also set in the source following a BIT operation, and, generally, to test that the result of the previous operation was 0.
BGE Branch if greater than or equal	PC	002000 PLUS 8-bit offset	PC ← PC + (2 × offset) if N⊻V = 0	N: unaffected Z: unaffected V: unaffected C: unaffected	Causes a branch if N equals V (i.e., either both clear or both set). BGE is the complementary operation to BLT. Thus, BGE always causes a branch when it follows an operation that caused addition to two positive numbers. BGE also causes a branch on a 0 result.
BGT Branch if greater than	PC	003000 PLUS 8-bit offset	PC ← PC + (2 × offset) if Zv(N⊻V) = 0	N: unaffected Z: unaffected V: unaffected C: unaffected	Causes a branch if Z is clear and N equals V. Thus, BGT never branches following an operation that added two negative numbers, even if

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
					overflow occurred. In particular, BGT never causes a branch if it follows a CMP instruction operating on a negative source and a positive destination (even if overflow occurred). Further, BGT always causes a branch when it follows a CMP instruction operating on a positive source and negative destination. BGT does not cause a branch if the result of the previous operation was 0 (without overflow).
BHI Branch if higher	PC	101000 PLUS 8-bit offset	PC ← PC + (2 × offset) if C = 0 and Z = 0	N: unaffected Z: unaffected V: unaffected C: unaffected	Causes a branch if the previous operation causes neither a carry nor a 0 result. This will happen in comparision (CMP) operations as long as the source has a higher unsigned value than the destination.

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
BHIS Branch if higher than or same	PC	103000 PLUS 8-bit offset	PC ← PC + (2 × offset) if C = 0	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the C bit and causes a branch if C is cleared.
BIC BICB Bit Clear	DO	04SSDD 14SSDD	(dst) ← ~ (src) Λ (dst)	N: set if high order bit of result set Z: set if result = 0 V: cleared C: not affected	Clears each bit in the destination that corresponds to a set bit in the source. The original contents of the destination are lost. The contents of the source are unaffected.
BIS BISB Bit Set	DO	05SSDD 15SSDD	(dst) ← (src)v(dst)	N: set if high order bit of result set Z: set if result = 0 V: cleared C: not affected	Performs inclusive OR operation between the source and destination operands and leaves the result at the destination address, i.e., corresponding bits set in the source are set in the destination. The contents of the destination are lost.

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
BIT BITB Bit Test	DO	03SSDD 13SSDD	(dst) Λ (src)	N: set if high order bit of result set Z: set if result = 0 V: cleared C: not affected	Performs logical AND comparison of the source and destination operands and modifies condition codes accordingly. Neither the source nor destination operands are affected. The BIT instruction may be used to test whether any of the corresponding bits that are set in the destination are clear in the source.
BLE Branch if less than or equal to	PC	003400 PLUS 8-bit offset	PC ← PC + (2 × offset) if Zv(N⊻V) = 1	N: unaffected Z: unaffected V: unaffected C: unaffected	Causes a branch if Z is set or if N does not equal V. Thus, BLE always branches following an operation that added two negative numbers, even if overflow occurred. In particular, BLE always causes a branch if it follows a CMP instruction operating on a negative source and a positive destination (even if overflow occurred). Further, BLE never causes a branch when it fol-

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
					lows a CMP instruction operating on a positive source and negative destination. BLE always causes a branch if the result of the previous operation was 0.
BLO Branch if lower	PC	103400 PLUS 8-bit offset	PC ← PC + (2 × offset) if C = 1	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the C bit and causes a branch if C is set. Used to test for a carry in the result of a previous operation.
BLOS Branch if lower or same	PC	101400 PLUS 8-bit offset	PC ← PC + (2 × offset) if CvZ = 1	N: unaffected Z: unaffected V: unaffected C: unaffected	Causes a branch if the previous operation caused either a carry or a 0 result. BLOS is the complementary operation to BHI. The branch occurs in comparison operations as long as the source is equal to or has a lower unsigned value than the destination.
BLT	PC	002400	PC ← PC +	N: unaffected	Causes a branch if the exclusive

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
Branch if less than		PLUS 8-bit offset	(2 × offset) if N ⊻ V = 1	Z: unaffected V: unaffected C: unaffected	OR of the N and V bits is 1. Thus, BLT always branches following an operation that added two negative numbers, even if overflow occurred. In particular, BLT always causes a branch if it follows a CMP instruction operating on a negative source and a positive destination (even if overflow occurred). Further, BLT never causes a branch when if follows a CMP instruction operating on a positive source and negative destination. BLT does not cause a branch if the result of the previous operation was 0 (without overflow).
BMI Branch if minus	PC	100400 PLUS 8-bit offset	PC ← PC + (2 × offset) if N = 1	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the N bit and causes a branch if N is set. Used to test the sign (most significant bit) of the result of the previous operation.

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
BNE Branch if not equal	PC	001000 PLUS 8-bit offset	PC ← PC + (2 × offset) if Z = 0	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the Z bit and causes a branch if the Z bit is clear. BNE is the complementary operation to BEQ. It is used to test inequality following a CMP, to test that some bits set in the destination were also in the source, following a BIT, and, generally, to test that the result of the previous operation was not 0.
BPL Branch if plus	PC	100000 PLUS 8-bit offset	PC ← PC + (2 × offset) if N = 0	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the N bit and causes a branch if N is clear. BPL is the complementary operation of BMI.
BPT Breakpoint Trap	PC	000003	−(SP) ← PS −(SP) ← PC PC ← (14) PS ← (16)	N: loaded from trap vector Z: loaded from trap vector V: loaded from trap	Performs a trap sequence with a trap vector address of 14. Used to call debugging aids. The user is cautioned against employing code 000003 in programs run under

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
				vector C: loaded from trap vector	these debugging aids. No information is transmitted in the low byte.
BR Branch (Unconditional)	PC	000400 PLUS 8-bit offset	PC ← PC + (2 × offset)	N: unaffected Z: unaffected V: unaffected C: unaffected	Provides a way of transferring program control within a range of −128 to +127 words with a one word instruction. An unconditional branch.
BVC Branch if V bit clear	PC	102000 PLUS 6-bit offset	PC ← PC + (2 × offset) if V = 0	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the V bit and causes a branch if the V bit is clear. BVC is the complementary operation to BVS.
BVS Branch if V bit set	PC	102400 PLUS 8-bit offset	PC ← PC + (2 × offset) if V = 1	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of V bit and causes a branch if the V bit is set. BVS is used to detect arithmetic overflow in the previous operation.

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
CLR CLRB Clear	SO	0050DD 1050DD	(dst) ← 0	N: cleared Z: set V: cleared C: cleared	Contents of specified destination are replaced with zeros.
C Clear selected condition code bits	CC	000240 PLUS 4-bit mask	Clear condition code bits. Selectable combinations of these bits may be cleared together. Condition code bits corresponding to bits in the condition code operator (Bits 0-3) are modified. Clears the bit specified by the mask; i.e., bit 0, 1, 2, or 3. Bit 4 is a 0. **Operation:** PSW <3:0> ← PSW <3:0>Λ[~mask <3:0>]		
CCC Clear all condition code bits	CC	00257	N, Z, V, C ← 0		
CLC Clear C	CC	000241	C ← 0		

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
CLN Clear N	CC	000250	N ← 0		
CLV Clear V	CC	000242	V ← 0		
CLZ Clear Z	CC	000244	Z ← 0		
CMP CMPB Compare	DO	02SSDD 12SSDD	(src) – (dst) [in detail (src) + ~ (dst) + 1]	N: set if result < 0 Z: set if result = 0 V: set if there is arithmetic overflow; i.e., operands of opposite signs and the sign of the destination is the same as the sign of the result C: set if there is a borrow into the most	Compares the source and destination operands and sets the condition codes, which may then be used for arithmetic and logical conditional branches. Both operands are unaffected. The only action is to set the condition codes. The compare is customarily followed by a conditional branch instruction. Note that unlike the subtract instruction, the order of

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
				significant bit, i.e., if (src)+~(dst)+1 was less than 2**16	operation is (src)−(dst), not (dst)−(src).
COM COMB Comple- ment	SO	0051DD 1051DD	(dst) ← ~ (dst)	N: set if most significant bit of result = 1 Z: set if result = 0 V: cleared C: set	Replaces the contents of the destination address by their logical complements (each bit equal to 0 set and each bit equal to 1 cleared).
CSM Call to Supervisor Mode (PDP- 11/44 only)	PC	0070DD	If MMR 3<3> = 1 **and** current mode ≠ kernel then: begin Supervisor SP ← current mode SP; temp <15:4> ← PSW <15:4>;	N: unaffected Z: unaffected V: unaffected C: unaffected	CSM may be executed in User or Supervisor Mode, but is an illegal instruction in Kernel mode. CSM copies the current stack pointer to the Supervisor Mode (SP), switches to Supervisor Mode, stacks three words on the Supervisor stack, (the PSW with the condition codes cleared, the PC, and the argument word addressed by the op-

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
			temp <3:0> ← 0; PSW <13:12> ← PSW <15:14>; PSW <15:14> ← 01; PSW <4> ← 0; −(SP) ← temp; −(SP) ← PC; −(SP) ← (dst); PC ← (10); end; else trap to 10 in kernel mode;		erand), and sets the PC to the contents of location 10 (in Supervisor space). The called program in Supervisor space may return to the calling program by popping the argument word from the stack and executing RTI. On return, the condition codes are determined by the PSW word on the stack. Hence, the called program in Supervisor space may control the condition code values following return.
DEC DECB Decrement	SO	0053DD 1053DD	(dst) ← (dst) − 1	N: set if result < 0 Z: set if result = 0 V: set if (dst) was M.N.I. C: not affected	Subtracts 1 from the contents of the destination.

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
DIV Divide	DO	071RSS	R,Rv1 ← R,Rv1/(src)	N: set if quotient < 0 (unspecified if V = 1) Z: set if quotient = 0 (unspecified if V = 1) V: set if source = 0 or if quotient cannot be represented as a 16-bit 2's complement number. R, Rv1 are unpredictable if V is set and C is clear. C: set if divide by 0 is attempted	The 32-bit 2's complement integer in R and Rv1 is divided by the source operand. The quotient is left in R; the remainder in Rv1 is of the same sign as the dividend. R must be even.
EMT Emulator Trap	PC	104000 to 104377	−(SP) ← PS −(SP) ← PC PC ← (30)	N: loaded from trap vector Z: loaded from trap	All operation codes from 104000 to 104377 are EMT instructions and may be used to transmit informa-

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
			PS ← (32)	vector V: loaded from trap vector C: loaded from trap vector	tion to the emulating routine (e.g., function to be performed). The trap vector for EMT is at address 30. The new PC is taken from the word at address 30; the new central processor status word (PS) is taken from the word at address 32. **Caution:** EMT is used frequently by DIGITAL system software and is therefore not recommended for general use.
HALT	MS	000000		N: unaffected Z: unaffected V: unaffected C: unaffected	Causes the processor operation to cease. The console is given control of the processor. The console data lights display the contents of the PC (which is the address of the HALT instruction plus 2). Transfers on the UNIBUS are terminated immediately. Pressing the continue

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
					key on the console causes processor operation to resume.
INC INCB Increment	SO	0052DD 1052DD	(dst) ← (dst) + 1	N: set if result < 0 Z: set if result = 0 V: set if (dst) was M.P.I. C: not affected	Adds 1 to the contents of the destination.
IOT I/O Trap	PC	000004	−(SP) ← PS −(SP) ← PC PC ← (20) PS ← (22)	N: loaded from trap vector Z: loaded from trap vector V: loaded from trap vector C: loaded from trap vector	Performs a trap sequence with a trap vector address of 20. Used to call the I/O executive routine IOX in the paper tape software system and for error reporting in the disk operating system. No information is transmitted in the low byte.

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
JMP Jump	PC	0001DD	PC ← dst	N: unaffected Z: unaffected V: unaffected C: unaffected	JMP provides more flexible program branching than provided with the branch instruction. It is not limited to $+177_8$ and -200_8 as are branch instructions. JMP does generate a second word, which makes it slower than branch instructions. Control may be transferred to any location in memory (no range limitation) and can be accomplished with the full flexibility of the addressing modes with the exception of register mode 0. Execution of a jump with mode 0 will cause an illegal instruction condition. (Program control cannot be transferred to a register.) Register deferred mode is legal and will cause program control to be transferred to the address held in the

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
					specified register. **Note that instructions are word data and therefore must be fetched from an even numbered address. A boundary error trap condition will result when the processor attempts to fetch an instruction from an odd address.**
JSR Jump to Subroutine	PC	004RDD	(tmp) ← (dst) (tmp is an internal processor register) ↓(SP) ← reg (push reg contents onto processor stack) reg ← PC (PC holds the location following JSR; this address	N: unaffected Z: unaffected V: unaffected C: unaffected	In execution of the JSR, the old contents of the specified register (the linkage pointer) are automatically pushed onto the R6 stack and new linkage information placed in the register. Thus, subroutines nested within subroutines to any depth may all be called with the same linkage register. There is no need either to plan the maximum depth at which any particular subroutine will be called or to include

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
			now put in reg) PC ← tmp (PC now points to subroutine address)		instructions in each routine to save and restore the linkage pointer. Further, since all linkages are saved in a re-entrant manner on the R6 stack, execution of a subroutine may be interrupted, and the same subroutine re-entered and executed by an interrupt service routine. Execution of the initial subroutine can then be resumed when other requests are satisfied. This process (called nesting) can proceed to any level. JSR PC, dst is a special case of the PDP-11 subroutine call suitable for subroutine calls that transmit parameters through the general purpose registers. JSR, with the PC as the linkage register, saves the use of an extra register.

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
					Note: If the register specified in the first operand register is autoincremented or autodecremented in the second operand (dst) evaluation, the modified register contact is pushed on SP. For example, JSR R5,@(R5)+ will cause the modified value of R5 to be pushed to SP.
LDUB Load Microbreak Register	MN	170003	In the 11/60, causes the lower 8 bits of general register 3 in the CPU to be loaded into the microbreak register. LDUB can be used for the functions described below, depending on the FMM bit (bit 04) in the program status word (FPS). The FMM bit in the status word is used to enable special maintenance logic. In order to set this bit, the CPU must be in Kernel mode. With the FMM bit set, the microprogram will be aborted through JAM μstate address 777 to the Ready state after the state specified by the address (next sequential μstate) in the microbreak register is detected. If the interrupt enable bit (bit 14) of the floating point processor status word is set, the CPU will trap to location 244. An exception code of 16 will be stored in the FEC (floating excep-			

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
			tion code) register. The contents of the FEC register can be transferred to the CPU by the STST (store status) instruction. A second function, available as a result of the LDUB instruction, is that the maintenance personnel can use the address match as a scope sync independent of the FMM bit. When the address matches the contents of the microbreak register, the micro MATCH signal is present. This output is pin DC1 (slot 8 in the FNUA module) and is used as a scope sync to allow visual observation of events that occur during a particular μstate.		
MARK	PC	0064NN	SP ← PC + 2 × NN PC ← R5 R5 ← (SP)+ NN = number of parameters	N: unaffected Z: unaffected V: unaffected C: unaffected	Used as part of the standard PDP-11 subroutine return convention. MARK facilitates the stack clean-up procedures involved in subroutine exit. Assembler format is: MARK N
MED Maintenance, Exam, and Dep	MN	076600	Used in the 11/60 for a processor-specific maintenance function. The first word is used as an escape, with the CODE specifying the operation and address. The instruction is executed only in Kernel mode. Its main purpose is to allow error logging of internal registers and examination of internal registers for diagnostic		

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description

purposes through the EXAM function. Instruction execution in user mode will result in a trap to 10.

The instruction also allows an alteration of registers through the write code.

Note: The cache is turned off via an internal UNIBUS address.

The OPERATION CODE is specified and is register- and operation-dependent. The code directly benefits 11/60 microcode.

R0, a general register, contains the information to be deposited or the results of an examination. The instuction is mainly for diagnostic purposes and failsafe features will not exist. The use of illegal operation codes will only be defined to the extent of completion of the instruction; no-op's will occur. Condition codes are unaltered for this instruction.

The operation codes for the registers and function are noted below:

MED CODE	REGISTER AND FUNCTION	MED CODE	REGISTER AND FUNCTION
XXX00X	LOW HALF ASP LOW (READ)	XXX154	CACHE INVALIDATE
XXX01X	HIGH HALF ASP LOW (READ)	XXX155	READ CACHE TAG
XXX02X	LOW HALF ASP HIGH (READ)	XXX20X	LOW HALF ASP LOW (WRITE)
XXX03X	HIGH HALF ASP HIGH (READ)	XXX21X	HIGH HALF ASP LOW (WRITE)

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
			XXX04X LOW HALF BSP LOW (READ)		XXX22X LOW HALP ASP HIGH (WRITE)
			XXX05X HIGH HALF BSP LOW (READ)		XXX23X HIGH HALF ASP HIGH (WRITE)
			XXX06X LOW HALF BSP HIGH (READ)		XXX24X LOW HALF BSP LOW (WRITE)
			XXX07X HIGH HALF BSP HIGH (READ)		XXX25X HIGH HALF BSP LOW (WRITE)
			XXX100 CSP(0) (READ)		XXX26X LOW HALF BSP HIGH (WRITE)
			XXX101 CSP(1) (READ)		XXX27 HIGH HALF BSP HIGH (WRITE)
			XXX102 CSP(2) (READ)		XXX300 CSP(0) (WRITE)
			XXX103 CSP(3) (READ)		XXX301 CSP(1) (WRITE)
			XXX104 CSP(4) (READ)		XXX302 CSP(2) (WRITE)
			XXX105 CSP(5) (READ)		XXX303 CSP(3) (WRITE)
			XXX106 CSP(6) (READ)		XXX304 CSP(4) (WRITE)
			XXX107 CSP(7) (READ)		XXX305 CSP(5) (WRITE)
			XXX110 CSP(10) (READ)		XXX306 CSP(6) (WRITE)
			XXX111 CSP(11) (READ)		XXX307 CSP(7) (WRITE)

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
			XXX112 CSP(12) (READ) XXX113 CSP(13) (READ) XXX114 CSP(14) (READ) XXX115 CSP(15) (READ) XXX116 CSP(16) (READ) XXX117 CSP(17) (READ) XXX140 JAM (READ XXX141 SERVICE (READ) XXX142 NOP XXX143 CUA (READ) XXX144 FLAG REGISTER (READ) XXX145 NOP XXX146 NOP XXX147 COUNT REGISTER (READ) XXX152 DCS REGISTER #1 (READ) XXX153 DCS REGISTER #2 (READ)	XXX310 CSP(10) (WRITE) XXX311 CSP(11) (WRITE) XXX312 CSP(12) (WRITE) XXX313 CSP(13) (WRITE) XXX314 CSP(14) (WRITE) XXX315 CSP(15) (WRITE) XXX316 CSP(16) (WRITE) XXX317 CSP(17) (WRITE) XXX344 FLAG REGISTER (WRITE) XXX345 D REGISTER (WRITE) XXX346 SHIFT REGISTER (WRITE) XXX347 COUNTER (WRITE) XXX350 NUA (WRITE) XXX351 RES REGISTER (WRITE) XXX352 INIT REGISTER (WRITE) XXX353 NOP	
MFPD Move from previous	MS	1065SS 0065SS	tmp ← (src) −(SP) ← tmp	N: set if the source < 0 Z: set if the source = 0 V: cleared	Pushes a word onto the current R6 stack from an address in previous space determined by PS<13:12>.

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
data space MFPI Move from previous instruction space				C: unaffected	The source address is computed using the current registers and memory map. When MFPI is executed and both previous mode and current mode are User, the instruction functions as though it were MFPD. Not implemented on the PDP-11/04. MFPD is identical to MFPI on the PDP 11/34A and 11/60, and on the 11/44 and 11/70 when D-space is disabled.
MFPS Move Byte from PSW	MS	1067DD	(dst) ← PS<7:0> dst lower 8 bits	N: set if PS bit 7 = 1 Z: set if PS <0:7> = 0 V: cleared C: not affected	The 8-bit contents of the PS are moved to the effective destination. If destination is mode 0, PS bit 7 is sign extended through the upper byte of the register. The destination operand is treated as a byte address. 11/34A only.

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
MFPT Move From Processor (PDP-11/44 ONLY	MS	000007	R0<7:0> ← processor model code R0<15:8> ←processor sub-code	N: not affected Z: not affected V: not affected C: not affected	No source operands are used. The MFPT instructions returns in the low byte of R0 a processor model code (1 on the PDP-11/44). The high byte of R0 is loaded with a processor specific subcode, (currently 0 on the PDP-11/44). The condition codes are not affected. The previous contents of R0 are lost. Note: On processors where this instruction is not implemented, a reserved instruction trap through vector 10_8 is taken.
MNS Maintenance normalization shift	MN	170004	On the 11/60, rounds the contents of FSPAD (0) in bit position 34 (02) for floating (double) precision number; left-shifts two places the results of the rounding operation (this action effectively drops the hidden bit); normalizes the resulting number using the NORMK indirect control of the shifter (result is left in FSPAD (1)); adjusts the exponent of ACO (E(0)) to reflect normalization. Result is left in E(1).		

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
MOV MOVB Move	DO	01SSDD 11SSDD	(dst) ← (src)	N: set if (src) < 0 Z: set if (src) = 0 V: cleared C: not affected	Moves the source operand to the destination location. The previous contents of the destination are lost. The source operand is not affected. Byte: Same as MOV. The MOVB to a register (unique among byte instructions) extends the most significant bit of the low order byte (sign extension). Otherwise MOVB operates on bytes exactly as MOV operates on words.
MPP Maintenance Partial Product	MN	170005	In the 11/60, used for diagnostic purposes to test the multiplication network (MULNET). A 36-bit partial product (MNETCARRY plus MNETSUM) and a 36-bit limited product (MNETSUM) is generated from: FSPA (0) <31:03> . FSPAD (0) <42:35>. The result is stored in FSPAD (1) <58:23> (MNETSUM), and FSPAD (2) .		

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
			<58:23> (MNETSUM plus MNETCARRY). The exponents of FSPAD (1) and FSPAD (2) save the information needed to establish the contents of the most significant bit (AR <58>) of MNETSUM and MNETSUM plus MNETCARRY.		
MTPD Move to previous data space MTPI Move to previous instruction space	MS	1066DD 0066DD	tmp ← SP+ (dst) ← tmp	N: set if the source < 0 Z: set if the source = 0 V: cleared C: unaffected	This instruction pops a word off the current R6 stack determined by PS (bits 15,14) and stores that word into an address in previous space determined by PS (bits 13,12). The destination address is computed using the current registers and memory map. Not implemented on the PDP-11/04. MTPD is identical to MTPI on the 11/34A and 11/60, and on the 11/44 and 11/70 when D-space is disabled.
MTPS	MS	1064SS	PS ← (src)	N: set according to ef-	The 8 bits of the effective operand

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
Move Byte to PSW				fective src operand 0-3 Z: same as above V: same as above C: same as above	replace the current contents of the PS <0:7>. The source operand address is treated as a byte address. Note that PS bit 4 cannot be set with this instruction. The src operand remains unchanged. 11/34A only.
MUL Multipy	DO	070RSS	R,Rv1 ← R × (src)	N: set if product < 0 Z: set if product = 0 V: cleared C: set if the result is less than -2^{15} or greater than or equal to 2^{15}. Condition codes set on 32-bit result even if R is odd.	The contents of the destination register and source taken as 2's complement integers are multiplied and stored in the destination register and the succeeding register (if R is even). If R is odd, only the low order product is stored. Assembler syntax is: MUL S,R. (Note that the actual destination is R, Rv1, which reduces to just R when R is odd.)

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
NEG NEGB Negate	SO	0054DD 1054DD	(dst) ← −(dst)	N: set if result < 0 Z: set if result = 0 V: set if result = M.N.I. C: cleared if result = 0; set otherwise	Replaces the contents of the destination address by its 2's complement. Note that 100000 is replaced by itself.
RESET	MS	000005		N: unaffected Z: unaffected V: unaffected C: unaffected	Sends INIT on the UNIBUS for 10ms. All devices on the unit are reset to their state at power-up. Within the PDP-11/60 processor, the stack limit and memory management register, MMR0, are intialized.
ROL ROLB Rotate Left	SO	0061DD 1061DD	(dst) ← (dst) rotate left one place	N: set if the high order bit of the result word is set (result < 0) Z: set if all bits of the result = 0 V: loaded with the ex-	Rotates all bits of the destination left one place. The high order bit is loaded into the C bit of the status word and the previous contents of the C bit are loaded into the low order bit of the destination.

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
				clusive OR of the N bit and C bit (as set by the completion of the rotate operation) C: set if the high order bit of the destination was set	
ROR RORB Rotate Right	SO	0060DD 1060DD	(dst) ← (dst) rotate right one place	N: set if high order bit of the result is set Z: set if all bits of result are 0 V: loaded with the exclusive OR of the N bit and the C bit as set by ROR C: set if the low order bit of the destination was set	Rotates all bits of the destination right one place. The low order bit is loaded into the C bit and the previous contents of the C bit are loaded into the high order bit of the destination.

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
RTI Return from Interrupt	MS	000002	PC ← (SP)+ PS ← (SP)+	N: loaded from current R6 stack Z: loaded from current R6 stack V: loaded from current R6 stack C: loaded from current R6 stack	Used to exit from an interrupt or trap service routine. The PC and PS are restored (popped) from the R6 stack. If the RTI sets the T bit in the PS, a trace trap will occur prior to executing the next instruction. When executed in Supervisor mode, the current and previous mode bits in the restored PS cannot be Kernel. When executed in User mode, the current and previous mode bits in the restored PS can only be User. RTI cannot clear PS<11> if it was already set.
RTS Return from Subroutine	PC	00020R	PC ← (reg) (reg) ← (SP)+	N: unaffected Z: unaffected V: unaffected C: unaffected	Loads contents of register into PC and pops the top element of the R6 stack into the specified register. Return from a non-reentrant subroutine is made through the same

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
					register that was used in its call. Thus, a subroutine called with a JSR PC,dst exits with an RTS PC, and a subroutine called with a JSR R5,dst may pick up parameters with addressing modes (R5)+, X(R5), or @X(R5) and finally exit, with an RTS R5.
RTT Return from Interrupt	MS	000006	PC ← (SP)+ PS ← (SP)+	N: loaded from current R6 stack Z: loaded from current R6 stack V: loaded from current R6 stack C: loaded from current R6 stack	This is the same as the RTI instruction (used to exit from an interrupt or trap service routine; the PC and PS are restored (popped) from the processor stack; if the RTI sets the T bit in the PS, a trace trap will occur prior to executing the next instruction) except that it inhibits a trace trap, while RTI permits a trace trap. If a trace trap is pending, the first instruction after the RTT will be executed prior to the

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
					next "T" trap. In the case of the RTI instruction, the "T" trap will occur immediately after the RTI. When executed in Supervisor mode, the current and previous mode bits in the restored PS cannot be Kernel. When executed in User mode, the current and previous mode bits in the restored PS can only be User. RTT cannot clear PS<11> if it was already set.
SBC SBCB Subtract Carry	SO	0056DD 1056DD	(dst) ← (dst) − C	N: set if result < 0 Z: set if result = 0 V: set if (dst) = M.N.I. C: set if (dst) was 0 and C was 1	Subtracts the contents of the C bit from the destination.
S Set selected	CC	000260 PLUS 4-bit mask	Set condition code bits. Selectable combinations of these bits may be set together. Condition code bits corresponding to bits in the condition code operator (Bits 0-3) are modified; set the bit specified by bit 0, 1, 2, or 3. Bit 4 is a 1.		

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
condition codes			**Operation:** PSW <3:0> ← PSW <3:0> v mask <3:0>		
SCC Set all condition codes	CC	000277	N, Z, V, C ← 1		
SEC Set C	CC	000261	C ← 1		
SEN Set N	CC	000270	N ← 1		
SEV Set V	CC	000262	V ← 1		
SEZ Set Z	CC	000264	Z ← 1		

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
SOB Subtract one and branch if not equal to 0	PC	077R00 PLUS 6-bit offset	R ← R−1 if this result does not = 0 then PC ← PC − (2 × offset)	N: unaffected Z: unaffected V: unaffected C: unaffected	The register is decremented. If it is not equal to 0, twice the offset is subtracted from the PC (now pointing to the following word). The offset is interpreted as a 6-bit positive number. This instruction provides a fast, efficient method of loop control. Assembler syntax is: SOB R,A where A is the address to which transfer is to be made if the decremented R is not equal to 0. Note that the SOB instruction cannot be used to transfer control in the forward direction.
SPL Set priority level	PC	00023N	PS (bits 7-5) ← Priority (priority = N)	N: not affected Z: not affected V: not affected C: not affected	The least significant three bits of the instruction are loaded into the program status word (PS), bits 7-5, thus causing a changed priority. The old priority is lost.

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
					Assembler syntax is: SPL N **Note:** If used on the 11/60, it results in a processor trap through vector address 10.
SUB Subtract	DO	16SSDD	(dst) ← (dst) − (src) {in detail (dst) ← (dst)+ ~ (src)+1	N: set if result < 0 Z: set if result = 0 V: set if there is arithmetic overflow as a result of the operation, i.e., if the operands were of opposite signs and the sign of the source is the same as the sign of the result C: set if there is a borrow into the most significant bit of the result, i.e., if	Subtracts the source operand from the destination operand and leaves the result at the destination address. The original contents of the destination are lost. The contents of the source are not affected. In double precision arithmetic, the C bit, when set, indicates a borrow.

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
				(dst) + ~ (src)+1 was less than 2**16.	
SWAB Swap Bytes	SO	0003DD	Byte 1 ← Byte 0 Byte 0 ← Byte 1	N: set if high order bit of low order byte (bit 7) of result is set Z: set if low order byte of result = 0 V: cleared C: cleared	Exchanges high order byte and low order byte of the destination word (destination must be a word address).
SXT Sign Extend	SO	0067DD	(dst) ← 0 if N bit is clear (dst) ← −1 if N bit is set	N: unaffected Z: set if N bit clear V: cleared C: unaffected	If the condition code bit N is set, then a −1 is placed in the destination operand; if the N bit is clear, then a 0 is placed in the destination operand. This instruction is particularly useful in multiple precision arithmetic because it permits the sign to be extended through multiple words.

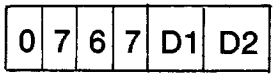
Table 4-1 PDP-11 Instruction Set, cont.

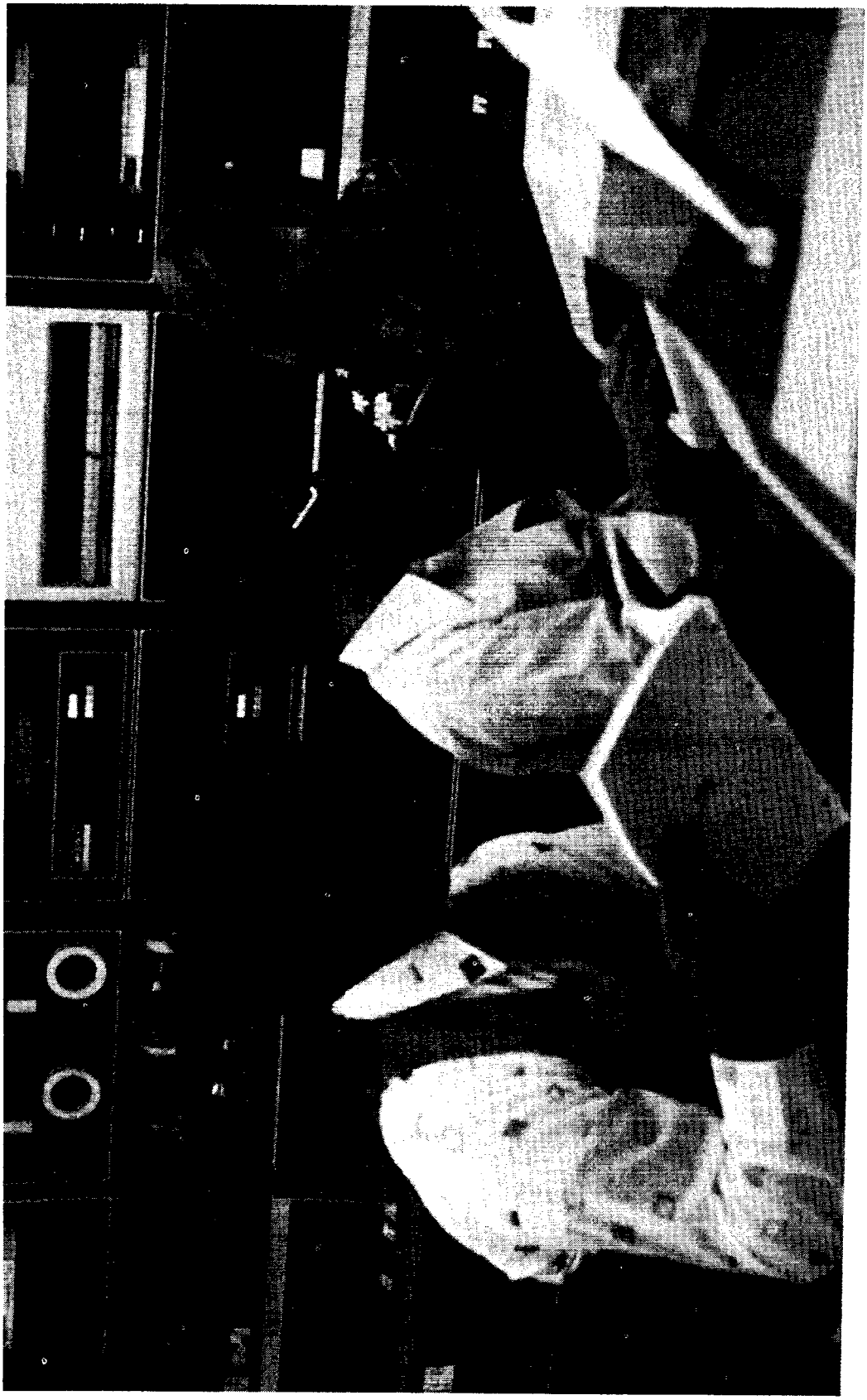
Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
TRAP	PC	104400 to 104777	−(SP) ← PS −(SP) ← PC PC ← (34) PS ← (36)	N: loaded from trap vector Z: loaded from trap vector V: loaded from trap vector C: loaded from trap vector	Operation codes from 104400 to 104777 are TRAP instructions. TRAPs and EMTs are identical in operation, except that the trap vector for TRAP is at address 34. **Note:** Since DIGITAL software makes frequent use of EMT, the TRAP instruction is recommended for general use.
TST TSTB Test	SO	0057DD 1057DD	tmp ← (dst)	N: set if result < 0 Z: set if result = 0 V: cleared C: cleared	Sets the condition codes N and Z according to the contents of the destination address.
WAIT Wait for Interrupt	MS	000001		N: unaffected Z: unaffected V: unaffected C: unaffected	Provides a way for the processor to relinquish use of the bus while it waits for an external interrupt. Having been given a WAIT command, the processor will not compete for the bus by fetching instructions or

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
					operands from memory. This permits higher transfer rates between device and memory, since no processor-induced latencies will be encountered by bus requests from the device. In WAIT, as in all instructions, the PC points to the next instruction following the WAIT operation. Thus, when an interrupt causes the PC and PS to be pushed onto the stack, the address of the next instruction following the WAIT is saved. The exit from the interrupt routine (i.e., execution of an RTI instruction) will cause resumption of the interrupted process at the instruction following the WAIT.
XFC					The UCS (User Control Store) option for the PDP-11/60 utilizes the XFC in-

Table 4-1 PDP-11 Instruction Set, cont.

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
Extended Function Code			struction. Details on use are contained in documentation associated with the UCS option. Extended Function Code (USER)  This instruction provides dispatch information to the user control store or extended control store. The D1 field is used for initial instruction group determination, with further instruction determination by D2 field or additional MACRO instruction words. If the option is not enabled, a trap through vector address 10 occurs.		
XOR Exclusive OR	DO	074RDD	(dst) ← R V (dst)	N: set if the result < 0 Z: set if result = 0 V: cleared C: unaffected	The exclusive OR of the register and destination operand is stored in the destination address. Contents of register are unaffected. Assembler format is XOR R,D



CHAPTER 5
PROGRAMMING TECHNIQUES

The PDP-11 processors offer you a great deal of programming flexibility and power. The combination of the instruction set, the addressing modes, and the programming techniques makes it possible to develop new software or to utilize old programs effectively. The programming techniques in this chapter show methods which exploit the unique capabilities of the PDP-11 processors. The techniques specifically discussed are: position-independent coding (PIC), stacks, subroutines, interrupts, reentrancy, coroutines, recursion, processor traps, and conversion.

POSITION-INDEPENDENT CODE

The output of a MACRO-11 assembly is a relocatable object module. The task builder or linker binds one or more modules together to create an executable task image. Once built, a task can generally be loaded and executed only at the address specified at link time. This is because the linker has had to modify some instructions to reflect the memory locations in which the program is to run. Such a body of code is considered position dependent (i.e., dependent on the virtual addresses to which it was bound).

All PDP-11 processors offer addressing modes that make it possible to write instructions that are not dependent on the virtual addresses to which they are bound. A body of such code is termed position-independent and can be loaded and executed at any virtual address. Position-independent code can improve system efficiency, both in the use of virtual address space and in the conservation of physical memory.

In multiprogramming systems like IAS and RSX-11M, it is important that many tasks be able to share a single physical copy of common code; for example, a library routine. To make the optimum use of a task's virtual address space, shared code should be position-independent. Code that is not position independent can also be shared, but it must appear in the same locations in every task using it. This restricts the placement of such code by the task builder and can result in the loss of virtual addressing space.

The construction of position-independent code is closely linked to the proper use of PDP-11 addressing modes. The remainder of this explanation assumes that you are familiar with the addressing modes described in Chapter 3.

All addressing modes involving only register references are position independent. These modes are:

R	register mode
(R)	register deferred mode
(R)+	autoincrement mode
@(R)+	autoincrement deferred mode
−(R)	autodecrement mode
@−(R)	autodecrement deferred mode

When using these addressing modes, you are guaranteed position independence, providing that the contents of the registers have been supplied independent of a particular memory location.

The relative addressing modes are position-independent when a relocatable address is referenced from a relocatable instruction. These modes are as follows:

A	PC relative mode
@A	PC relative deferred mode

Relative modes are not position-independent when an absolute address (that is, a non-relocatable address) is referenced from a relocatable instruction. In this case, absolute addressing (i.e., @#A) may be employed to make the reference position-independent.

Index modes can be either position-independent or position-dependent, according to their use in the program. These modes are as follows:

X(R)	index mode
@X(R)	index deferred mode

If the base, X, is an absolute value (e.g., a control block offset), the reference is position-independent. For example:

```
        MOV     2(SP),R0            ;POSITION INDEPENDENT
N=4
        MOV     N(SP),R0            ;POSITION INDEPENDENT
```

If, however, X is a relocatable address, the reference is position dependent. For example:

```
        CLR     ADDR(R1)            ;POSITION DEPENDENT
```

Immediate mode can be either position independent or not, according to its use. Immediate mode references are formatted as follows:

#N	immediate mode

When an absolute expression defines the value of N, the code is position-independent. When a relocatable expression defines N, the code is position-dependent. That is, immediate mode references are position independent only when N is an absolute value.

Absolute mode addressing is position-independent only in those cases where an absolute virtual location is being referenced. Absolute mode addressing references are formatted as follows:

```
@#A        absolute mode
```

An example of a position-independent absolute reference is a reference to the directive status word ($DSW) from a relocatable instruction. For example:

```
        MOV     @#$DSW,R0       ;RETRIEVE DIRECTIVE
                                ;STATUS
```

EXAMPLES

The RSX-11 library routine, PWRUP, is a FORTRAN-callable subroutine to establish or remove a user power failure AST (Asynchronous System Trap) entry point address. Imbedded within the routine is the actual AST entry point which saves all registers, effects a call to the user-specified entry point, restores all registers on return, and executes an AST exit directive. The following examples are excerpts from this routine. The first example has been modified to illustrate position-dependent references. The second example is the position-independent version.

Position-Dependent Code

```
PWRUP:
        CLR     -(SP)           ;ASSUME SUCCESS
        CALL    .X.PAA          ;PUSH (SAVE)
                                ;ARGUMENT ADDRESSES
                                ;ONTO STACK
        .WORD   1.,$DSW         ;CLEAR DSW, AND
                                ;SET R1=R2=SP
        MOV     $OTSV,R4        ;GET OTS IMPURE
                                ;AREA POINTER
        MOV     (SP)+,R2        ;GET AST ENTRY
                                ;POINT ADDRESS
        BNE     10$             ;IF NONE SPECIFIED,
                                ;SPECIFY NO POWER
        CLR     -(SP)           ;RECOVERY AST SERVICE
        BR      20$             ;
                                ;
10$:
        MOV     R2,F.PF(R4)     ;SET AST ENTRY POINT
        MOV     #BA,-(SP)       ;PUSH AST SERVICE
                                ;ADDRESS
```

```
20$:                                        ;
        CALL    .X.EXT                      ;ISSUE DIRECTIVE, EXIT.
        .BYTE   109.,2.                     ;
        .
        .
        .
BA:     MOV     R0,-(SP)                    ;PUSH (SAVE) R0
        MOV     R1,-(SP)                    ;PUSH (SAVE) R1
        MOV     R2,-(SP)                    ;PUSH (SAVE) R2
```

Position-Independent Code

```
PWRUP:
        CLR     -(SP)                       ;ASSUME SUCCESS
        CALL    .X.PAA                      ;PUSH ARGUMENT
                                            ;ADDRESSES ONTO
                                            ;STACK
        .WORD   1.,$DSW                     ;CLEAR DSW, AND
                                            ;SET R1=R2=SP.
        MOV     @#$OTSV,R4                  ;GET OTS IMPURE
                                            ;AREA POINTER
        MOV     (SP)+,R2                    ;GET AST ENTRY
                                            ;POINT ADDRESS
        BNE     10$                         ;IF NONE SPECIFIED,
                                            ;SPECIFY NO POWER
        CLR     -(SP)                       ;RECOVERY AST SERVICE
        BR      20$
10$:                                        ;
        MOV     R2,F.PF(R4)                 ;SET AST ENTRY POINT
        MOV     PC,-(SP)                    ;PUSH CURRENT LOCATION
        ADD     #BA-.,(SP)                  ;COMPUTE ACTUAL
                                            ;LOCATION
                                            ;OF AST
20$:
        CALL    .X.EXT                      ;ISSUE DIRECTIVE, EXIT.
        .BYTE   109.,2.
;
;ACTUAL AST SERVICE ROUTINE:
;
;       1) SAVE REGISTERS
:       2) EFFECT A CALL TO SPECIFIED SUBROUTINE
;       3) RESTORE REGISTERS
;       4) ISSUE AST EXIT DIRECTIVE
;
```

```
BA:     MOV     R0,-(SP)        ;PUSH (SAVE) R0
        MOV     R1,-(SP)        ;PUSH (SAVE) R1
        MOV     R2,-(SP)        ;PUSH (SAVE) R2
```

The position-dependent version of the subroutine contains a relative reference to an absolute symbol ($OTSV) and a literal reference to a relocatable symbol (BA). Both references are bound by the task builder to fixed memory locations. Therefore, the routine will not execute properly as part of a resident library if its location in virtual memory is not the same as the location specified at link time.

In the position-independent version, the reference to $OTSV has been changed to an absolute reference. In addition, the necessary code has been added to compute the virtual location of BA based upon the value of the program counter. In this case, the value is obtained by adding the value of the program counter to the fixed displacement between the current location and the specified symbol. Thus, execution of the modified routine is not affected by its location in the image's virtual address space.

STACKS

The stack is part of the basic design architecture of the PDP-11. It is an area of memory set aside by the programmer or by the operating system for temporary storage and linkage. It is handled on a LIFO (last-in/first-out) basis, where items are retrieved in the reverse of the order in which they were stored. On a PDP-11 processor, a stack starts at the highest location reserved for it and expands linearly downward to a lower address as items are added to the stack.

You do not need to keep track of the actual locations into which data is being stacked. This is done automatically through a stack pointer. To keep track of the last item added to the stack, a general register always contains the memory address when the last item is stored in the stack. Any register except register 7 (the PC) may be used as a stack pointer under program control; however, instructions associated with subroutine linkage and interrupt service automatically use register 6 as a *hardware* stack pointer. For this reason, R6 is frequently referred to as the system SP. Stacks may be maintained in either full word or byte units. This is true for a stack pointed to by any register except R6, which must be organized in full word units only. Byte stacks, Figure 5-1, require instructions capable of operating on bytes rather than full words.

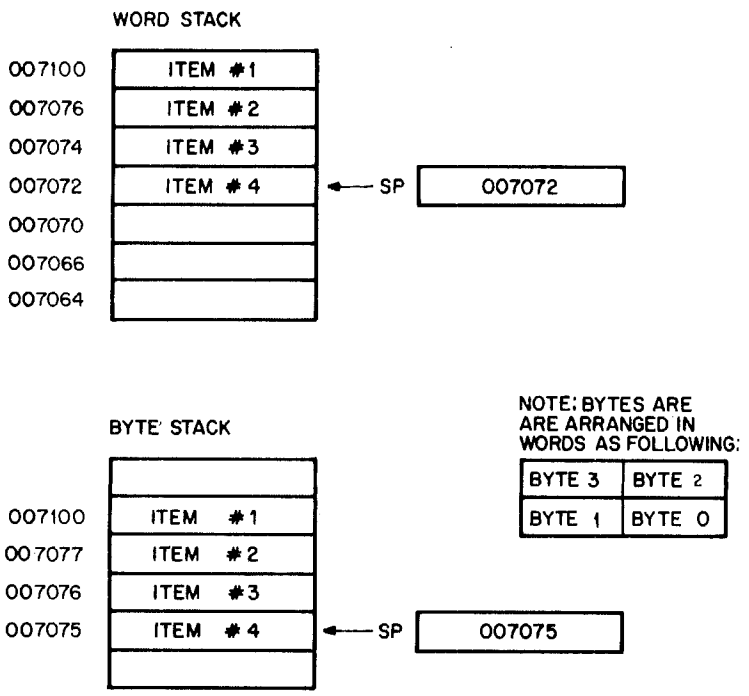


Figure 5-1 Word and Byte Stacks

Items are added to a stack using the autodecrement addressing mode. Adding items to the stack is called PUSHing, and is accomplished by the following instructions:

```
MOV     Source,-(SP)     ;MOV Contents of Source Word
                         ;onto the stack
                                  or
MOVB    Source,-(SP)     ;MOVB Source Byte onto
                         ;the stack
```

Data is thus PUSHed onto the stack.

Removing data from the stack is called a POP (popping from the stack). This operation is accomplished using the autoincrement mode:

```
MOV     (SP)+, Destination     ;MOV Destination Word
                               ;off the stack
                                        or
MOVB    (SP)+, Destination     ;MOVB Destination Byte
                               ;off the stack
```

After an item has been popped, its stack location is considered free and available for other use. The stack pointer points to the last used location, implying that the next lower location is free. Thus, a stack may represent a pool of temporary storage locations.

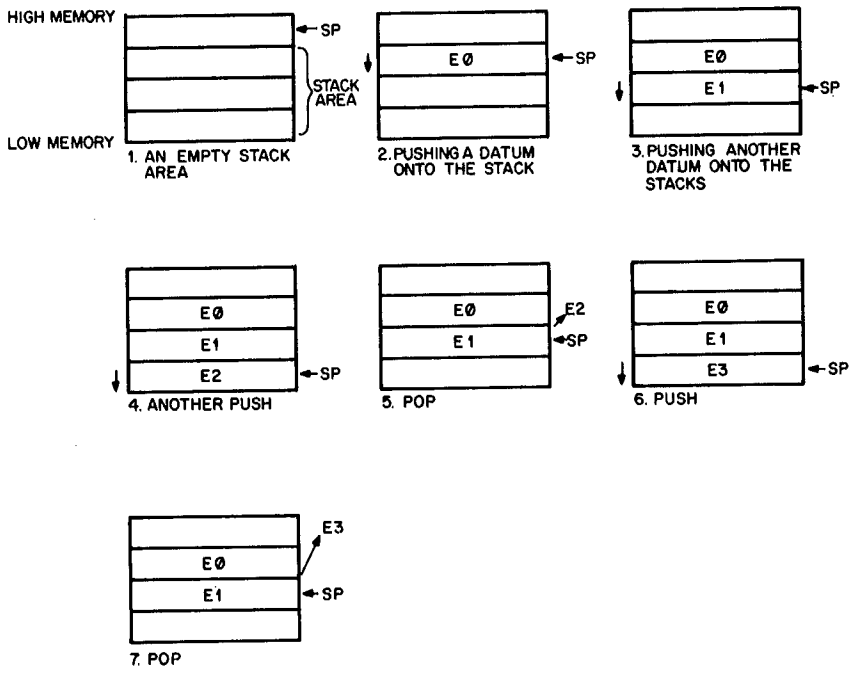


Figure 5-2 Illustration of Push and Pop Operations

Uses for the stack

- Often one of the general purpose registers must be used in a subroutine or interrupt service routine and then returned to its original value. The stack can be used to store the contents of the registers involved.
- The stack is used in storing linkage information between a subroutine and its calling program. The JSR instruction, used in calling a subroutine, requires the specification of a linkage register along with the entry address of the subroutine. The content of this linkage register is stored on the stack, so as not to be lost, and the return address is moved from the PC to the linkage register. This provides a pointer back to the calling program so that successive arguments may be transmitted easily to the subroutine.
- If no arguments need be passed by stacking them after the JSR instruction, the PC may be used as the linkage register. In this case, the result of the JSR is to move the return address in the calling program from the PC onto the stack and replace it with the entry address of the called subroutine.
- In many cases, the operations performed by the subroutine can be applied directly to the data located on or pointed to by a stack without the need ever actually to move the data into the subroutine area.

```
;CALLING PROGRAM
MOV     SP,R1           ;R1 IS USED AS THE STACK
JSR     PC,SUBR         ;POINTER HERE

;SUBROUTINE
ADD     (R1)+,(R1)      ;ADD ITEM #1 to #2,PLACE
                        ;RESULT IN ITEM #2,
                        ;R1 POINTS TO
                        ;ITEM #2 NOW
```

Because the hardware already uses general purpose register R6 to point to a stack for saving and restoring PC and processor status word (PS) information, it is convenient to use this same stack to save and restore immediate results and to transmit arguments to and from subroutines. Using R6 in this manner permits extreme flexibility in nesting subroutines and interrupt service routines.

Since arguments may be obtained from the stack by using some form of register indexed addressing, it is sometimes useful to save a temporary copy of R6 in some other register which has been saved at the beginning of a subroutine. If R6 is saved in R5 at the beginning of the subroutine, R5 may be used to index the arguments while R6 is free to be incremented and decremented in the course of being used as a stack pointer. If R6 had been used directly as the base for indexing and not "copied," it might be difficult to keep track of the position in the argument list, since the base of the stack would change with every autoincrement/decrement which occurs.

However, if the contents of R6 (SP) are saved in R5 before any arguments are pushed onto the stack, the position relative to R5 would remain constant.

Return from a subroutine also involves the stack, as the return instruction, RTS, must retrieve information stored there by the JSR.

When a subroutine returns, it is necessary to "clean up" the stack by eliminating or skipping over the subroutine arguments. One way this can be done is by insisting that the subroutine keep the number of arguments as its first stack item. Returns from subroutines then involve calculating the amount by which to reset the stack pointer, resetting the stack pointer, then storing the original contents of the register used as the copy of the stack pointer.

- Stack storage is used in trap and interrupt linkage. The program counter and the processor status word of the executing program are pushed on the stack.

- When using the system stack, nesting of subroutines, interrupts, and traps to any level can occur until the stack overflows its legal limits.
- The stack method is also available for temporary storage of any kind of data. It may be used as a LIFO list for storing inputs, intermediate results, etc.

As an example of stack use consider this situation: a subroutine (SUBR) wants to use registers 1 and 2, but these registers must be returned to the calling program with their contents unchanged. The subroutine could be written as follows:

Address	Octal Code	Assembler Syntax	Comments
076322	010167 SUBR:	MOV R1,TEMP1	;save R1
076324	000074	*	
076326	010267	MOV R2,TEMP2	;save R2
076330	000072	*	
.	.	.	
.	.	.	
.	.	.	
076410	016701	MOV TEMP1,R1	;restore R1
076412	000006	*	
076414	0167902	MOV TEMP2,R2	;restore R2
076416	000004	*	
076420	000297	RTS PC	
076422	000000	TEMP1: 0	
076424	000000	TEMP2: 0	

* Index Constants

OR: Using the Stack

R3 has been previously set to point to the end of an unused block of memory.

Address	Octal Code	Assembler Syntax	Comments
010020	010143 SUBR:	MOV R1,−(R3)	;push R1
010022	010243	MOV R2,−(R3)	;push R2
.	.	.	
.	.	.	
.	.	.	
.	.	.	

```
010130      012302      MOV (R3)+,R2    ;pop R2
010132      012301      MOV (R3)+,R1    ;pop R1
010134      000207      RTS PC
```

Note: In this case R3 was used as a stack pointer.

The second routine uses four fewer words of instruction code and two words of temporary "stack" storage. Another routine could use the same stack space at some later point. Thus, the ability to share temporary storage in the form of a stack is a way to save on memory use.

As another example of stack use, consider the task of managing an input buffer from a terminal. As characters come in, you may wish to delete characters from the line; this is accomplished very easily by maintaining a byte stack containing the input characters. Whenever a backspace is received, a character is "popped" off the stack and eliminated from consideration. In this example, you have the choice of "popping" characters to be eliminated by using either the MOVB (MOVE BYTE) or INC (INCREMENT) instructions.

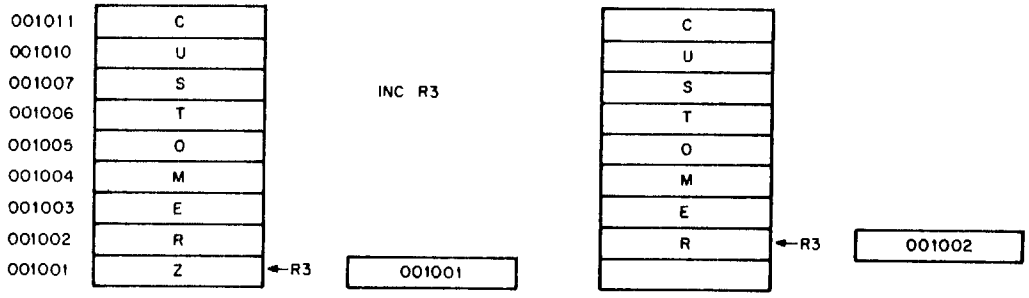


Figure 5-3 Byte Stack Used as a Character Buffer

NOTE

In this case the increment instruction (INC) is preferable to MOVB, since it accomplishes the task of eliminating the unwanted character from the stack by readjusting the stack pointer without the need for a destination location. Also, the stack pointer (SP) used in this example cannot be the system stack pointer (R6) because R6 may point only to word (even) locations.

DELETING ITEMS FROM A STACK

To delete one item:

INC SP or TSTB(SP)+ for a byte stack

To delete two items:

ADD#2,SP or TST(SP)+ for word stack

To delete fifty items from a word stack:

ADD #100.,SP

SUBROUTINE LINKAGE

The contents of the linkage register are saved on the system stack when a JSR is executed. The effect is the same as if a MOV reg,−(R6) had been performed. Following the JSR instruction, the same register is loaded with the memory address (the contents of the current PC), and a jump is made to the entry location specified.

The JSR figure, Figure 5-4, gives the before and after conditions when executing the subroutine instructions JSR R5,1064.

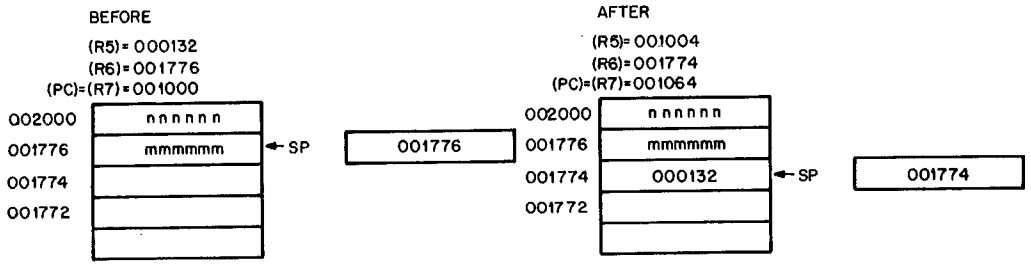


Figure 5-4 JSR

Because the PDP-11 hardware already uses general purpose register R6 to point to a stack for saving and restoring PC and PS (processor status word) information, it is convenient to use this same stack to save and restore intermediate results and to transmit arguments to and from subroutines. Using R6 this way permits nesting subroutines and interrupt service routines.

Return from a Subroutine

An RTS instruction provides for a return from the subroutine to the calling program. The RTS instruction must specify the same register as the one the JSR instruction used in the subroutine call. When the RTS is executed, the register specified is moved to the PC, and the top of the stack to be placed in the register specified. Thus, an RTS PC has the effect of returning to the address specified on the top of the stack.

Subroutine Advantages

There are several advantages to the PDP-11 subroutine calling procedure, effected by the JSR instruction.

- Arguments can be passed quickly between the calling program and the subroutine.
- If there are no arguments, or the arguments are in a general register or on the stack, the JSR PC,DST mode can be used so that none of the general purpose registers are used for linkage.
- Many JSRs can be executed without the need to provide any saving procedure for the linkage information, since all linkage information is automatically pushed onto the stack in sequential order. Returns can be made by automatically popping this information from the stack in the order opposite to the JSRs.

Such linkage address bookkeeping is called automatic "nesting" of subroutine calls. This feature enables you to construct fast, efficient linkages in a simple, flexible manner. It also permits a routine to call itself in those cases where this is meaningful.

INTERRUPTS

An interrupt is similar to a subroutine call, except that it is initiated by the hardware rather than by the software. An interrupt can occur after the execution of an instruction.

Interrupt-driven techniques are used to reduce CPU waiting time. In direct program data transfer, the CPU loops to check the state of the DONE/READY flag (bit 7) in the peripheral interface. Using interrupts, the system actually ignores the peripheral, running its own low-priority program until the peripheral initiates service by setting the DONE bit. The interrupt enable bit in the control status register must have been set at some prior point. The CPU completes the instruction being executed and then is interrupted and vectors to an interrupt service routine. The service routine will transfer the data and may perform calculations with it. After the interrupt service routine has been completed, the computer resumes the program that was interrupted by the peripheral's high-priority request.

With interrupt service routines, linkage information is passed so that a return to the main program can be made. More information is necessary for an interrupt sequence than for a subroutine call because of the random nature of interrupts. The complete machine state of the program immediately prior to the occurrence of the interrupt must be preserved in order to return to the program without any noticeable effects. This information is stored in the processor status word (PS).

Upon interrupt, the contents of the program counter (PC) (address of next instruction) and the PS are automatically pushed onto the R6 system stack. The effect is the same as if:

```
MOV PS,-(SP)      ;Push PS
MOV PC,-(SP)      ;Push PC
```

had been executed.

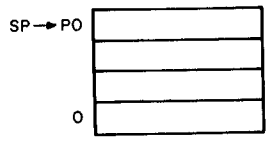
The new contents of the PC and PS are loaded from two preassigned consecutive memory locations which are called "vector addresses." The first word contains the interrupt service routine entry address (the address of the service routine program sequence), and the second word contains the new PS which will determine the machine status, including the operational mode and register set to be used by the interrupt service routine. The contents of the vector address are set under program control.

After the interrupt service routine has been completed, an RTI (return from interrupt) is performed. The top two words of the stack are automatically "popped" and placed in the PC and PS respectively, thus resuming the interrupted program.

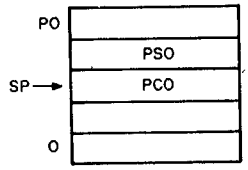
Nesting

Interrupts can be nested in much the same manner that subroutines are nested. In fact, it is possible to nest any arbitrary mixture of subroutines and interrupts without any confusion. By using the RTI and RTS instructions, respectively, the proper returns are automatic.

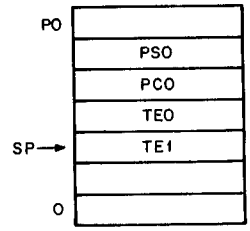
1. Process 0 is running; SP is pointing to location P0.



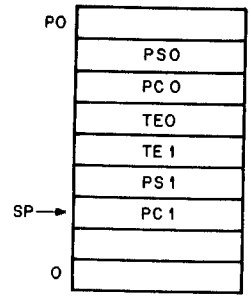
2. Interrupt stops process 0 with PC = PC0, and status = PS0; starts process 1.



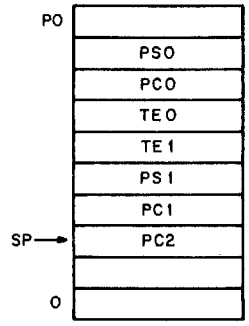
3. Process 1 uses stack for temporary storage (TEO, TE1).



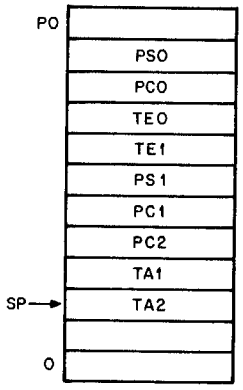
4. Process 1 interrupted with PC = PC1 and status = PS1; process 2 is started.



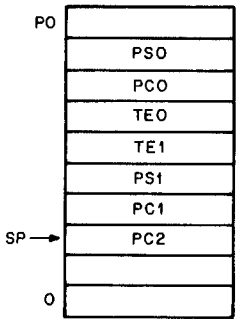
5. Process 2 is running and does a JSR R7,A to subroutine A with PC = PC2.



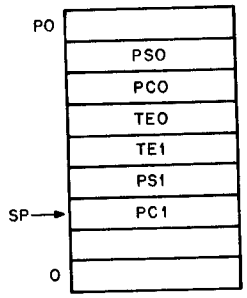
6. Subroutine A is running and uses stack for temporary storage.



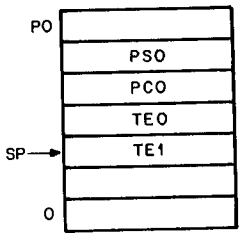
7. Subroutine A releases the temporary storage holding TA1 and TA2.



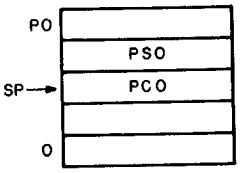
8. Subroutine A returns control to process 2 with an RTS R7; PC is reset to PC2.



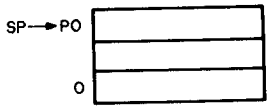
9. Process 2 completes with an RTI instructions (dismisses interrupt) PC is reset to PC1 and status is reset to PS1; process 1 resumes.



10. Process 1 releases the temporary storage holding TE0 and TE1.



11. Process 1 completes its operation with an RTI, PC is reset to PC0, and status is reset to PS0.



Nested Interrupt Service Routines and Subroutines

Note that the area of interrupt service programming is intimately involved with the concept of CPU and device priority levels.

REENTRANCY

Other advantages of the PDP-11 stack organization are obvious in programming systems that are engaged in concurrent handling of several tasks. Multi-task program environments range from simple single-user applications which manage a mixture of I/O interrupt service and background data processing, as in RT-11, to large complex multi-programming systems that manage an intricate mixture of executive and multi-user programming situations, as in RSX-11. In all these situations, using the stack as a programming technique provides flexibility and time/memory economy by allowing many tasks to use a single copy of the same routine with a simple straightforward way of keeping track of complex program linkages.

The ability to share a single copy of a program among users or among tasks is called **reentrancy**. Reentrant program routines differ from ordinary subroutines in that it is not necessary for reentrant routines to finish processing a given task before they can be used by another task. Multiple tasks can exist at any time in varying stages of completion in the same routine. Thus the following situation may occur.

(ART)	(ART)
PDP-11 Approach	Conventional Approach
Programs 1, 2, and 3 can share Subroutine A.	A separate copy of Subroutine A must be provided for each program.

Figure 5-6 Reentrant Routines

Reentrant Code

Reentrant routines must be written in pure code, code that is not self-modifying and consists entirely of instructions and constants.

Pure code (any code that consists exclusively of instructions and constants) may be used when writing any routine, even if the completed routine is not to be reenterable. The value of using pure code whenever possible is that the resulting code:

- is generally considered easier to debug
- can be kept in read-only memory (is read-only protected)

Using reentrant code, control of a routine can be shared as follows:

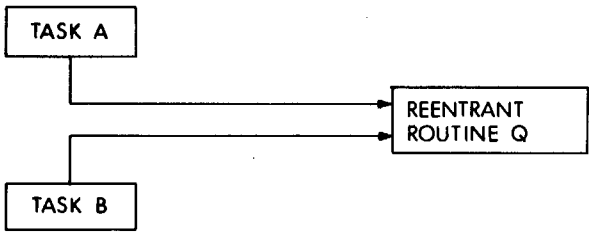


Figure 5-7 Sharing Control of a Routine

- Task A requests processing by Reentrant Routine Q.
- Task A temporarily relinquishes control of Reentrant Routine Q before it completes processing.
- Task B starts processing the same copy of Reentrant Routine Q.
- Task B completes processing by Reentrant Routine Q.
- Task A regains use of Reentrant Routine Q and resumes where it stopped.

Writing Reentrant Code

In an operating system environment, when one task is executing and is interrupted to allow another task to run, a context switch occurs which causes the processor status word and current contents of the general purpose registers (GPRs) to be saved and replaced by the appropriate values for the task being entered. Therefore, reentrant code should use the GPRs and the stack for any counters, pointers, or data that must be modified or manipulated in the routine.

The context switch occurs whenever a new task is allowed to execute. It causes all of the GPRs, the PS, and often other task-related information to be saved in an impure area, then reloads these registers and locations with the appropriate data for the task being entered. Notice that one consequence of this is that a new stack pointer value is loaded into R6, therefore causing a new area to be used as the stack when the second task is entered.

The following should be observed when writing reentrant code:

- All data should be in or pointed to by one of the general purpose registers.
- A stack can be used for temporary storage of data or pointers to impure areas within the task space. The pointer to such a stack would be stored in a GPR.

- Parameter addresses should be used by indexing and indirect reference rather than by putting them into instructions within the code.
- When temporary storage is accessed within the progam, it should be by indexed addresses, which can be set by the calling task in order to handle any possible recursion.

Use of Reentrant Code

Reentrant code is used whenever more than one task may reference the same code without requiring that each task complete processing with the code before the next may use it.

COROUTINES

In some programming situations it happens that several program segments or routines are highly interactive. Control is passed back and forth between the routines, each going through a period of suspension before being resumed. Since the routines maintain a symmetric relationship with each other, they are called **coroutines.**

Coroutines are two program sections, neither subordinate to the other, which can call each other. The nature of the call is "I have processed all I can for now, so you can execute until you are ready to stop, then I will continue."

The coroutine call and return are identical, each being a jump to subroutine instruction with the destination address being on top of the stack and the PC serving as the linkage register, i.e.,

```
JSR PC,@(R6)+
```

Coroutine Calls

The coding of coroutine calls is made simple by the PDP-11 stack feature. Initially, the entry address of the coroutine is placed on the stack and from that point the

```
JSR PC,@(R6)+
```

instruction is used for both the call and the return statements. The result of this JSR instruction is to exchange the contents of the PC and the top element of the stack, and so permit the two routines to swap control and resume operation where each was terminated by the previous swap.

For example:

Routine A	**Stack**	**Routine B**	**Comments**
.		.	LOC is pushed
.		.	onto the stack
.		.	to prepare for
MOV #LOC,–(SP)	LOC ←SP		the corou-
.			tine call.
.			
.		LOC:	
JSR PC,@(SP)+	PC0 ←SP	.	When the call
(PC0)		.	is executed,
		.	the PC from
			routine A is
			pushed on the
			stack and exe-
			cution contin-
			ues at LOC.
		JSR PC,@(SP)+	Routine B can
.	PC1←SP	(PC1)	return control
.		.	to routine A
.		.	by another
		.	coroutine call.
			PC0 is popped
			from the stack
			and execution
			resumes in
			routine A just
			after the call
			to Routine B,
			i.e., at PC0.
			PC1 is saved
			on the stack
			for a later
			return to
			Routine B.

Figure 5-8 Coroutine Example

Notice that the coroutine linkage cleans up the stack with each transfer of control.

Coroutines Versus Subroutines

- A subroutine can be considered to be subordinate to the main or calling routine, but a coroutine is considered to be on the same level, as each coroutine calls the other when it has completed current processing.
- A subroutine executes, when called, to the end of its code. When called again, the same code will execute before returning. A coroutine executes from the point after the last call of the other coroutine. Therefore, the same code will not be executed each time the coroutine is called. For example,

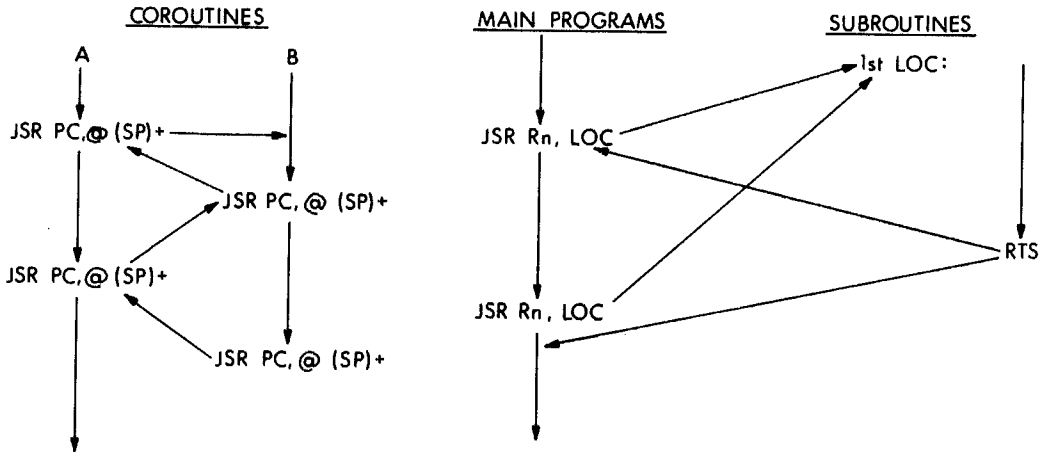


Figure 5-9 Coroutines vs. Subroutines

- The call and return statements for coroutines are the same:

  JSR PC,@(SP)+

  This one instruction also cleans up the stack with each call.

  The last coroutine call will leave an address on the stack that must be popped if no further calls are to be made.
- Each coroutine call returns to the coroutine code at the point after the last exit with no need for a specific entry point label, as would be required with subroutines.

Using Coroutines

- Coroutines should be used whenever two tasks must be coordinated in their execution without obscuring the basic structure of the program. For example, in decoding a line of assembly language

code, the results at any one position might indicate the next process to be entered. Where a label is detected, it must be processed. If no label is present, the operator must be located, etc.

- Coroutines should be employed to add clarity to the process being performed, to ease in the debugging phase, etc.

Examples

An assembler must perform a lexicographic scan of each assembly language statement during pass one of the assembly process. The various steps in such a scan should be separated from the main program flow to add to the program clarity and to aid in debugging by isolating many details. Subroutines would not be satisfactory here, as too much information would have to be passed to the subroutine each time it was called. This subroutine would be too isolated. Coroutines could be effectively used here with one routine being the assembly-pass-one routine and the other extracting one item at a time from the current input line.

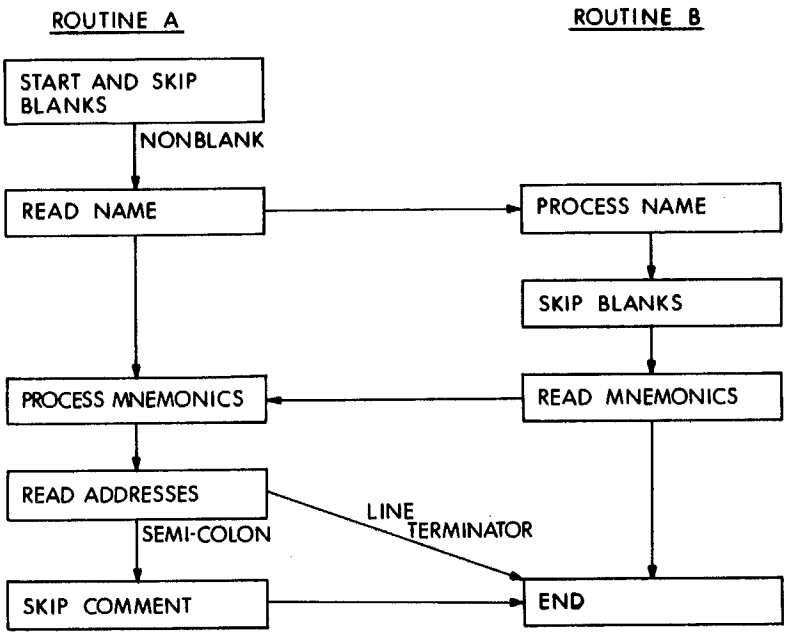


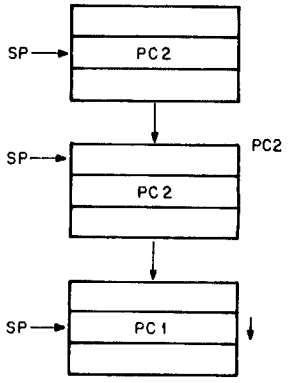
Figure 5-10 Coroutine Path

Coroutines can be utilized in I/O processing. The example shows co-routines used in double-buffered I/O using IOX. The flow of events might be described as:

```
Write 01
Read 11          concurrently
Process 12
```

then

Write 02
Read 12 concurrently
Process 11



Routine #1 is operating, it then executes:

```
MOV #PC2,-(R6)
JSR PC,@(R6)+
```

with the following results:

1. PC2 is popped from the stack and the SP autoincremented.
2. SP is autodecremented and the old PC (i.e. PC1) is pushed.
3. Control is transferred to the location PC2 (i.e. Routine #2).

Routine #2 is operating, it then executes:

```
JSR PC,@(R6)+
```

with the result that PC2 is exchanged for PC1 on the stack and control is transferred back to Routine #1.

Figure 5-11 Coroutine Interaction

RECURSION

An interesting aspect of a stack facility, other than its providing for automatic handling of nested subroutines and interrupts, is that a program may call on itself as a sub-routine just as it can call on any other routine. Each new call causes the return linkage to be placed on the stack, which, as it is a last-in/first-out queue, sets up a natural unraveling to each routine just after the point of departure.

Typical flow for a recursive routine might be something like this:

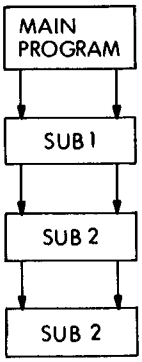


Figure 5-12 Recursive Routine Flow

The main program calls function one, SUB 1, which calls function two, SUB 2, which recurses once before returning.

Example:

```
DNCF:   '
        '
        '
        BEQ 1$              ;TO EXIT RECURSIVE LOOP
        JSR R5,DNCF         ;RECURSE
1$      '
        '
        '
        RTS R5              ;RETURN TO 1$ FOR
                            ;EACH CALL, THEN TO
                            ;MAIN PROGRAM
```

The routine DNCF calls itself until the variable tested becomes equal to zero, then it exits to 1$ where the RTS instruction is executed, returning to the 1$ once for each recursive call and one final time to return to the main program.

In general, recursion techniques will lead to slower programs than the corresponding interactive techniques, but the recursion will give shorter programs in memory space used. Both the brevity and clarity produced by recursion are important in assembly language programs.

Uses of Recursion

Recursion can be used in any routine in which the same process is required several times. For example, a function to be integrated may contain another function to be integrated, i.e., to solve for XM

where:

$$XM = 1 + \int_{0}^{x} F(X)$$

and:

$$F(X) = \int_{x}^{0} G(X)$$

Another use for a recursive function could be in calculating a factorial function because

$$FACT(N) = FACT(N-1) * N$$

Recursion should terminate when N = 1.

The macro processor within MACRO-11, for example, is itself recursive, as it can process nested macro definitions and calls. For example, within a macro definition, other macros can be called. When a macro call is encountered within definition, the processor must work recursively, i.e., to process one macro before it is finished with another, then to continue with the previous one. The stack is used for a separate storage area for the variables associated with each call to the procedure.

As long as nested definitions of macros are available, it is possible for a macro to call itself. However, unless conditionals are used to terminate this expansion, an infinite loop could be generated.

PROCESSOR TRAPS

There are a series of errors and programming conditions which will cause the central processor to trap to a set of fixed locations. These include power failure, odd addressing errors, stack errors, time out errors, memory parity errors, memory management violations, floating point processor exception traps, use of reserved instructions, use of the T bit in the processor status word, and use of the IOT, EMT, and TRAP instructions.

Power Failure

Whenever AC power drops below 95 volts for 115V power (190 volts for 230V) or outside a limit of 47 to 73 Hz, as measured by dc voltage, the power-fail sequence is initiated. The central processor automatically traps to location 24 and the power-fail program has 2 msec. to save all volatile information (data in registers), and to condition peripherals for power failure.

When power is restored, the processor traps to location 24 and executes the power-up routine to restore the machine to its state prior to power failure.

Odd Addressing Errors
This error occurs whenever a program attempts to execute a word instruction on an odd address (in the middle of a word boundary). The instruction is aborted and the CPU traps through location 4.

Time-out Errors
These errors occur when a master synchronization pulse is placed on the UNIBUS and there is no slave pulse within a certain length of time. This error usually occurs in attempts to address non-existent memory or peripherals. The typical UNIBUS time-out is 10 microseconds.

The offending instruction is aborted and the processor traps through location 4.

Reserved Instructions
There is a set of illegal and reserved instructions which cause the processor to trap through location 10.

Vector Address and Trap Errors

000	(reserved)
004	CPU errors
010	Illegal and reserved instructions
014	BPT, breakpoint trap
020	IOT, input/output trap
024	Powerfail
030	EMT, emulator trap
034	TRAP instruction

TRAP INSTRUCTIONS
Trap instructions provide for calls to emulators, I/O monitors, debugging packages, and user-defined interpreters. A trap is effectively an interrupt generated by software. When a trap occurs, the contents of the current program counter (PC) and program status word (PS) are pushed onto the processor stack and replaced by the contents of a 2-word trap vector containing a new PC and new PS. The return sequence from a trap involves executing an RTI or RTT instruction which restores the old PC and old PS by popping them from the stack. Trap vectors are located at permanently assigned fixed addresses.

The EMT (trap emulator) and TRAP instructions do not use the low-order byte of the word in their machine language representation. This

allows user information to be transferred in the low-order byte. The new value of the PC loaded from the vector address of the TRAP or EMT instructions is typically the starting address of a routine to access and interpret this information. Such a routine is called a **trap handler.**

The trap handler must accomplish several tasks. It must save and restore all necessary GPRs, interpret the low byte of the trap instruction and call the indicated routine, serve as an interface between the calling program and this routine by handling any data that need be passed between them, and, finally, cause the return to the main routine.

Uses of Trap Handlers

The trap handler can be useful as a patching technique. Jumping out to a patch area is often difficult because a 2-word jump must be performed. However, the 1-word TRAP instruction may be used to dispatch to patch areas. A sufficient number of slots for patching should first be reserved in the dispatch table of the trap handler. The jump can then be accomplished by placing the address of the patch area into the table and inserting the proper TRAP instruction where the patch is to be made.

The trap handler can be used in a program to dispatch execution to any one of several routines. Macros may be defined to cause the proper expansion of a call to one of these routines. For example,

```
.MACRO SUB2 ARG
MOV ARG, R0
TRAP +1
.ENDM
```

When expanded, this macro sets up the one argument required by the routine in R0 and then causes the trap instruction with the number 1 in the lower byte. The trap handler should be written so that it recognizes a 1 as a call to SUB2. Notice that ARG here is being transmitted to SUB2 from the calling program. It may be data required by the routine or it may be a pointer to a longer list of arguments.

In an operating system environment like RT-11, the EMT instruction is used to call system or monitor routines from a user program. The monitor of an operating system necessarily contains coding for many functions, i.e., I/O, file manipulation, etc. This coding is made accessible to the program through a series of macro calls, which expand into EMT instructions with low bytes indicating the desired routine, or group of routines to which the desired routine belongs. Often a GPR is designated to be used to pass an identification code to further indicate to the trap handler which routine is desired. For example, the macro expansion for a resume execution command in RT-11 is as follows:

```
        .MACRO .RSUM
        CM3, 2.
        .ENDM
```

and CM3 is defined as

```
        .MACRO CM3 CHAN, CODE
        MOV #CODE *400,R0
.IIF NB CHAN,BISB CHAN,R0
        EMT 374
        .ENDM
```

Notice the EMT low byte is 374. This is interpreted by the EMT handler to indicate a group of routines. Then the contents of R0 (high byte) are tested by the handler to identify exactly which routine within the group is being requested, in this case routine number 2. (The CM3 call of the .RSUM is set up to pass the identification code.)

Summary of PDP-11 Processor Trap Vectors:

VECTOR ADDRESS	FUNCTION SERVED
4	Illegal instructions (JSR, JMP for mode 0) Bus errors (odd address error, timeout) Stack limit (Red Zone, Yellow Zone) Illegal internal address Microbreak
10	Reserved instruction XFC with UCS disabled SPL, MTPS, MFPS FADD, FSUB, FMUL, FDIV HALT in user mode
14	Trace (T bit)
20	IOT
24	Power fail
30	EMT
34	TRAP
114	Cache parity error UNIBUS memory parity error UCS parity error
244	Floating point exception
250	Memory management (KT) abort

CONVERSION ROUTINES

Almost all assembly language programs require the translation of data or results from one form to another. Coding that performs such a transformation will be called a conversion routine in this handbook. Several commonly used conversion routines are included in the following pages.

Almost all assembly language programs involve some type of conversion routines, octal to ASCII, octal to decimal, and decimal to ASCII being a few of the most widely used.

Arithmetic multiply and divide routines are fundamental to many conversion routines.

Division is typically approached in one of two ways.

1. The division can be accomplished through a combination of rotates and subtractions.

   Examples:
   Assume the following code and register data; to make the example easier, also assume a 3-bit word.

```
DIV:  MOV #3,-(SP)      ;SET UP DIGIT COUNTER
      CLR -(SP)         ;CLEAR RESULT
1$    ASL (SP)
      ASL R1
      ROL R0
      CMP R0,R3
      BLT 2$
      SUB R3,R0         ;R0 CONTAINS REMAINDER
      INC (SP)          ;INCREMENT RESULT
2$    DEC 2 (SP)        ;DECREMENT COUNTER
      BNE $1
```

   Therefore, to divide 7 by 2:

```
      R0=000            remainder
      R1=111            seven-multiplicand
      R3=010            two-multiplier
      C bit=0

      STACK
      011               counter
      000               quotient
```

   Following through the coding, the quotient, remainder, and dividend all shift left, manipulating the most significant digit first, etc.

At the conclusion of the routine:

R0=001	remainder
R1=000	
R3=010	
STACK	
000	counter
011	quotient

1. A second method of division occurs by repeated subtraction of the powers of the divisor, keeping a count of the number of subtractions at each level.

   Example:
   To divide 221_{10} by 10, first try to subtract powers of 10 until a non-negative value is obtained, counting the number of subtractions of each power.

```
  221
−1000
```

negative so go to next lower power, count for $10^3 = 0$.

```
  221
 −100

  121   count for 10² = 1.
 −100

   21   count = 2
 −100
```

negative, so reduce power count for $10^2 = 2$

```
   21
  −10

   11   count for 10¹ = 1.

   11
  −10

    1   count=2
  −10
```

negative, so count for $10^1 = 2$.

No lower power, so remainder is 1.

Answer = 022_{10}, remainder 1.

Multiplication can be done through a combination of rotates and additions or through repetitive additions.

Example:
Assume the following code and a 3-bit word.

```
        CLR R0          ;HIGH HALF OF ANSWER
        MOV #3,CNT      ;SET UP COUNTER
        MOV R1,MULT;    ;MULTIPLICAND

MORE:                   ROR R2
                        BCC NOW
                        ADD MULT,R0 ;IF INDICATED,
ADD
                        ;MULTIPLICAND
NOW:                    ROR R0
                        ROR R1
                        DEC CNT
                        BNE MORE
MULT:                   0
CNT:                    0
```

The following conditions exist for 6 times 3:

R0 = 000 — high order half of result
R1 = 110 — multiplicand
R3 = 011 — multiplier

After the routine is executed:

R0 = 010 — high order half of result
R1 = 010 — low order half of result
R2 = 100
CNT = 0
MULT = 110

Example:
Multiplication of R0 by 50_8 (101000).

```
MUL50:          MOV R0,-(SP)
                ASL R0
                ASL R0
                ADD (SP)+,R0
                ASL R0
```

```
        ASL R0
        ASL R0
        RETURN
```

If R0 contains 7:

R0 = 111

After execution;

R0 = 100011000
($7*50_8 = 430_8$)

ASCII CONVERSIONS

The conversion of ASCII characters to the internal representation of a number as well as the conversion of an internal number to ASCII in I/O operations presents a challenge. The following routine takes the 16-bit word in R1 and stores the corresponding six ASCII characters in the buffer addressed by R2.

```
OUT:    MOV     #5,R0           ;LOOP COUNT
LOOP:   MOV     R1,-(SP)        ;COPY WORD INTO STACK
        BIC     #177770,@SP     ;ONE OCTAL VALUE
        ADD     #'0,@SP         ;CONVERT TO ASCII
        MOVB    (SP)+,-(R2)     ;STORE IN BUFFER
        ASR     R1              ;SHIFT
        ASR     R1              ;RIGHT
        ASR     R1              ;THREE
        DEC     R0              ;TEST IF DONE
        BNE     LOOP            ;NO, DO IT AGAIN
        BIC     #177776,R1      ;GET LAST BIT
        ADD     #'0,R1          ;CONVERT TO ASCII
        MOVB    R5,-(R2)        ;STORE IN BUFFER
        RTS     PC              ;DONE,RETURN
```

PROGRAMMING EXAMPLES

The programming examples on the following pages show how the PDP-11 instruction set, the addressing modes, and the programming techniques can be used to solve some simple problems. The format used is either PAL-11 or MACRO-11.

Program Address	Program Contents	Label	Op Code	Operand	Comments
			;PROGRAMMING EXAMPLE		
			;SUBTRACT CONTENTS OF LOCS 700-710		
			;FROM CONTENTS OF LOCS 1000-1010		
	000000		R0=%0		
	000001		R1=%1		
	000002		R2=%2		
	000003		R3=%3		
	000004		R4=%4		
	000005		R5=%5		
	000006		SP=%6		
	000007		PC=%7		
	000500		.=500		
000500	012706	START:	MOV	#.,SP	;INIT STACK POINTER
	000500				
000504	012701		MOV	#700,R1	
	000700				
000510	012702		MOV	#712,R2	
	000712				
000514	012703		MOV	#1000,R3	
	001000				
000520	012704		MOV	#1012,R4	
	001012				

Program Address	Program Contents	Label	Op Code	Operand	Comments
000524	005000		CLR	R0	
000526	005005		CLR	R5	
000530	062105	SUM1:	ADD	(R1)+,R5	;START ADDING
000532	020102		CMP	R1,R2	;FINISHED ADDING?
000534	001375		BNE	SUM1	;IF NOT BRANCH BACK
000536	062300	SUM2:	ADD	(R3)+,R0	;START ADDING
000540	020304		CMP	R3,R4	;FINISHED ADDING?
000542	001375		BNE	SUM2	;IF NOT BRANCH BACK
000544	160500	DIFF:	SUB	R5,R0	;SUBTRACT RESULTS
000546	000000		HALT		;THAT'S ALL FOLKS
	000700		.=700		
000700	000001		.WORD 1, 2, 3, 4, 5		
000702	000002				
000704	000003				
000706	000004				
000710	000005				
	001000		.=1000		
001000	000004		.WORD 4, 5, 6, 7, 8		
001002	000005				

Program Address	Program Contents	Label	Op Code	Operand	Comments
001004	000006				
001006	000007				
001010	000010				
					A−30
	000500		END		

```
                ;PROGRAM TO COUNT NEGATIVE NUMBERS
                ;IN A TABLE
                ;20. SIGNED WORDS
                ;BEGINNING AT LOC VALUES
                ;COUNT HOW MANY ARE NEGATIVE IN R0

                R0=%0
                R1=%1
                R2=%2
                SP=%6
                PC=%7

                .=500

START:          MOV #.,SP               ;SET UP STACK
                MOV #VALUES,R1          ;SET UP POINTER
                MOV #VALUES+40.,R2      ;SET UP COUNTER
                CLR R0

CHECK:          TST (R1)+               ;TEST NUMBER
                BPL NEXT                ;POSITIVE?
                INC R0                  ;NO. INCREMENT
                                        ;COUNTER
NEXT:           CMP R1,R2               ;YES, FINISHED?
                BNE CHECK               ;NO, GO BACK
                HALT                    ;YES, STOP


VALUES:         .BLK 20.
                0
                .END
```

```
               ;PROGRAM TO COUNT ABOVE AVERAGE QUIZ SCORES
               ;LIST OF 16. QUIZ SCORES
               ;BEGINNING AT LOC SCORES
               ;KNOWN AVERAGE IN LOC AVRAGE
               ;COUNT IN R0 SCORES ABOVE AVERAGE

               R0=%0
               R1=%1
               R2=%2
               R3=%3
               SP=%6
               PC=%7
```

```
        .=500

START:  MOV #.,SP           ;SET UP STACK
        MOV #16.,R1         ;SET UP COUNTER
        MOV #SCORES, R2     ;SET UP POINTER
        MOV #AVRAGE,R3
        CLR R0

CHECK:  CMP (R2)+, (R3)     ;COMPARE SCORE AND AVRAGE
        BLE NO              ;LESS THAN OR EQUAL
                            ;TO AVRAGE?
        INC R0              ;NO, COUNT
NO:     DEC R1              ;YES, DECREMENT COUNTER
        BNE CHECK           ;FINISHED? NO, CHECK
        HALT                ;YES, STOP
AVERAGE: 65.

SCORES: 25.,70.,100.,60.,80.,80.,40.
        55.,75.,100.,65.,90.,70.,65.,70.

        .END


            ;PROGRAMMING EXAMPLE
            ;ACCEPT (IMMEDIATE ECHO) AND
            ;STORE 20. CHARS
            ;FROM THE KEYBOARD, OUTPUT CR & LF
            ;ECHO ENTIRE STRING FROM STORAGE

    R0=%0
    R1=%1
    SP=%6
    CR=15
    LF=12
    TKS=177560
    TKB=TKS+2
    TPS=TKB+2
    TPB=TPS+2

    .TITLE ECHO

    .=1000
```

```
START:  MOV     #.,SP           ;INITIALIZE STACK POINTER
        MOV     #SAVE+2,R0      ;SA OF BUFFER
                                ;BEYOND CR & LF
        MOV     #20.,R1         ;CHARACTER COUNT

IN:     TSTB    @#TKS           ;CHAR IN BUFFER?
        BPL     IN              ;IF NOT BRANCH BACK
                                ;AND WAIT
ECHO:   TSTB    @#TPS           ;CHECK TELEPRINTER
                                ;READY STATUS
        BPL     ECHO
        MOVB    @#TKB,@#TPB     ;ECHO CHARACTER
        MOVB    @#TKB,(R0)+     ;STORE CHARACTER AWAY
        DEC     R1
        BNE     IN              ;FINISHED INPUTTING?

        MOV     #SAVE,R0        ;SA OF BUFFER INCLUDING
                                ;CR & LF
        MOV     #22.,R1         ;COUNTER OF BUFFER
                                ;INCLUDING CR & LF

OUT:    TSTB    @#TPS           ;CHECK TELEPRINTER
                                ;READY STATUS
        BPL     OUT
        MOVB    (R0)+,@#TPB     ;OUTPUT CHARACTER
        DEC     R1
        BNE     OUT             ;FINISHED OUTPUTTING?
        HALT

SAVE:   .BYTE   CR,LF
        .=.+20.

        .END
```

```
        ;PROGRAMMING EXAMPLE
        ;SUBROUTINE TO INPUT TEN VALUES
INPUT:  MOV #BUFFER,R0      ;SET UP SA OF
                            ;STORAGE BUFFER
        MOV #-10.,R1        ;SET UP COUNTER
IN:     TSTB @#TKS          ;TEST KYBD READY STATUS
        BPL IN
OUT:    TSTB @#TPS          ;TEST TTO READY STATUS
```

```
        BPL OUT
        MOVB @#TKB,@#TPB;ECHO CHARACTER
        MOVB @#TKB,(R0)+  ;STORE CHARACTER
        INC R1            ;INC COUNTER
        BNE IN
        RTS PC            ;EXIT


        ;PROGRAMMING EXAMPLE
        ;SUBROUTINE TO SORT TEN VALUES
SORT:   MOV #-10.,R4
NEXT:   MOV COUNT,R3
        MOV #BUFFER+9.,R0
        ADD R3,R0
        MOVB (R0)+,R1
LOOP:   CMPB (R0)+,R1
        BGE GT
LT:     MOVB -(R0),R2
        MOVB R1,(R0)+
        MOV R2,R1
GT:     INC R3
        BNE LOOP
INSERT: MOVB R1,BUFFER+10.(R4)
        INC R4
        INC COUNT
        BNE NEXT
        MOV #-9.,COUNT    ;RESTORE LOCATION COUNT
        RTS PC            ;EXIT

COUNT:  .WORD -9.
LINE1:  .ASCII/INPUT ANY TEN SINGLE DIGIT VALUES (0-9);
I'LL/
        .ASCII/SORT AND OUTPUT THEM IN/
LINE2:  .ASCII/SMALLEST TO LARGEST ORDER./
BUFFER: .=.+10.
        .END INITSP       ;FINISHED!!!


        ;PROGRAMMING EXAMPLE
        ;SUBROUTINE EXAMPLE
        ;INPUT TEN VALUES, SORT, AND
        ;OUTPUT THEM IN SMALLEST TO LARGEST ORDER
```

```
          R0=%0
          R1=%1
          R2=%2
          R3=%3
          R4=%4
          R5=%5
          SP=%6
          PC=%7
          TKS=177560 (address of teletype control status register)
          TKB=TKS+2 — (teletype data buffer register)
          TPS=TKB+2 (teletype output control and status registers)
          TPB=TPS+2 — (teletype output data buffer)

          .=3000

INITSP:   MOV #.,SP          ;INITIALIZE STACK POINTER
          JSR PC,CRLF        ;GO TO CRLF SUBROUTINE
          JSR R5, OUTPUT     ;GO TO OUTPUT SUBROUTINE
          LINE1              ;SA OF LINE 1 BUFFER
          69.                ;NUMBER OF OUTPUTS
          JSR PC,CRLF        ;GO TO CRLF SUBROUTINE
          JSR R5,OUTPUT      ;GO TO OUTPUT SUBROUTINE
          LINE2              ;SA OF LINE 2 BUFFER
          26.                ;NUMBER OF OUTPUTS
          JSR PC,CRLF        ;GO TO CRLF SUBROUTINE
          JSR PC,INPUT       ;GO TO INPUT SUBROUTINE
          JSR PC,SORT        ;GO TO SORT SUBROUTINE
          JSR PC,CRLF        ;GO TO CRLF SUBROUTINE
          JSR R5,OUTPUT      ;GO TO OUTPUT SUBROUTINE
          BUFFER             ;INPUT BUFFER AREA
          10.                ;NUMBER OF OUTPUTS
          JSR PC,CRLF
          HALT               ;THE END!!!


          ;PROGRAMMING EXAMPLE
          ;SUBROUTINE TO OUTPUT A CR & LF
CRLF:     TSTB @#TPS         ;TEST TTO READY STATUS
          BPL CRLF
          MOVB #15,@#TPB     ;OUTPUT CARRIAGE RETURN
LNFD:     TSTB @#TPS         ;TEST TTO READY STATUS
          BPL LNFD
          MOVB #12,@#TPB     ;OUTPUT LINE FEED
          RTS PC             ;EXIT
```

```
        ;SUBROUTINE TO OUTPUT A
        ;VARIABLE LENGTH MESSAGE
OUTPUT: MOV (R5)+,R0        ;PICK UP SA OF DATA BLOCK
        MOV (R5)+,R1        ;PICK UP NUMBER OF OUTPUTS
        NEG R1              ;NEGATE IT
AGAIN:  TSTB @#TPS          ;TEST TTO READY STATUS
        BPL AGAIN
        MOVB (R0)+,@#TPB    ;OUTPUT CHARACTER
        INC R1              ;BUMP COUNTER
        BNE AGAIN
        RTS R5
```

LOOPING TECHNIQUES

PROGRAM SEGMENTS BELOW USED TO CLEAR
A 50.WORD TABLE

1. AUTOINCREMENT (POINTER ADDRESS IN GPR)

```
        R0=%0
        MOV #TBL,R0
LOOP:   CLR (R0)+
        CMP R0,#TBL+100.
        BNE LOOP
```

2. AUTODECREMENT (POINTER AND LIMIT VALUES IN GPR)

```
        R0=%0
        R1=%1
        MOV #TBL,R0
        MOV #TBL+100.,R1
LOOP:   CLR – (R1)
        CMP R1,R0
        BNE LOOP
```

3. COUNTER (DECREMENTING A GPR CONTAINING COUNT)

```
        R0=%0
        R1=%1
        MOV #TBL,R0
        MOV #50.,R1
LOOP:   CLR (R0)+
        DEC R1
        BNE LOOP
```

4. INDEX REGISTER MODIFICATION (INDEXED MODE; MODIFYING INDEX VALUE)

```
        R0=%0
        CLR R0
LOOP:   CLR TBL (R0)
        ADD #2,R0
        CMP R0,#100.
        BNE LOOP
```

5. FASTER INDEX REGISTER MODIFICATION (STORING VALUES IN GPR)

```
        R0=%0
        R1=%1
        R2=%2
        MOV #2,R1
        MOV #100.,R2
        CLR R0
LOOP:   CLR TBL (R0)
        ADD R1,R0
        CMP R0,R2
        BNE LOOP
```

6. ADDRESS MODIFICATION (INDEXED MODE; MODIFYING BASE ADDRESS)

```
        R0=%0
        MOV #TBL,R0
LOOP:   CLR 0 (R0)
        ADD #2,LOOP+2
        CMP LOOP+2,#100.
        BNE LOOP
```

CHAPTER 6
MEMORY MANAGEMENT

PDP-11 programs address memory using a 16-bit **virtual address**. On the PDP-11/04 (always) and other PDP-11 processors (with memory management disabled), the 16-bit virtual address provides direct access to 56K bytes of main memory and 8K bytes of peripheral and processor registers. Addresses in the lower 56K bytes are presented directly to the UNIBUS by the CPU. The high 8K bytes are mapped by the CPU to the UNIBUS I/O page address 760 000 to 777 777 (see Figure 6-1).

PDP-11 processors with memory management can address up to 64K bytes of main memory by mapping all of the virtual address space to physical memory.

Memory mapping is available in three forms on PDP-11s:

16-bit mapping:	PDP-11/04, 11/34A, 11/60, 11/44, 11/70
18-bit mapping:	PDP-11/34A, 11/60, 11/44, 11/70
22-bit mapping:	PDP-11/44, 11/70

CPU MAPPING ON THE 11/34A AND 11/60

Mapping of processor addresses is performed in one of two possible ways: 16-bit mapping (with Memory Management disabled), or 18-bit mapping (with Memory Management enabled).

16-bit Mapping (Memory Management Disabled)

There is a fixed relocation mapping from virtual to physical addresses. The lowest 56K virtual addresses correspond to the same physical addresses. The top 8K addresses cause UNIBUS cycles to addresses 760 000 to 777 777. (Refer to Figure 6-1.) 16-bit mapping occurs after Power-Up, Console Start, or the RESET instruction.

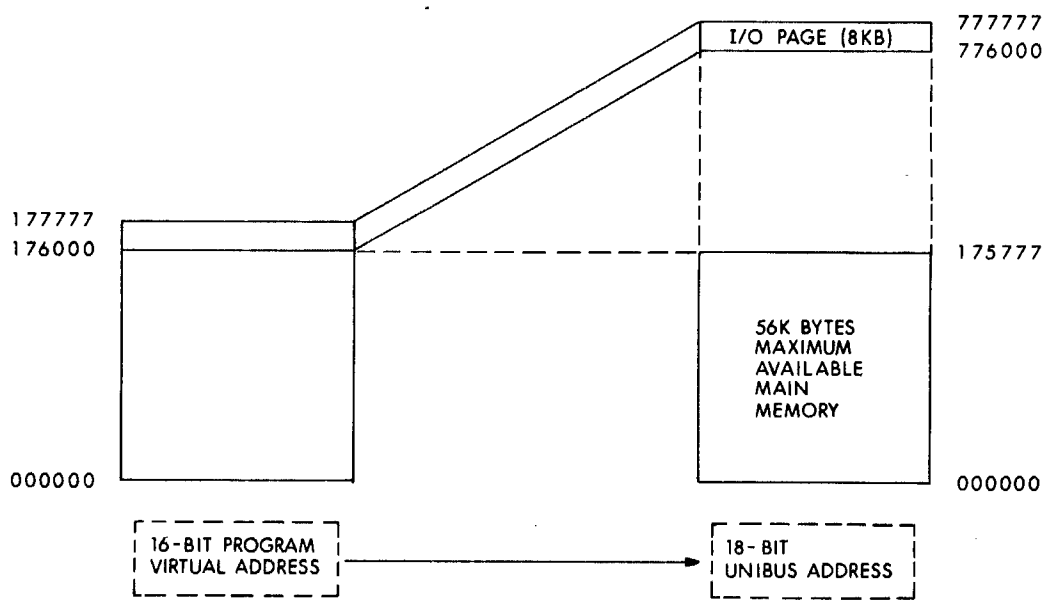


Figure 6-1 16-Bit Mapping on the 11/04, 11/34A, 11/60

18-Bit Mapping (Memory Management Enabled)

64K bytes of virtual address space for each of the two modes (Kernel and User) are mapped into 256K bytes of physical address space. The lowest 248K byte addresses reference physical memory. The top 8K byte addresses access peripheral page registers. (Refer to Figure 6-2).

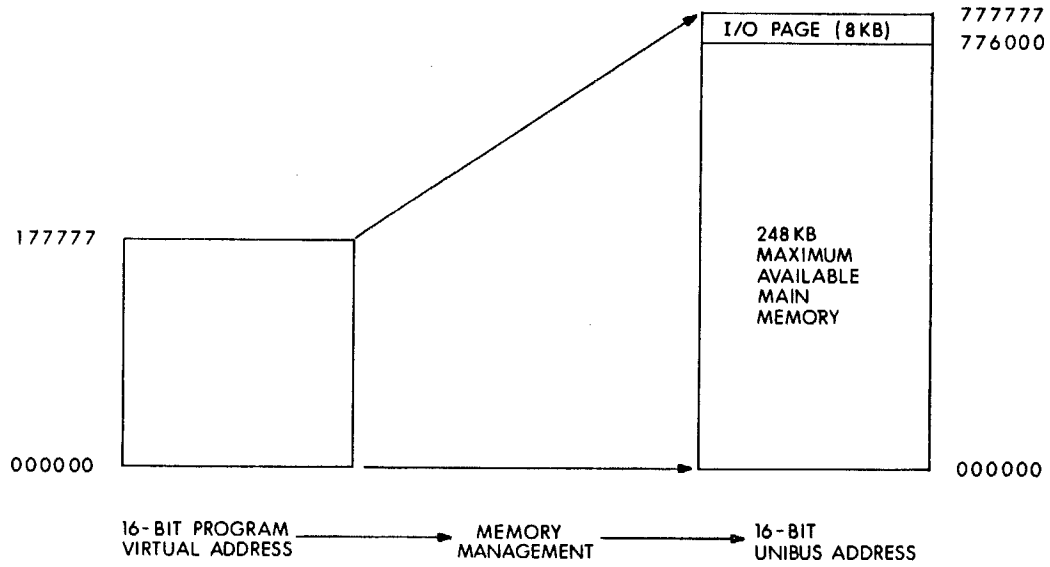


Figure 6-2 Memory Management on the 11/34A, 11/60

CPU MAPPING ON 11/44 AND 11/70

Mapping of processor addresses is performed in one of three possible ways: 16-bit mapping, 18-bit mapping and 22-bit mapping.

16-Bit Mapping

There is fixed relocation mapping from virtual to physical addresses. The lowest 56K virtual addresses are treated as corresponding to the same physical addresses. The top 8K addresses cause UNIBUS cycles to addresses 17 760 000 to 17 777 777. (Refer to Figure 6-3.) 16-bit mapping operation occurs after Power-Up, Console Start, or the RESET instruction.

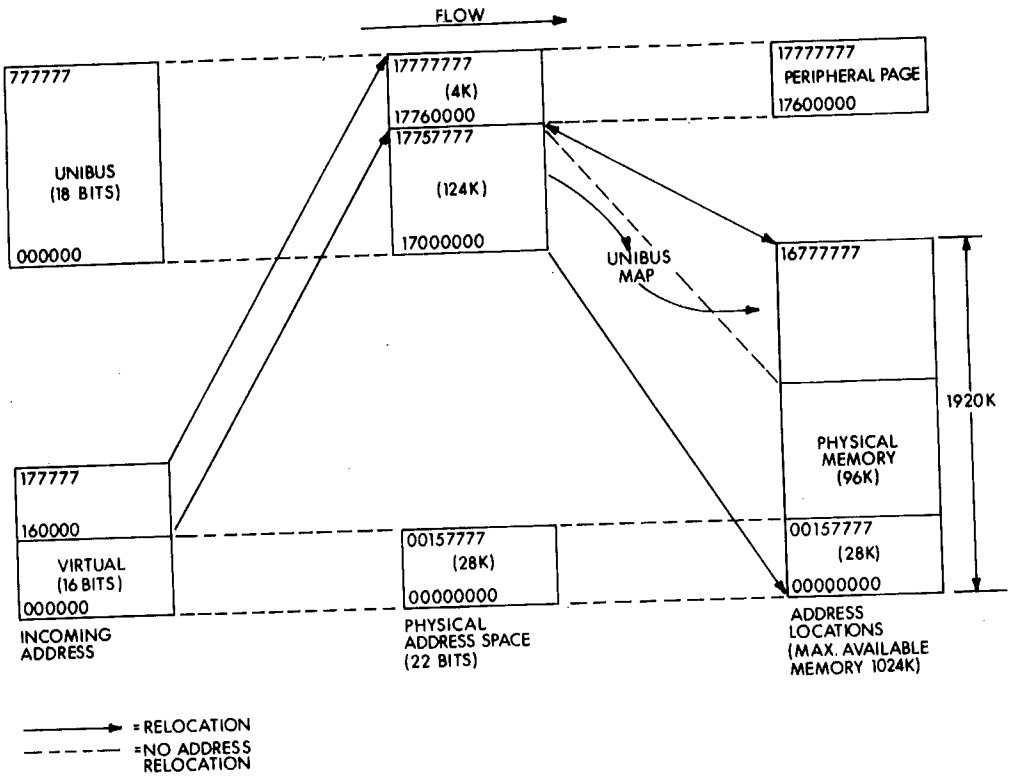


Figure 6-3 16-Bit Mapping on 11/44, 11/70

18-Bit Mapping

64K virtual addresses for each of the three modes (Kernel, Supervisor, and User) are mapped into 256K of physical address space. The lowest 248K addresses reference physical memory. The top 8K addresses cause UNIBUS cycles to addresses 17 760 000 to 17 777 777. (Refer to Figure 6-4.)

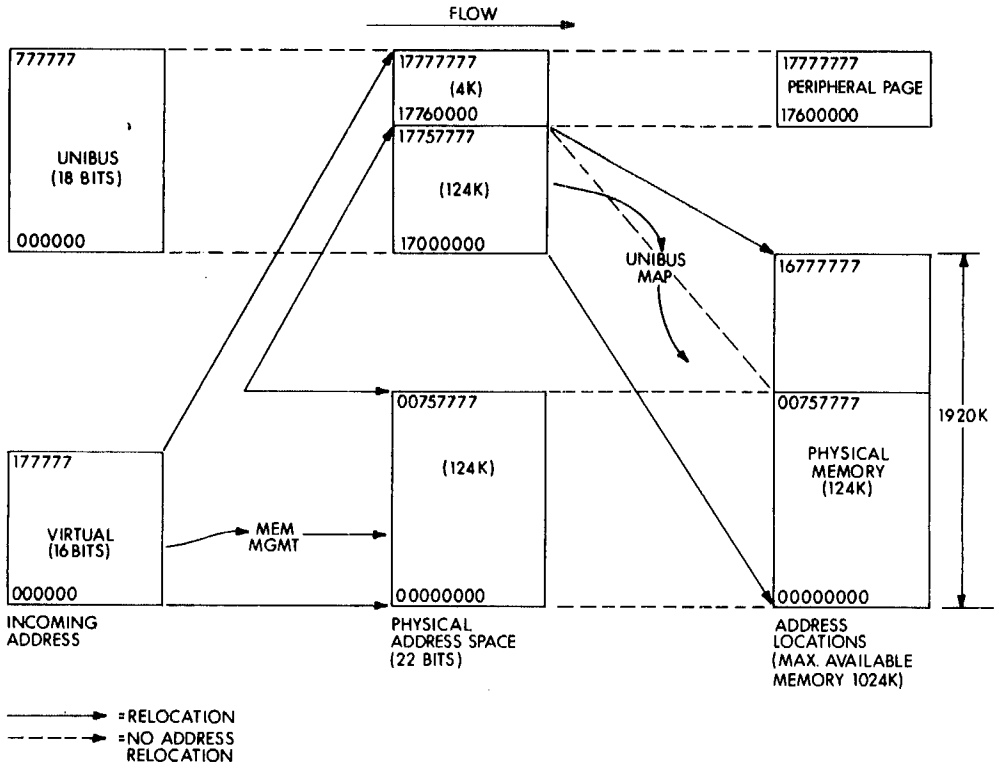


Figure 6-4 18-Bit Mapping on 11/44, 11/70

22-Bit Mapping

This mode produces full 22-bit addresses for accessing all of physical memory. The top 256K addresses cause UNIBUS cycles to addresses 17 000 000 to 17 777 777. (Refer to Figure 6-5.)

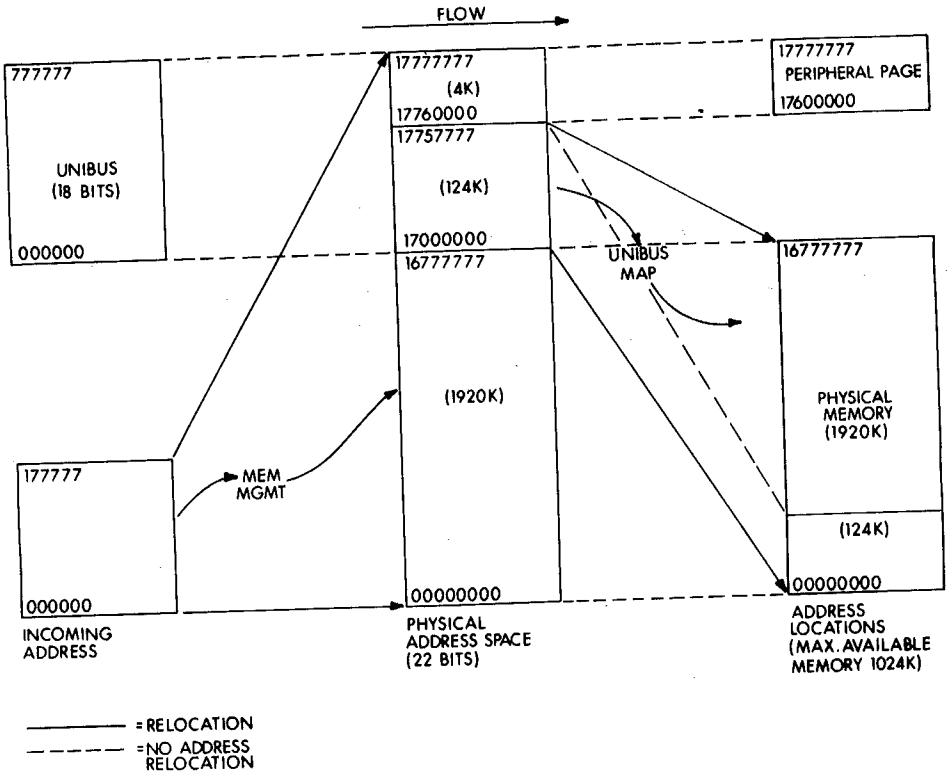


Figure 6-5 22-Bit Mapping on 11/44, 11/70

COMPATIBILITY

16-bit and 18-bit mapping can used so that the computer is compatible with other PDP-11 computers, such as the PDP 11/20 or the PDP-11/45. Thus, software written for another PDP-11 can be run on the 11/44 or PDP-11/70 without modification.

UNIBUS

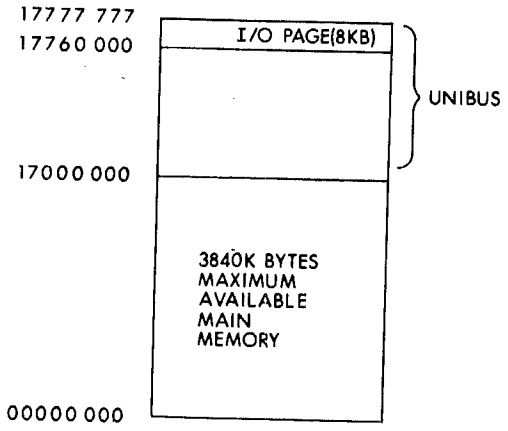
Mapping	Mem Mgt	Map Relocation	Compatible With
16 Bit	Off	Off	PDP-11/05,11/40 11/15, 11/20, 11/04
18 Bit	On	Off	PDP-11/35, 11/40 11/45, 11/50, 11/34A, 11/60
22 Bit	On	Off or On	PDP-11/70, 11/44

Although all machines are started in 16-bit mapping mode at boot time, DIGITAL operating systems run 11/34As and 11/60s with 18-bit mapping and 11/44s and 11/70s with 22-bit mapping.

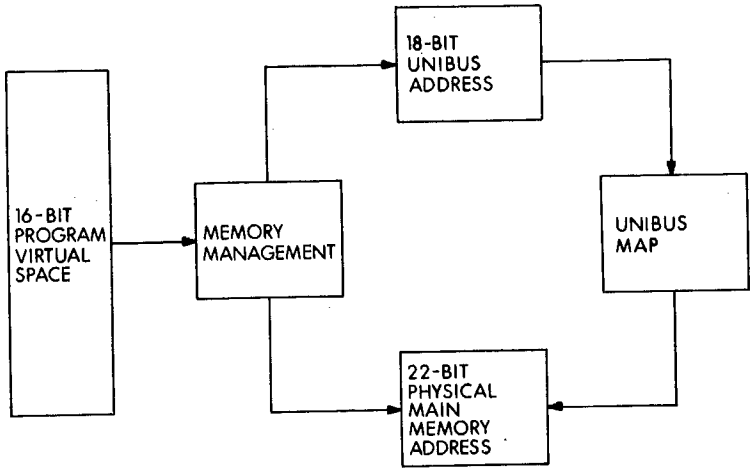
On the PDP-11/34A, 11/44, 11/60 and 11/70, memory management can be used to map the 64K bytes of program virtual address space into a larger physical address space. This mapping allows the program to access 64K bytes of memory at any given time. The mapping can be changed to address a different set of 64K bytes. This mapping mechanism allows several programs to occupy different portions of physical memory without risk of unintended accesses by other programs, as is necessary in multiprogramming systems. The operating system can also allow several programs to share physical memory locations without requiring them to use the same virtual address.

On the PDP-11/34A and 11/60, memory management hardware maps 16-bit program virtual addresses to 18-bit UNIBUS addresses providing access to 256K bytes of which the high order 8K bytes are reserved for peripheral and processor registers (I/O page). This leaves 248K bytes of addressable main memory on the UNIBUS (see Figure 6-2).

On the PDP-11/44 and 11/70, memory management hardware converts a 16-bit program virtual address to a 22-bit physical address, which provides access to over 4 million bytes (the 11/44 is currently limited to 1M bytes of physical memory). The high order 256K byte addresses are used to access the UNIBUS. The remaining 3,840K byte addresses are available for access to physical main memory. Thus, physical addresses between 00 000 000 and 16 777 777 access main memory



If an address is in the top 256K bytes of the 22-bit physical address space, i.e., 17 000 000 to 17 777 777, the lower 18 bits of the address are placed on the UNIBUS. The UNIBUS map can convert the 18-bit UNIBUS address back to a 22-bit physical main memory address.



ADDRESSING AND MEMORY MANAGEMENT

PDP-11 memory management allows addressing of physical memories larger than 64K bytes, and provides other enhancements of the PDP-11 memory addressing capability.

The 16-bit addresses which appear in the PDP-11 programs refer to physical memory on processors, such as the PDP-11/04, which do not have memory management. However, on processors which support larger physical memories, these 16-bit addresses are transformed by memory mangement into physical address which are 18 bits long on the PDP-11/34A and 11/60, and 22 bits long on the 11/44 and 11/70.

The 16-bit addresses which are part of the PDP-11 program are called *Virtual Addresses* because the program operates as if it were addressing a 64K byte memory even though it is actually addressing a part of a much larger physical memory. Because the PDP-11 program operates without explicitly invoking memory management, the relocation of programs by memory is said to be *transparent* to the program.

The mechanism described below by which memory management maps virtual addresses to physical addresses provides other functions such as:

- extension of the physical memory address
- separation of system and user functions

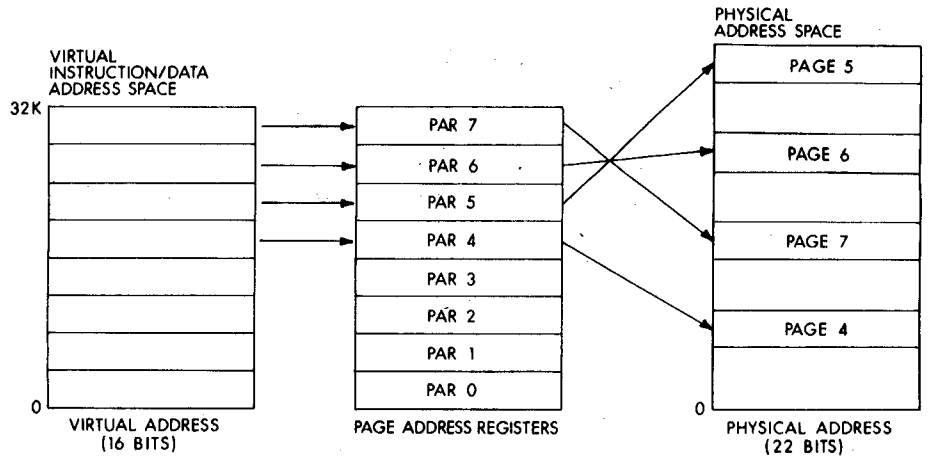


Figure 6-6 Virtual Address Mapping into Physical Address

The starting physical address for each page is an integral multiple of 32 words, and each page has a maximum size of 4,096 words. Pages may be located anywhere within the Physical Address space. The determination of which set of page registers is used to form a Physical Address is made by the current mode of operation of the CPU, i.e., Kernel, Supervisor, or User mode.

NOTE

All references to Supervisor Mode or Data Space pertain to the PDP-11/44 and 11/70 only. When Data Space is disabled and not in Supervisor Mode, the memory management facility on the PDP-11/44 and PDP-11/70 operates in the same way as PDP-11/34A and PDP-11/60 memory management.

Interrupt Conditions Under Memory Management Control

The Memory Management Unit relocates all addresses. Thus, when it is enabled, all trap, abort, and interrupt vectors are considered to be in Kernel mode virtual address space. When a vectored transfer occurs, control is transferred according to a new Program Counter (PC) and Processor Status Word (PS) contained in a two-word vector relocated through the Kernel Page Address Register set. Relocation of trap addresses means that the hardware is capable of recovering from a failure in the first physical bank of memory.

When a trap, abort, or interrupt occurs, the "push" of the old PC and old PS is to the User/Supervisor/Kernel R6 stack specified by CPU

mode bits 15, 14 of the new PS in the vector. (00 = Kernel, 01 = Supervisor, 11 = User.) The CPU mode bits also determine the new Page Address Register set. Thus, it is possible for a Kernel mode program to have complete control over service assignments for all interrupt conditions, since the interrupt vector is located in Kernel space. The Kernel program may assign the service of some of these conditions to a Supervisor or User mode program by simply setting the CPU mode bits of the new PS in the vector to return control to the appropriate mode.

Construction of a Physical Address

With memory relocation enabled, all addresses either reference information in Instruction (I) space or Data (D) space. I space is used for all instruction fetches, index words, absolute addresses and immediate operands. D Space is used for all other references. I Space and D Space each have eight PAR's in each mode of CPU operation (Kernel, Supervisor, and User). Using Memory Management Register #3, the operating system may select to disable D space and map all references (Instructions and Data) through I space, or to use both I and D space.

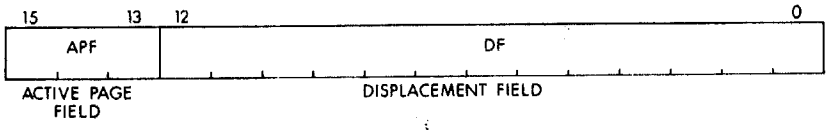


Figure 6-7 Interpretation of Virtual Address

The Virtual Address consists of:

1. The Active Page Field (APF). This 3-bit field determines which of eight Page Address Registers (PAR0-PAR7) will be used to form the Physical Address.
2. The Displacement Field (DF). This 13-bit field contains an address relative to the beginning of a page. This permits page lengths up to 4K words (8K bytes). The DF is further subdivided into two fields as shown in Figure 6-8.

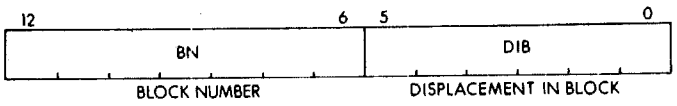


Figure 6-8 Displacement Field of Virtual Address

The Displacement Field consists of:

1. The Block Number (BN). This 7-bit field is interpreted as the block number within the current page.
2. The Displacement in Block (DIB). This 6-bit field contains the displacement within the block referred to by the Block Number.

The remainder of the information needed to construct the Physical Address comes from the Page Address Field (PAF) which is 16 bits on the 11/44 and 11/70, and 12 bits on the 11/34A and 11/60. The Page Address Register (PAR) specifies the starting address of the memory page which that PAR describes. The PAF is actually a block number in the physical memory, e.g., PAF = 3 indicates a starting address of 96 (3 × 32) words in physical memory.

The formation of the Physical Address (PA) is illustrated in Figure 6-9.

The logical sequence involved in constructing a Physical Address is:

1. Select a set of Page Address Registers depending on the space being referenced.
2. The Active Page Field of the Virtual Address is used to select a Page Address Register (PAR0-PAR7).
3. The Page Address Field of the selected Page Address Register contains the starting address of the currently active page as a block number in physical memory.
4. The Block Number from the Virtual Address is added to the Page Address Field to yield the number of the block in physical memory (PBN-Physical Block Number) which will contain the Physical Address being constructed.
5. The Displacement in Block from the Displacement Field of the Virtual Address is joined to the Physical Block Number to yield a Physical Address.

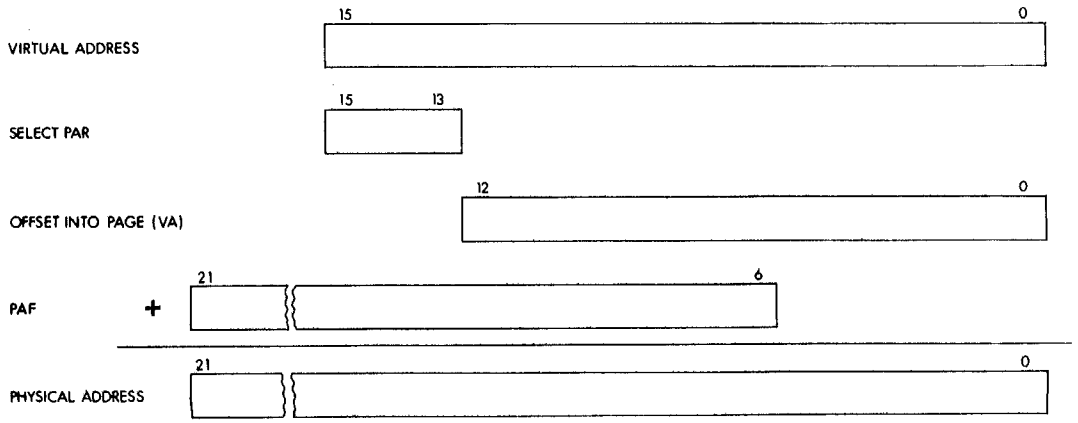


Figure 6-9 Construction of a Physical Address

Memory Management Registers on the PDP-11/44 and PDP-11/70

The Memory Management Units on the PDP-11/44 and 11/70 implement three sets of 32 16-bit registers. One set of registers is used in Kernel mode, another in Supervisor, and the third in User mode. The choice of which set is to be used is determined by the current CPU mode contained in the Processor Status word. Each set is subdivided into two groups of 16 registers. One group is used for references to Instruction (I) Space, and one to Data (D) Space. The I Space group is used for all instruction fetches, index words, absolute addresses and immediate operands. The D Space group is used for all other references, providing it has not been disabled by Memory Management Register #3. If D space is disabled, then I space is used for all references. Each group is further subdivided into two parts of eight registers. One part is the Page Address Register (PAR) whose function has been described in previous paragraphs. The other part is the Page Descriptor Register (PDR). PARs and PDRs are always selected in pairs by the top three bits of the virtual address. A PAR/PDR pair contains all the information needed to describe and locate a currently active memory page.

The Memory Management registers are located in the uppermost 8K bytes of PDP-11 physical address space along with the UNIBUS I/O device registers.

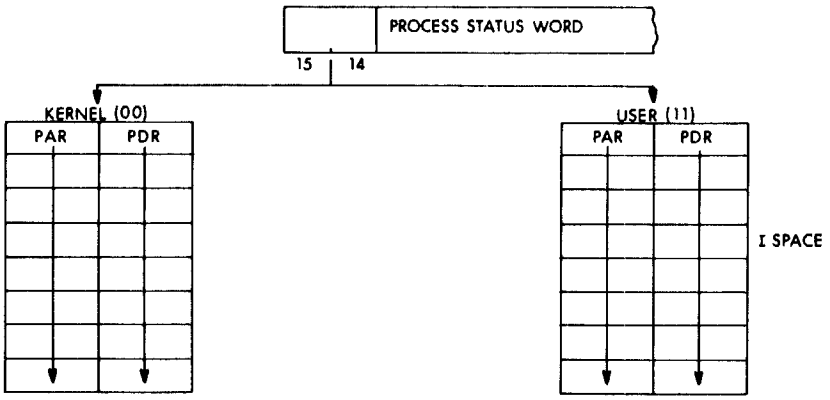


Figure 6-10 Active Page Registers on the PDP-11/44 and 11/70

Memory Management Registers on the PDP-11/34A and PDP-11/60

The Memory Management Units on the PDP-11/34A and PDP-11/60 implement a subset of the capability described above for the 11/44 and 11/70. The Supervisor mode and Data space have been omitted and addressing is limited to 18-bit physical addresses. Hence, on the 11/34A and 11/60, the Memory Management Units implement two sets of 16 registers.

Page Address Register (PAR)
The Page Address Register (PAR) contains the Page Address Field (PAF), a 16-bit field on the 11/44 and 11/70, and a 12-bit field on the 11/34A and 11/60, which specifies the starting address of the page as a block number in physical memory.

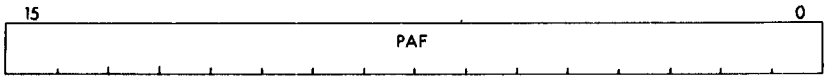


Figure 6-11 Page Address Register

The Page Address Register, which contains the Page Address Field, may be alternatively thought of as a relocation register containing a relocation constant, or as a base register containing a base address. Either interpretation indicates the basic importance of the Page Address Register as a relocation tool.

Page Descriptor Register
The Page Descriptor Register contains information relative to page expansion, page length, and access control.

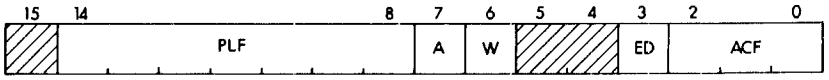


Figure 6-12 Page Descriptor Register

Access Control Field (ACF)
This 3-bit field, occupying bits 2-0 of the Page Descriptor Register contains the access rights to this particular page. The access codes, or "keys," specify the manner in which a page may be accessed and whether or not a given access should result in a trap or an abort of the current operation. A memory reference which causes an abort is not completed while a reference causing a trap is completed. In fact, when a memory reference causes a trap to occur, the trap does not occur until the entire instruction has been completed. Aborts are used to catch "missing page faults," prevent illegal access, etc.; traps are used as an aid in gathering memory management information.

In the context of access control the term "write" is used to indicate the action of any instruction which modifies the contents of any addressable word. "Write" is synonymous with what is usually called a "store" or "modify" in many computer systems.

The modes of access control are as follows:

000	non-resident	abort all accesses
001	read-only*	abort on write attempt, memory management trap on read
010	read-only	abort on write attempt
011	unused	abort all accesses—reserved for future use
100	read/write	memory management trap upon completion of a read or write
101	read/write*	memory management trap upon completion of a write
110	read/write	no system trap/abort action
111	unused	abort all accesses—reserved for future use

*Read-only and read/write traps are implemented on the PDP-11/70 only.

It should be noted that using I Space provides the user with a further form of protection on the PDP-11/44 and PDP-11/70, execute only.

Access Information Bits

A Bit (bit 7)—This bit, implemented on the PDP-11/70 only, is used by software to determine whether or not any accesses to this page met the trap condition specified by the Access Control Field (ACF) (A = 1 is Affirmative). The A Bit is used in the process of gathering memory management statistics.

W Bit (bit 6)—This bit indicates whether or not this page has been modified (written into) since either the PAR or PDR was loaded. (W = 1 is Affirmative). The W Bit is useful in applications which involve disk swapping and memory overlays. It is used to determine which pages have been modified and hence must be saved in their new form and which pages have not been modified and can be simply overlaid.

Note that A and W bits are reset to "0" whenever either PAR or PDR is modified (written into).

Expansion Direction (ED)
Bit 3 of the Page Descriptor Register specifies in which direction the page expands. If ED = 0 the page expands upwards from block number 0 to include blocks with high addresses; if ED = 1, the page expands downward from block number 127 to include blocks with lower addresses. Upward expansion is usually used for program space while downward expansion is used for stack space.

Page Length Field (PLF)
This 7-bit field, occupying bits 14-8 of the Page Descriptor Register, specifies the block number, which defines the boundary of that page. The block number of the Virtual Address is compared against the Page Length Field to detect Length Errors. An error occurs when expanding upwards if the block number is greater than the Page Length Field, and when expanding downwards if the block number is less than the Page Length Field.

Bypass Cache bit (BC) (11/44 only)
When bit 15 of Page Descriptor Register is set, and relocation is enabled, all CPU references to this page go directly to main memory.

Reserved Bits
Bits 5 and 4 are spare and always read as 0, and should never be written. They are unused and reserved for possible future expansion.

Fault Recovery Registers
Aborts and traps generated by the Memory Management hardware are vectored through Kernel virtual location 250. Memory Management registers #0, #1, and #3 are used to differentiate an abort from a trap, determine why the abort or trap occurred, and allow for easy program restarting. Note that an abort or trap to a location which is itself an invalid address will cause another abort or trap. Thus the Kernel program must insure that Kernel Virtual Address 250 is mapped into a valid address, otherwise a loop will occur which will require console intervention.

Memory Management Register #0 (MMR0)
MMR0 contains error flags, the page number whose reference caused the abort, and various other status flags. The register is organized as shown in Figure 6-13.

Setting bit 0 of this register enables address relocation and error detection. This means that the bits in MMR0 become meaningful.

Bits 15-12 are the error flags. They may be considered to be in a priority queue in that flags to the right are less significant and should

be ignored. That is, a non-resident fault-service routine would ignore length, access control, and memory management flags. A page length service routine would ignore access control and memory management faults, etc.

Bits 15-13, when set (error conditions), cause Memory Management to freeze the contents of bits 1-7 and Memory Management Registers #1 and #2. This has been done to facilitate error recovery.

These bits may also be written under program control. No abort will occur, but the contents of the Memory Management registers will be locked up as in an abort.

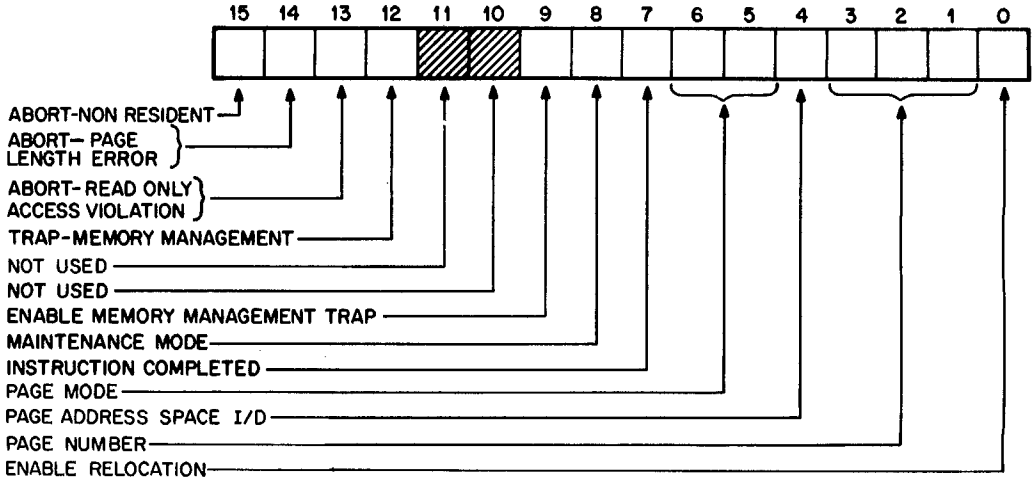


Figure 6-13 Format of Memory Management Register #0 (MMR0)

Abort—Non-Resident Bit 15 — Bit 15 is the Abort Non-Resident bit. It is set by attempting to access a page with an Access Control Field (ACF) key equal to 0, 3, or 7. It is also set by attempting to use Memory Relocation with a processor mode of 2 (undefined/invalid mode).

Abort—Page Length, Bit 14 — Bit 14 is the Abort Page Length bit. It is set by attempting to access a location in a page with a block number (Virtual Address bits 12-6) that is outside the area authorized by the Page Length Field of the Page Descriptor Register for that page. Bits 14 and 15 may be set simultaneously by the same access attempt. Bit 14 is also set by attempting to use Memory Relocation with a processor mode of 2.

Abort—Read Only, Bit 13 — Bit 13 is the Abort Read Only bit. It is set by attempting to write in a read-only page. Read-only pages have access keys of 1 or 2.

Trap—Memory Management, Bit 12 (PDP-11/70 only) — Bit 12 is the Trap Memory Management bit. It is set whenever a Memory Management trap condition occurs; that is, a read operation which references a page with an Access Control Field of 1 or 4, or a write operation to a page with an ACF key of 4 or 5.

Bits 11-10 — Bits 11 and 10 are spare and are always read as 0, and should never be written. They are unused and reserved for future use.

Enable Memory Management Traps, Bit 9 (PDP-11/70 only) — Bit 9 is the Enable Memory Management Traps bit. It is set or cleared by doing a direct write into MMR0. If bit 9 is 0, no Memory Management traps will occur. The A and W bits will, however, continue to log Memory Management Trap conditions. When bit 9 is set to 1, the next Memory Management trap condition will cause a trap, vectored through Kernel Virtual Address 250.

NOTE

If an instruction which sets bit 9 to 0 (disable Memory Management Trap) causes a Memory Management trap condition in any of its memory references prior to and including the one actually changing MMR0, the trap will occur at the end of the instruction.

Maintenance/Destination Mode, Bit 8 — Bit 8 specifies that only destination mode references will be relocated using Memory Management. This mode is only used for maintenance purposes.

Instruction Completed, Bit 7 (PDP-11/70 only) — Bit 7 indicates that the current instruction has been completed. It will be set to 0 during T bit, Parity, Odd Address, and Time Out traps and interrupts. This provides error handling routines with a way of determining whether the last instruction will have to be repeated in the course of an error recovery attempt. Bit 7 is read-only (it cannot be written). It is initialized to a 1. Note that EMT, TRAP, BPT, and IOT do not set bit 7.

Processor Mode, Bits 6-5 — Bits 6 and 5 indicate the CPU mode associated with the page causing the abort (Kernel = 00, Supervisor = 01, User = 11, illegal mode = 10). If an illegal mode is specified, bits 15 and 14 will be set.

Page Address Space, Bit 4 (PDP-11/44 and 11/70) — Bit 4 indicates the type of address space (I or D) the Unit was in when a fault occurred (0 = I Space, 1 = D Space). It is used in conjunction with bits 3-1, Page Number.

Page Number, Bits 3-1 — Bits 3-1 contain the page number of a reference causing a Memory Management fault. Note that pages, like blocks, are numbered from 0 upwards.

Enable Relocation, Bit 0 — Bit 0 is the Enable Relocation bit. When it is set to 1, all addresses are relocated by the unit. When bit 0 is set to 0 the Memory Management Unit is inoperative and addresses are not relocated or protected.

Memory Management Register #1 (MMR1) (PDP-11/44 and 11/70)
MMR1 records any autoincrement/decrement of the general purpose registers, including explicit references through the PC. MMR1 is cleared at the beginning of each instruction fetch. Whenever a general purpose register is either autoincremented or autodecremented, the register number and the amount by which the register was modified (in 2's complement notation) is written into MMR1.

The information contained in MMR1 is necessary to accomplish an effective recovery from an error resulting in an abort. The low order byte is written first and it is not possible for a PDP-11 instruction to autoincrement/decrement more than two general purpose registers per instruction before an abort-causing reference. Register numbers are recorded MOD 8; thus it is up to the software to determine which set of registers (User/Supervisor/Kernel—General Set 0/General Set 1) was modified, by determining the CPU and Register modes as contained in the PS at the time of the abort. The 6-bit displacement of R6 (SP) that can be caused by the MARK instruction cannot occur if the instruction is aborted.

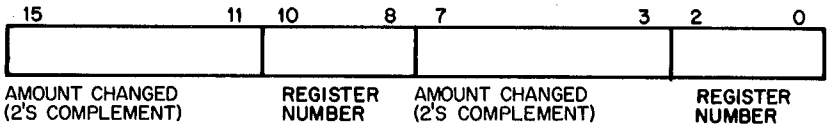


Figure 6-14 Format of Memory Management Register #1 (MMR1)

Memory Management Register #2 (MMR2)
MMR2 is loaded with the 16-bit Virtual Address (VA) at the beginning of each instruction fetch, or with the address Trap Vector at the beginning of an interrupt, T Bit trap, Parity, Odd Address, and Timeout aborts and parity traps. Note that MMR2 does not get the Trap Vector on EMT, TRAP, BPT and IOT instructions. MMR2 is read-only; it cannot be written. MMR2 is the Virtual Address Program Counter.

Memory Management Register #3 (MMR3) (PDP-11/44 and 11/70)
The Memory Management Register #3 (MMR3) enables or disables the use of the D space PARs and PDRs, 22-bit mapping and UNIBUS

mapping. When D space is disabled, all references use the I space registers; when D space is enabled, both the I space and D space registers are used. Bit 0 refers to the User's registers, Bit 1 to the Supervisor's, and Bit 2 to the Kernel's. When the appropriate bits are set, D space is enabled; when clear, it is disabled. Bit 3 is used to enable the change to Supervisor mode (CSM) instruction in the 11/44. It is reserved for future use. Bit 4 enables 22-bit mapping. If Memory Management is not enabled, bit 4 is ignored and 16-bit mapping is used.

If bit 4 is clear and Memory Management is enabled (bit 0 of MMR0 is set), the computer uses 18-bit mapping. If bit 4 is set and Memory Management is enabled, the computer uses 22-bit mapping. Bit 5 is set to enable relocation in the UNIBUS map; the bit is cleared to disable relocation. Bits 15-6 are unused. On initialization this register is set to 0 and only I space is in use.

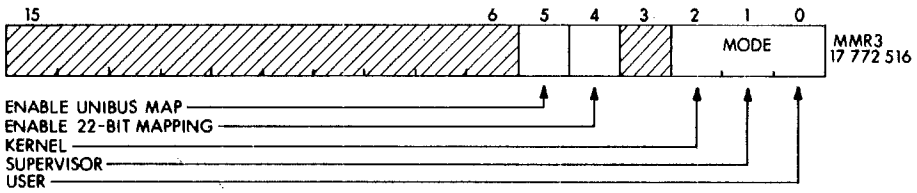


Figure 6-15 Format of Memory Management Register #3 (MMR3)

Bit	State	Operation
5	0	UNIBUS Map relocation disabled
	1	UNIBUS Map relocation enabled if bit 0 of MMR0 is set
4	0	Enable 18-bit mapping
	1	Enable 22-bit mapping
2	1	Enable Kernel D Space
1	1	Enable Supervisor D Space
0	1	Enable User D Space

Instruction Back-Up/Restart Recovery

The process of backing-up and restarting a partially completed instruction involves:

1. Performing the appropriate memory management tasks to alleviate the cause of the abort (e.g., loading a missing page).
2. Restoring the general purpose registers indicated in MMR1 to their original contents at the start of the instruction by subtracting the modify value specified in MMR1.

3. Restoring the PC to the abort time PC by loading R7 with the content of MMR2, which contains the value of the Virtual PC at the time the abort-generating instruction was fetched.

Note that this back-up/restart procedure assumes that the general purpose register used in the program segment will not be used by the abort recovery routine. This is automatically the case if the recovery program uses a different general register set (available on the PDP-11/70 only).

Clearing Status Registers Following Trap/Abort

At the end of a fault service routine, bits 15-12 of MMR0 must be cleared (set to 0) to resume error checking. On the next memory reference following the clearing of these bits, the various registers will resume monitoring the status of the addressing operations. MMR2 will be loaded with the next instruction address, MMR1 will store register change information and MMR0 will log Memory Management status information.

Multiple Faults

Once an abort has occurred, any subsequent errors that occur will not affect the state of the machine. The information saved in MMR0 through MMR2 will always refer to the first abort detected. However, when multiple traps occur, the information saved will refer to the most recent trap that occurred.

In the case that an abort occurs after a trap, but in the same instruction, only one stack operation will occur; and the PC and PS at the time of the abort will be saved.

EXAMPLES

Normal Usage

The Memory Management Unit provides a general purpose memory management tool. It can be anything from a simple memory expansion device to a complete memory management facility.

With the facilities offered by the Memory Management Unit, both single users and multi-users can make whatever memory management decisions best suit their needs. Although certain methods of using the Memory Management Unit will be more common than others, there is no limit to the ways to use these facilities.

In most normal applications, it is assumed that control over the actual memory page assignments and their protection resides in a supervisory program which would operate at the nucleus of a CPU's executive (Kernel) mode. It is further assumed that this Kernel mode program would set access keys in such a way as to protect itself from willful or

accidental destruction by other Supervisor mode or User mode programs. The nucleus can dynamically assign memory pages of varying sizes in response to system needs.

When in the Kernel mode, the program has complete control and can execute all instructions. Monitors and supervisory programs are executed in this mode. When in the user mode, the program is prevented from executing certain instructions that could:

- cause the modification of the Kernel program
- halt the computer
- use memory space assigned to the Kernel or to other users

Typical Memory Page

When the Memory Management Unit is enabled, the Kernel mode program, Supervisor mode program (11/44 and 11/70 only) and User mode program each have eight active pages, described by the appropriate Page Address Registers and Page Descriptor Registers; on the PDP-11/70 and 11/44, when D space is enabled, there are an additional eight register pairs in each mode for data. Each segment is made up of from 1 to 128 blocks and is pointed to by the Page Address Field of the corresponding Page Address Register as illustrated in Figure 6-16.

The memory segment illustrated in Figure 6-16 has the following attributes:

1. Page length: 40 blocks
2. Virtual Address range: 140000-144777
3. Physical Address range: 312000-316777
4. No trapped access has been made to this page
5. Nothing has been modified (i.e., written) in this page
6. Read-only protection
7. Upward expansion

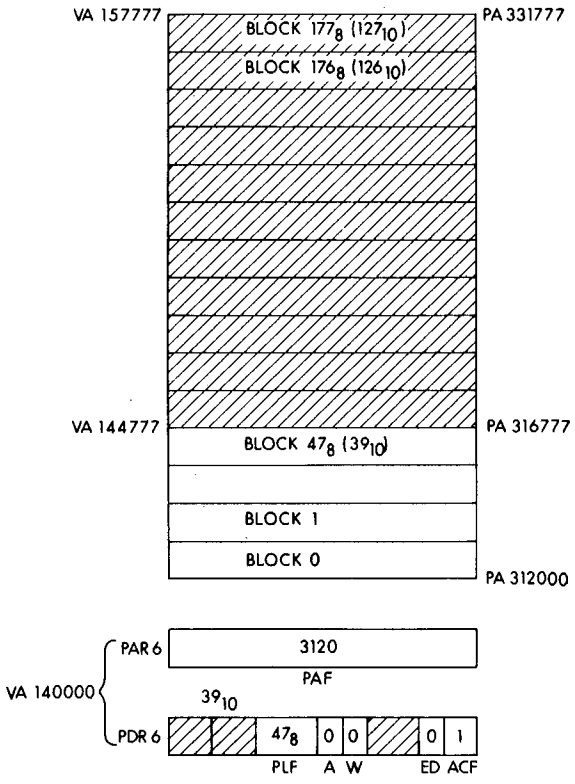


Figure 6-16 Typical Memory Page

The attributes were determined by the following scheme:

1. Page Address Register (PAR6) and Page Descriptor Register (PDR6) were selected by the Active Page Field of the Virtual Address. (Bits 15-13 of the VA = 6.)
2. The initial address of the page was determined from the Page Address Field of PAR6 (312000 = 3120 blocks × (32_{10} words per block × 2 bytes per word)).

   Note that the PAR which contains the PAF constitutes what is often referred to as a base register containing a base address or a relocation register containing a relocation constant.
3. The page length (47 + 1 = 40 blocks) was determined from the PLF contained in PDR6. Any attempts to reference beyond these 40 blocks in this page will cause a Page Length Error, which will result in an abort, vectored through Kernel Virtual Address 250.
4. The Physical Addresses were constructed according to the scheme illustrated in Figure 6-22.
5. The A-bit of PDR6, (implemented on the 11/70 only), indicates that no trapped access has been made to this page (A-bit = 0).

When an illegal or trapped reference (i.e., a violation of the Protection Mode specified by the ACF for this page) or a trapped reference (i.e., read in this case) occurs, the A-bit will be set to 1.

6. The Written bit (W-bit) indicates that no locations in this page have been modified (i.e., written). If an attempt is made to modify any location in this particular page, an Access Control Violaton Abort will occur. If this page were involved in disk swapping in a memory overlay scheme, the W-bit would be used to determine whether it had been modified and thus required saving before overlay.
7. This page is read-only protected; no locations in this page may be modified. In addition, a memory management trap will occur upon completion of a read access. The mode of protection was specified by the Access Control Field of PDR6.
8. The direction of expansion is upward (ED = 0). If more blocks are required in this segment, they will be added by assigning blocks with higher relative addresses.

The various attributes which describe this page can all be determined under software control. The parameters describing the page are all loaded into the appropriate Page Address Register and Page Descriptor Register under program control. In a normal application, it is assumed that the particular page which itself contains these registers would be assigned to the control of a supervisory program operating in Kernel mode.

Non-Consecutive Memory Pages

It should be noted that although the correspondence between Virtual Addresses and PAR/PDR pairs is such that higher VAs have higher PAR/PDRs, this does not mean that higher Virtual Addresses necessarily correspond to higher Physical Addresses. It is quite simple to set up the Page Address Fields of the PARs in such a way that higher Virtual Address blocks may be located in lower Physical Address blocks as illustrated in Figure 6-17.

Although a single memory page must consist of a block of contiguous locations, memory pages as units do not have to be located in consecutive Physical Address locations. It also should be realized that memory page assignment is not limited to consecutive non-overlapping Physical Address locations.

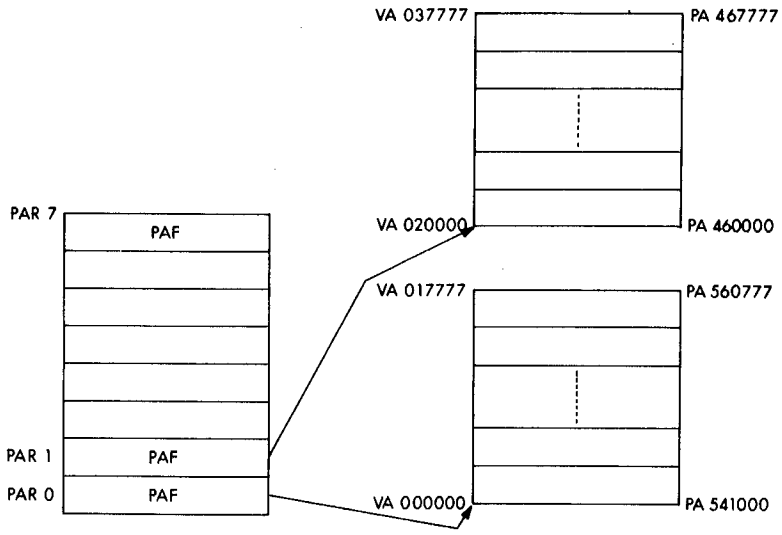


Figure 6-17 Non-Consecutive Memory Pages

Stack Memory Pages

When constructing programs, it is often desirable to isolate all program variables from pure code (i.e., program instructions) by placing them on a register indexed stack. These variables can then be pushed or popped from the stack area as needed (see Chapter 3, Addressing Modes). Since all PDP-11 family stacks expand by adding locations with lower addresses, when a memory page which contains stacked variables needs more room it must expand down, i.e., add blocks with lower relative addresses to the current page. This mode of expansion is specified by setting the Expansion Direction bit of the appropriate Page Descriptor Register to a 1. Figure 6-18 illustrates a typical stack memory page. This page will have the following parameters:

PAR6: PAR = 3120
PDR6: PLF = 175 or 125 (128 + 3)
ED = 1
A = 0 or 1 (implemented on the 11/70 only)
W = 0 or 1
ACF = nnn (to be determined by programmer)

Note: the A and W bits will normally be set by hardware.

In this case the stack begins 128 blocks above the relative origin of this memory page and extends downward for a length of three blocks. A PAGE LENGTH ERROR abort vectored through Kernel Virtual Address 250 will be generated by the hardware when an attempt is made

to reference any location below the assigned area, i.e., when the Block Number from the Virtual Address is less than the Page Length Field of the appropriate Page Descriptor Register.

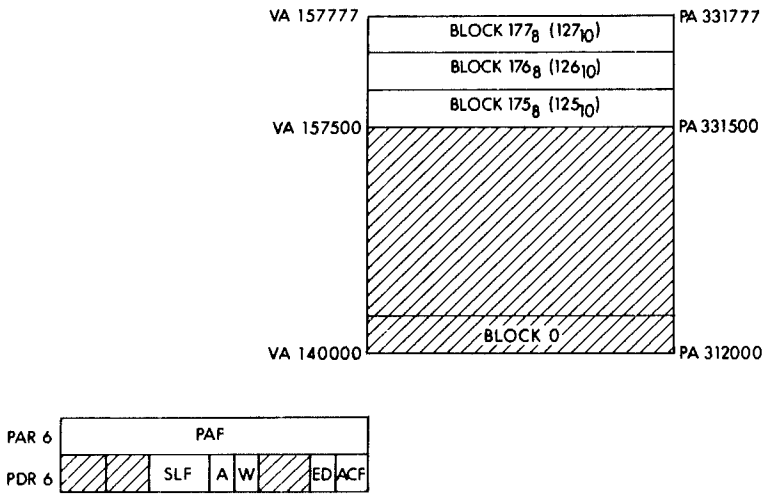


Figure 6-18 Typical Stack Memory Page

Transparency

It should be clear that in a multiprogramming application, it is possible for memory pages to be allocated so that a particular program seems to have a complete 32K memory configuration. Using relocation, a Kernel mode supervisory program can perform all memory management tasks transparently to a Supervisor or User mode program. In effect, a PDP-11 system can use its resources to provide maximum throughput and response to a variety of users, each of whom seems to have a powerful system all to himself.

Memory Management Unit—Register Map

REGISTER	ADDRESS
Memory Mgt Register #0 (MMR0)	17 777 572
Memory Mgt Register #1 (MMR1)*	17 777 574
Memory Mgt Register #2 (MMR2)	17 777 576
Memory Mgt Register #3 (MMR3)*	17 772 516
User I Space Descriptor Register (UIDR0)	17 777 600
.	.
.	.
.	.

User I Space Descriptor Register (UIDR7)	17 777 616
User D Space Descriptor Register (UDSDR0)*	17 777 620
.	.
.	.
.	.
User D Space Descriptor Register (UDSDR7)*	17 777 636
User I Space Address Register (UISAR0)	17 777 640
.	.
.	.
.	.
User I Space Address Register (UISAR7)	17 777 656
User D Space Address Register (UDSAR0)*	17 777 660
.	.
.	.
.	.
User D Space Address Register (UDSAR7)*	17 777 676
Supervisor I Space Descriptor Register (SISDR0)*	17 772 200
.	.
.	.
.	.
Supervisor I Space Descriptor Register (SISDR7)*	17 772 216
Supervisor D Space Descriptor Register (SDDR0)*	17 772 220
.	.
.	.
.	.
Supervisor D Space Descriptor Register (SDSDR7)*	17 772 236
Supervisor I Space Address Register (SISAR0)*	17 772 240
.	.
.	.
.	.
Supervisor I Space Address Register (SISAR7)*	17 772 256
Supervisor D Space Address Register (SDSAR0)*	17 772 260
.	.
.	.
.	.
Supervisor D Space Address Register (SDSAR7)*	17 772 276
Kernel I space Descriptor Register (KISDR0)	17 772 300
.	.
.	.
.	.
Kernel I Space Descriptor Register (KISDR7)	17 772 316
Kernel D Space Descriptor Register (KDSDR0)*	17 772 320

.	.
.	.
.	.
Kernel D Space Descriptor Register (KDSDR7)*	17 772 336
Kernel I Space Address Register (KISAR0)	17 772 340
.	.
.	.
.	.
Kernel I Space Address Register (KISAR7)	17 772 356
Kernel D Space Address Register (KDSAR0)*	17 772 360
.	.
.	.
.	.
Kernel D Space Address Register (KDSAR7)*	17 772 376

* These registers are implemented on the 11/44 and 11/70 only.

UNIBUS MAP (11/44 AND 11/70 ONLY)

The UNIBUS Map performs the conversion that allows devices on the UNIBUS to communicate with physical memory by means of Non-Processor Requests (NPRs). UNIBUS addresses of 18 bits are converted to 22-bit physical addresses using relocation hardware. This relocation is enabled (or disabled) under program control.

The top 8K byte addresses of the 256K UNIBUS addresses are reserved for CPU and I/O registers and are called the Peripherals or I/O Page; see Figure 6-19. The lower 248K addresses are used by the UNIBUS Map to reference physical memory.

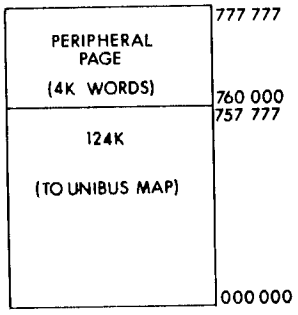


Figure 6-19 UNIBUS Address Space

The UNIBUS Map is the interface to memory from the UNIBUS. The operation is transparent to the user if it is disabled.

Relocation Disabled (11/44 and 11/70 only)
If the UNIBUS Map relocation is not enabled, an incoming 18-bit UNIBUS address has 4 leading zeroes added for referencing a 22-bit physical address. The lower 18 bits are the same. No relocation is performed.

Relocation Enabled (11/44 and 11/70 only)
There are a total of 31 mapping registers for address relocation. Each register is composed of a double 16-bit PDP-11 word (in consecutive locations) that holds the 22-bit base address; see Figure 6-20. These registers have UNIBUS addresses in the range 770 200 to 770 372.

If UNIBUS Map relocation is enabled, the five high order bits of the UNIBUS address are used to select one of the 31 mapping registers. The low-order 13 bits of the incoming address are used as an offset from the base address contained in the 22-bit mapping register; see Figure 6-21. To form the physical address, the 13 low-order bits of the UNIBUS address are added to 22 bits of the selected mapping register to produce the 22-bit physical address. Refer to Figure 6-22. The lowest order bit of all mapping registers is always a zero, since relocaton is alway on word boundaries.

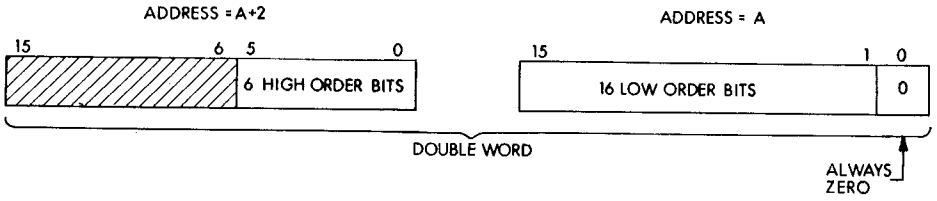


Figure 6-20 Single Mapping Register (1 or 31)

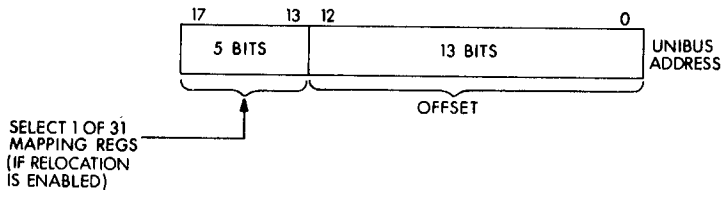


Figure 6-21 18-bit UNIBUS Address

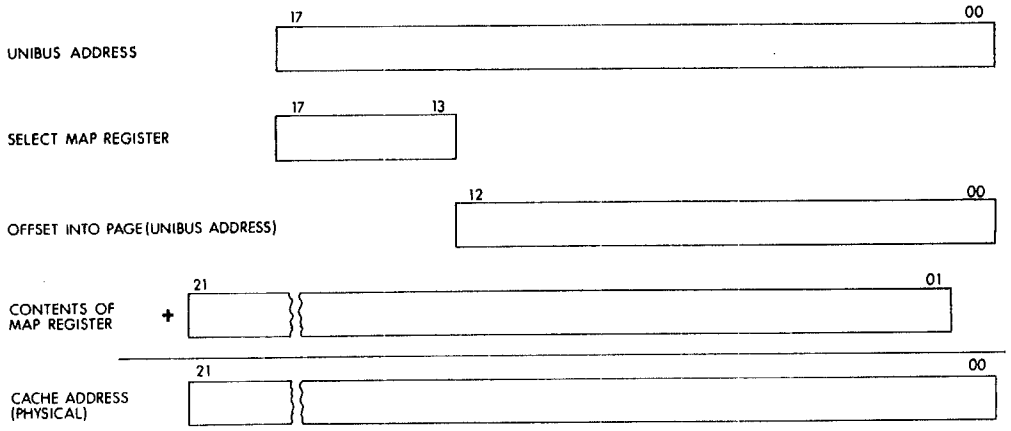


Figure 6-22 Construction of a Physical Address

Non-Existent Memory Errors

After a 22-bit physical address is generated (on the 11/70), the CPU looks at the four high order bits, bits 18 to 21, to see if they are all 1s. If this is true (range 17 000 000 to 17 17 777 777), the lower 18 bits are used for a UNBIUS address. If, after 10 to 20 μsec, there is no response, the CPU does a UNIBUS Timeout abort, and bit 4 in the CPU Error Register is set.

Non-Existent Memory Errors on the 11/34A, 11/44, and 11/60

If there is no response 10 to 20 μsec after a physical address is generated, the CPU does a UNIBUS timeout abort, and bit 4 in the CPU error register is set.

Key for Interpreting KT11 comparison chart

d - implemented but with differences

f - forces nonresident memory abort

i - illegal/invalid

k - field value signifies kernel mode

o - not implemented

r - implemented, read-only

s - field value signifies supervisor mode

u - field value signifies user mode

w - implemented, read/write

x - implemented

MEMORY MANAGEMENT COMPARISON CHART

KT11 registers, fields, and options	**/34A**	**/44**	**/60**	**/70**
MANAGEMENT REGISTERS				
Modes (Management Register Sets)				
Kernel				
I space	x	x	x	x
D space	o	x	o	x
Supervisor				
I space	o	x	o	x
D space	o	x	o	x
User				
I space	x	x	x	x
D space	o	x	o	x
Registers (per mode)				
PAR(ASK) fields				
PAF(SAF)	11-0	15-0	11-0	15-0
PDR fields				
ACF (2-0)				
000	x	x	x	x
001	o	o	o	x
010	x	x	x	x
011	o	o	o	o
100	x	x	x	x
101	o	o	o	x
110	x	x	x	x
111	o	o	o	o
ED (3)	x	x	x	x
W (6)	x	x	x	x
A (7)	o	o	o	x
PLF (14-8)	x	x	x	x
BC (15)	o	x	o	o

FAULT RECOVERY

Interrupt Vector	**250**	**250**	**250**	**250**
Registers				
MMR0 (SSR0) fields				
Abort—non resident (15)	w	w	w	w
Abort—length error(14)	w	w	w	w
Abort—read-only (13)	w	w	w	w
Trap—memory management (12)	o	o	o	w
Trap—programmer's aid (11)	o	o	o	o
Enable MM trap (9)	o	o	o	w
Maintenance mode (8)	w	w	w	w
Instruction completed (7)	o	o	o	r
Mode (6-5)				
00	x	x	x	x
01	i	x	i	x
10	i	i	i	f
11	x	x	x	x
ID(4)	o	x	o	x
Page number(3-1)	x	x	x	x
Enable(0)	w	w	w	w
MMR1(SSR1) fields				
Amount changed (15-11)	o	x	o	x
Register number (10-8)	o	x	o	x
Amount changed (7-3)	o	x	o	x
Register number (2-0)	o	x	o	x
MMR2(SSR2) fields				
VA (15-0)	r	r	r	r
MMR3(SSR3) fields				
Enable 18-bit (UNIBUS) map (5)	o	w	o	w
Enable 22-bit map (4)	o	w	o	w
Enable Kernel D space (2)	o	w	o	w
Enable Supervisor D space (1)	o	w	o	w
Enable User D space (0)	o	w	o	w
Enable CSM instruction	o	w	o	o

MEMORY MANAGEMENT INSTRUCTIONS

MFPI	x	x	x	x
MTPI	x	x	x	x
MFPD	x[1]	x	x[1]	x
MTPD	x[1]	x	x[1]	x

PROCESSOR STATUS BITS

PS (15-14)/PS (13-12)				
00	k	k	k	k
01	i	s	i[2]	s
10	i	i	i[2]	i
11	u	u	u	u

1 MFPD and MTPD are duplicates of MFPI and MTPI respectively.

2 Attempts to write 01, 10 to the PS bits 15:14 and 15:12 will result in 00 and 11 to be written.


decpack
decpack

CHAPTER 7
PDP-11/04, 11/34A

Although the PDP-11/04 and 11/34A have similar architecture, capabilities and features, the performance of these two processors is very different.

The PDP-11/04 is optimized for compactness, with the entire CPU logic confined to one circuit board. This allows extra chassis space for system expansion. By offering up to 56K bytes of core or MOS memory, the PDP-11/04 offers flexibility—now you can tailor both package and price to each application. This offers minimum hardware initially, and gives you room to grow.

The PDP-11/34A contains hardware multiply/divide instructions, memory management, an enhanced data path, and control signals for the addition of hardware Floating Point and Cache Memory options. These extra features require two modules instead of one. The PDP-11/34A looks like the PDP-11/04. Yet it has 2½ times the power of the 11/04. It also has memory expansion to 248K bytes, setting a standard of upward compatibility for 11/04-based systems.

FEATURES

The features common to the PDP-11/04 and 11/34A include:

- Self-test diagnostic routines which are automatically executed every time the processor is powered up, the console emulator routine is initiated, or the bootstrap routine is initiated. These allow system faults to be detected and avoid catastrophic failure during the running of the application program.
- Operator front panel with built-in CPU console emulator allows control from any ASCII terminal without the need for the conventional front panel with display lights and switches.
- Automatic bootstrap loader allows system restart from a variety of peripheral devices without manual switch toggling or key-pad operations.
- Choice of core or MOS memory, with parity memory optional, expandable from a minimum of 16K Bytes on the 11/04 and 32K bytes on the 11/34A to as much as 56K bytes on the 11/04, and 248K bytes on the 11/34A. This choice gives exceptional configuring flexibility, and allows tailoring of memory size to precisely fit application requirements.

- Slot-independent backplane with power and space available for significant expansion within the 5¼ or 10½ inch chassis. This provides easier system configuring than the single mounting chassis most systems have.

In addition, the PDP-11/34A includes these features:

- Integral extended instruction set (EIS) that provides hardware fixed-point arithmetic in double precision mode (32-bit operands). This significantly improves performance, compared with software implementations.
- Hardware Floating Point option allows ten times the performance of software implementations of floating point functions.
- Cache Memory option can mean up to 60 percent system performance improvement (application dependent).

MEMORY

The PDP-11/04 and the PDP-11/34A are available with MOS or Core memory. All memories are available with parity to enhance system integrity. Parity is generated and checked on all references between the CPU and memory, and any parity errors are flagged for resolution under program control. Odd parity is used, with one parity bit per 8-bit byte, for a total of 18 bits per word.

A double height module, M7850, contains parity control logic. Its control and status register (CSR) address is selectable between 772 100 and 772 136.

The CSR captures the high order address bits of a memory location with a parity error.

Memory Capacity

The PDP-11/04 has 16 address lines, which provide 64K unique byte addresses. The upper 8K addresses are reserved for UNIBUS I/O device registers, so that there remain 56K memory addresses. The PDP-11/34A, however, contains Memory Management, which extends the addressing to 18 bits (248K bytes for memory plus 8K for I/O). (See Chapter 6).

Memory Management

Memory Management is a hardware feature in the PDP-11/34A. It serves two functions: it extends memory addresses to 18 bits (248K bytes) and provides protection and relocation features for multiuser applications. The processor can be operated in either of two modes: Kernel and User. In Kernel mode, the program has complete control and can execute all instructions. Monitors and supervisor programs would be executed in this mode. In User Mode, the program is pre-

vented from executing certain instructions that could modify the Kernel program, halt the computer, gain access outside the assigned memory, or issue a restart.

Core

Each core memory unit requires two hex mounting spaces.

Core memories are non-volatile; their contents are not lost when power is shut down or lost.

The core memories available on the PDP-11/04 and PDP-11/34A are:

Model Number	Size (Bytes)	Access Time (nsec)	Cycle Time (μsec)
MM11-YP (18 Bit)	64K	450 (no parity) 600 (parity)	1.3
MM11-DP (18 Bit)	32K	560 (parity)	1

MOS

Several MOS memories are available for the PDP-11/04 and the PDP-11/34A. All are available with partially depopulated boards for smaller capacities. Since the PDP-11/04 does not have Memory Management, it cannot use memories larger than 64K bytes.

MOS memory is volatile. Information is stored as charge, and must be continuously refreshed. If there were a power loss, or if power were shut down, the MOS memory contents would be lost. To preserve the integrity of the memory during a power failure, a battery backup option is available.

Battery Backup

With the 5¼" and 10½" CPUs, there is an optional battery backup unit which can preserve the contents of 64K bytes of MOS memory for about two hours. This auxiliary power unit is a battery which is charged up by the main AC power when the computer system is operating normally. Under normal operation, the battery backup has no effect on MOS memory. But if power is interrupted, voltage sensing circuitry within the battery backup will automatically cause the MOS to be powered from this auxiliary source. Information will be retained by being refreshed at a low cycle rate, using minimum power.

	Size (Bytes)	Access Time (nsec)	Cycle Time (nsec)	Refresh
MS11-JP (18 Bit)	32K	550	700	700 nsec every 24 μsec
MS11-LB (18 Bit)	128K	360 for DATI 95 for DATO	450	560 nsec every 12.5 μsec
MS11-LD (18 Bit)	256K	360 for DATI 95 for DATO	450	560 nsec every 12.5 μsec

Cache Memory

Cache memory is an option available on the PDP-11/34A.

Cache memory reduces the cycle time for accessing frequently used main memory addresses by storing the contents of these addresses in a small, high speed memory attached directly to the CPU. This architecture bypasses the UNIBUS, thus eliminating the access and transmission times associated with the UNIBUS. A cache system offers significantly faster system speed for a small quantity of fast memory plus associated logic.

The cache option on the PDP-11/34A uses a 2K byte direct mapping approach. Without operator or programming intervention, it copies the contents of every memory location fetched from the main memory, unless it has already been copied. When this address is called again (as is very likely, because of the repetitive nature of most programs) the cache memory registers a cache "hit," places the memory contents on the CPU's internal data bus, and aborts the UNIBUS transfer going to main memory.

FLOATING POINT OPTION

The Floating Point Processor is a hardware option that enables the PDP-11/34A central processor to execute floating point arithmetic operations. It performs high speed numerical data handling much faster and more effectively than software floating point routines. Floating point representation permits a greater range of number values than is possible with the conventional integer mode. The option provides a faster alternative to the use of software floating point routines, and system speed is increased without complex arithmetic coding routines that consume valuable CPU time. The option features both single- and double-precision (32- or 64-bit) capability and floating point modes.

The option is an integral part of the central processor. It operates using similar address modes, and the same memory management facilities as the central processor. Floating Point Processor instructions can reference the floating point accumulators, the central processor's general registers, or any location in memory.

Floating Point Instruction Set Features

- 32-bit (single-precision) and 64-bit (double-precision) data modes
- Addressing modes compatible with existing PDP-11 addressing modes
- Special instructions that can improve input/output routines and mathematical subroutines
- Allows execution of in-line code (i.e., floating point instructions and other instructions can appear in any sequence desired)
- Multiple accumulators for ease of data handling
- Can convert 32- or 64-bit floating point numbers to 16- or 32-bit integers during the Store instructions
- Can convert 32-bit floating point numbers to 64-bit floating-point numbers and vice-versa during the Load or Store instructions

CONSOLES

Either of two consoles are available for the PDP-11/04 and PDP-11/34A. They are the Operator's console and the Programmer's console.

The Operator's console (KY11-LA) contains only three switches, providing control of Power ON/OFF, Initialization and Boot, and Halt/Continue.

Power	OFF	DC power to the computer is off.
	ON	Power is applied to the computer (and the system)
	STNDY	Standby; no DC power to the computer, but DC power is applied to MOS memory (to retain data). In the 5¼" box, the fans remain on.
CONT/ HALT	CONT	The program is allowed to continue.
	HALT	The program is stopped.
BOOT/ INIT	INIT	The switch is spring-returned to the BOOT position. When the switch is depressed to INITialize and then returned to BOOT, the

operation depends on the setting of the CONT/HALT switch.

If the switch is set to HALT, then only the processor is initialized and no "UNIBUS INIT" is generated. Upon lifting the CONT/HALT switch, the M9312 routine is executed, allowing examination of system peripherals without clearing their contents with "UNIBUS INIT."

If the switch is set to CONT, then initialization and execution of the M9312 program begins.

Details of the sequence of operations which occur upon booting are described in this chapter under the BOOT module section.

Console Emulation

The normal console functions traditionally performed through front panel switches can be obtained by typing simple commands on the console terminal. LOAD, EXAMINE, DEPOSIT, START, and BOOT functions are available.

The BOOT module contains a console emulator routine. When the routine is used in conjunction with the terminal, functions quite similar to those found on the programmer's console of traditional PDP-11 family computers are generated.

Booting is performed by a read-only memory which then transfers control of the CPU to an ASCII terminal, if it is present in the system. This terminal, through another read-only memory called the Console Emulator, can then function as the conventional console to execute instructions such as LOAD, EXAMINE, and DEPOSIT.

Summary of Console Emulator Functions

LOAD ADDR	Loads the address to be manipulated into the system.
EXAMINE	Allows the operator to examine the contents of the address that was loaded.
DEPOSIT	Allows the operator to write into the address that was loaded and/or examined
START	Initializes the system and starts execution of the program at the address loaded.
BOOT	Allows the booting of a device specified by a 2-character code and optional unit number

Entry into the Console Emulator

There are four ways of entering the console emulator:

- move the power switch to the ON position
- depress the BOOT switch
- enter automatically on return from a power failure
- load the address manually

After the console emulator routine has started and the basic CPU diagnostics have all run successfully, a series of numbers representing the contents of R0, R4, SP and PC will be printed on the terminal. This sequence will be followed by an @ on the next line.

Example—a typical printout on power up:

XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
(R0)	*(R4)*	*(R6, Stack Pointer,SP)*	*(PC, Program Counter)*

@ *(Prompt Character)*

NOTE

X signifies an octal numeral (0-7). Whenever there is a power-up routine, or the BOOT switch is released from the INIT position, the PC at the time will be stored. The stored value is printed out as above (noted as the PC).

Detailed instructions about using the console emulator can be found in user instruction documents, the *PDP-11/34 User's Guide* and the associated hardware manual.

Both the BOOT read-only memory and the Console Emulator read-only memory are contained in the Boot Module (M9312) described in this chapter.

The Programmer's Console (KY11-LB) contains a 20-key keypad which is functionally divided into two distinct modes: Console Mode and Maintenance Mode.

In Console Mode, facilities exist for displaying and addressing data, for depositing data and examining the contents of the UNIBUS addresses including processor registers, and for single-stepping the processor one instruction cycle at a time. This is a useful aid for program development. Note that the Operator's Console also contains these features through the use of the Console Emulator and an ASCII terminal. (This console emulator feature is still available with the Programmer's Console.)

In Maintenance Mode, the above facilities are locked out. Instead, features useful for system error diagnostics are provided. In this mode, the Programmer's Console enables the CPU's microcode to be single-stepped one clock cycle at a time and allows the UNIBUS addresses and their contents to be displayed or printed. Note that this feature is not available with an Operator's Console.

Boot Module (M9312)

The Boot Module provides the following four functions:

- It contains diagnostic routines in ROM for verifying computer operation.
- It contains the several bootstrap loader programs in ROM for starting up the system.
- It contains the console emulator routine in ROM for issuing console commands from the terminal.
- It provides termination resistors for the UNIBUS.

Diagnostics

The M9312 contains diagnostics to check both the processor and memory in a GO/NOGO mode. Execution of the diagnostics occurs automatically but may be disabled by switches on the module.

Bootstrap Loader

The M9312 contains independent programs that can bootstrap programs into memory from a selected peripheral device. Through front panel control or following power-up, the computer can execute a bootstrap directly, without the operator's keying in the initial program manually. The M9312 contains four sockets for peripheral bootstrap loader programs encoded in ROMs. The choice of ROMs is determined by the system configuration.

After execution of the CPU diagnostics, the M9312 turns control of the system over to the user at the console terminal. The system prints out status information and is ready to accept simple user commands for checking and modifying information within the computer, starting a program already in memory, or executing a device bootstrap.

The inclusion of a bootstrap loader in non-destructible read-only memory is a tremendous convenience in system operation. Bootstrap programs do not have to be loaded manually into the computer for system initialization.

BACKPLANE CONFIGURATIONS

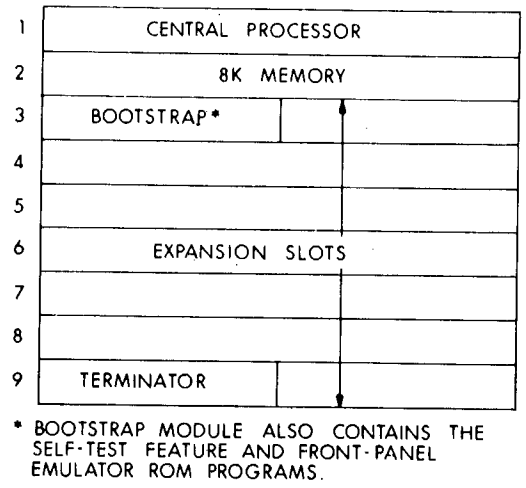


Figure 7-1 PDP-11/04 Processor Backplane Configuration

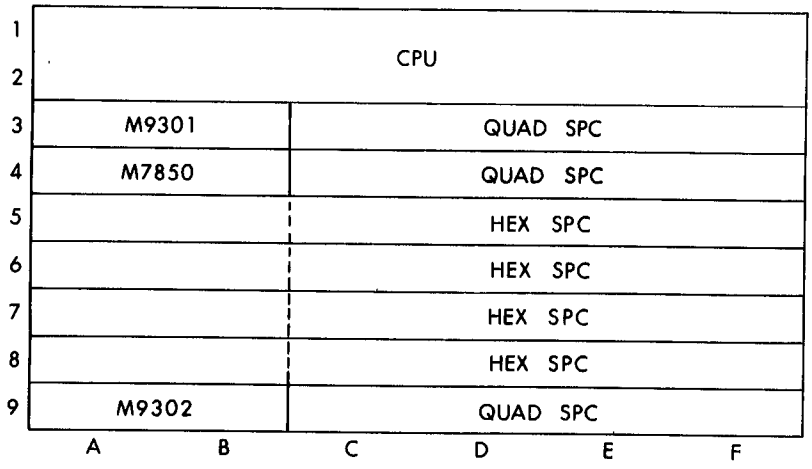


Figure 7-2 PDP-11/34A Processor Backplane Configuration

The processor backplane consists of a double system unit (SU) comprising nine hex slots. All PDP-11/34A systems contain the CPU, M9312 Bootstrap/Terminator, M7850 parity control, and M9302 (or a UNIBUS jumper to the next SU) as shown in Figure 7-2. Memory is added as follows depending on whether the system uses core or MOS.

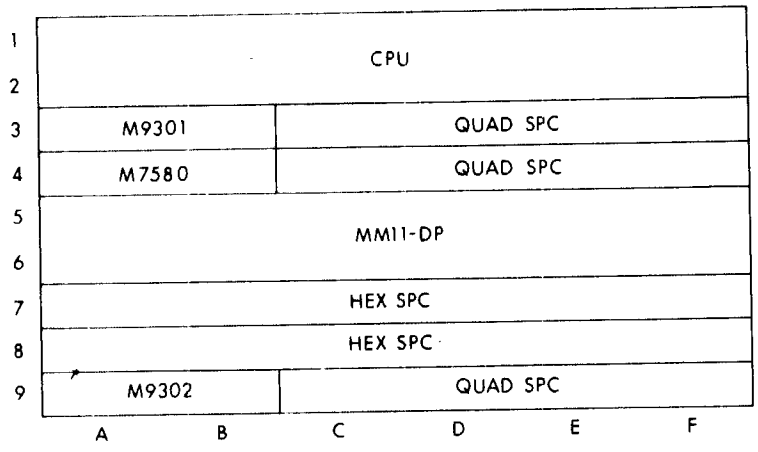


Figure 7-3 PDP-11/34A Backplane Configuration with Core Memory

Additional memory or quad and hex SPC options (DL11-W, TA11 controller, RX11 controller, etc.) may be added to the processor backplane as space allows.

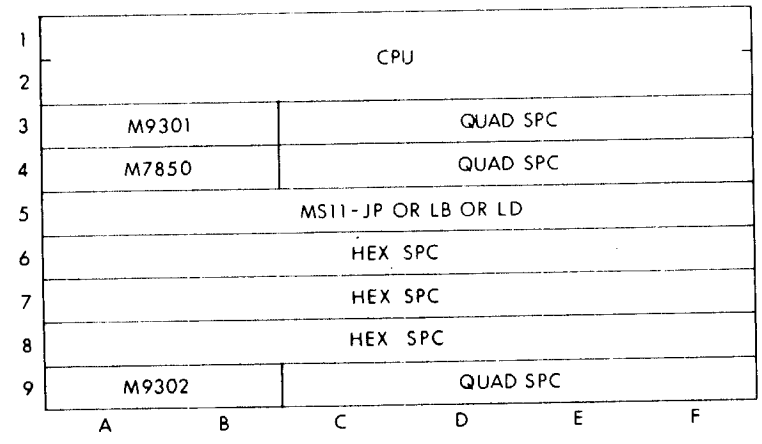


Figure 7-4 PDP-11/34A Backplane Configuration with MOS Memory

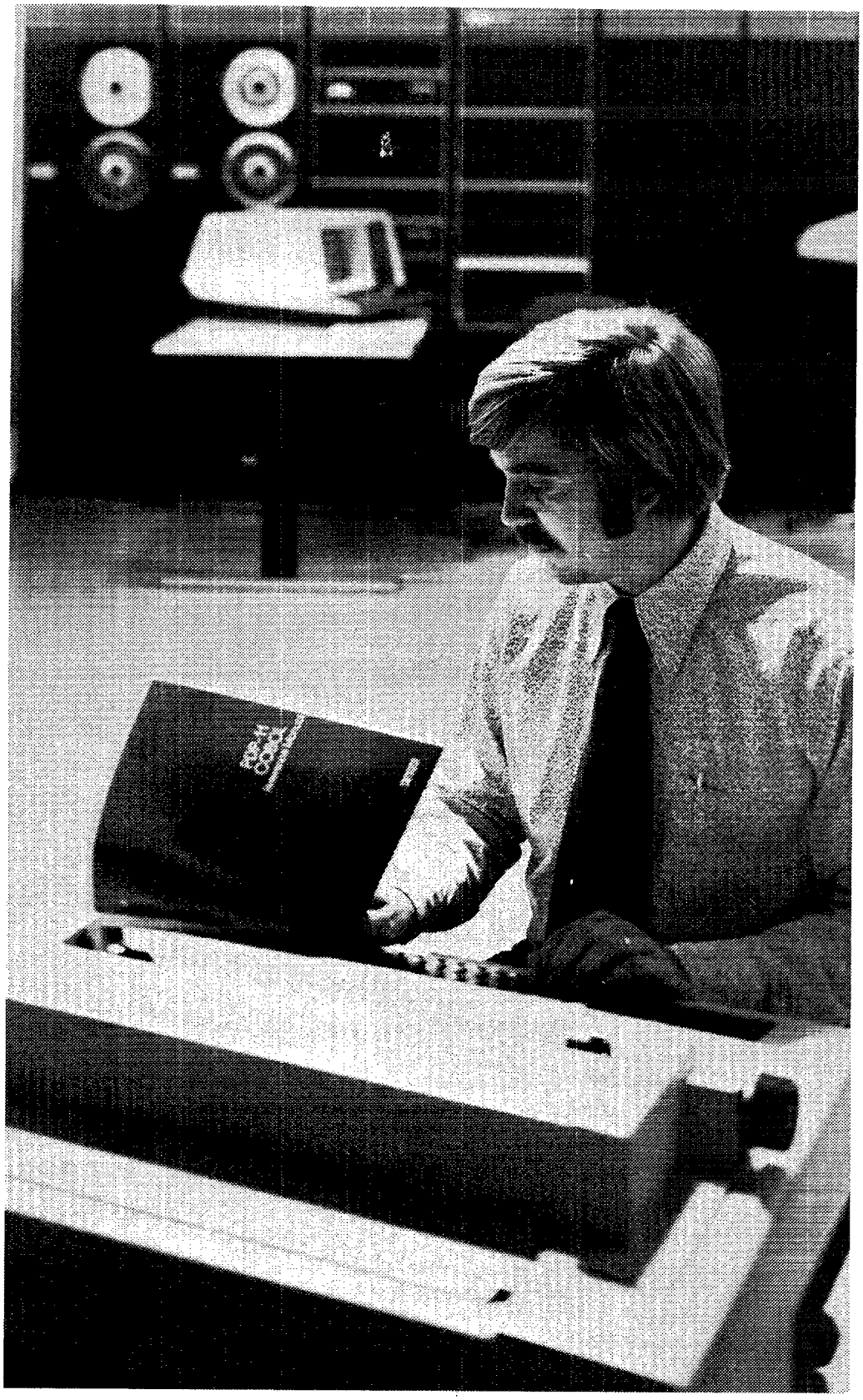
SPECIFICATIONS
PDP-11/04 AND PDP-11/34A

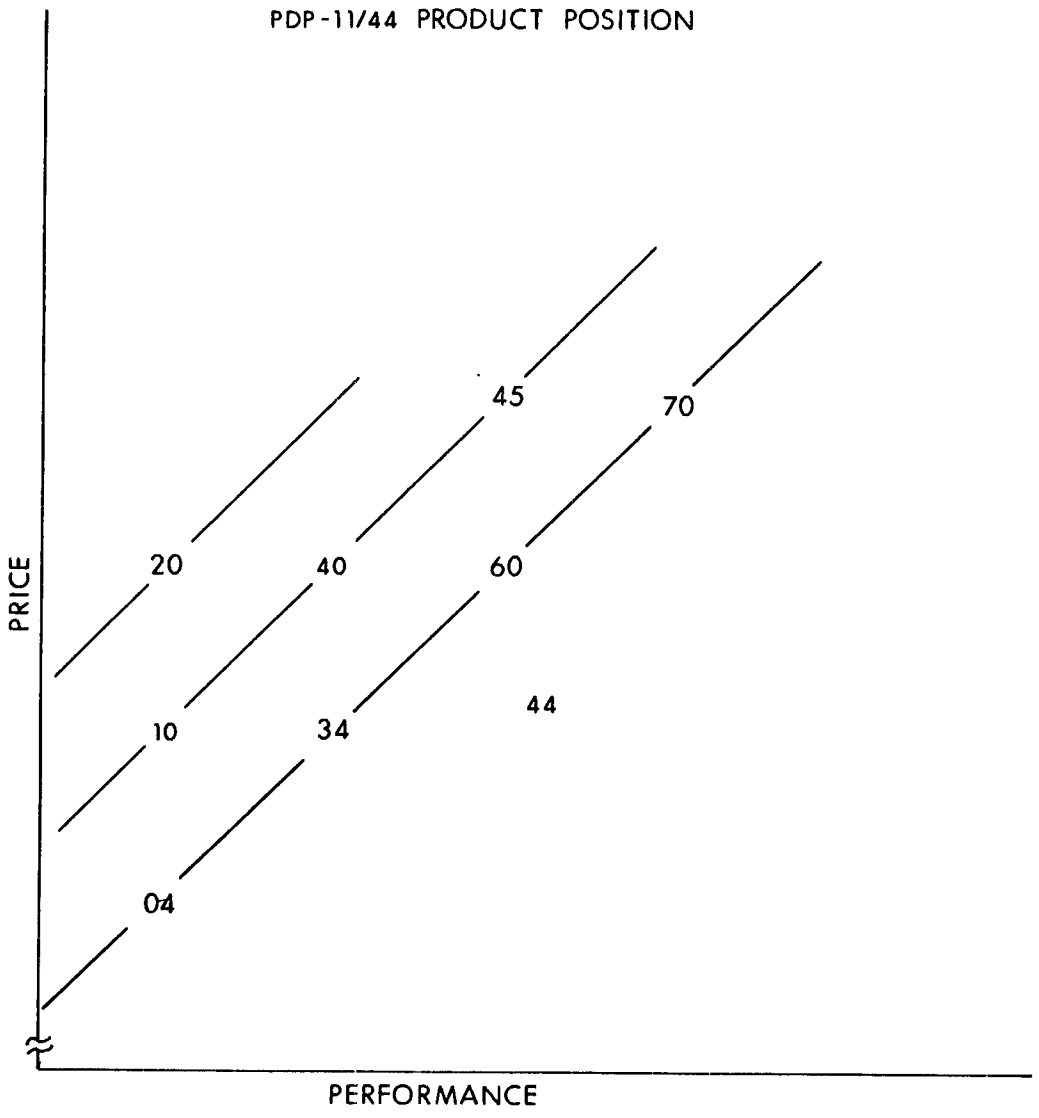
Environment

Operating Temperature:	10°-40°C, (50°-104°F)
Relative Humidity:	10-90%, non-condensing

Mechanical

Weight: 45lbs. (20 Kg)	110 lbs. (50 Kg)
Height: 5.25 in. (13.3 cm)	10.5 in. (26.7 cm)
Width: 19 in. (48.3 cm)	19 in. (48.3 cm)
Depth: 25 in. (63.5 cm)	25 in. (63.5 cm)




PDP-11/44 PRODUCT POSITION
PRICE
PERFORMANCE
20
45
70
10
40
60
04
34
44

CHAPTER 8
PDP-11/44

THE MID-RANGE MINICOMPUTER STANDARD

The 11/44, a new fourth generation mid-range PDP-11, offers new levels of functionality and performance for a machine in its price range. It offers many large, high performance machine features such as a high-speed central processor, support of megabyte memory, large 8,192 byte high-speed cache memory, optional floating point processor, optional commercial instruction set processor and optional battery backup unit. The 11/44 will provide more system up-time since it is designed to meet a rigorous reliability and maintainability program (RAMP).

FEATURES

The PDP-11/44 contains, as an integral part of the central processor unit, the following hardware features and expansion capabilities:

- Cache memory organization to provide very fast program execution speed and high system throughput
- Extended Instruction Set (EIS) for better integer arithmetic throughput
- Optional remote diagnosis
- Memory management for relocation and protection in multi-user, multi-task environments
- Intelligent ASCII console emulator with which the user can operate the computer without having access to the front panel
- TU58 cartridge tape interface port makes it easy to load software patches or field service diagnostic programs
- Ability to access up to 1 million bytes of main memory (1 byte = 8 bits) provides ample memory space for application programs
- Real-time clock which provides KW11-L compatible line-frequency clock
- Optional Commercial Instruction Set for better COBOL throughput
- Optional Floating Point Processor with advanced features, operating with 32-bit and 64-bit numbers for better FORTRAN or BASIC throughput

- Optional battery backup unit for data integrity during most power outages
- 256 Kbyte ECC MOS main memory modules provide high density, low cost memories with error correcting codes to insure better memory reliability

SYSTEM ARCHITECTURE

The PDP-11/44 is a medium scale general purpose computer using an enhanced, upwardly-compatible version of the basic PDP-11 architecture. A block diagram of the computer is shown in Figure 8-1.

The central processor performs all arithmetic and logical operations required in the system. Memory Management is standard with the basic computer, allowing expanded memory addressing, relocation, and protection. Also standard is a UNIBUS Map which translates UNIBUS addresses to physical memory addresses. The cache contains 8,192 bytes of fast, MOS memory that buffers the data from main memory.

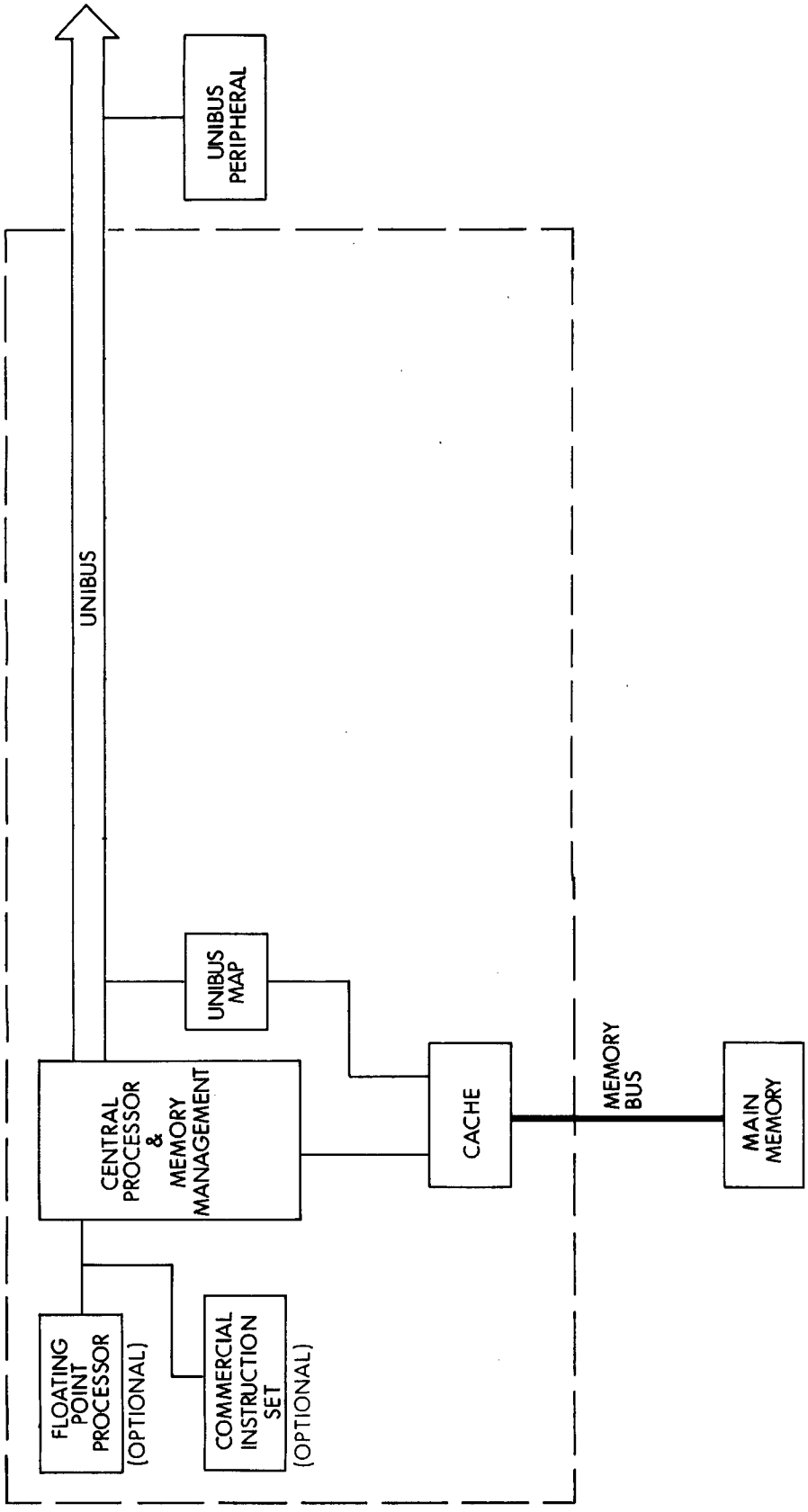


Figure 8-1 PDP-11/44 Block Diagram

The PDP-11/44 System has an expanded internal implementation of the PDP-11 architecture for greatly improved system throughput. All the memory is on its own high data rate bus. The processor has a direct connection to the cache memory system for very high-speed memory access.

The UNIBUS remains the primary control path in the 11/44 system. It is conceptually identical with previous PDP-11 systems; the memory in the system still appears to be on the UNIBUS to all UNIBUS devices. This expanded internal implementation of the PDP-11 architecture is generally compatible with earlier 11/70 programs.

CENTRAL PROCESSOR

The PDP-11/44 processor performs all arithmetic and logical operations required in the system. It also acts as the arbitration unit for UNIBUS control by regulating bus requests and transferring control of the bus to the requesting device with the highest priority.

The central processor contains arithmetic and control logic for a wide range of operations. These include fixed point arithmetic with hardware multiply and divide, extensive test and branch operations, and other control operations. It also provides room for the addition of the Floating Point Processor, Commercial Instruction Set, and UNIBUS options.

The machine operates in three modes: Kernel, Supervisor, and User. When the machine is in Kernel mode, a program has complete control of the machine; when the machine is in any other mode the processor is inhibited from executing certain instructions and can be denied direct access to the peripherals on the system. This hardware feature can be used to provide complete executive protection in a multiprogramming environment.

The central processor contains 10 general registers which can be used as accumulators, index registers, or as stack pointers. Stacks are extremely useful for nesting programs, creating re-entrant coding, and as temporary storage where a Last-In/First-Out structure is desirable. One of the general registers is used as the PDP-11/44's program counter. Three others are used as Processor Stack Pointers, one for each operational mode.

The CPU performs all of the computer's computation and logic operations in a parallel binary mode through step by step execution of individual instructions.

General Registers

The general registers can be used in many ways, the uses varying with requirements. The general registers can be used as accumulators,

index registers, autoincrement registers, autodecrement registers, or as stack pointers for temporary storage of data. Chapter 3 on Addressing describes these uses of the general registers in more detail. Arithmetic operations can be from one general register to another, from one memory or device register to another, or between memory or a device register and a general register.

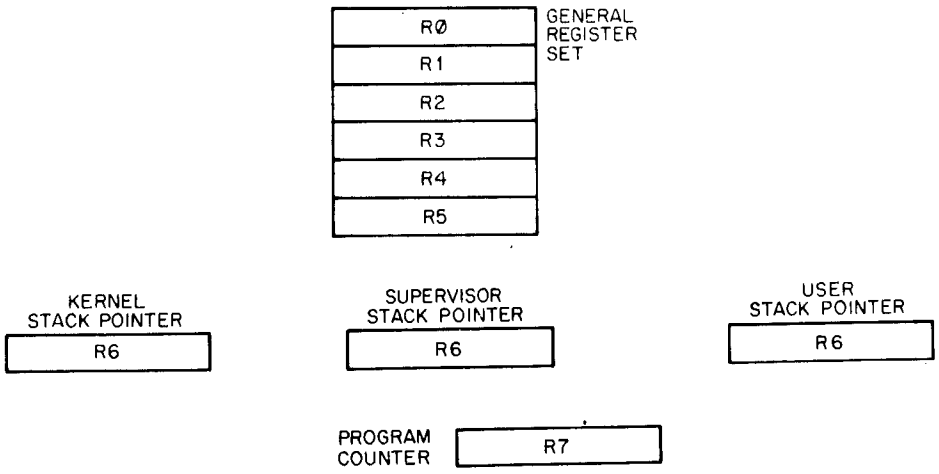


Figure 8-2 The General Registers

R7 is used as the machine's program counter (PC) and contains the address of the next instruction to be executed. It is a general register normally used only for addressing purposes and not as an accumulator for arithmetic operation.

The R6 register is normally used as the Processor Stack Pointer indicating the last entry in the appropriate stack (a common temporary with "Last-In/First-Out" characteristics). (For information on the programming uses of stacks, please refer to Chapter 5). The three stacks are called the Kernel Stack, the Supervisor Stack, and the User Stack. When the central processor is operating in Kernel mode it uses the Kernel Stack, in Supervisor mode, the Supervisor stack, and in User mode, the User Stack. When an interrupt or trap occurs, the PDP-11/44 automatically saves its current status on the Processor Stack selected by the service routine. This stack-based architecture facilitates re-entrant programming.

The remaining six registers are R0-R5. The current register set in operation is determined by the Processor Status Word.

The set of registers can be used to increase the speed of real-time data handling or facilitate multiprogramming. The six registers in the

General Register Set could each be used as an accumulator and/or index register for a real-time task or device, or as general registers.

Processor Status Word

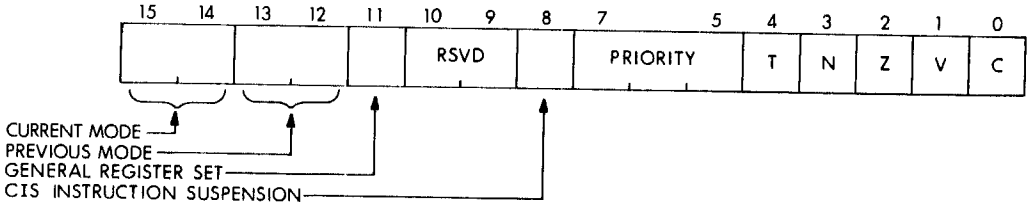


Figure 8-3 Processor Status Word

The Processor Status Word, at location 17 777 776, contains information on the current status of the PDP-11. This information includes current processor priority; current and previous operational modes; the condition codes describing the results of the last instruction; and an indicator for detecting the execution of an instruction to be trapped during program debugging.

Bit 8, when set, indicates that a commercial instruction is in process. Since commercial instructions can be suspended (interrupted), this bit will be pushed onto the stack with the rest of the Processor Status Word so that when control is returned to the routine, the commercial instruction can continue where it left off.

Modes

Mode information includes the present mode, either User, Supervisor, or Kernel (bits 15, 14), and the mode the machine was in prior to the last interrupt or trap (bits 13, 12).

The three modes permit a fully protected environment for a multi-programming system by providing the user with three distinct sets of Processor Stacks and Memory Management Registers for memory mapping.

In all modes, except Kernel, a program is inhibited from executing a "HALT" instruction and the processor will trap through location 4 if an attempt is made to execute this instruction. Furthermore, the processor will ignore the "RESET" and "SPL" (Set Priority Level) instructions, and will execute No Operation. In Kernel mode, the processor will execute all instructions.

A program operating in Kernel mode can map users' programs anywhere in memory and thus explicitly protect key areas (including

the device's registers and the Processor Status Word) from the User operating environment.

Processor Priority
The central processor operates at any of eight levels of priority, 0-7. When the CPU is operating at level 7, an external device cannot interrupt it with a request for service. The central processor must be operating at a lower priority than the priority of the external device's request in order for the interruption to take effect. The current priority is maintained in the Processor Status Word (bits 5-7). The eight processor levels provide an effective interrupt mask, which can be dynamically altered through use of the Set Priority Level (SPL) instruction described in Chapter 4 (which can only be used by the Kernel mode.) This instruction allows a Kernel mode program to alter the central processor's priority without affecting the rest of the Processor Status Word.

Condition Codes
The condition codes contain information on the result of the last CPU operation. They include: a carry bit (C), which is set by the previous operation if the operation caused a carry out of its most significant bit; a negative bit (N), set if the result of the previous operation was negative; a zero bit (Z), set if the result of the previous operation was zero; and an overflow bit (V), set if the result of the previous operation resulted in an arithmetic overflow.

Trap
The trap bit (T) can be set or cleared under program control. When set, a processor trap will occur through location 14 on completion of instruction execution and a new Processor Status Word will be loaded. This bit is especially useful for debugging programs as it provides an efficient method of installing breakpoints.

Interrupt and trap instructions both automatically cause the previous Processor Status Word and Program Counter to be saved and replaced by the new values corresponding to those required by the routine servicing the interrupt or trap. The user can, thus, cause the central processor to automatically switch modes (context switching), switch register sets, alter the CPU's priority, or disable the Trap Bit whenever a trap or interrupt occurs.

Stack Limit
The 11/44 has a Kernel Stack Overflow Boundary at location 400.

Once the Kernel stack exceeds its boundary, the Processor will complete the current instruction and then trap to location 4 (Stack Overflow).

MEMORY SYSTEM

MOS Memory with ECC

ECC (error correcting code) is a technique for checking the contents of memory to detect errors and correct them before sending them to the processor. The process of checking is accomplished by combining the bits in a number of unique ways so that parity, or syndrome, bits are generated for each unique combination and stored along with the data bits in the same word as the data. The memory word length is extended to store these unique bits. When memory is read, the data word is checked against the syndrome bits stored with the word. If they match, the word is sent on to the processor. If they do not match, an error exists and the mismatch of the syndrome bits determines which data bit is in error. The bit in error is then corrected and sent on to the processor. The error correcting code which is employed in MOS memory will detect and correct single bit errors in a word, and detect double bit errors in a word. Where a double bit error is detected, the processor is notified, as happens with a parity error.

ECC provides maximum system benefits when used in a storage system which fails in a random single bit mode rather than in blocks or large segments. Single bit error (or failure) is the predominant failure mode for MOS memory.

ECC memory provides fault tolerance with the result that multiple single bit failures can be present in a memory system without measurable degradation in either performance or reliability.

MOS memory by its nature is volatile. It cannot retain data without proper DC voltages. DIGITAL MOS memories, therefore, have battery backup (BBU) power provisions standard on the PDP-11/44, so that data may be retained during short-term loss of AC line power.

Generally, the incidence of AC line power loss varies inversely with the severity of loss. That is, there are an extremely small number of complete failures of AC power, and a relatively larger number of short-term failures or drops in voltage. No economically feasible battery backup unit can store sufficient energy to accommodate a complete AC power failure for more than several minutes.

Battery backup units are not intended to preserve data overnight or over weekends, but rather to overcome infrequent, short-term failures of AC power.

Memory Management

The Memory Management hardware is standard with the PDP-11/44 computer. It is a hardware relocation and protection facility that can convert the 16-bit program virtual addresses to 22-bit addresses. The

unit may be enabled and disabled under program control. There is a small speed advantage when in the 16-bit mode.

UNIBUS Map

The UNIBUS Map responds as memory on the UNIBUS. It is the hardware relocation facility for converting the 18-bit UNIBUS addresses to 22-bit addresses. The relocation mapping may be enabled or disabled under program control.

CACHE MEMORY

11/44 Cache Specification and Design Description

An overall block diagram of the PDP-11/44 is shown in Figure 8-4. From a function standpoint, main memory and the cache can be treated as a single unit of memory.

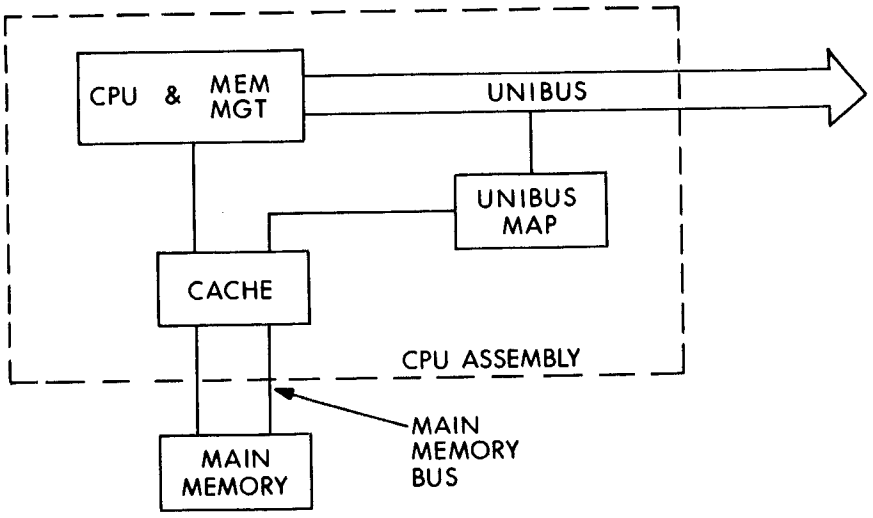


Figure 8-4 Overall Block Diagram PDP-11/44

Introduction

The 11/44 cache memory is integral to the 11/44 processor and is designed to increase the CPU performance by decreasing the CPU-to-memory read access time. It is a 8,192-byte high speed RAM memory, organized as a direct mapped cache with write-through, on a hex module.

Prerequisites

The cache module plugs into slot 7 of the processor backplane, requiring no extra connectors or modifications for its installation.

Physical Description

The 11/44 cache memory module is implemented on a hex high density (12 mil etch, 13 mil spacing) multilayer module which interfaces to

the processor via the processor backplane. Two user-accessible switches (S1 and S2) enable the cache to be shut off by causing a force-miss condition in either upper or lower cache address space. The power requirement is 7 amps of +5 Vdc. No other voltages are required. Software bits or switches for enabling or disabling cache are also provided in the MMU registers, discussed later in this chapter.

General System Architecture

The cache operates as an associative memory in parallel with the main memory and is connected to the CPU via the high speed internal data path in the 11/44 ("PAX Data Lines"). This high speed data path is isolated from the internal data path that is shared by the floating point and commercial instruction set options ("AMUX Lines"). The cache is logically connected to the PAX address and memory address buses, but is isolated from them by a set of independent receivers. When memory DATI or DATIP transfers are initiated by the CPU, the cache is strobed 100 ns later to determine if it is a valid hit with no errors. If the access is a cache hit, the processor clock is immediately restarted. This clocks in the cache data which ends the transfer from the CPU. If the strobe resulted in a read miss, then main memory MSYN is asserted and the access is to main memory with the cache performing an automatic write-through to update itself. During DATO and DATOB transfers, a write is performed to main memory with the cache updating itself if that location is presently cached. DMA, DATO or DATOB transfers from the UNIBUS are monitored by the cache and result in invalidation of cached locations. Only CPU transfers to main memory are cached. Any memory appearing on the UNIBUS will not be cached.

	DMA		CPU	
	Miss	**Hit**	**Hit**	**Miss**
Read	Nothing	Invalidate	Cached	Update
Read Bypass	Nothing	Nothing	Nothing or Invalidate	Nothing
Write Bypass	Nothing	Invalidate	Invalidate	Nothing
Write	Nothing	Invalidate	Update	Nothing

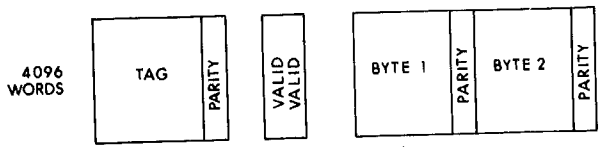
Figure 8-5 Cache Response Matrix

The response of the cache to a CPU read bypass hit is jumper selectable. In its normal configuration, jumper W1 is in and jumper W2 is

removed to allow a forced miss only to occur for a CPU read hit bypass. If the 11/44 and the KK11-B cache are to be used in a multi-ported memory system, jumper W1 is removed and jumper W2 is inserted, to allow a CPU read hit with bypass to cause an invalidation to occur to that location. This allows the software to clean potentially stale cache data that might arise in a multiported memory system.

Cache Memory Organization

The cache memory array consists of 30 4,096 × 1 RAM chips arranged in the following way:



TAG	Consists of nine tag store bits plus one bit of parity.
VALID	Consists of two bits, one of which is currently active, allowing the other bit to be cleared concurrently. By having two bits, a fast flush may be accomplished by switching to the set which has been previously cleared.
DATA	Consists of two 8-bit bytes plus a parity bit for each byte.

I/O PAGE REGISTERS

The following I/O page registers will be implemented on the 11/44 cache. All bits will be cleared by processor INIT, but not a CPU RESET instruction.

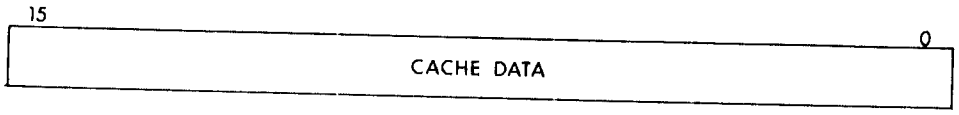


Figure 8-6 17 777 754 Cache Data Register

CDR<15:00> Cache Data Register Bits 15:00

These bits are read-only and are loaded from the 16-bit data array section of the cache RAM on every read access to main memory, except the top 256K bytes. This register can be used with the Hit on Destination Only bit to aid the cache diagnostics in identifying failures in the data section of the cache array.

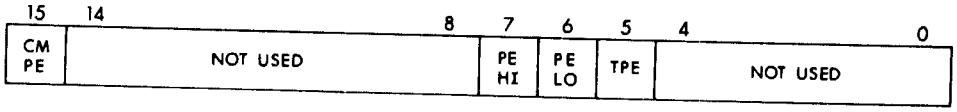


Figure 8-7 17 777 744 Cache Memory Error Register

CME<15> Cache Memory Parity Error (CMPE)

Set if a cache parity error is detected while the cache parity abort bit CCR<07> is set, or if a memory parity error occurs. If set, cache will force a miss. Cleared by any write to the CME Register or by console INIT. If the cache detects a parity error in itself, the LED mounted on the right side of the board will be on.

CME<07> Parity Error High Byte (PEHI)
CME<06> Parity Error Low Byte (PELO)
CME<05> Tag Parity Error (TPE)

Set individually when a parity error occurs in the high data byte, low data byte or tag field, respectively, if the cycle is aborted (CCR<07> is set). If the cycle is not aborted, bits 5, 6 and 7 are all set upon any cache parity error occurrence as an aid to system software compatibility. Cleared by any write to the CME register or by console INIT.

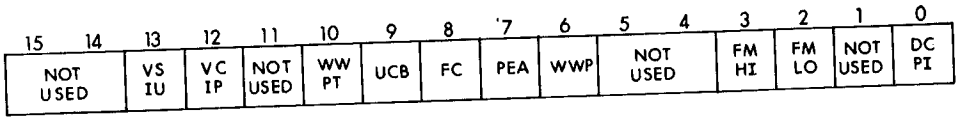


Figure 8-8 17 777 746 Cache Control Register

CCR<13> Valid Store in Use (VSIU)

This bit controls which set of valid store bits is currently being used to determine the validity of the contents of the tag store memory. It is read-only and is complemented each time that the cache is flushed.

When set, valid bit B is in use.
When clear, bit A is in use.

CCR<12> Valid Clear in Progress (VCIP)

This bit is read-only and is set to indicate that the cache is currently in the process of clearing a valid store set. The clear cycle occurs on Power-Up and when the flush cache bit is set.

NOTE: The hardware clear cycle will take approximately 800 microseconds to be accomplished. While a valid store set is being cleared the other set is in use, allowing the cache to continue functioning.

CCR<10> Write Wrong Parity Tag (WWPT)

This bit is read/write, and when set causes tag parity bits to be written with wrong parity on CPU read misses and write hits. A parity error will thus occur on the next access to that location.

CCR<09> Unconditional Cache Bypass (UCB)

This bit is read/write, and when set, all references to memory by the CPU will be forced to go to main memory. Read or write hits will result in invalidation of those locations in the cache and misses will not change the contents.

CCR<08> Flush Cache (FC)

This bit is write-only. It will always read a "0." Writing a "1" into it will cause the entire contents of the cache to be declared invalid. Writing a "0" into this bit will have no effect.

CCR<07> Parity Error Abort (PEA)

This bit is read/write and controls the response of the cache to a parity error. When set, a cache parity error will cause a force miss and an abort to occur (asserts UNIBUS signal PB L). When cleared, this bit inhibits the abort and enables an interrupt to parity error vector 114. All cache parity errors result in force misses.

CCR<06> Write Wrong Parity Data (WWPD)

This bit is read/write, and when set causes high and low parity bytes to be written with wrong parity on all update cycles (CPU read misses and write hits). This will cause a cache parity error to occur on the next access to that location.

CCR<03> Force Miss High (FMHI)

This bit is read/write, and when set causes forced misses to occur on CPU reads of addresses where address bit 12 is a 1. This bit can also be set by moving the toggle switch S1 to the right side of the board.

CCR<02> Force Miss Low (FMLO)

This bit is read/write, and when set causes forced misses to occur on CPU reads of addresses where address bit 12 is a 0. This bit can also be set by moving the toggle switch S2 to the right side of the board.

NOTE: Setting bits 03 and 02 will cause all CPU reads to be misses.

CCR<00> Disable Cache Parity Interrupt (DCPI)

This bit is read/write. When set, this bit overrides the cleared condition of the parity error abort bit, disabling the interrupt to location 114.

CCR<07>	CCR<00>	Result of Cache Parity Error
0	0	Interrupt to 114 and force miss
0	1	Force miss only
1	X	Abort and force miss.

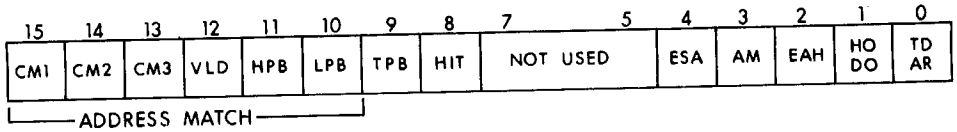


Figure 8-9 17 777 750 Cache Maintenance Register

CMR<15:10> Address Match Bits <21:16>

These bits are write-only. This register is used to set bits 21:16 of the address match register, which provides a scope sync pulse to a user accessible test point when the memory address lines (21:00) match the address match register (21:00). This feature is useful for troubleshooting the cache and 11/44 system.

CMR<15> Compare 1

CMR<14> Compare 2

CMR<13> Compare 3

CMR<12> Valid

CMR<11> High Parity bit

CMR<10> Low Parity bit

CMR<09> Tag Parity bit

CMR<08> Hit

These bits are key points in the cache that the diagnostic can use to help localize errors. This register is loaded on any read to main memory. Like the cache data register, these bits can be used with the Hit on Destination Only bit to aid the cache diagnostic in tracing cache failures.

CMR<04> Enable Stop Action

This bit can be set to allow the cache to stop the CPU clock upon detection of a cache parity error or address match condition.

CMR<03> Address Matched (AM)

This bit is read/write, and is set when the 22-bit address match register is equal to the 22-bit cache address. The bit being set is indicated by the left LED mounted on top of the board.

CMR<02> Enable Halt Action

This bit can be set to allow the cache to halt the CPU upon detection of a cache parity error or address match condition.

CMR<01> Hit on Destination Only (HODO)

This bit is read/write. When set this bit causes the cache to be enabled during the destination memory access only of an instruction. Read hits and updates will only happen during the final destination access. This feature will be a very powerful tool for cache diagnostics. When cleared, this bit has no effect on the cache. This bit should be used with caution, as it can cause stale data in the cache.

CMR<00> Tag Data from Address Match Register (TDAR)

This bit is read/write. When set, this bit enables the tag field of the cache to be written with data from bits 08:00 of the address match register. Once this bit is set, it will cause all cache writes to clear the valid bit in these locations. This feature allows the cache diagnostics to identify failures in the tag field of the cache array.

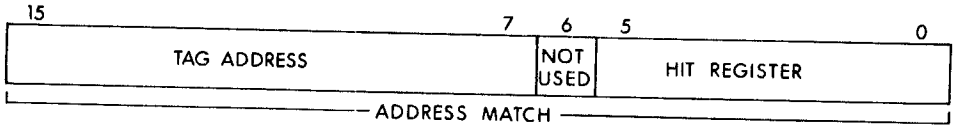


Figure 8-10 17 777 752 Cache Hit Register

CHR<15:00> Address Match Bits 15:00

These bits are write-only. This register is used to set bits 15:00 of the address match register, which provides a scope sync pulse to a user accessible test point when the memory address lines (21:00) match the address match register (21:00). This feature is useful for troubleshooting the cache and 11/44 system.

CHR<15:07> Tag Address

This field is read-only. These bits contain the nine bits of the tag store memory of the last access by the CPU to main memory (except the top 256K bytes). When used with the Hit on Destination Only and Tag Data from Address Match register bits, this field will allow the cache diagnostics to read any tag field of any location in the array.

CHR<05:00> Hit Register

This 6-bit field is read-only and shows the number of cache hits (read and write hits) on the last six CPU accesses to non-I/O page memory. The bits flow from LSB to MSB of the field with a "1" indicating a hit and a "0" indicating a miss.

OTHER CPU EQUIPMENT

Floating Point Processor

The PDP-11/44 Floating Point Processor fits integrally into the central processor. It provides a supplemental instruction set for performing single and double precision floating point arithmetic operations and floating-integer conversion in parallel with the CPU. The Floating Point processor provides both speed and accuracy in arithmetic computations. It provides 7 decimal digit accuracy in single word calculations and 17 decimal digit accuracy in double calculations.

ASCII CONSOLE

The 11/44 serial console is a standard feature which replaces the "lights and switches" programmer's console of earlier processors with logic that interprets ASCII characters to perform equivalent panel functions.

Physically, the I/O port used for the serial console function is shared with the standard system terminal (also called the "system console"),

and is mode (or state) switchable by the typing of ASCII characters at the system terminal (the LA120 or equivalent which serves as system console/programmer console).

In this section "Console State" defines the serial console mode of operation in which ASCII commands are interpreted and result in the programmer's console functions (load, examine, halt, continue, etc.) being performed. The term "Program I/O State" will be used to refer to that state in which the LA120 functions simply as the standard system terminal (also referred to as the system console).

NOTE

The console state can be entered only when the key switch is not in the local disable position.

Console State

The console state is entered by typing a reserved input character, Control P (ASCII ↑P <020>), called the CONSOLE BREAK character. The reserved character is not passed to a running program, and CONSOLE state is entered after printing the current output character, if any. While in the CONSOLE state, all input characters are interpreted by the console logic as commands to the CPU control interface. The console performs all character echoing while in the CONSOLE state.

A program running in the processor is inhibited by the console logic from sending or receiving any characters (see NOTE). The CONSOLE state is exited to the PROGRAM I/O state by typing a specific console command.

Program I/O State

The PROGRAM I/O state is entered from the CONSOLE state by typing the CONTINUE command. A running program will then resume any input/output that might have been interrupted by the CONSOLE BREAK character. Any ASCII character may be output by the program, and any ASCII character, except the CONSOLE BREAK character, may be input to the program. Character echoing is the responsibility of the CPU software in PROGRAM I/O state.

The PROGRAM I/O state is exited to the CONSOLE state by typing the CONSOLE BREAK character.

NOTE

This is accomplished by inhibiting the "ready" and "done" bits from being set. If a program just sends output to the printer without testing status bits, fewer characters will be printed.

CONSOLE COMMAND SYNTAX & SEMANTICS

< >	Angle brackets are used to denote category names. For example, the category name <ADDRESS> may be used to represent any valid address, instead of actually listing all the strings of characters that can represent an address.
[ ]	Brackets surrounding part of an expression indicate that part of the expression may not have to be typed.
<SP>	Represents one space.
<COUNT>	Represents a numeric count in OCTAL.
<ADDRESS>	Represents an address argument.
<DATA>	Represents a numeric argument.
<QUALIFIER>	A command modifier (switch).
<INPUT-PROMPT>	Represents the console's input prompt string ">>>."
<CR>	Carriage return.
<LF>	Line feed.

Console defaults:

Address defaults:	a physical 22-bit address is always assumed.
Data defaults:	all transfers are 16-bit WORD transfers.

Control Characters & Special Characters

This section lists the control characters and special characters recognized by the console adaptor, and describes their functions. All control characters, with the exception of ↑P, are optional.

CONTROL C (↑C)

Causes the suspension of all repetitive console operations such as:

1. Successive operations as a result of a /N qualifier.
2. Repeated command executions as a result of a REPEAT command.

CONTROL O (↑O)

Suppresses/enables console terminal output (toggle). Console terminal output is always enabled at the next console input prompt.

CONTROL P (↑P)	Enters CONSOLE mode (if key switch is not in local disable position). Characters typed are now fielded by the console. If the console was already in CONSOLE mode another console <INPUT-PROMPT> is typed.
CONTROL U (↑U)	↑U typed before a line terminator causes the deletion of all characters typed since the last line terminator. The console echoes: "↑U <CR> <LF>"
CONTROL S (↑S)	Will cause cease of execution of current command and of character transmission until either a ↑Q is received, or a ↑C is received. A system power failure will cause the flag that ↑S sets to be cleared before exiting CONSOLE mode (transmission of characters stops).
CONTROL Q (↑Q)	Will cause execution to continue of current command and of character transmission. If no command or character transmmission is in process, there is no response to this command. (Transmission of characters, if any, continues.)
CARRIAGE RETURN <CR>	Terminates a console command line.

The following is a list of allowable qualifiers along with their descriptions:

/G	Specifies general register space. This is a shorthand method to get to the general registers. The user need only type an E or D (examine or deposit) followed by the "/G" qualifier, and then, instead of a full 22-bit address, simply enter the register number (e.g. 0, 1, 2, 3.....). Example: E/G <SP> 7 <CR> will examine R7 as compared to: E <SP> 17777707 <CR>
/N	The /N qualifier is provided to permit examine and deposit commands to operate on multiple sequential addresses. The syntax of the /N qualifier is: <SLASH>N(:<COUNT>) The <COUNT> argument specifies the number of additional executions of the command to be performed after the initial execution. The default value for <COUNT> is one.

/M

The /M qualifier allows the operator to examine the various data and control paths in the 11/44, and, in one special case, allows the operator to change (deposit to) the CPU's MPC.

Example: E/M <SP> 0 <CR> will examine the data that is on the data bus internal to the floating point option. A list of machine dependent addresses follows:

ADDRESS	=	DATA EXAMINED
0	=	Floating Point Data
1	=	CIS MPC
2	=	CIS Data
3	=	CPU Data
4	=	CPU MPC
5	=	CACHE Data
6	=	CPU Error Register
7	=	Illegal

/E

Extensive test used only with T (TEST) command

BOOT

B [<SP> <DEVICE-NAME>] <CR>

<DEVICE-NAME> is of the following format: "DDn" where "DD" is a 2-letter device mnemonic (such as DT for DECtape), and "n" is a 1-digit unit number.

If no <DEVICE-NAME> is given with the boot command, the console will perform the boot sequence for the default system device. This is the equivalent of using the front panel "boot" switch.

The boot command is executed only if the CPU is halted, otherwise, an error message is generated.

CONTINUE

C <CR>

The CPU begins instruction execution at the address currently contained in the CPU program counter (PC) or continues execution if already running. CPU initialization is NOT performed. Additionally, the console enters PROGRAM I/O mode (see CONSOLE state and PROGRAM I/O state sections) at the same time as issuing the CONTINUE to the CPU. This command may be used to return the console to PROGRAM I/O mode even if the CPU was already running.

FILL

F [<SP> <COUNT>] <CR>

Until a power failure has occurred, the console will send <COUNT> (in system radix) null characters after each <CR> before any further transmission. A power failure will clear <COUNT>. Also, neither entering/exiting CONSOLE mode nor execution of **any** other console command (including Self-Test) affects <COUNT>. F <CR> sets fill to zero. Response: <CR> <LF> <??>

DEPOSIT

D [<QUALIFIER-LIST>] <SP>

/M, /N, /G

Deposits <DATA> into the <ADDRESS> specified. The address space used will depend upon the qualifiers specified with the command (e.g. general registers if /G, or machine dependent register if /M, or the default physical address, if no qualifiers are specified).

<ADDRESS> is a 1- to 8-digit octal number (see Note). Non-specified upper bits are set to zero. Additionally, the address may be specified by one of the mnemonics listed below.

<DATA> is a 1- to 6-digit octal number, and as with the address, non-specified upper bits are set to zero.

NOTE: When the /M (machine dependent register) qualifier is used, the value of <ADDRESS> may not be anything but 4 (this is the machine dependent register which is writable.)

When the /G (general register) qualifier is used, the value of <ADDRESS> may not exceed 20_8 .

SW — deposits to the Switch Register.

\+ — deposits to the location immediately following the last location referenced.

\− — deposits to the location immediately preceeding the last location referenced.

* — deposits to the location last referenced.

@ — deposits to the address represented by the last data examined or deposited.

Deposits are legal only when the CPU is halted. Otherwise, an error message is generated.

EXAMINE E [<QUALIFIER-LIST>] <SP> <ADDRESS> <CR>

/M, /N, /G

Examine the contents of the specified <ADDRESS>.

<ADDRESS> is a 1- to 8-digit octal number (see note) with non-specified upper address bits set to 0, or one of the mnemonics described below.

NOTE: Where the /M (machine dependent register) qualifier is specified, the value of <ADDRESS> may not exceed 7.

Where the /G (general register) qualifier is specified the value of <ADDRESS> may not exceed 17_8.

SW	examines the Switch Register.
+	examines the location immediately following the last location referenced.
–	examines the location immediately preceding the last location referenced.
*	examines the location last referenced.
@	examines the address represented by the last data examined or deposited.

The EXAMINE command is legal whether or not the CPU is running.

HALT H <CR>

The CPU will stop instruction execution after completing the execution of the instruction being executed when the console presents the HALT request to the CPU.

Upon halting the CPU, the console will display the contents of the PC.

INITIALIZE I <CR>

A CPU system initialize is performed.

The INITIALIZE command is executed only if the CPU is halted. Otherwise, an error message is generated.

MICROSTEP M [<SP> <COUNT>] <CR>

The CPU is allowed to execute the number of MICRO-instructions indicated by <COUNT>. If no <COUNT> is specified, one instruction is performed, and the console enters SPACE-BAR-STEP mode. (See below.) The Console enters PROGRAM I/O mode immediately before issuing the step, and re-enters CONSOLE mode as soon as the step completes. The CPU may be restarted by typing "C," and will continue executing the current instruction before halting.

The MICROSTEP command is executed only if the CPU is halted. Otherwise, an error message is generated.

NEXT N [<SP> <COUNT>] <CR>

The CPU is allowed to execute the number of MACRO instructions indicated by <COUNT>. If no <COUNT> is specified, one instruction is performed, and the console enters SPACE-BAR-STEP mode.

The console enters PROGRAM I/O mode immediately before issuing the step, and re-enters CONSOLE I/O mode as soon as the step completes.

SPACE-BAR-STEP FEATURE

1. Each time a NEXT or MICROSTEP command with no <COUNT> argument is given, the step is executed and SPACE-BAR-STEP mode is entered. Each depression of the SPACE-BAR will cause a single step of the microcycle or instruction.
2. To exit SPACE-BAR-STEP mode, type any character except SPACE.

REPEAT R <SP> <CONSOLE COMMAND>

This causes the console to repeatedly execute the <CONSOLE COMMAND> specified, until execution is terminated by a Control-C (↑C). Any valid console command may be specified for <CONSOLE COMMAND> except the REPEAT command.

START S [<SP> <ADDRESS>] <CR>

The START command performs the equivalent of the following sequence of console commands:

1. A system INITIALIZE is performed.
2. <ADDRESS> is deposited into the CPU Program Counter (PC). If no address is specified, the console uses the address which was associated with the last START command executed.
3. A CONTINUE is issued to begin CPU execution.

TEST

T [<QUALIFIER-LIST>]

The console subsystem will execute a self test, checking to insure its own integrity.

BINARY LOAD

X <SP> <ADDRESS> <SP> <COUNT> <CR>

This command instructs the console to prepare to load or unload <COUNT> binary data bytes from the address space specified by the <QUALIFIER-LIST>, starting from location <ADDRESS>

A count with bit 15 set indicates that the data are to be sent to the requester. The remaining bits in the count field are considered a positive number indicating the number of bytes to load or unload.

All CHECKSUMS used by this command are calculated by performing a 2's complement addition of each character into a register initially set to zeroes. If no errors occurred, the low eight bits of the register should be zero after the checksum has been received and added into it.

Once the command has been parsed, the console will stop echoing input bytes. A byte of binary data will immediately follow the command (after the <CR>). This byte of data is a byte CHECKSUM of the ASCII characters which made up the command string (including the <CR>), and will not be loaded into memory. <COUNT> will not be decremented.

If the checksum is correct, the console will respond with the input prompt, but remain in binary mode (echo suppressed) and either send data to the requester or be prepared to receive data.

If the checksum calculation detects an error, the console will respond within one second with an error message, re-enable echo of received characters, issue its input prompt and await another command. This

will prevent inadvertent operator entry into a mode where the console is accepting the next several thousand input characters as data with no escape sequence possible from the keyboard. Note that this entire command **including** the checksum may be received by the console as a single burst of characters at the console's specified character rate.

BINARY LOADING

A binary string of data of length <COUNT>$_1$ will be sent once the requester receives the input prompt indicating that the console has accepted the command. The console will deposit all but the last byte into the specified address space. The data length actually deposited during each memory reference is implementation-dependent. As the console is receiving the data it is also adding the bytes together to form another checksum, and reading back from memory (if possible) the data it just stored to assure data integrity.

Once the <COUNT> is exhausted, the final byte transmitted will be the block CHECKSUM of all the data. The console will compute the CHECKSUM as above, and respond within one second with an error message if an error is detected. In any case, the console will re-enable echo, issue an input prompt, and await the next command.

BINARY UNLOAD

As in the load command, the console processes the command and checks the CHECKSUM. If the CHECKSUM is correct, the console responds with a normal input prompt, followed by a string of bytes which is the binary data requested. As each byte is sent, it as added to the CHECKSUM and at the end of the transmission, the 2's complement of the low byte of the CHECKSUM is sent. The console then re-enables echo, issues an input prompt, and awaits the next command.

If the original CHECKSUM fails, the console will respond with an error message. It will then issue an input prompt and await the next command.

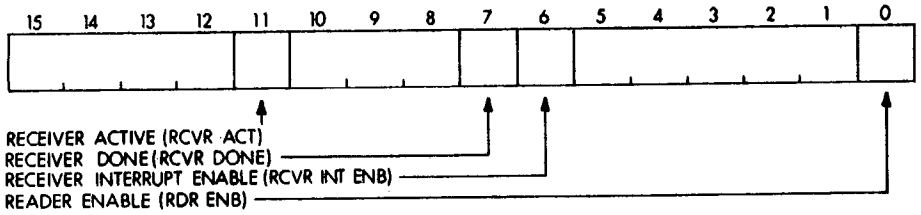
NOTE

If the console detects line parity error or memory read-after-write errors while receiving the above command or data streams, it will continue processing the incoming data and will send a single appropriate error message after the command is completed.

To make this command useful for automated test systems, the consoles should be able to receive at least 2K bytes of data in a single "X" command.

REGISTERS

Receiver Status Register (RCSR)



Bit: 15-12 **Name:** Unused
Function:

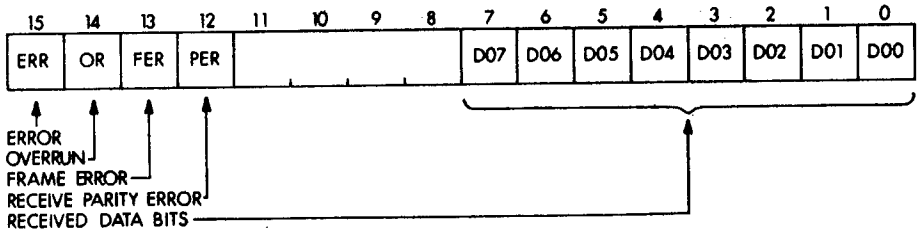
Bit: 7 **Name:** RCVR DONE
Function: Read-only. Set when an entire character has been received and is ready for transfer to the UNIBUS. Cleared by setting RDR ENB, addressing (READ or WRITE) RBUF or INIT. Starts an interrupt sequence when RECEIVER INTERRUPT ENABLE (bit 6) is also set.

Bit: 6 **Name:** RECEIVER INTERRUPT ENABLE
Function: Read/Write. Cleared by INIT. Starts an interrupt sequence when RCVR DONE is set.

Bit: 5-1 **Name:** Unused
Function:

Bit: 0 **Name:** READER ENABLE
Function: Write-only. Cleared by INIT or at middle of START bit. Advances paper tape reader of ASR teletypes. Clears RCVR DONE. 20 mA current loop circuit output associated with this bit.

Receiver Data Buffer (RBUF)



Bit: 15 **Name:** ERROR
Function: Read-only. Logical OR of OVERRUN ERROR, FRAMING ERROR and PARITY ERROR. Cleared by removing the error conditions. ERROR is not tied to the interrupt logic, but RCVR DONE is.

Bit: 14 **Name:** OVERRUN
Function: Read-only. Set if previously received character is not read (RCVR DONE not reset) before the present character is read.

Bit: 13 **Name:** FRAMING ERROR
Function: Read-only. Set if the character read has no valid bit. Also used to detect break.

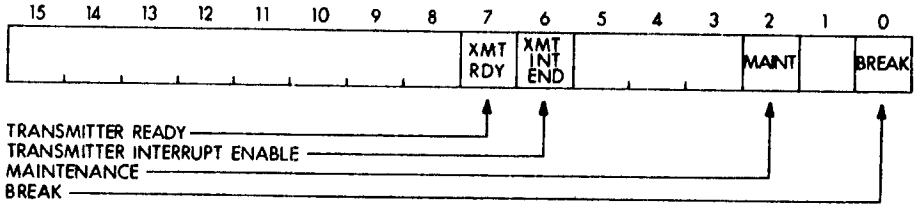
Bit: 12 **Name:** RECEIVE PARITY ERROR
Function: Read-only. Set if received parity does not agree with the expected parity. Always 0 if no parity is selected.

NOTE: Error conditions remain present until the next character is received, at which time the error bts are updated. INIT does not necessarily clear the error bits. Error bits may be disabled via a switch.

Bit: 11-8 **Name:** Unused
Function:

Bit: 7-0 **Name:** RECEIVED DATA
Function: Read-only. These bits contain the character just read. If less than eight bits are selected, the buffer will be right justified with the unused bits read as 0's. Not cleared by INIT.

Transmitter Status Register (XCSR)



Bit: 15-9 **Name:** Unused.
Function:

Bit: 7 **Name:** TRANSMITTER READY
Function: Read-only. Set by INIT. Cleared when XBUF is loaded; set when XBUF can accept another character. When set, it will start an interrupt sequence if XMIT INT ENB is also set.

Bit: 6 **Name:** TRANSMITTER INTERRUPT ENABLE
Function: Read/Write. Cleared by INIT. When set it will start an interrupt sequence if XMIT READY is also set.

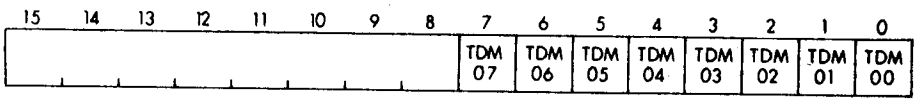
Bit: 5-3 **Name:** Unused
Function:

Bit: 2 **Name:** MAINTENANCE
Function: Read/Write. Cleared by INIT. When set it disables the serial line input to the RECEIVER and sends the serial output of the TRANSMITTER into the serial input of the RECEIVER. Forces receiver to run at transmitter speed.

Bit: 1 **Name:** Unused.
Function:

Bit: 0 **Name:** BREAK
Function: Read/Write. Cleared by INIT. When set, it transmits a continuous space. May be disabled with a switch.

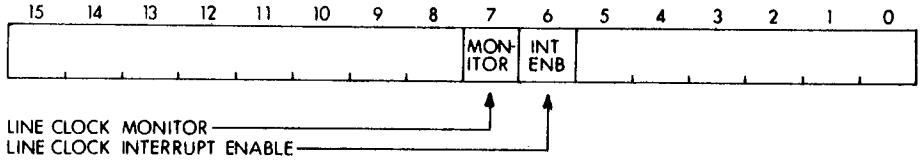
Transmitter Data Buffer (XBUF)



Bit: 15-8 **Name:** Unused.
Function:

Bit: 7-0 **Name:** TRANSMITTED DATA BUFFER
Function: Write-only. If less than eight bits are selected, then the character must be right justified.

Clock Status Register (CLKS)



Bit: 15-8 **Name:** Unused.
Function:

Bit: 7 **Name:** LINE CLOCK MONITOR
Function: Read/Clear. Set only by the line frequency clock signal and cleared only by the program. Set by INIT.

Bit: 6 **Name:** LINE CLOCK INTERRUPT ENABLE
Function: Read/Write. Cleared by INIT. When set, starts an interrupt sequence if Line Clock monitor is also set.

Bit: 5-0 **Name:** Unused.
Function:

NOTE: Line Clock circuit must be disabled via a switch when serial line portion is used as other than console interface (Address 77756X).

Floating vectors for serial line interface portion are switch selectable.

INTERRUPTS

The real-time clock has three interrupt channels: one for the receiver section (vector = XX0), one for the transmitter section (vector = XX4) and one for the clock section (vector = 100). These circuits operate independently.

ADDRESS AND VECTOR ASSIGNMENTS

The serial line port follows the same address and vector assignments as the KL11, DL11-A, B, C, D.

	Address	Vector	Priority
Line Clock	777546 777560	100	BR6
Console	777662 777564 777566 776XX0	60/64	BR4
Additional Units	776XX2 776XX4 776XX6	Floating	BR4

Where XX = 50 to 67

	77XXX0 77XXX2 77XXX4 77XXX6	Floating	BR4

Where XXX = 561 to 617.

TIMING CONSIDERATIONS

Receiver

The RCVR DONE flag sets when the UART has assembled a full character, which occurs at the middle of the first stop bit.

Since the UART is double buffered, data remain valid until the next character is received and assembled. This allows one full character time for servicing the RCVR DONE flag.

NOTE

The UART (Universal Asynchronous Receiver/transmitter) is an asynchronous subsystem. The transmitter accepts parallel characters and converts them to a serial asynchronous output. The receiver accepts asynchronous serial characters and converts them to a parallel output.

Transmitter

The UART's transmitter section is also double buffered. After initialization, the XMIT RDY flag is set. When the buffer is loaded with the first character, the flag clears but sets again within a fraction of a bit time. A second character can then be loaded, clearing the flag again. The flag then remains clear for nearly a full character time.

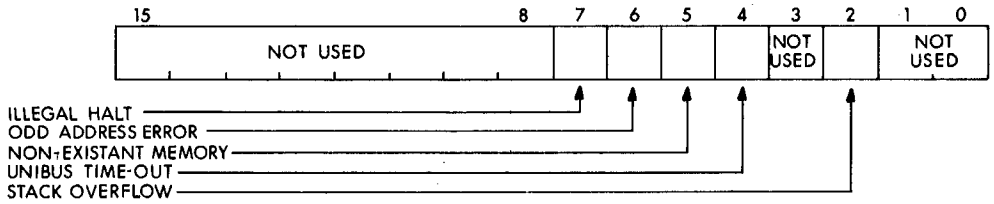
Break Generation

Setting the break bit causes the transmission of a continuous space. Since the XMIT RDY flag continues to function normally, the duration of break can be timed by the "pseudo-transmission" of a number of characters. However, since the transmitter is double buffered, a null character (all zero) should precede transmission of break to insure the previous character clears the line. Likewise, the last "pseudo-transmitted" character under break should be null.

Registers

The following five CPU registers are not accessible from the UNIBUS. They are accessed by program or console control.

CPU Error Register 17 777 766



This register identifies the source of the abort or trap that used the vector at location 4.

Bit: 7 **Name:** Illegal Halt
Function: Set when trying to execute a HALT instruction when the CPU is in User or Supervisor mode (not Kernel).

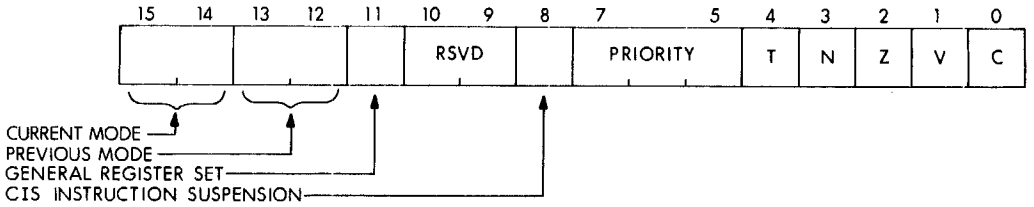
Bit: 6 **Name:** Odd Address Error
Function: Set when a program attempts to do a word reference to an odd address.

Bit: 5 **Name:** Non-Existent Memory
Function: Set when the CPU attempts to read a word from a non-existent location. This does not include UNIBUS addresses.

Bit: 4 **Name:** UNIBUS Timeout
Function: Set when there is no response on the UNIBUS within approximately 20 μsec.

Bit: 2 **Name:** Stack Overflow
Function: Set when a Stack Limit violation occurs.

Processor Status Word 17 777 776



The Processor Status Word contains information on the current status of the CPU. This information includes current processor priority; current and previous operational modes; the condition codes describing the results of the last instruction; an indicator for detecting the execution of an instruction to be trapped during program debugging; and an indicator to determine whether a commercial instruction was in progress.

PROCESSOR TRAPS

These are a series of errors and programming conditions which will cause the central processor to trap to a set of fixed locations. These include Power Failure, Odd Addressing Errors, Stack Errors, Timeout Errors, Non-Existent Memory References, Memory Errors, Memory Management Violations, Floating Point Processor Exception Traps, use of Reserved Instructions, use of the T bit in the Processor Status Word, and use of the IOT, EMT, and TRAP instructions.

Power Failure

Whenever AC power drops below 90 volts for 120V power (180 volts for 240V) or outside a limit of 47 to 63 Hz, as measured by DC power, the power-fail sequence is initiated. The central processor automatically traps to location 24 and the power-fail program has 5 msec. to save all volatile information (data in registers), and to condition peripherals for power failure.

If battery backup is present, and batteries are not depleted when power is restored, the processor traps to location 24 and executes the power-up routine to restore the machine to its state prior to power failure. If batteries are not present, a boot to default device is executed.

Odd Addressing Errors
This error occurs whenever a program attempts to execute a word instruction on an odd address (in the middle of a word boundary). The instruction is aborted and the CPU traps through location 4.

Time-Out Error
This error occurs when a Master Synchronization pulse is placed on the UNIBUS and there is no slave pulse within 20 μsec. This error usually occurs in attempts to address non-existent memory or peripherals.

The instruction is aborted and the processor traps through location 4.

Non-Existent Memory Errors
This error occurs when a program attempts to reference a non-existent memory location. The cycle is aborted and the processor traps through vector 4.

Reserved Instruction
There is a set of illegal and reserved instructions which cause the processor to trap through location 10. The set is fully described in Appendix A.

Trap Handling
Appendix A includes a list of the reserved Trap Vector locations and System Error Definitions which cause processor traps. When a trap occurs, the processor follows the same procedure for traps as it does for interrupts (saving the PC and PS on the new Processor Stack, etc.).

In cases where traps and interrupts occur concurrently, the processor will service the conditions according to the priority sequence illustrated.

Trap Priorities
1. HALT (Instruction, Switch, or Command)
2. Memory Management Fault
3. Memory Parity Errors
4. Bus Error Traps
5. Floating Point Traps
6. TRAP Instruction
7. TRACE Trap
8. OVFL Trap
9. Power Fail Trap
10. Console Bus Request (Console Operation)
11. Program Interrupt Request (PIR) level 7

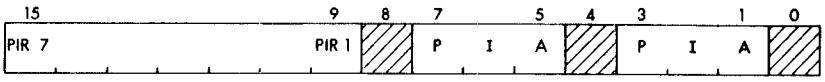
12. Bus Request (BR) level 7
13. PIR 6
14. BR 6
15. PIR 5
16. BR 5
17. PIR 4
18. BR 4
19. PIR 3
20. PIR 2
21. PIR 1
22. WAIT LOOP

Stack Limit Violations

When instructions cause a stack address to exceed (go lower than) 400_8, a Stack Violation occurs. Operations that cause a Stack Violation are completed, then a bus error trap is effected (TRAP to 4). The error trap, which itself uses the stack, executes without causing an additional violation.

PROGRAM INTERRUPT REQUESTS

A request is booked by setting one of bits 15 through 9 (for PIR 7—PIR 1) in the Program Interrupt Register at location 17 777 772. The hardware sets bits 7-5 and 3-1 to the encoded value of the highest PIR bit set. This Program Interrupt Active (PIA) should be used to set the Processor Level and also index through a table of interrupt vectors for the seven software priority levels. The figure below shows the layout of the PIR Register.



Program Interrupt Request Register

When the PIR is granted, the Processor will trap to location 240 and pick up the PC in 240 and the PSW in 242. It is the interrupt service routine's responsibility to queue requests within a priority level and to clear the PIR bit before the interrupt is dismissed.

The actual interrupt dispatch program should look like:

```
MOVB PIR,PS          ;places Bits 5-7 in PSW
                     ;Priority Level Bits
MOV R5,-(SP)         ;save R5 on the stack
```

```
MOV PIR,R5
BIC #177761,R5          ;Gets Bits 1-3
JMP @DISPAT(R5)         ;use to index through table
                        ;which requires 15 core.
                        ;locations
```

SPECIFICATIONS

Packaging

A basic PDP-11/44 consists of a 10.5″ box with a 14-slot backplane, power supply, CPU, 256 Kbyte memory, and two cabinets.

There are prewired areas within the backplane for expansion with optional equipment.

Component Parts

The basic PDP-11/44 system has:

Included Equipment

11/44 CPU
Memory Management
Bootstrap Loader
Line Frequency Clock
Serial Bus Interface for TU58
Terminal Interface
8 Kbyte Cache Memory
256 Kbyte ECC MOS Memory
BA11-A Box with Power Supply

Prewired Expansion Space for Optional Equipment

Floating Point Processor
Commercial Instruction Set
2 SPC Slots for Peripherals
768 Kbyte ECC MOS memory (up to 1024 Kbytes maximum)
3 SU open space in CPU Box

OTHER SPECIFICATIONS

AC Power

90-128 VRMS, 47 to 63Hz, 1 phase power, 19 amps RMS maximum @ 120 Vac
180-256 VRMS, 47 to 63Hz, 1 phase power, 9.5 amps RMS maximum @ 240 Vac.

Size

Each cabinet is 26.34cm × 42.21cm × 66.01cm (10.38″ high × 16.22″ wide × 26″ deep).

Weight

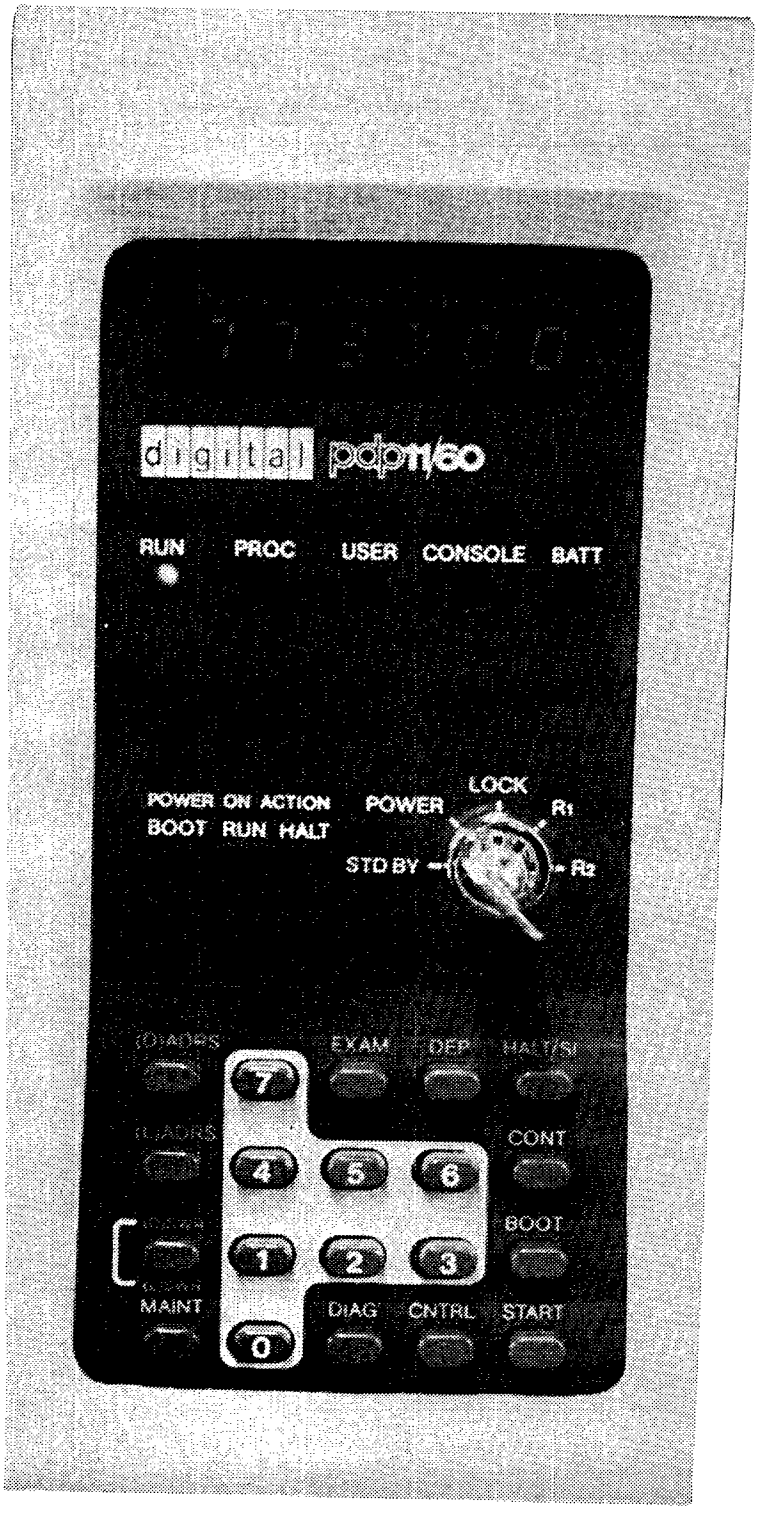
CPU Box: 40.5 Kg (90 lbs.)

Operating Environment

Temperature:	5°C to 50°C (41°F to 122°F)
Humidity:	10% to 95% with max wet bulb 32°C (89.6°F) and minimum dew point 2°C (36°F)
Altitude:	to 2.4 km (8000 ft.)

Non-Operating Environment

Temperature:	−40°C to 80°C (−40°F to 176°F)
Humidity:	to 95%
Altitude:	to 9.1 km (30,000 ft.)


digital pdp11/60
RUN
PROC
USER
CONSOLE
BATT
POWER ON ACTION
BOOT RUN HALT
POWER
LOCK
R1
R2
STD BY
EXAM
DEP
CONT
BOOT
MAINT
DIAG
CNTRL
START
7
4
5
6
1
2
3
0

CHAPTER 9
PDP-11/60

The PDP-11/60 is at the top of the mid-range PDP-11 processors for floating point applications. It is designed for both real-time applications and multi-user timesharing applications, offering a combination of features normally found only in larger computers.

The unique combination of UNIBUS-interfaced MOS or core memory and processor cache memory allows I/O transfers to memory to occur simultaneously with CPU accesses from cache memory. Since the cache memory is an integral part of the processor, the standard PDP-11 operations of the UNIBUS, I/O devices, and memory are unaffected.

FEATURES

Features of the PDP-11/60 that are explained in this chapter include:

- cache memory system
- keypad, numeric programmers' display console
- system integral floating point instructions and an optional parallel floating point processor
- internal extended instruction set (EIS)
- four levels of priority interrupt
- maintenance features
- reliability and maintainability (R.A.M.P.)

User microprogramming capability is described later in this chapter.

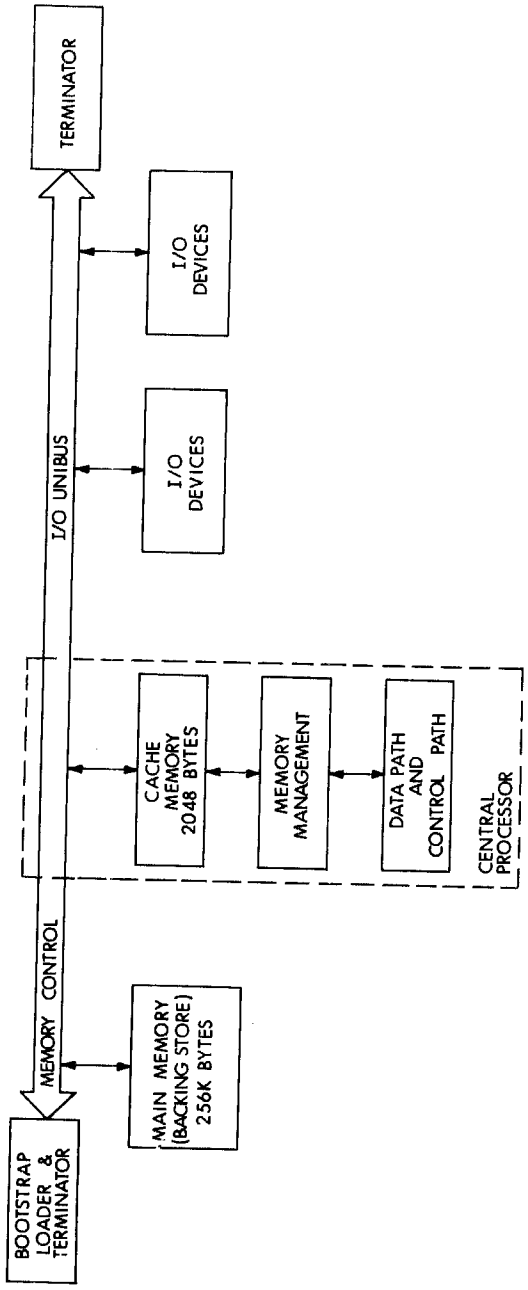


Figure 9-1 Simplified PDP-11/60 System

PDP-11/60 MEMORY

Memory for the PDP-11/60 is a combination of a 2048-byte high-speed bipolar cache memory and up to 248K bytes main memory which can be either ECC MOS or parity core memory. Cache memory provides for rapid execution of instructions, while the main memory provides cost-effective bulk storage.

Cache Memory

Cache memory is a small, high-speed memory that maintains a copy of previously selected portions of main memory for faster access of instructions and data. The PDP-11/60 computer system appears to be a conventional PDP-11 system with UNIBUS-connected memory, except that the execution of programs is noticeably faster. The only difference is in system timing; there are no changes in programming.

Cache memory is physically located within the processor and is a part both of the processor and of main memory, as shown in Figure 9-2. The high-speed bipolar cache memory is synchronized with the processor and eliminates long bus transmission and access times associated with main memory. Allocation mechanisms in the PDP-11/60 processor update the cache memory automatically and dynamically and extend the speed effect of cache across the entire main memory.

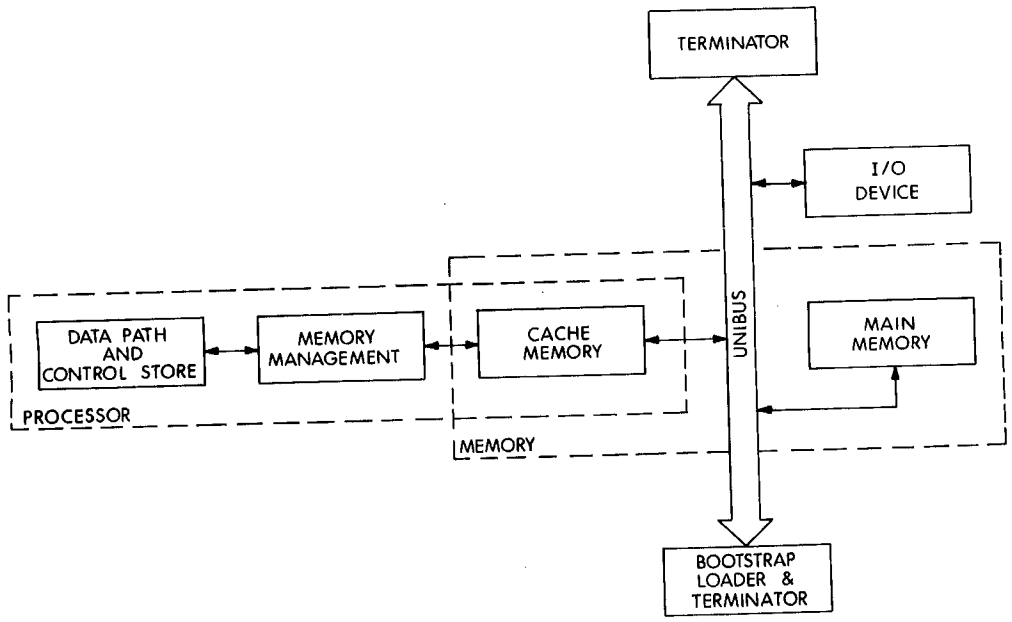


Figure 9-2 Cache Memory System Relationships

All instructions are stored in main memory; a copy of some of this information is stored in the cache. If most of the time the needed data is in the fast cache memory, the program will execute quickly, slowing down only for access to main memory. The cache system loads cache memory automatically and dynamically, in a way that gives a high probability that desired data will be in the fast memory.

The principle of program locality states that programs have a tendency to make most accesses in the neighborhood of locations accessed in the recent past. Programs typically execute instructions in straight lines or small loops, with future accesses likely to be within a few words of the last reference. Stacks grow and shrink from one end, with the next few accesses near the current bottom. Data elements are often scanned sequentially. Cache makes effective use of this program behavior by keeping copies of recently used words.

A cache system offers faster system speed for the cost of a small quantity of fast memory plus associated logic, while main memory can be implemented economically. An increase in system speed depends on the size and organization of cache, not on the type or speed of main memory. You receive a substantial speed improvement for a modest cost, and there are no programming changes. Although the exact speed improvement depends on the particular program, a judicious choice of architecture and algorithm will produce good results for all programs.

The fundamental concern is instruction execution speed. This is affected by the speed of fast and slow memory and by the percentage of time that memory references will find the data within the cache, allowing faster execution. When the needed data is found in the cache, a **hit** is said to occur. A **miss** occurs when the data is not in the cache.

The cache system within the 11/60 processor provides an additional advantage of lower UNIBUS utilization by the processor, since read memory references that are hits do not access the UNIBUS. Consequently, the UNIBUS is available more often for I/O device-to-memory transfers.

PDP-11/60 Cache Implementation

Cache memory organization can be implemented in different ways. The PDP-11/60 cache implementation is summarized in Table 9-1.

Table 9-1 PDP-11/60 Cache Implementation

CACHE CHARACTERISTICS	PDP-11/60 IMPLEMENTATION
Address mechanism	Direct mapping
Block size	Block size one
Set size	Set size one
Allocation mechanism	Write through
Replacement algorithm	Not applicable with set size of one

Direct mapping address mechanism This type of mechanism allows each word from main memory only one possible location in cache and consequently requires only one address comparison, as opposed to the fully associative cache, for example, which requires many address comparisons.

Block size The PDP-11/60 has a block size of one, which means that every time a fetch to main memory occurs, one word is fetched. One word is allocated to cache in the event of a miss.

Set size The PDP-11/60 has a set size of one, which means that there is a unique location in cache for any given word from main memory. Consequently, if a miss occurs, only one cache location is available for the data to be written into.

Write through The PDP-11/60 method of handling stale data in main memory is write through. In the write through method, the data stored in cache is immediately copied into main memory; main memory always has a valid copy of all data.

Cache Memory Data Format

Figure 9-3 shows the basic data format of the PDP-11/60 cache memory. The 2048 bytes of memory data are organized in 1024 words of 27 bits each. These 1024 words are index positions and are organized into a direct mapping cache. Bits 10 through 1 of the physical address access these index positions upon a memory reference. A complete address match requires a comparison of bits 17 through 11 of the physical address with the address information contained in the tag field of the index position. The tag field contains seven address bits, a valid bit, and a parity bit. The data field of the index position consists of two 8-bit bytes of data, each with byte parity.

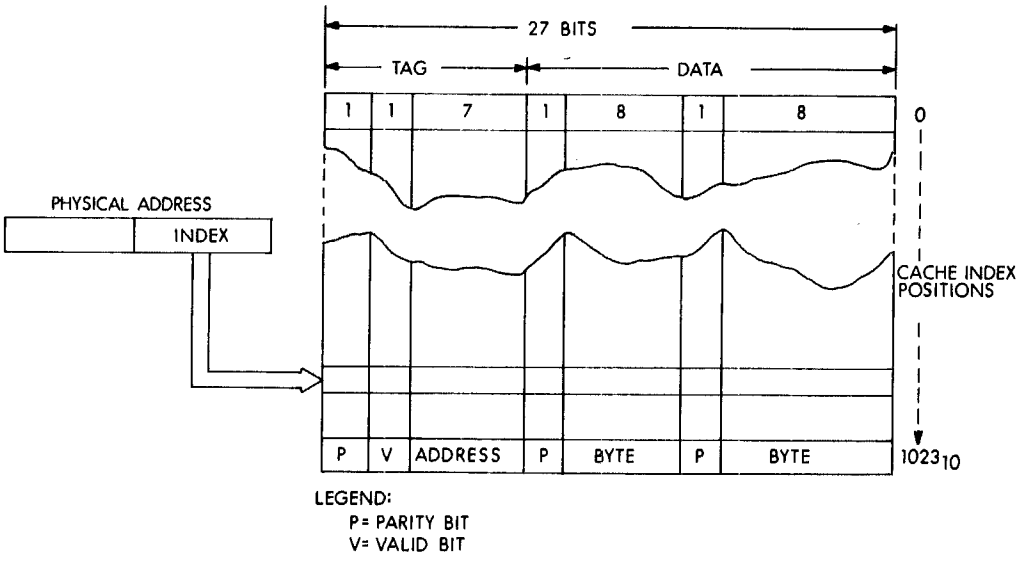


Figure 9-3 Cache Memory Data Format

Physical and Cache Address

Since the physical address space is 256K bytes, an address mapping technique is necessary to allow the 1K-word cache to be mapped directly onto any one of the 128 blocks. The physical address is divided into a tag field, an index field, and a byte field, as shown in Figure 9-4.

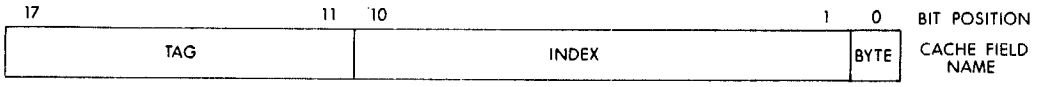


Figure 9-4 Physical Address Format

The **byte** field selects the high or low byte. The **Index** field determines which cache index position is used to store the copy of the data. This 10-bit index field specifies one of 1024 index values and is the address of a 27-bit word in the cache (see Figure 9-3).

For each of the index words, however, the remaining bits of the physial address can specify one of the 128 blocks. These bits constitute the **tag** field and are stored with the memory data in the cache index location. They prevent ambiguous determination of a specific physical address by uniquely specifying one of the 128 1K blocks.

Addressing cache then consists of applying the lower part of a physical address <10:1> against the 1K cache memory matrix and

checking the higher order physcial address <17:11> against the tag field of the index word obtained. If the tag field in the address matches the tag field stored with the data in the index word, the word obtained is the word specified by the physical address. This is designated as a hit. If the word is not the same (the fields do not match), it is designated as a miss. On a processor write, a main memory reference occurs and the new data and tag portion of its physical address will be stored in the still accessed index position. This allocation keeps current data in the cache for processor use.

Processor Memory Reference

Cache memory within the PDP-11/60 operates synchronously with processor memory references. Address information from the processor is translated to physical addresses by the memory management unit (if enabled).

The processor always looks for data in the fast cache memory first.

If the data is in the cache memory, a hit occurs, and there is no change to cache or main memory. The UNIBUS is not accessed and instruction proceeds at the fastest rate. If a miss occurs, the data and tag of a cache location are changed to correspond to the information obtained in a bus cycle to a main memory location (allocating cache). During a write into memory, if a hit occurs, both main memory and cache are updated. If a miss occurs during a word write memory reference, main memory is written, and the tag and data of the cache location are changed to correspond to the main memory location (allocating cache). For a write byte into memory, the process is similar except that cache is not allocated upon a miss.

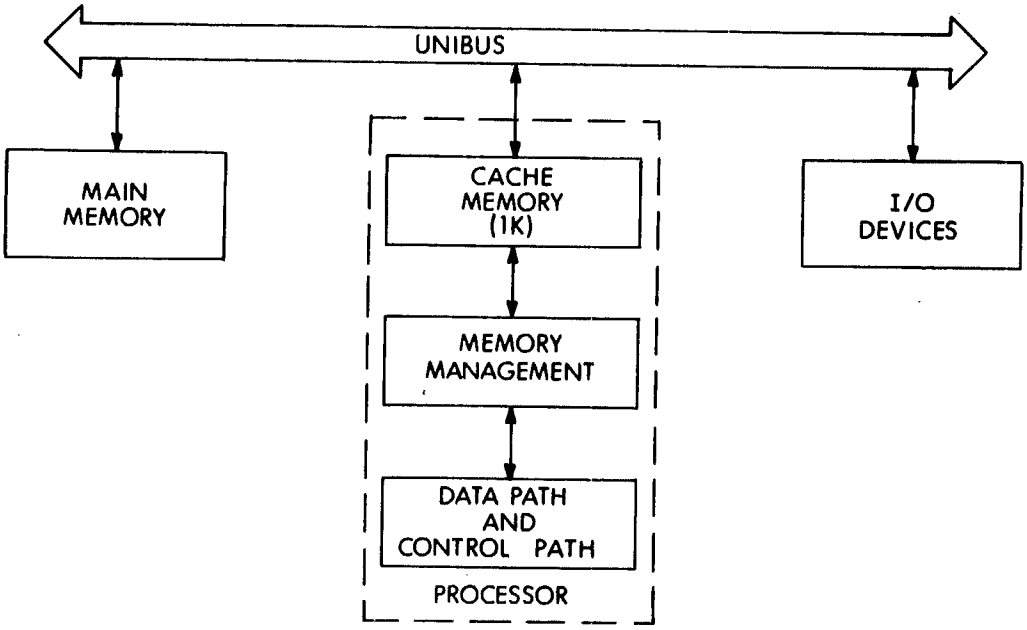


Figure 9-5 Cache Addressing Scheme

Table 9-2 Hit or Miss Operations

Processor Operation	**What Happens In Cache**	**What Happens In Main Memory**
Read (word, byte)		
Hit	No change	No change
Miss	Allocated[1]	No change
Write (word)		
Hit	Updated[2]	Written Into
Miss	Allocated[1]	Written Into
Write (byte)		
Hit	Updated[2]	Written Into
Miss	No change	Written Into
NPR Operations	**What Happens In Cache**	**What Happen In Main Memory**

Processor Operation	What Happens In Cache	What Happens In Main Memory
Read (word)		
Hit (not checked)	No change	No change
Miss (not checked)	No change	No change
Write (word or byte)		
Hit	Invalidated[3]	Written Into
Miss	No change	Written Into

1 Allocated — The data and tag of the cache location are changed to correspond to the main memory location.

2 Updated — The data in cache is changed to correspond to the data in main memory.

3 Invalidated — Valid bit in the cache word is cleared to show that the data is stale and does not correspond to the data in main memory.

NPR Memory References

Exterior UNIBUS memory references (NPRs) that alter memory (write into memory) are monitored by the cache control logic. Physical address bits 1-10 are used as an index to access the corresponding index position in cache. If the tag bits of the physical address match the address bits in the cache tag field, the index position is invalidated by clearing the valid bit in the tag field to 0. If the tag bits of the physical address do not match the address bits in the cache tag field, no change occurs (see Table 9-2).

The I/O monitoring is synchronized by the processor logic to maximize overlap of processor operations and to have a negligible effect on I/O transfer rates.

Power Failure

When power is first applied, the valid bit is cleared in all cache index values prior to any memory reference. First memory references are to the main memory. If power is lost, cache data will become invalid, but main memory, if non-volatile core, will have a correct copy of the data. If the machine contains MOS memory, with battery backup, a power fail will operate just as with core, provided the battery is functioning properly. If the battery is depleted, defective, or no battery backup is present, the machine will boot upon an automatic restart in panel lock mode. Otherwise, restart will be according to console switch setting.

Registers

The registers described in this section provide information about pari-

ty errors, memory status, and processor status. These hardware registers have program addresses in the top 4K words of physical address space (peripheral page).

Register	Address
Memory System Error	777744
Control	777746
Hit/Miss	777752

Some bit positions of the registers are not used (not implemented with hardware). These bits are always read as zeros by the program. The memory system error register is assembled from data within various error log registers and has certain restrictions. These registers are all accessed by processor program execution or console actions.

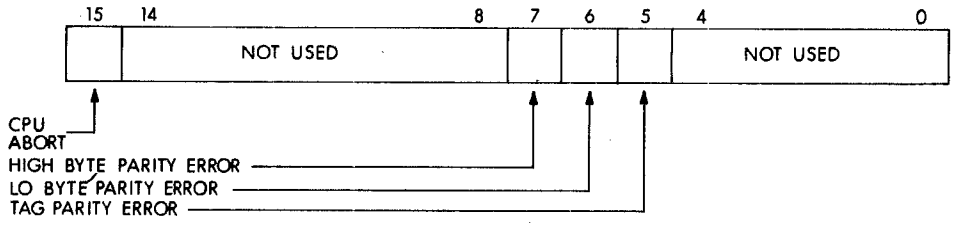


Figure 9-6 Memory System Register 777744

Bit	Name
15	CPU ABORT

Function
Set if an error occurs that caused the processor to abort an operation. The errors that cause this action are: UNIBUS memory parity error; cache parity error if the cache parity error abort bit of the cache control register is set; and user control store parity error.

Bit	Name
14-8	Not Used

Bit	Name
7	HI BYTE
6	LO BYTE
5	TAG PARITY

Function
These bits are set for cache parity errors. The bits are set for parity errors in the high byte of data, the low byte of data, or the tag field (which includes the valid bit), if the cycle is aborted. If the cycle is not

aborted (cache parity error, abort bit of cache control register is cleared and backing store references occur), all the bits (7, 6, 5) are set upon an error to aid compatibility with the PDP-11/70 system software. Then if a cache parity error occurs, the disable traps bit of the cache control register should be set to prevent the operating system from looping in the parity handler.

Bit	Name
4-0	Not Used

In the PDP-11/60, the memory system error register is assembled from error log information and is subject to the restrictions on the error log. The error log is stored, upon an error, in scratch-pad registers used for floating point constants. If error information is to be obtained, floating point instructions cannot be executed between the parity error trap (location 114) and register access. The contents of this register are undefined if the last trap is not to location 114.

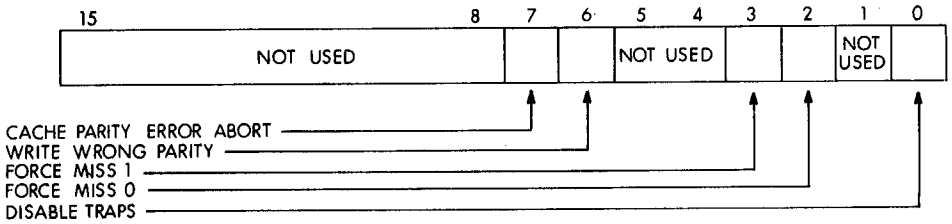


Figure 9-7 Cache Control Register 777746

Bit	Name
15-8	Not Used

Bit	Name
7	Cache Parity Error Abort

Function

This bit is cleared on power-up. It is set only during maintenance diagnostics and will cause an abort when a cache parity error occurs.

Bit	Name
6	Write Wrong Parity

Function

This bit is cleared on power-up. It is used during maintenance diagnostics and, if set, will write wrong parity in the tag, high byte, and low byte when cache is updated.

Bit	Name
5-4	Not Used

Bit	Name
3-2	Force Miss

Function
Setting these bits forces misses on reads to the cache and on attempts to invalidate the cache on NPR DATO references. Bit 3 forces misses on words 512 to 1023. Bit 2 forces misses on words 0 to 511. Setting both bits forces all cycles to main memory (degraded operation).

Bit	Name
1	Not Used

Bit	Name
0	Disable Traps

Function
Set by the cache parity error handler when it is desired to disable traps occurring as a result of non-fatal cache errors.

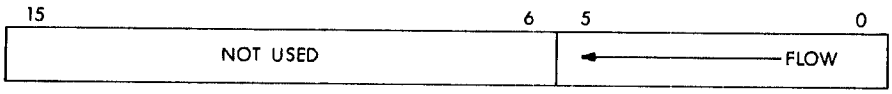


Figure 9-8 Hit/Miss Register 777752

This register indicates whether the six most recent references by the processor were hits or misses. A *one* (1) indicates a read hit; a *zero* (0) indicates a read miss or a write. The lower numbered bits are for the more recent cycles.

All the bits are read only. The bits are undetermined after a power up. They are not affected by a console start clear.

PDP-11/60 MAIN MEMORY

The 11/60 is available with ECC MOS and parity core configurations. The use of ECC MOS memory provides the following advantages:

- lower power consumption
- greater packaging density
- more reliable systems
- lower maintenance costs

Mos Memory with ECC

ECC (error correcting code) is a technique for checking the contents of memory to detect errors and correct them before sending them to

the processor. The process of checking is accomplished by combining the bits in a number of unique combinations so that parity, or **syndrome**, bits are generated for each unique combination and stored along with the data bits in the same word as the data. The memory word length is extended to store these unique bits. When memory is read, the data word is again checked, syndrome bits are regenerated and compared with the syndrome bits stored with the word. If they match, the word is sent on to the processor. If they do not match, an error exists and the mismatch of the syndrome bits determines which data bit is in error. The bit in error is then corrected and sent on to the processor. The error correcting code which is employed in MOS memory will detect and correct single bit errors in a word, as well as detect multiple bit errors in a word. Where a multiple bit error is detected, the processor is notified, as happens with a parity error.

ECC provides the maximum system benefits when used in a storage system which fails in a random single bit mode rather than in blocks or large segments. Single bit error (or failure) is the predominant failure mode for MOS. ECC provides fault tolerance with the result that multiple single bit failures can be present in a memory system without measurable degradation in either performance or reliability.

PARITY

Parity is used extensively in the PDP-11/60 to insure the integrity of data handling and to enhance the reliability of system operation.

- UNIBUS memory parity is isolated to 1K blocks. When a memory parity error occurs on the UNIBUS, examination of the memory parity register (in memory) will localize the error to the nearest 1K block.
- Cache parity has parity bits associated with the tag field (including valid bit), high byte of the data word, and low byte of the data word.
- There is a parity bit for each 16-bit segment of the 48-bit Writable Control Store (WCS) word.

Software routines are used to log the occurrence of parity errors, to handle recovery from errors, and to provide information on system reliability and performance. Diagnostic software uses parity to isolate errors for rapid repair.

Error Response

The PDP-11/60 has two basic responses to parity errors:

1. The operation is aborted, an error log is generated concerning conditions at the point of error, and a macro trap is generated immediately.
2. The operation continues, an error log cannot be generated, and a macro trap occurs at the end of the instruction. The macro trap

can be suppressed for cache errors if the disable traps bit of the cache control register is set.

The first response (abort) is necessary for UNIBUS memory parity errors, non-existent address, time-out, and writable control store parity errors. In these instances, there is no way to continue or to reconstruct operation. For cache parity errors, the abort mode with its error log can be used for diagnostic purposes. This mode is enabled by setting the cache parity error abort bit of the Cache Control Register. Multiple errors within MOS memory will result in memory cycle abort with an immediate macro trap. The error correcting logic will correct the error and will set the single error bit in the MOS memory control and status registers. The register can be analyzed by system software to note degradation of memory operation. Continued operation depends upon the ability to obtain correct information. For cache parity errors, a reference to memory can provide this information. This reference occurs automatically if the cache parity error abort bit of the cache control register is not set. Certain bits of the memory system error register are set for compatibility with the PDP-11/70, and a macro trap through location 114 (parity error trap) occurs at instruction end. For error correcting codes in MOS memory, single errors are corrected in the memory to provide correct information.

The PDP-11/60 has been designed to allow recovery from cache parity errors, and to allow operation in a degraded mode if a section of the memory system is not operating properly. This type of operation is possible under program control by using the built-in control registers.

If data found in a location in cache does not have correct parity, a memory reference can automatically occur to allow program execution to proceed. If a number of locations in cache fail, it is possible to turn off the cache using the force miss bits of the cache control register. All of the read data is then brought from the memory. Operation of programs will be slower, but the system will yield correct results. A decision to force misses in cache at the system level should be considered irrevocable until the system is restarted or diagnostic corrections have occurred. Restart requires an update of the full 1024-word cache during the absence of I/O device intervention.

If the macro trap after an automatic memory reference takes too much system time, it can be suppressed by the disable traps bit of the cache control register. This disable is also used in the service routine for the cache error to prevent endless traps.

If part of the main memory is not working, the memory management unit can be used to map around the malfunctioning memory. Indication of main memory failure comes from the UNIBUS memory

parity error bit for single core memory failures and multiple MOS memory errors.

For MOS memory, the error correcting logic will correct a single bit error and will set the single error bit in the MOS memory control and status register. No direct macro program indication of an error is made. The control and status register of the MOS memory does contain a single error bit that is set and remains set until cleared by program action. This register also contains a disable correction code bit to provide diagnostic determination of the exact error.

Cache Parity Error and Cache Control Register (CCR)

The system response to cache parity errors depends on the state of the Cache Control Register bits CRR<07> (Cache Parity Error Abort) and CCR<00> (Disable Traps).

In most operations, CCR<07> and CCR<00> are zero. On a cache parity error, a trap will occur at the end of the current instruction. In this mode, where a cache parity error occurs, an internal control bit is set that will cause a trap through location 114, and a memory reference occurs to obtain correct data. In the error handling routine, the CPU abort bit (bit 15) in the memory system error register is examined. It will be zero, indicating that the instruction was not aborted. Bits 7, 6, and 5 (high byte, low byte and tag parity) will all be set for compatibility with PDP-11/70 software.

In certain situations (the parity handler routine, for example), it is desirable to disable traps because of cache parity errors. The disable is done by making CCR<07> equal to zero and CCR<00> equal to one. In this mode, a cache parity error results in a memory reference and no macro trap occurs.

If more detailed information about a cache parity error is required, as in a diagnostic, the current instruction is aborted. This mode occurs with CCR<07> set to one. When the error occurs, the memory reference cycle is aborted, an error log is constructed, and a macro trap through location 114 occurs. The information in the error log includes exact parity error location to the address and byte level. When the memory system error register is examined, it will contain a value of one, indicating that an instruction was aborted.

Table 9-3 summarizes cache parity operations.

Table 8-3 summarizes cache parity operations.

Table 9-3 Actions Upon Cache Parity

Cache Control		Memory System Error		System Action
CCR<07>	CCR<00>	MSE<15>	MSE<07,06,05>	
0	0	0	All Set	Memory references, trap through location 114 at instruction end.
0	1	0	All Set	Memory references, no trap to location 114.
1	0	1	Set per Error	Abort current operation, construct error log, trap through location 114.
1	1	1	Set per Error	Abort current operation, construct error log, trap through location 114.

PDP-11/60 PROGRAMMERS' CONSOLE

The Programmers' Console, KY11-P, is designed for both computer operation and maintenance. The console maintenance function supplements other PDP-11/60 features such as a single clock, microbreak, processor error log, error status registers, and device-specific macrodiagnostics. Microdiagnostics are also available with the microprogramming options.

The PDP-11/60 console allows direct control of the computer system. It contains a power switch that is used as the master switch for the system. The console is used for starting, stopping, resetting, and debugging programs. Lights, switches, and a numeric display provide for monitoring operation, system control, and maintenance. Debugging and detailed tracing of operations can be accomplished by executing single instructions. Contents of all memory locations and internal registers can be examined and data entered manually from the console control switches and numeric keypad.

Power-up

Power is turned on by turning the rotary switch to POWER. What occurs after power-up depends on the position of the BOOT/RUN/HALT slide switch prior to the power-up. The slide switch allows three modes of power-up: BOOT, RUN, and HALT.

BOOT:	Position allows the system to boot directly from the bootstrap loader (M9301-YX). The boot procedure is accomplished by selecting the device to be bootstrapped by the microswitches, placing the slide switch in BOOT position and turning the rotary switch to POWER.
RUN:	Position allows automatic restart on power-fail recovery. Power-up is to location 24 for automatic restart and occurs in all except MOS memory systems where the battery is depleted or absent; in that case, a boot occurs.
HALT:	Position allows the use of the console keypad after power-up.

NOTE

To initialize the computer, depress the HALT/SI key while holding the START key down. You should have the slide switch in the desired position, as it is examined during the initialization. This procedure can be used to clear a hung bus without turning off power.

Starting and Stopping

If you wish to start a program from a given address, turn the power on after placing the slide switch in HALT position. The keypad is active and the desired address can be loaded into the temporary switch register (and also in the display) by pressing the numeric switches. After checking the desired address as displayed, press the LADRS key. Then press START, holding the CNTRL key down. This starts the program. The CONSOLE light goes out and the RUN light comes on; the system is now in run mode. The only keys which are active are the numerics, DADRS, D/LSWR, and HALT/SI.

To terminate the execution of a program, depress the HALT/SI key. This stops the program, the CONSOLE light comes on and the RUN light goes out. The system is in console mode and all the keys in the keypad are active. The display contains the PC. In this mode of operation, a single instruction is executed each time the HALT/SI key is depressed.

Console Indicators and Switches

The PDP-11/60 Programmers' Console provides the following facilities:

- 6-digit octal display for address and data indication
- Processor state lights:
  RUN
  PROC (Processor)
  USER
  CONSOLE
  BATT (Battery)
- BOOT/RUN/HALT slide switch for power-up action
- 5-position rotary switch for selection of machine status
  STD BY
  POWER
  LOCK (panel lock)
  R1 (Remote 1)
  R2 (Remote 2)
- Keypad switches (four rows of five switches each, noted below)
  DADRS (Display address)
  7 (Numeric)
  EXAM (Examine)
  DEP (Deposit)
  HALT/SI (Halt/Single Instruction)
  (L)ADRS (Load Address)
  4 (Numeric)
  5 (Numeric)

6 (Numeric)
CONT (Continue)
(D)SWR, (L)SWR (Display Switch Register, Load Switch Register)
1 (Numeric)
2 (Numeric)
3 (Numeric)
BOOT (Bootstrap)
MAINT (Maintenance)
0 (Numeric)
DIAG (Diagnostic)
CNTRL (Control)
START

NOTE

The CNTRL interlocks the action of other keys. The functions labeled in blue on the control panel cause irrevocable change in machine status and therefore are interlocked with CNTRL. CNTRL must be depressed when the other key is activated for action to occur.

Console Internal Registers

The console has the following four internal registers (in the A and B Scratchpads) for its exclusive use. Each is 16 bits wide and has the functions noted below:

CNSL.CNTL, Console Control, is a 16-bit register containing various control bits used in the console microcode. It also contains the upper two bits of the temporary switch register, the console switch register, and console address register.

CNSL.TMPSW, Console Temporary Switch Register, is 18 bits wide and is made up of the CNSL.TMPSW register and two bits in the control register. The temporary switch register is used as a buffer to collect the numerics and is also used for display.

CNSL.ADRS, Console Address Register, is also 18 bits wide and is composed of the CNSL.CNTL to allow 18-bit physical addresses.

CNSL.SW, Console Switch Register, is also 18 bits wide and is composed of the CNSL.SW register and two bits in the CNSL.CNTL register. This register has a UNIBUS address of 777570 and is a read-only register. If a write is attempted at this address, the data will be written in the console address register and then displayed on the console if the DLOCK bit in the CNSL.CNTL is not set. This bit is cleared in (D)ADRS and START functions and set in every other function. (D)ADRS can be used to unlock the display and provide a positive indication of movements by the program to 777570.

Switches and Indicators

Octal Display

The octal display is a 6-digit, 7-segment display used to display address or data information. The display allows 18 bits (octally coded) to be displayed.

Processor State Lights

RUN — If illuminated, indicates that the processor is executing instructions. The light will not remain illuminated during an extended WAIT instruction.

PROC — If illuminated, indicates that the processor is the master device and has control of the UNIBUS.

USER — If illuminated, indicates that the processor is in user mode and certain restrictions on instruction operation and Processor Status word (PS) loading exist.

CONSOLE — If illuminated, indicates that the processor is in console mode and is under control of the console keypad switches (manual operation).

BATT — Battery monitor indicator. This indicator will function only in machines containing the battery backup options and has the following four states:

OFF — Indicates either no battery present, or battery depletion, if battery is present.

ON (Continuous) — Indicates that battery is present and is charged.

Flashing (Slow) — Indicates battery is charging.

Flashing (Fast) — Indicates loss of power, and also that battery is discharging while maintaining MOS memory contents.

BOOT/RUN/HALT Slide Switch

Power-up action is determined by this switch position, in conjuction with PANEL LOCK status. If the rotary switch is in LOCK position (deactivating all keypad functions), inadvertent operation of the slide switch has no effect. Upon power-up, the slide switch is treated as if it were in the RUN position, regardless of its physical position. If the battery is depleted for a MOS memory system, RUN is altered to a BOOT action.

If the console is not in LOCK position, and a power fail occurs, three choices of recovery (BOOT, RUN, and HALT) are available.

BOOT — Power-up to the M9301 bootstrap terminator.

RUN — Power-up to location 24, which contains the power-up vector.

Note that this action occurs independent of battery status on a MOS memory system.

HALT — Power-up to the console. The CONSOLE light is illuminated and the console keypad switches are active.

Rotary Switch

STD BY — Removes DC power from processor and core memory (MOS memory battery charger is still on).

POWER — Applies power to all units. All console controls are operable in console mode.

LOCK — Deactivates all keypad functions. With power switch in LOCK position, the position of the BOOT/RUN/HALT slide switch has no effect when power-up occurs; power-up is to RUN, unless a battery depletion causes BOOT upon a MOS memory system.

R1 — Local control is deactivated to allow operation from a remote console. The octal display on the console will be blanked.

R2 — Console action is the same as R1.

NOTE

The CNTRL (Control) key is used in conjunction with some keys to prevent accidental operation of certain functions. When these are used, the CNTRL key must be depressed.

Those keys which are interlocked with the CNTRL key are indicated with an asterisk.

Keypad Switches

The keypad contains twenty switches which are priority-encoded into a unique 5-bit code. Simultaneous operation of the keys will allow the operation of the switch with the higher priority. The switches are listed in order of their priorities, with the highest priority described first.

0-7 NUMERICS — Activation of any of the numeric keys causes the binary value of that key to be entered into the low-order three bits of the temporary switch register. The previous contents are left-shifted three bits. Each 3-bit binary value is displayed in octal representation for each additional numeric depressed; the temporary switch register (one of four internal registers) is left-shifted three bits; and the octal display is left-shifted one digit. Consequently, a 6-digit octal number is generated as octal digits are entered from the right and left-shifted. Operation of the numerics occurs in both console mode and run mode.

HALT/SI (Halt/Single Instruction) — Depressing this switch while the processor is in run mode halts the processor between instructions, after outstanding trap sequences, and before bus requests. The processor is now in console mode and the CONSOLE indicator is lighted. The octal display indicates the program counter for both HALT and SI functions. Depressing the HALT/SI switch now causes a single instruction to be executed.

To initialize the system without a program start, it is necessary to depress the HALT/SI key while holding the START switch down.

NOTE

The PDP-11/60 differs from other PDP-11 processors regarding the single instruction step function. An operator cannot simply load an address and immediately start single-stepping. To start from an arbitrary address, the PC must be loaded using the maintenance key function; one can then single-step by pressing the HALT/SI switch.

(D)SWR, *(L)SWR (Display Switch Register, Load Switch Register) — Displays the contents of the console address register in both console and run modes. If this switch is depressed while the CNTRL switch is held, the contents of the temporary switch register are loaded into the console switch register. The contents of the console switch register are displayed. Operative in both console and run modes.

(D)ADRS — Displays the contents of the console address register and clears the display lock bit, thus enabling the program movements to 777570. Operation occurs in both console and run modes.

Console Mode Functions

Console operations are word-ordered operations. If an odd bus address (bit 00 enabled) is used, the odd address is stored in the console address register (CAR). Examine or deposit operations in this address will be treated as word operations (bit 00 ignored).

An EXAM or a DEP operation that references a non-existent address causes the machine to display the console address with all the decimal points lighted. Time-out trap sequences to non-existent addresses will not be activated.

NOTE

The following switches are active only in console mode.

(L)ADRS (Load Address) — Depressing this switch transfers the con-

tents of the temporary switch register to the console address register to be used in subsequent DEP or EXAM operations. The contents of the console address register are displayed in the octal display and all decimal points are lighted.

EXAM (Examine) — Depressing this key accesses the UNIBUS address specified in the console address register and displays the contents of that address in the octal display. Sequential examination increments the address by 2 and displays the contents of the incremented addresses. This incrementation process is stopped if any key other than the numeric keys is depressed.

DEP (Deposit) — Depressing this switch deposits the contents of the temporary switch register at the UNIBUS address specified by the console address register. The console switch register is not changed. To deposit data into sequential addresses, all that is necessary is to press the DEP key. This automatically word-increments the console address register and deposits the data into the incremented address. This process is stopped if any key other than the numeric keys is depressed.

***CONT** (Continue) — Depressing this switch allows the processor to leave console mode and continue operation at the present Program Counter (PC) location without a BUS INIT. The display is unaltered.

***START** — Depressing this switch begins machine operation at the address (PC) specified by the console address register after a BUS INIT signal. Operation occurs only in console mode and the CONSOLE mode light is turned off. The display is unaltered.

***BOOT** (Bootstrap) — Depressing this switch will cause a BUS INIT and will start the boot program of the M9301 module. The display is unaltered.

***DIAG** (Diagnostic) — Depressing this switch transfers control to the DCS (Diagnostic Control Store) module, if present. Otherwise, the computer enters console mode. The display is unaltered.

MAINT (Maintenance) — This key is used to read and write the internal registers. The procedure for reading an internal register is:

1. Load the temporary switch register with the read code of the register that you wish to read. The opcodes for the internal registers are listed in Table 9-4.
2. Depress the (L)SWR keypad switch while holding the CNTRL keypad switch depressed. This transfers the contents of the temporary switch register to the console switch register.

3. Depress the MAINT keypad switch while holding the CNTRL switch depressed. The console display will display the contents of the register specified by the opcode in step 1.

The procedure for writing an internal register is:

1. Load the temporary switch register with the write opcode of the register that you wish to write. The internal register function codes are listed in Table 9-4.
2. Depress the (L)SWR keypad switch while holding the CNTRL keypad switch depressed. This transfers the contents of the temporary switch register to the console switch register.
3. Load the temporary switch register with the data to be written by depressing the applicable numeric switches.
4. Depress the MAINT keypad switch while holding the CNTRL switch depressed. The console display will display the data that has been written into the specified register.

NOTE

In Table 9-4, a register can have several names, depending upon its use at a given time. For example, in the C Scratchpad, the register with the read/write code of 100/300 can be used as a floating point (FP) register or as the log jam register.

Table 9-4 Internal Registers Read/Write Function Codes

ASP LO: A SCRATCHPAD [0:15]	
Register	**Read/Write Code**
R0	000/200
R1	001/202
R2	002/202
R3	003/203
R4	004/204
R5	005/205
R6	006/206
R7	007/207
FAC3[0]	010/210
FAC3[1]	011/211
FAC3[2]	012/212

Register	Read/Write Code
FAC3[3]	013/213
FAC3[4]	014/214
FAC3[5]	015/215
USER R6	016/216
FDST3	017/217

ASP HI: A SCRATCHPAD [16:31]

Register	Read/Write Code
WCSA[0]	020/220
WCSA[1]	021/221
WCSADR	022/222
CNSL.CADR	023/223
R(SRC)	024/224
R(SRC X)	025/225
R(SRC I)	
R(T1A)	
R(VECT)	
R(DST)	026/226
R(DST X)	
R(T2A)	
R(DST I)	
CNSL.SW	
CNSL.TMPSW	027/227
FAC1[0]	030/230
FAC1[1]	031/231
FAC1[2]	032/232
FAC1[3]	033/233
FAC1[4]	034/234
FAC1[5]	035/235
GEN, WHAMI	036/236
FPSHI, FEC FDST 1	037/237

BSP LO: B SCRATCHPAD [0:15]

Register	Read/Write Code
R0	040/240
R1	041/242
R2	042/242
R3	043/243
R4	044/244
R5	045/245
R6	046/246
R7	047/247
FAC2[0]	050/250
FAC2[1]	051/251
FAC2[2]	052/252
FAC2[3]	053/253
FAC2[4]	054/254
FAC2[5]	055/255
USER R6	056/256
FDST2	057/257

BSP HI: B SCRATCHPAD [16:31]

Register	Read/Write Code
WCSB[0]	060/260
WCSB[1]	061/261
WCSB[2]	062/262
R(ZERO)	063/263
R(SRC)	064/264
R(SRC, X)	
R(ES)	
R(T1B)	
R(DST)	065/265
R(DST X)	
R(T2B)	
R(ES)	
R(IR)	066/266
CNSL.CNTL	067/267

CSP: C SCRATCHPAD [0:15]

Register	Read/Write Code
FP, LOG JAM	100/300
FP, LOG SERVICE	101/301
FP, LOG PBA	102/302
FP, LOG CUA	103/303
FP, LOG FLAG/INTR	104/304
FP, LOG WHAMI	105/305
FP, LOG CACHE DATA	106/306
FP, LOG TAGE CPU	107/307
FP, CONSOLE	110/310
FP, CONSOLE	111/311
FP, CONSOLE	112/312
FP, CONSOLE	113/313
CONST 2	114/314
MD	115/315
CONST 0	116/316
CONST 1	117/317

OTHER REGISTERS

Register	Read/Write Code
JAM	140/ — Read-only
SERVICE	141/ — Read-only
PBA	142/ — Read-only
CUA	143/ — Read-only
FLAG	144/344
REV	146/ —
DCSO	152/ —
DCS1	153/ —
D REG	— /345 Write-only
S REG	— /346
COUNT	147/347
NUA	— 350
RES	— 351
INIT	— 352

Register	Read/Write Code
	NO-OPS @
	340-343
	150-177
	120-137
	320-337
	352-377

PROGRAMMABLE STACK LIMIT

The stack limit allows program control of the lower limit for permissible stack addresses. This limit may be varied in increments of $(400)_8$ words, up to a maximum address of 177400, almost the top of a 32K word memory.

The normal boundary for stack addresses is 400. The stack limit option allows this lower limit to be raised, providing more address space for interrupt vectors or other data that should not be destroyed by a program.

There is a stack limit register, with the following format:

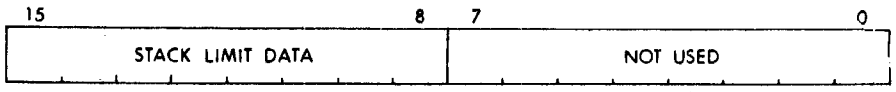


Figure 9-9 Stack Limit Register Format

The stack limit register can be addressed as a word at location 777774, or as a byte at location 777775. The register is accessible to the processor and to the console, but not to any bus device.

Bits 15 through 8 contain the stack limit information. These bits are cleared by system reset, console start, or the RESET instruction. The lower 8 bits are not used. Bit 8 corresponds to a value of 400_8 or 256_{10}.

The contents of the stack limit register (SL) are compared to the stack address to determine if a violation has occurred (although memory references that do not alter memory are always allowed). The least significant bit of the register (bit 8) has a value of 400_8. The determination of the violation zones is as follows:

- Yellow Zone = (SL) + $(340 \text{ through } 377)_8$ execute, then trap.
- Red Zone = (SL) + $(337)_8$ abort, then trap to location 4.

If the stack limit register contents were zero:

- Yellow Zone = 340 through 377
- Red Zone = 000 through 337

INTEGRAL FLOATING POINT INSTRUCTIONS

The PDP-11/60 contains integral floating point hardware which can execute the full complement of PDP-11 floating point instructions. The instructions are noted in Chapter 10.

High-Speed Floating Point Processor Option

The FP11-E floating point processor is an optional, asynchronous, parallel processor capable of doing high-speed arithmetic calculations. The FP11-E is logically contained on four hex modules that fit into the processor backplane.

The FP11-E provides 17 digits of decimal accuracy, does 32-bit single precision or 64-bit double precision arithmetic, and contains six 64-bit accumulators. Additional information about the FP11-E may be found in Chapter 11.

EXTENDED INSTRUCTION SET

The Extended Instruction Set (EIS) allows hardware fixed-point arithmetic and direct implementation of multiply, divide, and multiple shifting. A double-precision 32-bit word can be handled. The Extended Instruction Set executes compatibly with the EIS available on the PDP-11/34A.

PRIORITY INTERRUPT

The PDP-11/60 interrupt system has four priority levels, each of which can handle an almost unlimited number of devices. The priority of the device is a function of the device's electrical location on the UNIBUS—the closer to the processor, the higher its priority on that level.

The priority system makes excellent use of the PDP-11's hardware stacks. When the processor services an interrupt, it first saves important program information on the stack. This information enables the processor to return automatically to the same point in the program and the same conditions, once the current interrupt has been serviced.

The device causing the interrupt(s) provides a direct vector to its own service routine—eliminating the slow and tedious operation of polling all devices to see which one interrupted.

The system also allows interrupts to be enabled or disabled, through software, during program operation. Such masking allows priorities to change dynamically in response to system conditions.

For example, a real-time program can disable data entry terminals whenever critical analog data is being collected. As soon as the scan is complete, the terminals can automatically be enabled and ready to input data.

RELIABILITY, ACCESSABILITY AND MAINTAINABILITY

The significant maintenance feature of the PDP-11/60 is the availability of a wide spectrum of reliabilty and maintenance aids. The spectrum ranges from software (system, diagnostics, error logging, microdiagnostics) to hardware (packaging, parity, error status registers, microbreak). These aids are coordinated via the Reliability, Accessability, and Maintenance Program (RAMP).

RAMP is a DIGITAL corporate program whose purpose is the development of trade-off data for use by DIGITAL's engineering groups in hardware design. Reliability means minimizing failures, and accessability and maintainability mean planning for ease of maintenance and for minimum time spent isolating faults and making repairs.

The design and packaging of the PDP-11/60 has placed great emphasis on RAMP. This means reduced mean time between failures (MTBF) and reduced mean time to repair (MTTR).

Computer System Specifications

Environment

Operating Temperature: 10° C to 40° C
Relative Humidity: 20% to 80%, non-condensing

Mechanical (double-width lowboy)

Height:	50.5 inches (128.3 cm)
Width:	46.5 inches (118.11 cm)
Depth:	30 inches (76.20 cm)
Weight:	379 lbs. (172 Kg.) 11 × 60 with 64Kb MOS memory 409 lbs. (186 Kg.) 11 Y 60 with 64Kb MOS memory

MICROPROGRAMMING

The user microprogramming capability of the PDP-11/60 offers you an opportunity to custom tailor the processor's performance to meet your particular needs precisely. This feature is best utilized by those whose programming requirements include bit manipulation of data or by those who wish to increase the speed of a specific type of data handling, for example, certain scientific calculations. A scientist who is

working with dynamic graphic display data may wish to increase the speed and specificity of the calculation by utilizing one of the microprogamming options, either permanently or temporarily modifying the way the processor implements the software.

DIGITAL offers excellent tutorial user documentation to support the Writable Control Store software option. The programmer who wishes to use the microprogramming options on the PDP-11/60 should have extensive experience in assembly language programming and should be familiar with the RSX-11M operating system.

For the user who wishes to take advantage of the features of microprogramming but who does not wish to do the actual programming, DIGITAL offers the option of consultation with software specialists who are experienced in microprogram development. Specific microprogramming application packaged systems are also available through DIGITAL's network of OEMs and independent software suppliers.

Three microprogramming options are offered with the PDP-11/60. They are:

- **User Control Store** — 1,024 48-bit words of random access memory, used for storing user microprograms and data. The UCS includes the Writable Control Store (WCS) hardware and the WCS software tools: the MICRO-11/60 Assembler, the Microprogram Loader, and the Microdebugging Tool.
- **Extended Control Store** — 1,536 48-bit words of read-only memory for a microprogram. With ROM, there is no loss of microprogram either through inadvertent program modification or through power failure.
- **Diagnostic Control Store** — a hardware aid using microcode analysis of processor operations. It provides a read-only memory that quickly allows isolation and analysis of many central processor faults.

You may use only one microprogramming option at a time, but you may find it useful to have all three options, using whichever is appropriate at any particular time.

The term Writable Control Store (WCS) is the industry-wide generic term used to describe various options which enable the user to control basic processor logic. These options vary widely in their capabilities. Efforts to clarify the functions and capabilities of DIGITAL's control store options have led to each option's being named individually, i.e., UCS, ECS, and DCS. In discussion of the PDP-11/60 microprogramming capabilities, the term WCS refers to the hardware board and to the accompanying software tools, all of which are considered part of the UCS option.

Before explaining further the microprogramming options available with the PDP-11/60, it is helpful to consider some of the basic concepts of microprogramming and some of the variables which can influence your decision about whether or not to utilize microprogramming capabilities.

Microprogramming is a method of controlling the functions of a computer. The essential ideas of microprogramming were first outlined by M.V. Wilkes in 1951 (Wilkes, M.V., "The Best Way to Design an Automatic Calculating Machine," Manchester University Inaugural Conference, 1951, pp16-21). Wilkes proposed a structured hardware design technique to replace prevailing methods of logic design. He observed that a machine-language instruction could be subdivided into a sequence of elementary operations which he called micro-operations, and he compared the execution of the individual steps to the execution of the individual instructions in a program. This concept is the basis of all microprogramming.

For many years, microprogramming remained the province of the hardware designer. As new machines were designed that incorporated advances in theory and technology, the software for the older, slower machines became obsolete. Microprogramming proved to be an attractive solution to this problem of incompatibility. New machines could be provided with additional read-only memory, or control store, which allowed them to emulate earlier computers. The use of emulation, or the interpretive execution of a foreign instruction set, was later extended to provide upward and downward compatibility among a number of computers in a family.

Microprogramming as a tool of the user has evolved slowly. Three things had to happen before its use became feasible. First, technological advances in the field of fast random-access memories were required. The use of read-only memories in a user environment was troublesome and expensive, because correction of programming errors, or bugs, required new memories. Second, user microprogramming required the spread of previously specialized knowledge. When only those engineers actually involved in the design of microprogrammed computers knew what microprogramming involved, users and educators were at a severe disadvantage. In recent years, microprogramming has found a place in computer science curricula, and has been widely used throughout the electronics and scientific industry. The third, and most important, prerequisite for user microprogramming is the inclusion of generality and extendability in the design of a computer. A machine designed solely to implement a given instruction set, with no address space for user control programs,

makes alteration an onerous task. A corollary to this point is that software tools had to be developed, so that the user would not have to work solely with binary patterns.

The USC options and the software microprogramming tools developed for the PDP-11/60 now make user microprogramming a reality.

MICROINSTRUCTIONS

The heart of the 11/60 is a 3-board microprocessor, whose operational unit is the data path. A data path is composed of three types of components:

1. combinational units, such as adders, decoders, or other logical circuits
2. sequential units, such as registers and counters
3. connections, such as wires

The execution of a PDP-11 instruction involves a sequence of transfers from one register in the data path to another; some of these transfers take place directly, others involve an adder or other logical circuit. Each step in this sequence is controlled by a microinstruction; a set of such microinstructions is known as a microprogram.

Microprograms are held in a control store, a block of high-speed memory that can be accessed once per machine cycle. A machine cycle is the basic unit of time within a processor.

PROCESSOR STATE

The **processor state** of a computer is the set of registers and flags that hold the information left upon the completion of one instruction available for use during the execution of the next instruction.

Programmers working at different levels of a machine see different machine states; an applications programmer may never be concerned with machine state at all. A machine-language or macro-level programmer knows the PDP-11 processor state to be defined by the contents of R0 through R7 and the processor status word. Nearly **100 registers** are included in the machine state known to 11/60 microprogrammers. At the nano- or hardware level, even more machine state is seen.

This concept of machine, or processor, state is fundamental to an understanding of microprogrammable processors like the 11/60. State changes at the microprogramming level can affect the macro-level processor state.

A computer is unique, or defined, by the functions it performs and the machine states it enters while performing those functions. Because of this, two machines can be built differently and yet perform identically.

A microprogrammed machine changes state as it reads successive locations in the control store, emulating the state changes that would take place in a completely hard-wired machine. Additionally, the macro-level state, which is a subset of the micro-level machine state, changes as if there were no machine but the macro-level machine.

ARCHITECTURE AND ORGANIZATION

To distinguish the micro-level machine from the macro-level machine, it is useful to differentiate between the terms architecture and organization.

Architecture refers to that set of a computer's features that are visible to the programmer. To a PDP-11 machine-language programmer, this includes the general registers, the instruction set, and the processor status word.

Organization describes a level below architecture, and is concerned with many items that are invisible to the programmer. The term architecture describes *what* facilities are provided, while organization is concerned with *how* those facilities are provided. Occasionally, another term is included in this hierarchy: realization. This term is used to characterize the components used in a particular machine implementation, such as the type of logic and chips used.

The macro-level organization, transparent to the macro-level programmer, defines the micro-level architecture of the machine. The concept is illustrated graphically in Figure 9-10.

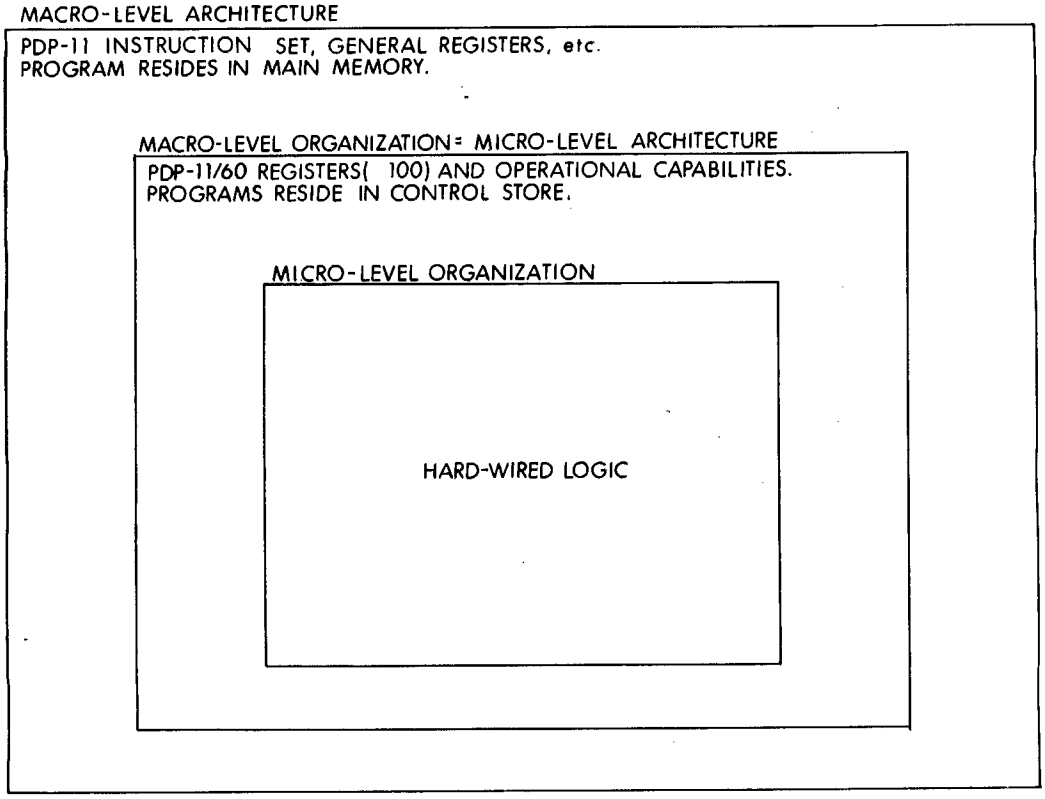


Figure 9-10 Hierarchical Structure of Memories, Architecture, and Organization

The micro-level architecture of the 11/60 is radically different from the standard PDP-11 structure visible to the macro-level programmer. To microprogram the 11/60 successfully, you must familiarize yourself with the details of its micro-level architecture.

The 11/60 can be divided into five logical sections. The microprogrammer's task is to control the flow of data within each of these five basic sections, and sometimes between them.

- the **data-path** section, where most data handling functions are performed
- the **bus control** section, which contains the UNIBUS control logic, the timing generator, and the console interface
- the **KT/cache** section, which contains the memory management logic (KT), the stack limit register (KJ), and 1024 words of high-speed cache memory
- the **processor control** section, which contains the control store for the base machine in the form of a read-only memory, ROM; other control logic, the processor status word (PS) and the floating point status register (FPS)

MICRODEBUGGING TOOL

The MicroDebugging Tool (MDT) is a stand-alone program that provides an efficient tool for debugging 11/60 microprograms. Using MDT you can monitor the execution of your microprogram. You can set breakpoints, examine and change data or instructions in main or micro memory, and alter the control of the program.

MDT is intended for debugging microprograms. Usually, the program to be debugged consists of a small main memory program and a microprogram. The main memory program's purpose is to call the microprogram and, in some cases, provide data for the microprogram to manipulate. MDT takes over the machine and controls all I/O vectors and, consequently, all the interrupts. Therefore, the processing that can be done by the main memory program is limited. It cannot, for example, perform any input or output unless you make special provisions for handling I/O.

Because MDT is used to debug microprograms, it saves the state of the machine.

WCS

WCS enables you to tailor, or bias, the PDP-11 to your particular special purpose needs. Such tailoring can be classified hierarchically as follows:

Class 0	**Instruction Set Extensions** Some functions were considered to be too special-purpose in nature to be included in the original PDP-11 design. These functions, such as block move and decimal arithmetic, can become new PDP-11 instructions. Their definition should conform to 11-instruction format and style.
Class 1	**Application Kernels** Most applications and systems programs have sections which are executed much more frequently than others. A useful rule of thumb is that 10% of the code is executed 90% of the time. Kernels within these critical sections can be microprogrammed for better throughput. Examples include the Fast Fourier Transform, the operating system's memory allocation routine, and Cyclic Redundancy Check calculations.

Class 2	**Emulation** The interpretive execution of an instruction set by software is generally called simulation. When this interpretation is done by hardware it is called emulation. Microprogramming provides a means for inexpensively emulating several different instruction sets on one piece of hardware. The tasks involved in emulation include instruction decode, address calculation, operand fetch, and I/O operation, as well as instruction execution.

Class 0 applications are relatively simple and straightforward uses of microprogramming. Class 1 applications require more intensive study and possibly statistical analysis if they are to improve performance significantly.

The final class of applications, emulation, is best served by a machine specifically designed as a general purpose emulator. The 11/60 was designed to emulate a PDP-11; hence, the organization of its data path is keyed to the 16-bit PDP-11 word and to the other characteristics of a PDP-11 computer system. These factors in large part determine what other computers can be emulated by the 11/60.

WCS MICROPROGRAMMING

To gain real benefit from use of the UCS option, you should invest time and resources in two areas of study before attempting any WCS microprogramming. These two areas are: 1) understanding the 11/60, and 2) analyzing your proposed application.

To microprogram the 11/60 effectively, you must study the internal details of the microprocessor—particularly the data path. Although this is not a difficult task per se, the largely unprotected nature of the microprogramming environment may seem overly complex and unpredictable.

Use of microprogramming will not *always* result in significant performance gains. Applications well suited to microprogramming may improve performance by a factor of 5 to 10; poorly suited ones, not at all. You must understand your application and analyze the execution of its individual instructions. This section is aimed at helping such analysis, but it is in no way a complete treatment of performance analysis.

A machine-language instruction goes through the following processing phases:

I-phase	Instruction fetched from memory and decoded.
O-phase	Operand addresses calculated; operands fetched from memory.
E-phase	Operation executed upon operands.

Each of these phases takes one or more micro-cycles. The total execution time, assuming no overlap of the phase, is the sum of these microcycles. Each phase can be seen as a candidate for elimination or for cycle reduction through microprogramming, with resulting gains in performance.

The following generalizations can be made.

Composite operations save I-cycles.

A block move on the PDP-11 can be programmed as:

```
        MOV COUNT,R0        ;INSTRUCTION 1
        MOV #A,R1           ;2:FIRST SOURCE ADDRS TO R1
        MOV #B,R2           ;3:FIRST DESTINATION ADDRS
                            ;TO R2
LOOP:   MOV (R1)+,(R2)+     ;4:MOVE AND INCREMENT
                            ;BOTH ADDRS
        SOB R0, LOOP        ;5:DECREMENT AND TEST
                            ;COUNTER
```

Combining these operations into one instruction,

```
        BLOCKMOV #A, #B, COUNT
```

elimates I-cycles, with the predominant savings coming from instructions four and five.

Using processor storage saves O-cycles.

The microprogrammer can use internal CPU storage (the hardware registers) for intermediate results. There are a number of hardware registers, in addition to the general registers R0-PC, which can be used by the microprogrammer to avoid memory cycles.

Because there is more parallelism at the micro-level, the inner machine (the microprocessor) is potentially more efficient than the outer machine (the PDP-11). Moverover, the microbranching logic structure of the microprocessor provides a broader decision logic capability which can be exploited, for example, in table search and string-edit operations.

In general, most cycle reductions which result from microprogramming come for the I- and O-phases of instructions.

When analyzing instructions, you must also consider the ratio of the time used by the I- and O-phases to that of the E-phase:

$$\frac{I + O}{E}$$

In vector scalar multiplication, for example, the cycles saved by a composite instruction are a small fraction of the overall execution time.

In summary, you should analyze your application to develop candidate sections for microprogramming, then apply detailed analysis to the instruction execution sequence before coding a microprogram.

INSTRUCTION FORMATS

An instruction, whether at the macro-level or the micro-level, is the basic mechanism that allows a procedure to be invoked. Instructions usually take two source operands and produce a single result. This kind of instruction has five logical functions:

1) and 2 specify the address (location in storage) of the two source operands.
2) Specify the address (location in storage) of the two source operands.
3) Specify the address at which the result of the operation is to be stored.
4) Specify the operation to be performed on the two source operands.
5) Specify the address of the next instruction in the sequence.

These specifications may be explicit or implicit. Implicit specification saves space in the instruction at the expense of additional instructions in the sequence.

There are four common formats for instructions: 3-address, 2-address, single-address, and zero-address (stack-type). These categories indicate how many of the address specifications are explicit in the instruction.

A normal PDP-11 instruction of the form OPR SRC DST uses a 2-address instruction format. The addresses of both the source operands are explicitly specified. The result address is implicitly specified by the address of the destination operand. The next instruction to be executed is implicitly identified by the contents of the program counter.

The 11/60 microword, on the other hand, uses a 4-address instruction format: two source operand addresses, result address, and next in-

struction address are all explicitly identified in each instruction. There is no microprogram counter analogous to the PDP-11 PC.

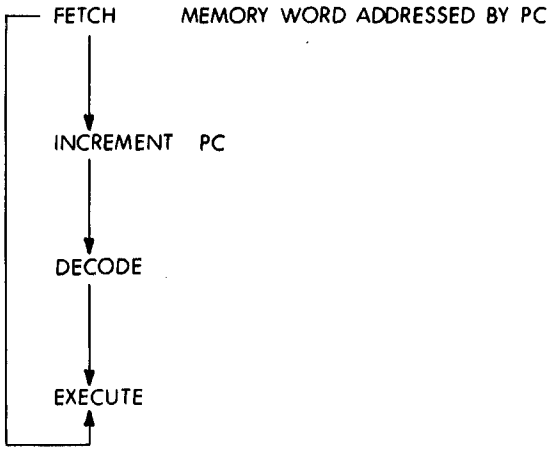
Sequencing and Branching

Because there is no incremental program counter at the microprogramming level in the 11/60, each microinstruction specifies the address of its successor. Therefore, there is no requirement that microinstructions execute sequentially according to their storage address.

Moreover, each microinstruction can also specify a branch condition to be tested before the next microinstruction is fetched. The result of the test can cause a different microinstruction to be fetched.

MICROPROGRAM FLOW

The basic interpretive loop of instruction execution in 11/60 microcode is as follows:



Every microprogram invoked by a PDP-11 opcode follows this pattern. The instruction currently pointed to by the contents of the PC is brought into the processor from main memory and stored in the instruction register, or IR. The PC is incremented by two so that it points at the next location to be accessed. The decode step identifies what instruction is to be executed, and dispatches control to the proper section of microcode. After the operaton is performed, another instruction is fetched.

A slightly more detailed flow structure is shown in Figure 9-11. Note that at the completion of the instruction execution, a test is made for service conditions. If no service condition, such as an interrupt, exists, the next instruction is fetched. If a service condition does exist, control passes to another microprogram which handles the interrupt or other condition. The I-, O-, and E-phases are noted at the left side of the diagram.

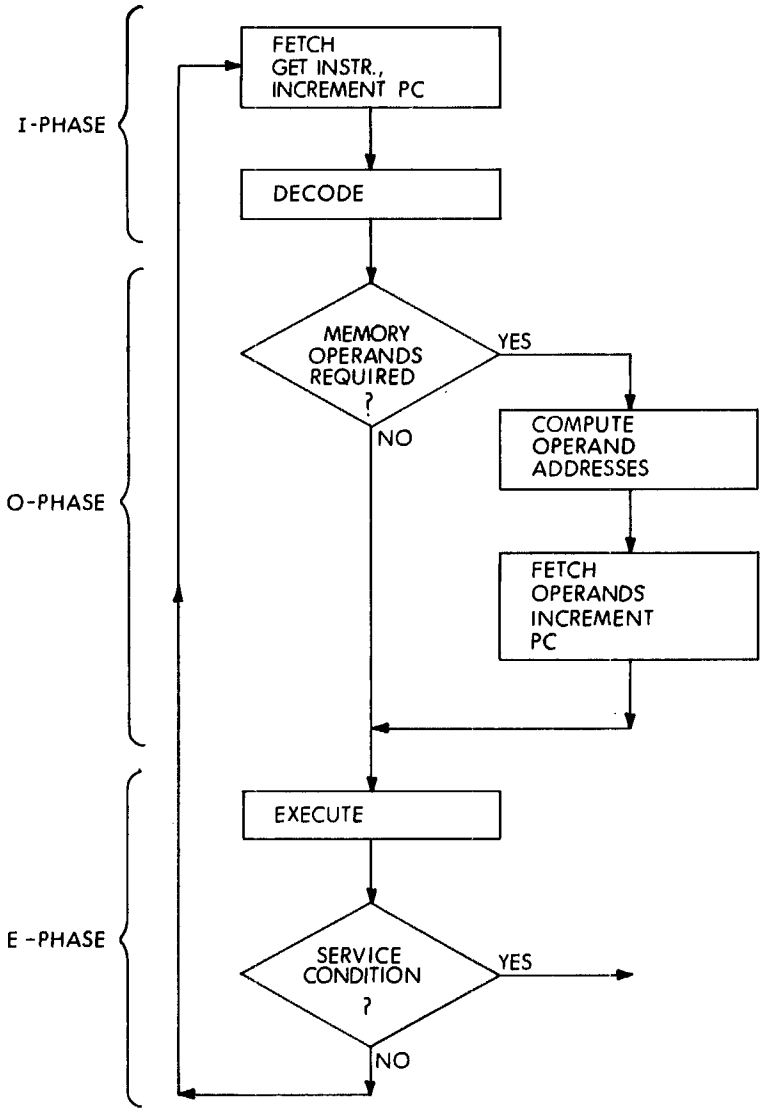


Figure 9-11 Program Flow in the PDP-11/60

WCS

WCS enables you to tailor, or bias, the PDP-11 to your particular special purpose needs. Such tailoring can be classified hierarchically as follows:

Class 0

Instruction Set Extensions
Some functions were considered to be too special-purpose in nature to be included in the original PDP-11 design. These functions, such as block move and decimal arithmetic, can become new PDP-11 instructions. Their definition should conform to 11-instruction format and style.

Class 1

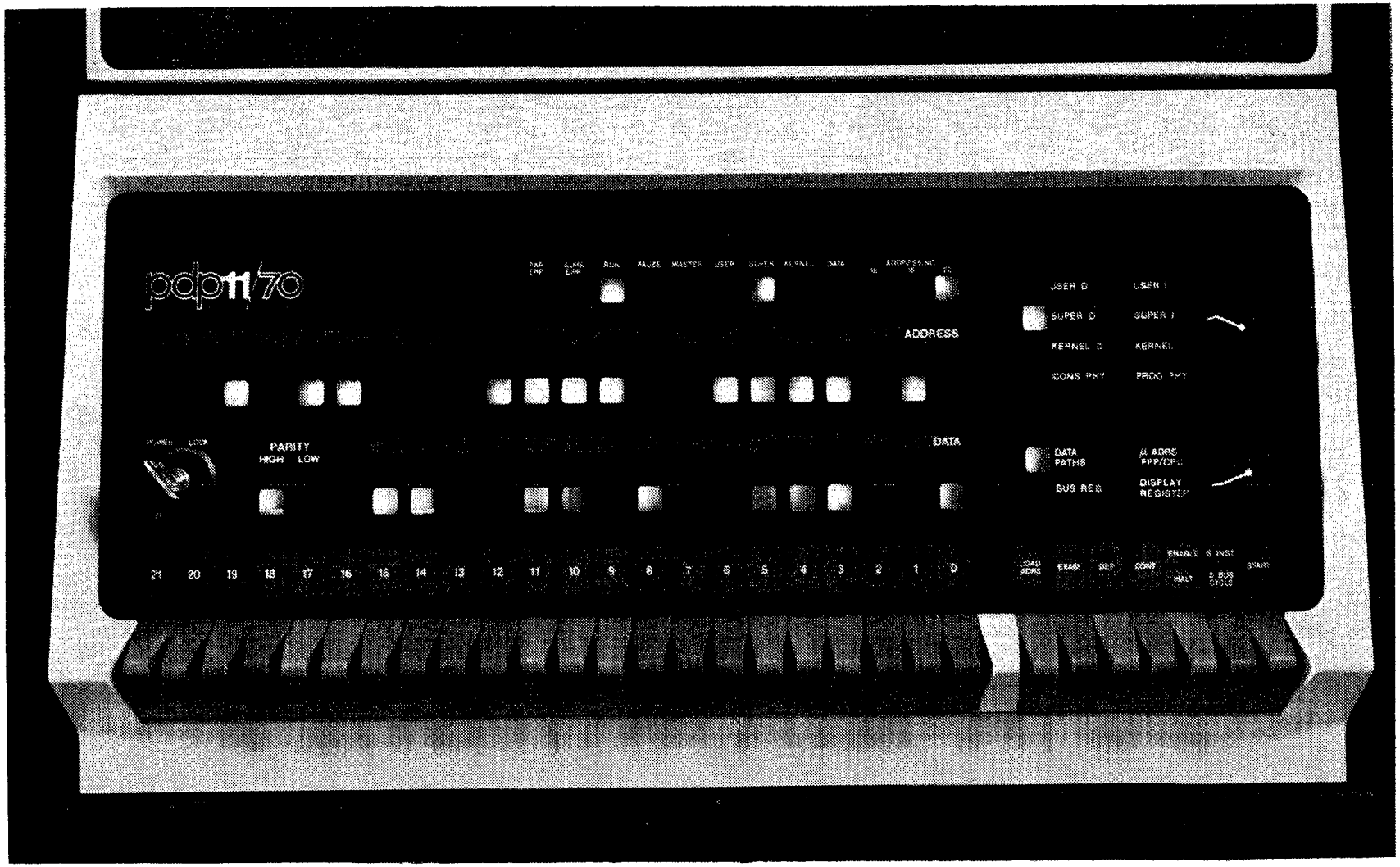
Application Kernels
Most applications and systems programs have sections which are executed much more frequently than others. A useful rule of thumb is that 10% of the code is executed 90% of the time. Kernels within these critical sections can be microprogrammed for better throughput. Examples include the Fast Fourier Transform, and operation system's memory allocation routine, and Cyclic Redundancy Check calculations.

Class 2

Emulation
The interpretive execution of an instruction set by software is generally called simulation. When this interpretation is done by hardware it is called emulation. Microprogramming provides a means for inexpensively emulating several different instruction sets on one piece of hardware. The tasks involved in emulation include instruction decode, address calculation, operand fetch, and I/O operation, as well as instruction execution.

Class 0 applications are relatively simple and straightforward uses of microprogramming. Class 1 applications require more intensive study and possibly statistical analysis if they are to improve performance significantly.

The final class of applications, emulation, is best served by a machine specifically designed as a general purpose emulator. The 11/60 was designed to emulate a PDP-11; hence, the organization of its data path is keyed to the 16-bit PDP-11 word and to the other characteristics of a PDP-11 computer system. These factors in large part determine what other computers can be emulated by the 11/60.


pdp11/70
ADDRESS
DATA
PARITY
HIGH LOW
USER D
USER I
SUPER D
SUPER I
KERNEL D
KERNEL I
CONS PHY
PROG PHY
DATA PATHS
BUS REG
DISPLAY REGISTER
21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

CHAPTER 10
PDP-11/70

The PDP-11/70 is the most powerful computer in the PDP-11 family. It is designed to operate in large, sophisticated, high-performance systems. It can be used as a powerful computational tool for high-speed, real-time applications and for large multi-user, multi-tasking, time-shared applications requiring large amounts of addressable memory space. It is the systems-level PDP-11 that applies the power of 32-bit hardware architecture to demanding, multifunction computing requirements.

FEATURES

The PDP-11/70 contains, as an integral part of the central processor unit, the following hardware features and expansion capabilities:

- Cache memory organization to provide very fast program execution speed and high system throughput.
- Memory management for relocation and protection in multi-user, multi-task environments.
- Ability to access up to 4 million bytes of main memory (1 byte = 8 bits).
- Optional high-speed, mass storage controllers as an integral part of the CPU, to provide dedicated paths to high performance storage devices.
- Optional Floating Point processor with advanced features, operating with 32-bit and 64-bit numbers.

SYSTEM ARCHITECTURE

The PDP-11/70 is a medium scale general purpose computer using an enhanced, upwardly-compatible version of the basic PDP-11 architecture. A block diagram of the computer is shown in Figure 10-1.

The central processor performs all arithmetic and logical operations required in the system. Memory Management is standard with the basic computer, allowing expanded memory addressing, relocation, and protection. Also standard is a UNIBUS Map which translates UNIBUS addresses to physical memory address. The cache contains 2,048 bytes of fast, bipolar memory that buffers the data from main (core or MOS) memory.

Also within the CPU assembly are pre-wired areas for a floating point processor, and up to four high-speed I/O controllers.

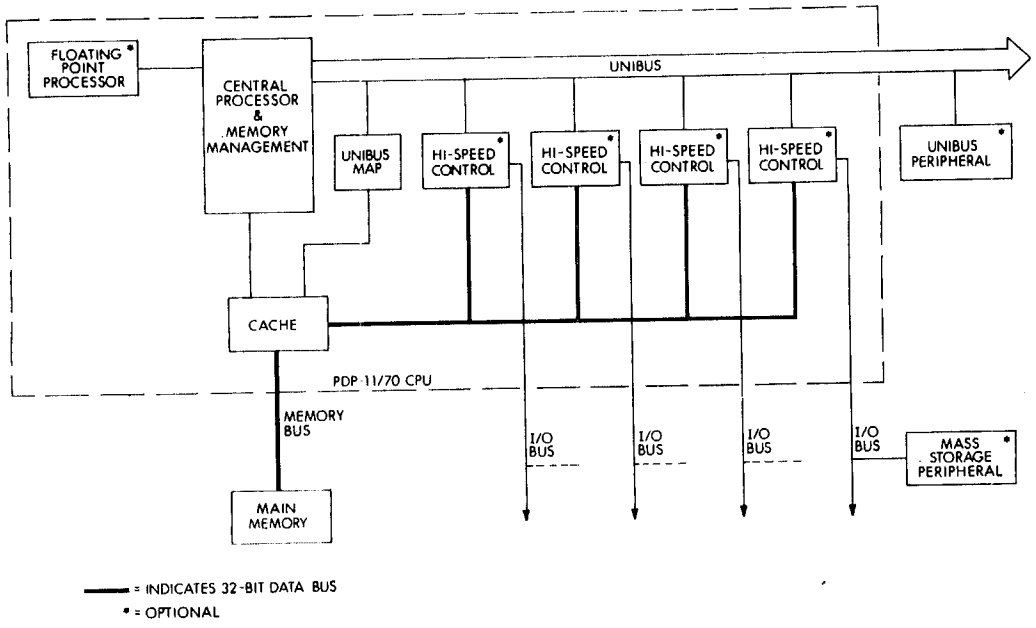


Figure 10-1 PDP-11/70 Block Diagram

The PDP-11/70 system has an expanded internal implementation of the PDP-11 architecture for greatly improved system throughput. All the memory is on its own high data rate bus. The internal high-speed I/O controllers for mass storage devices have direct connections through the cache to memory for transferring data (using the cache only for timing purposes). The processor has a direct connection to the cache memory system for very high-speed memory access.

The UNIBUS remains the primary control path in the 11/70 system. It is conceptually identical with previous PDP-11 systems; the memory in the system still appears to be on the UNIBUS to all UNIBUS devices. Control and status information to and from the high speed I/O controllers is transferred over the UNIBUS. This expanded internal implementation of the PDP-11 architecture has no effect on PDP-11/70 programming.

CENTRAL PROCESSOR

The PDP-11/70 CPU performs all arithmetic and logical operations required in the system. It also acts as the arbitration unit for UNIBUS control by regulating bus requests and transferring control of the bus to the requesting device with the highest priority.

The central processor contains arithmetic and control logic for a wide range of operations. These include high-speed fixed point arithmetic with hardware multiply and divide, extensive test and branch operations, and other control operations. It also provides room for the addi-

tion of the high-speed Floating Point Processor, and high-speed controllers.

The machine operates in three modes: Kernel, Supervisor, and User. When the machine is in Kernel mode, a program has complete control of the machine. When the machine is in any other mode, the processor is inhibited from executing certain instructions and can be denied direct access to the peripherals on the system. This hardware feature can be used to provide complete executive protection in a multi-programming environment.

The central processor contains 16 general registers which can be used as accumulators, index registers, or as stack pointers. Stacks are extremely useful for nesting programs, creating re-entrant coding, and as temporary storage where a Last-In/First-Out structure is desirable. One of the general registers is used as the PDP-11/70's program counter. Three others are used as Processor Stack Pointers, one for each operational mode.

The CPU performs all computation and logic operations in a parallel binary mode through step by step execution of individual instructions.

General Registers

The general registers can be used for many purposes: the uses vary with requirements. The general registers can be used as accumulators, index registers, autoincrement registers, autodecrement registers, or as stack pointers for temporary storage of data. Chapter 3 on Addressing describes these uses of the general registers in more detail. Arithmetic operations can be from one general register to another, from one memory or device register to another, or between a memory or a device register and a general register.

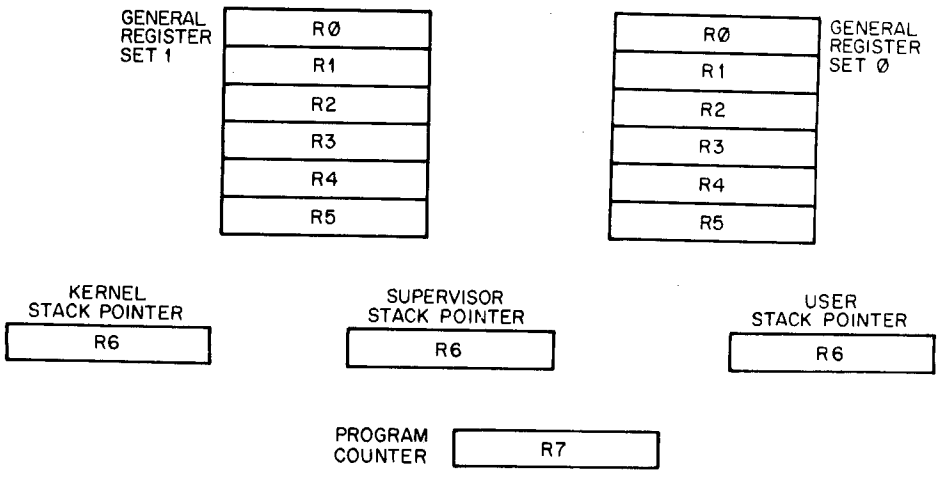


Figure 10-2 The General Registers

R7 is used as the machine's program counter (PC) and contains the address of the next instruction to be executed. It is a general register normally used only for addressing purposes and not as an accumulator for arithmetic operations.

The R6 register is normally used as the Processor Stack Pointer (SP) indicating the last entry in the appropriate stack, a common temporary storage area with Last-In/First-Out characteristics. (For information on the programming use of stacks, please refer to Chapter 5.) The three stacks are called the Kernel Stack, the Supervisor Stack and the User Stack. When the central processor is operating in Kernel mode, it uses the Kernel Stack; in Supervisor mode, the Supervisor Stack; and in User mode, the User Stack. When an interrupt or trap occurs, the PDP-11/70 automatically saves its current status on the Processor Stack selected by the service routine. This stack-based architecture facilitates re-entrant programming.

The remaining 12 registers are divided into two sets of unrestricted registers, R0-R5. The current register set in operation is determined by the Processor Status Word.

The two sets of registers can be used to increase the speed of real-time data handling or facilitate multi-programming. The six registers in General Register Set 0 could each be used as an accumulator and/or index register for a real-time task or device, or as general registers for a Kernel or Supervisor mode program. General Register Set 1 could be used by the remaining programs or User mode programs. The Supervisor can therefore protect its general registers and stacks from User programs, or other parts of the Supervisor.

Processor Status Word

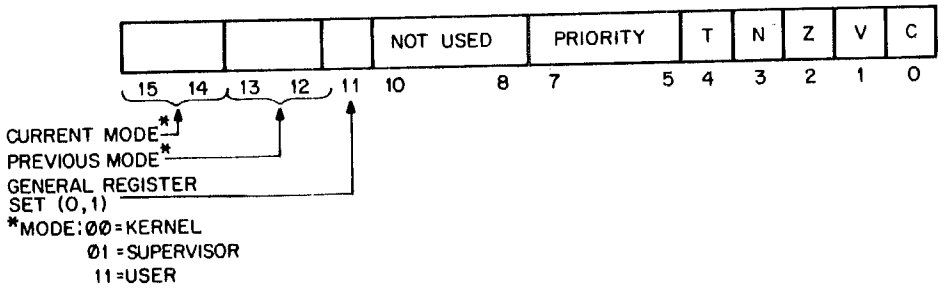


Figure 10-3 Processor Status Word

The Processor Status Word, located at location 17 777 776, contains information on the current status of the PDP-11/70. This information

includes the register set currently in use; current processor priority; current and previous operational modes; the condition codes describing the results of the last instruction; and an indicator for detecting the execution of an instruction to be trapped during program debugging.

Modes — Mode information includes the present mode, either User, Supervisor or Kernel (bits 15, 14); the mode the machine was in prior to the last interrupt or trap (bits 13, 12); and which register set (General Register Set 0 or 1) is currently being used (bit 11).

The three modes permit a fully protected environment for a multiprogramming system by providing the user with three distinct sets of Processor Stacks and Memory Management Registers for memory mapping. In all modes except Kernel, a program is inhibited from executing a "HALT" instruction and the processor will trap through location 4 if an attempt is made to execute this instruction. Furthermore, in other than Kernel mode, the processor will ignore the "RESET" and "SPL" (Set Priority Level) instructions. In Kernel mode, the processor will execute all instructions.

A program operating in Kernel mode can map users' programs anywhere in main memory and thus explicitly protect key areas (including the device registers and the Processor Status Word) from the User operating environment.

Processor Priority — The central processor operates at any of eight levels of priority, 0-7. When the CPU is operating at level 7, an external device cannot interrupt it with a request for service. The central processor must be operating at a lower priority than the priority of the external device's request for the interruption to take place. The current priority is maintained in the Processor Status Word (bits 5-7). The eight processor levels provide an effective interrupt mask, which can be dynamically altered through use of the Set Priority Level (SPL) instruction (described in Chapter 4). The SPL instruction can only be used in Kernel mode. This instruction allows a Kernel mode program to alter the central processor's priority without affecting the rest of the Processor Status Word.

Condition Codes — The condition codes contain information on the result of the last CPU operation. They include: a carry bit (C), set by the previous operation if the operation caused a carry out of its most significant bit; a negative bit (N), set if the result of the previous operation was negative; a zero bit (Z), set if the result of the previous operation was zero; and an overflow bit (V), set if the result of the previous operation resulted in arithmetic overflow.

Trap — The trap bit (T) can be set or cleared under program control. When set, the processor trap will occur through location 14 on

completion of instruction execution and a new Processor Status Word will be loaded. This bit is especially useful for debugging programs, as it provides an efficient method of installing breakpoints.

Interrupts and trap instructions both automatically cause the previous Processor Status Word and Program Counter to be saved and replaced by new values corresponding to those required by the routine servicing the interrupt or trap. The user can thus cause the central processor to automatically switch modes (context switching), switch registers sets, alter the CPU's priority, or disable the Trap Bit whenever a trap or interrupt occurs.

Stack Limit Register

All PDP-11s have a Stack Overflow Boundary at location 400_8. The Kernel Stack Boundary, in the PDP-11/70, is a variable boundary set through the Stack Limit Register found in location 17 777 774.

Once the Kernel stack exceeds its boundary, the processor will complete the current instruction and then trap to location 4 (Yellow, or Warning Stack Violation). If, for some reason, the program persists beyond the 16-word limit, the processor will abort the offending instruction, set the stack pointer (R6) to 4 and trap to location 4 (Red, or Fatal Stack Violation). A description of these traps is contained in Appendix A.

MEMORY

Memory Organization

A memory can be viewed as a series of locations, with a number (address) assigned to each location. Thus a 16,384-byte PDP-11 memory could be shown as in Figure 10-4.

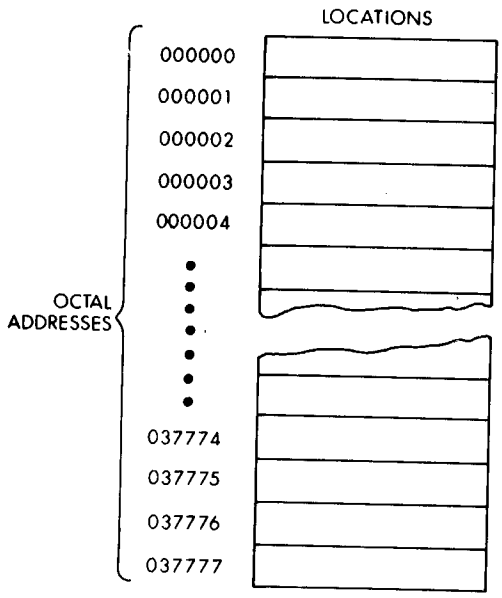


Figure 10-4 Memory Address

Because PDP-11 memories are designed to accommodate both 16-bit words and 8-bit bytes, the total number of addresses does not correspond to the number of words. An 8K-word memory can contain 16K bytes and consist of 037777 octal locations. Words always start at even-numbered locations.

A PDP-11 word is divided into a high byte and low byte as shown in Figure 10-5.

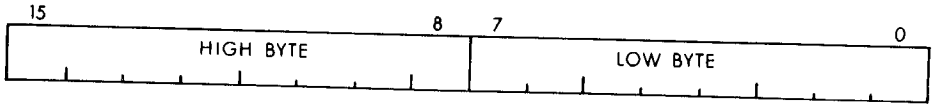


Figure 10-5 High and Low Byte

Low bytes are stored at even-numbered memory locations and high bytes at odd-numbered memory locations. Thus it is convenient to view the PDP-11 memory as shown in Figure 10-5.

Certain memory locations have been reserved by the system for interrupt and trap handling, processor stacks, general registers, and peripheral device registers. Addresses from 0 to 370_8 are always reserved, and those to 777_8 are reserved on large system configurations for traps and interrupt handling.

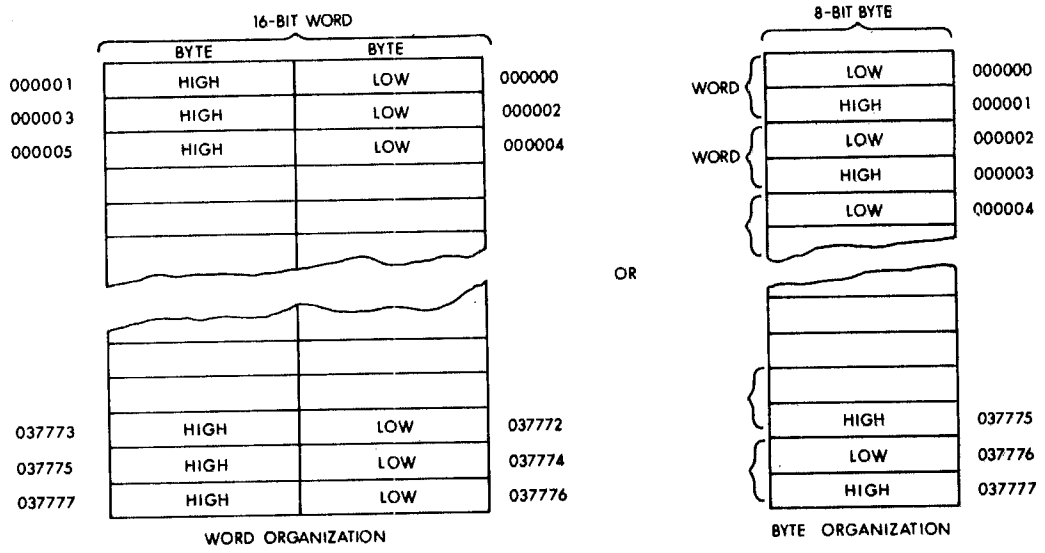


Figure 10-6 Word and Byte Addresses

MOS Memory with ECC

ECC (error correcting code) is a technique for checking the contents of memory to detect errors and correct them before sending them to the processor. The process of checking is accomplished by combining the bits in a number of unique ways, so that parity, or **syndrome**, bits are generated for each unique combination and stored along with the data bits in the same word as the data. The memory word length is extended to store these unique bits. When memory is read, the data word is again checked, syndrome bits are regenerated and compared with the syndrome bits stored with the word. If they match, the word is sent on to the processor. If they do not match, an error exists and the mismatch of the syndrome bits determines which data bit is in error. The bit in error is then corrected and sent on to the processor. The error correcting code which is employed in MOS memory will detect and correct single bit errors in a word, as well as detect double bit errors in a word. Where a double bit error is detected, the processor is notified, as happens with a parity error.

ECC provides the maximum system benefits when used in a storage system which fails in a random single bit mode rather than in blocks or large segments. Single bit error (or failure) is the predominant failure mode for MOS memory.

ECC memory provides fault tolerance with the result that multiple single-bit failures can be present in a memory system without measurable degradation in either performance or reliability.

MOS memory by its nature is volatile. It cannot retain data without proper DC voltages being applied. DIGITAL MOS memories, therefore, have battery backup (BBU) power provisions, standard on the PDP-11/70, so that data may be retained during short-term loss of AC line power.

Generally, the incidence of AC line power loss varies inversely with the severity of loss. That is, there are an extremely small number of complete failures of AC power, and in relatively larger number of short-term failures or drops in voltage. No economically feasible battery backup unit can store sufficient energy to accomodate a complete AC power failure for more than several minutes.

Battery backup units are not intended to preserve data overnight or over weekends, but rather to overcome infrequent, very short-term failures of AC power.

Parity

Parity is used extensively in the PDP-11/70 to ensure the integrity of information. All memory has byte parity. Parity for both data and addresses is generated on transfers to memory and is checked on all transfers from memory. Registers are provided within the CPU to provide information on the location of parity errors, types of errors, and other relevant information so that software can respond to the situation, take corrective action, and log the occurrence of errors.

MEMORY SYSTEM

Address Space

The PDP-11/70 uses 22 bits for addressing physical memory. This represents a total of 2^{22} (over 4 million) byte locations.

Of the over 4 million byte locations possible with the 22-bit address, the top 256K are used to reference the UNIBUS rather than physical memory. Maximum main memory is therefore $2^{22} - 2^{18}$, or a total of 3,932,160 bytes.

Three separate address spaces are used with the PDP-11/70. Main memory uses 22-bit addresses, the UNIBUS uses an 18-bit address, and the computer program uses a 16-bit virtual address. The information is summarized below:

		Bytes
16 bits	program virtual space	2^{16} = 64K
18 bits	UNIBUS space	2^{18} = 256K
22 bits	physical memory space	4 million

Memory Management

The Memory Management hardware is standard with the PDP-11/70 computer. It is a hardware relocation and protection facility that can convert the 16-bit program virtual addresses to 22-bit addresses. The unit may be enabled and disabled under program control. There is no increase in access time when the Memory Management unit is enabled.

UNIBUS Map

The UNIBUS Map responds as memory on the UNIBUS. It is the harware relocation facility for converting the 18-bit UNIBUS addresses to 22-bit addresses. The relocation mapping may be enabled or disabled under program control.

Cache

The cache memory is a very high-speed memory that buffers data between the processor and main memory. The cache is completely transparent to all programs; programs are treated as if there were one continuous bank of memory.

Whenever a request is made to fetch data from memory, the cache circuitry checks to see if that data is already in the cache. If it is, it is fetched from there and no main memory read is required. If the data is not already in cache memory, four bytes are fetched from main memory and stored in the cache, with the requested word or byte being passed directly to the CPU.

When a request is made to write data into memory, it is written both to the cache and to main memory, assuring that both stores are always updated immediately.

The key to the effectiveness of PDP-11/70's cache memory is its size. Because it holds 2,048 bytes at any given point in time, the PDP-11/70 cache already contains the next needed data a very high percentage of the time.

A detailed description of cache memory and the other parts of memory are contained later in the chapter.

OTHER CPU EQUIPMENT

Floating Point Processor

The PDP-11/70 Floating Point Processor fits integrally into the central processor. It provides a supplemental instruction set for performing single and double precision floating point arithmetic operations and floating-integer conversion in parallel with the CPU. The floating point processor provides speed and accuracy in arithmetic computations. It

provides 7 decimal digit accuracy in single word calculations and 17 decimal digit accuracy in double word calculations.

Floating point calculations take place in the FPP's six 64-bit accumulators. The 46 floating point instructions include hardware conversion from single or double precision floating point to single or double precision integers. There is a detailed description in Chapter 11.

High Speed Mass Storage

The PDP-11/70 busing structure is optimized for high-speed device transfers. Up to four such devices can be plugged directly into the processor with a dedicated 32-bit bus feeding through to the main memory. Present DIGITAL devices that use this bus structure are the RP04/05/06, RS03/04, TU16, TE16, and RM03. Refer to the Specifications Section for device specifications.

SYSTEM INTERACTION

High-speed Non-Processor Request (NPR) devices use separate dedicated buses to the individual high-speed I/O controllers. From the controllers there is a single 4-byte wide bus that interfaces to the cache. The order of priorities in the system is:

1. UNIBUS (via UNIBUS Map)
2. High-speed I/O controllers (A through D)
3. CPU

Control information and lower speed data transfers are carried out through the UNIBUS.

A device will request the UNIBUS for one of two purposes:

1. To make an NPR transfer of data (direct data transfers such as DMA), or
2. To interrupt program execution and force the processor to branch to a service routine.

There are two sources of interrupts, hardware and software.

Hardware Interrupt Requests

A hardware interrupt occurs when a device wishes to indicate to the program, or the central processor, that a condition has occurred (such as transfer completed, end of tape, etc.). The interrupt can occur on any one of the four Bus Request levels and the processor responds to the interrupt through a service routine.

Program Interrupt Requests

Hardware interrupt servicing is often a two-level process. The first level is directly associated with the device's hardware interrupt and

consists of retrieving the data. The second is a software task that manipulates the raw information. The second process can be run at a lower priority than the first, because the PDP-11/70 provides the user with the means of scheduling his software servicing through seven levels of Program Interrupt Requests. The Program Interrupt Request Register is located at address 17 777 772. An interrupt is generated by the programmer setting a bit on the high-order byte of this register.

Priority Structure on the UNIBUS

When a device capable of becoming bus master requests use of the bus, handling of the request depends on the hierarchical position of that device in the priority structure.

The relative priority of the request is determined by the processor's priority level and the level at which the request is made.

- The processor's priority is set under program control to one of eight levels using bits 7-5 in the Processor Status Word. Bus requests are inhibited on the same or lower levels.
- Bus requests from external devices can be made on any one of the five request lines. An NPR has the highest priority, and its request is granted between bus cycles of an instruction execution. Bus Request 7 (BR 7) is the next highest priority and Bus Request 4 (BR 4) is the lowest. The four lower priority level requests (BR 7—BR 4) are granted by the processor between instructions providing they occur on higher levels than the processor's. Therefore, an interrupt may only occur on a Bus Request Level and not on a Non-Processor Request level.
- Any number of devices can be connected to a specific BR or NPR line.
- If two devices with the same priority request the bus, the device physically closest to the processor on the UNIBUS has the higher priority.
- Program Interrupt Requests can be made on any one of seven levels (PIR 7—PIR 1). Requests are granted by the processor between instructions providing that they occur on higher levels than the processor's.
- Program Interrupt Requests take precedence over equivalent level Bus Requests.

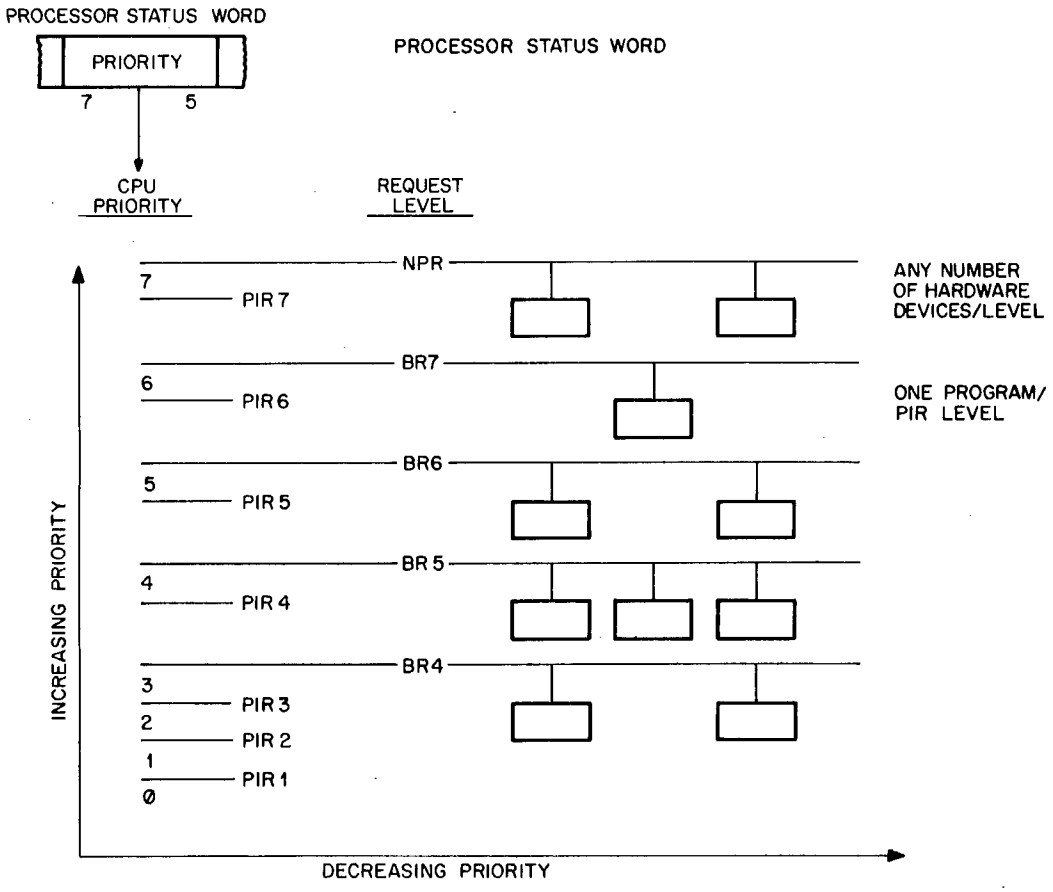


Figure 10-7 UNIBUS Priority Structure

Non-Processor Data Transfers

Direct memory or direct data tranfers can be accomplished between any two peripherals without processor supervision. These non-processor transfers, called NPR level data transfers, are usually made for Direct Memory Access (memory to/from mass storage) or direct device transfers (disk refreshing a CRT display).

An NPR device provides extremely fast access to the UNIBUS and can transfer data at high rates once it gains control of the bus. The state of the processor is not affected by this type of transfer, and, therefore, the processor can relinquish bus control while an instruction is still in progress. The bus can be released at the end of any bus cycle, except during a read-modify-write cycle sequence. (This occurs, for example, in destructive read-out devices such as core memory for certain instructions.) In the PDP-11/70, an NPR device can gain control in 3.5 microseconds or less (depending on the number of devices on the UNIBUS), and can tranfer 16-bit words to memory at the same speed as the effective cycle time of the memory being addressed.

Using the Interrupts

Devices that gain bus control with one of the Bus Request Lines (BR 7—BR 4) can take full advantage of the central processor by requesting an interrupt. In this way, the entire instruction set is available for manipulating data and status registers.

When a service routine is to be run, the current task being performed by the central processor in interrupted, and the device service routine is initiated. Once the request has been satisfied, the processor returns to its former task. Interrupts may also be used to schedule program execution by using the Program Interrupt Request.

Interrupt Procedure

Interrupt handling is automatic in the PDP-11/70. No device polling is required to determine which service routine to execute. The operations required to service an interrupt are as follows:

1. Processor relinquishes control of the bus, priorities permitting.
2. When a master gains control, it sends the processor an interrupt command and a unique memory address which contains the address of the device's service routine in Kernel virtual address space, called the interrupt vector address. Immediately following this pointer address is a word (located at vector address + 2) which is to be used as a new Processor Status Word.
3. The processor stores the current Processor Status Word (PS) and the current Program Counter (PC) into CPU temporary registers.
4. The new PC and PS (the interrupt vector) are taken from the specified address. The old PS and PC are then pushed onto the current stack as indicated by bits 15, 14 of the new PS and the previous mode in effect is stored in bits 13, 12 of the new PS. The service routine is then initiated.

These operations are performed in approximately 2.5 μsec from the time the central processor receives the interrupt command until the time it starts executing the first instruction of the service routine. This time interval assumes no NPR transfer occurred.

5. The device service routine can cause the processor to resume the interrupted process by executing the Return from Interrupt (RTI or RTT) instruction, described in Chapter 4, which pops the two top words from the current processor stack and uses them to load the PC and PS registers.

This instruction requires approximately 1.5 μsec providing there is no NPR request.

A device routine can be interrupted by a higher priority bus request any time after the new PC and PS have been loaded. If such an

interrupt occurs, the PC and PS of the service routine are automatically stored in the temporary registers and then pushed onto the new current stack, and the new device routine is initiated.

Interrupt Servicing

Every hardware device capable of interrupting the processor has a unique pair of locations reserved for its interrupt vector. The first word contains the location of the device's service routine, and the second, the Processor Status Word that is to be used by the service routine. Through proper use of the PS, the programmer can switch the operational mode of the processor, alter the General Register Set in use (context switching) and modify the processor's priority level to mask out lower level interrupts.

There is one interrupt vector for the Program Interrupt Request. It will generally be necessary in a multi-processing environment to determine which program generated the PIR and where it is located in memory.

Processor Traps

There is a series of errors and programming conditions which will cause the central processor to trap to a set of fixed locations. These include Power Failure, Odd Addressing Errors, Stack Errors, Timeout Errors, Non-Existent Memory Errors, Memory Parity Errors, Memory Management Violations, Floating Point Processor Exception Traps, use of Reserved Instructions, use of the T Bit in the Processor Status Word, and use of the IOT, EMT and TRAP instructions.

Input/Output Devices

The LA36 DECwriter is the standard PDP-11 system terminal. It has several advantages over standard electromechanical typewriter terminals, including higher speed, fewer mechanical parts and very quiet operation. I/O capabilities can be increased with high-speed paper tape readers-punches, line printers, card readers or alphanumeric display terminals.

PDP-11 I/O devices include:

DECwriter teleprinter, LA36, LA38, LA120
DECterminal alphanumeric display VT50, VT52, VT100
High-speed line printers LP11, LS11
High-speed paper tape reader-punch PC11
Card readers CR11, CD11
Synchronous and asynchronous communication interfaces

Storage Devices

Storage devices range from convenient, small reel magnetic tape units to mass storage magnetic tapes and disk memories. A large number of storage devices, in any combination, may be connected to a PDP-11 system. TU56 DECtapes are ideal for applications with modest storage requirements. Each DECtape provides storage for 144K 16-bit words. For applications which require handling large volumes of data, DIGITAL offers the industry-compatible TU16 magtape.

Disk storage devices include fixed-head disk units and moving-head removable cartridge and disk pack units. PDP-11 storage devices include:

DECtape: TU56, TU58
Magtape: TU16, TE16, TS03, TS11, TU45, TU77
512K byte dual floppy disk: RX01
1 M byte dual floppy disk: RX02
512K byte fixed head disk: RS03
1,024K byte fixed head disk: RS04
2.4M byte moving head cartridge disk: RK05
5.2M byte moving head cartridge disk: RL01
14M byte moving head disk pack: RK06
28M byte moving head disk pack: RK07
67M byte moving head disk pack: RM03
88M byte moving head disk pack: RP04/05
176M byte moving head disk pack: RP06

SPECIFICATIONS

PACKAGING

A basic PDP-11/70 consists of two H960 cabinets (see Figure 10-7), or a double width corporate cabinet (see Figure 10-8):

H960 Cabinet

1. A CPU cabinet which contains the processor, CPU related equipment and interface equipment.
2. A Memory Cabinet which contains the first 128K bytes of parity core or MOS memory (with expansion capability to 2,048K bytes within the cabinet. Another H960 memory cabinet located next to it can house an additional 2,048K bytes of core or MOS memory).

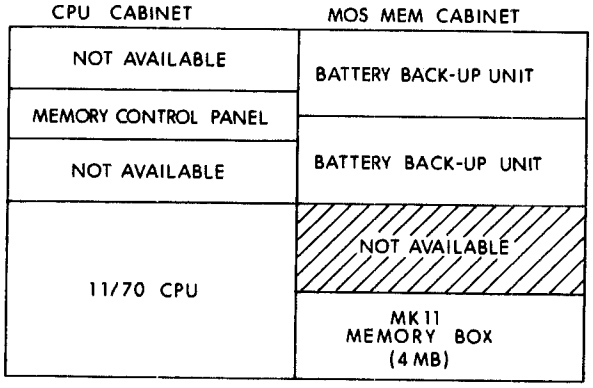


Figure 10-7 11/70 Equipment in H960 Cabinets

Corporate Cabinet*

1. A CPU cabinet which contains the processor, CPU-related equipment, interface equipment, and the first 128K bytes of parity core or MOS memory (with expansion capability to 2,048K bytes within the cabinet).
2. Another memory corporate cabinet located next to it can house an additional 2,048K bytes of core or MOS memory.

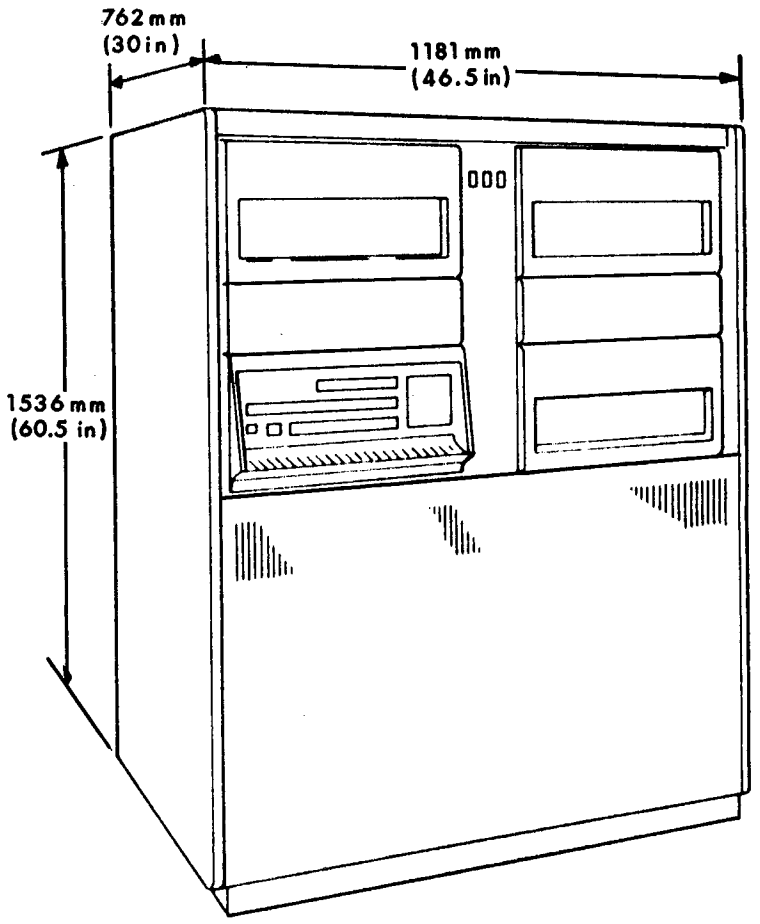


Figure 10-8 11/70 Equipment in Corporate Cabinet

An LA36 DECwriter II console terminal is included with the 11/70 system. There are prewired areas within the mounting assemblies for expansion with optional equipment.

* NOTE: By using the 256K byte memory arrays, the entire 11/70 main memory is contained in a single BA11-K Box.

COMPONENT PARTS

The basic PDP-11/70 system has:

Included Equipment
11/70 CPU
Memory Management
Bootstrap loader
DECwriter (LA36)
Terminal interface (DL11-W) with integral line clock
2K byte cache memory

128K byte parity core or MOS memory
CPU cabinet
Memory cabinet

Prewired Expansion Space for Optional Equipment
Floating Point Processor
Four High-speed I/O controllers
Four SPC slots for peripherals
128K byte parity core or MOS (within 1st memory expansion frame)

OTHER SPECIFICATIONS

AC Power
120 Vac ± 10%, 47 to 63Hz, 3 phase power
240 Vac ± 10%, 47 to 63Hz, 3 phase power

	120 Vac	240 V ac
Basic CPU cabinet (maximum current on each of 2 phases)	15A	7.5A
Memory, each BA11-K Box (maximum current on 1 phase)	12A	6A

Size
Each H960 cabinet is 72″ high × 21″ wide × 30″ deep.
Each double width corporate cabinet is 60.5″ high × 46.5″ wide × 30″ deep.

Weight (H960 cabinet)

CPU cabinet:	500 lbs. (227 Kg.)
Memory cabinet:	250 lbs. (including 1st 512K bytes) (114 Kg.)
Memory expansion frame:	150 lbs (each additional 512K bytes) (67.5 Kg.)

Operating Environment

Temperature:	15°C to 32°C (59°F to 90°F)
Humidity:	20% to 80% with max wet bulb 28°C (82°F) and minimum dew point 2°C (36°F)
Altitude:	to 2.4 km (8000 ft.)

Non-Operating Environment

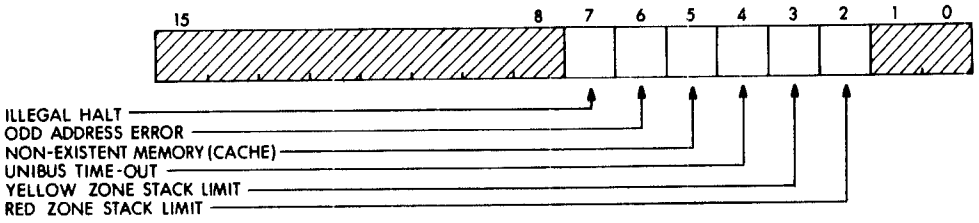
Temperature:	−40°C to 66°C (−40°F to 151°F)
Humidity:	to 95%
Altitude:	to 9.1 km (30,000 ft)

PROCESSOR CONTROL

REGISTERS

The following five CPU registers are not accessible from the UNIBUS. They are accessed by program or console control.

CPU Error Register 17 777 766



This register identifies the source of the abort or trap that used the vector at location 4.

Bit: 7 **Name:** Illegal Halt
Function: Set when trying to execute a HALT instruction when the CPU is in User or Supervisor mode (not kernel).

Bit: 6 **Name:** Odd Address Error
Function: Set when a program attempts to do a word reference to an odd address.

Bit: 5 **Name:** Non-Existent Memory
Function: Set when the CPU attempts to read a word from a location higher than indicated by the System Size register. This does not include UNIBUS addresses.

Bit: 4 **Name:** UNIBUS Timeout
Function: Set when there is no response on the UNIBUS within approximately 10 μsec.

Bit: 3 **Name:** Yellow Zone Stack Limit
Function: Set when a yellow zone trap occurs.

Bit: 2 **Name:** Red Zone Stack Limit
Function: Set when a red zone trap occurs.

Lower Size Register 17 177 760

This read-only register specifies the memory size of the system. It is defined to indicate the last addressable block of 32 words in memory (bit 0 is equivalent to bit 6 of the Physical Address).

Upper Size Register 17 777 762

This register is an extension of the system size, which is reserved for future use. It is read-only and its contents are always read as zero.

System I/D Register 17 777 764
This read-only register contains information uniquely identifying each system.

Microprogram Break Register 17 77 770
This register is used for maintenance purposes only. It is used with maintenance equipment to provide synchronization and testing facilities.

Processor Status Word 17 777 776
The Processor Status Word contains information on the current status of the CPU. This information includes the register set currently in use; current processor priority; current and previous operational modes; the condition codes describing the results of the last instruction; and an indicator for detecting the execution of an instruction to be trapped during program debugging.

PROCESSOR TRAPS
There are a series of errors and programming conditions which will cause the central processor to trap to a set of fixed locations. These include Power Failure, Odd Addressing Errors, Stack Errors, Timeout Errors, Non-Existent Memory References, Memory Parity Errors, Memory Management Violations, Floating Point Processor Exception Traps, use of Reserved Instructions, use of the T bit in the Processor Status Word, and use of the IOT, EMT, and TRAP instructions.

Power Failure
Whenever AC power drops below 95 volts for 110V power (190 volts for 220V) Or outside a limit of 47 to 63 Hz, as measured by DC power, the power fail sequence is initiated. The central processor automatically traps to location 24 and the power fail program has 2 msec. to save all volatile information (data in registers), and to condition peripherals for power failure.

When power is restored, the processor traps to location 24 and executes the power up routine to restore the machine to its state prior to power failure.

Odd Addressing Errors
This error occurs whenever a program attempts to execute a word instruction on an odd address (in the middle of a word boundary). The instruction is aborted and the CPU traps through location 4.

Time-Out Error
This error occurs when a Master Synchronization pulse is placed on the UNIBUS and there is no slave pulse within 10 μsec. This error usually occurs in attempts to address non-existent memory or peripherals.

The offending instruction is aborted and the processor traps through location 4.

Non-Existent Memory Errors
This error occurs when a program attempts to reference a memory address that is larger than indicated by the system size register. The cycle is aborted and the processor traps through location 4.

Reserved Instruction
There is a set of illegal and reserved instructions which cause the processor to trap through location 10. The set is fully described in Chapter 5.

Trap Handling
Chapter 5 includes a list of the reserved Trap Vector locations, and System Error Definitions which cause processor traps. When a trap occurs, the processor follows the same procedure for traps as it does for interrupts (saving the PC and PS on the new Processor Stack, etc.).

In cases where traps and interrupts occur concurrently, the processor will service the conditions according to the priority sequence illustrated.

Trap Priorities
- Parity error
- Memory Management violation
- Stack Limit Yellow
- Power Failure (power down)
- Floating Point exception trap
- Program Interrupt Request (PIR) level 7
- Bus Request (BR) level 7
- PIR 6
- BR 6
- PIR 5
- BR 5
- PIR 4
- BR 4
- PIR 3

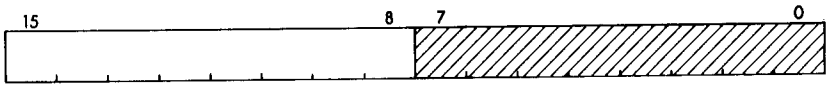
- PIR 2
- PIR 1
- Trace trap

STACK LIMIT

The Stack Limit allows program control of the lower limit for permissable stack addresses. This limit may be varied in increments of $(400)_8$ bytes (200_8 words), up to a maximum address of 177 400 (almost the top of a 64 Kb memory).

The normal boundary for stack addresses is 400. The Stack Limit option allows this lower limit to be raised, providing more address space for interrupt vectors or other data that should not be destroyed by the program.

There is a Stack Limit Register, with the following format:



The Stack Limit Register can be addressed as a word at location 17 777 774, or as a byte at location 17 777 775. The register is accessible to the processor and console, but not to any bus device.

The eight bits, 15 through 8, contain the stack limit information. These bits are cleared by System Reset, Console Start, or the RESET instruction. The lower eight bits are not used. Bit 8 corresponds to a value of $(400)_8$ or $(256)_{10}$.

Stack Limit Violations

When instructions cause a stack address to exceed (go lower than) a limit set by the programmable Stack Limit Register, a Stack Violation occurs. There is a Yellow Zone (grace area) of 32 bytes below the Stack Limit which provides a warning to the program so that corrective steps can be taken. Operations that cause a Yellow Zone Violation are completed, then a bus error trap is effected. The error trap, which itself uses the stack, executes without causing an additional violation, unless the stack has entered the Red Zone.

A Red Zone Violation is a Fatal Stack Error. (Odd Stack or Non-Existent Stack are the other Fatal Stack Errors.) When detected, the operation causing the error is aborted, the stack is repositioned to address 4, and a bus error occurs. The old PC and PS are pushed into locations 0 and 2, and the new PC and PS are taken from locations 4 and 6.

Stack Limit Addresses

The contents of the Stack Limit Register (SL) are compared to the stack address to determine if a violation has occurred. The least significant bit of the register (bit 8) has a value of $(400)_8$. The determination of the violation zones is as follows:

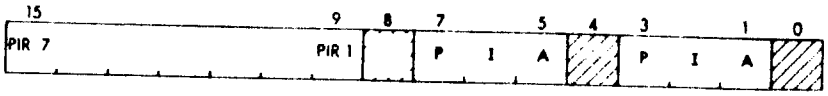
Yellow Zone = (SL) + $(340—377)_8$ execute, then trap
Red Zone ≤ (SL) + $(337)_8$ abort, then trap to location 4

If the Stack Limit Register contents were zero:

Yellow Zone = 340 through 377
Red Zone = 000 through 337

PROGRAM INTERRUPT REQUESTS

A request is booked by setting one of the bits 15 through 9 (for PIR 7—PIR 1) in the Program Interrupt Register at location 17 777 772. The hardware sets bits 7-5 and 3-1 to the encoded value of the highest PIR bit set. This Program Interrupt Active (PIA) should be used to set the Processor Level and also index through a table of interrupt vectors for the seven software priority levels. The figure below shows the layout of the PIR Register.



Program Interrupt Request Register

When the PIR is granted, the Processor will trap to location 240 and pick up the PC in 240 and the PSW in 242. It is the interrupt service routine's responsibility to queue requests within a priority level and to clear the PIR bit before the interrupt is dismissed.

The actual interrupt dispatch program should look like:

```
MOVB PIR, PS          ;places Bits 5-7 in PSW Priority
                      ;Level Bits

MOV R5,-(SP)          ;save R5 on the stack

MOV PIR,R5
BIC #177761,R5        ;Gets Bits 1-3
JMP @DISPAT(R5)       ;use to index through table
                      ; which requires 15 core locations.
```

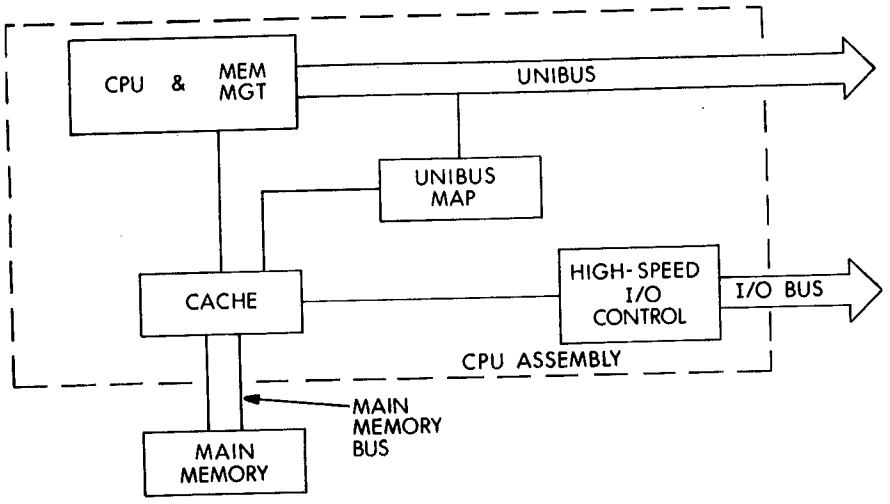


Figure 10-9 Block Diagram of PDP-11/70

MEMORY SYSTEM

An overall block diagram of the PDP-11/70 is shown in Figure 10-9. From a functional standpoint, main memory and the cache can be treated as a single unit of memory.

The PDP-11/70 Cache

The architecture of the cache chosen for the PDP-11/70 is described in this section. It represents a carefully thought-out approach, backed by extensive program simulations to determine hit statistics. The size of the cache memory is 1,024 words (2,048 bytes), organized as a two-way set associative cache with two-word blocks. There are two groups in the cache; each group contains 256 blocks of data, and each block contains two PDP-11 words (see Figures 10-10 and 10-11). Each block also has a tag field, which contains information to construct the address in main memory where the original copy of this data block resides. The data from main memory can be stored within the cache in one index position determined by its physical address. Refer to Figure 10-12 for the organization of the 22-bit physical address. The 8-bit index field (bits 2 to 9) determines which element of the array will contain the data (it can be in either Group 0 or Group 1).

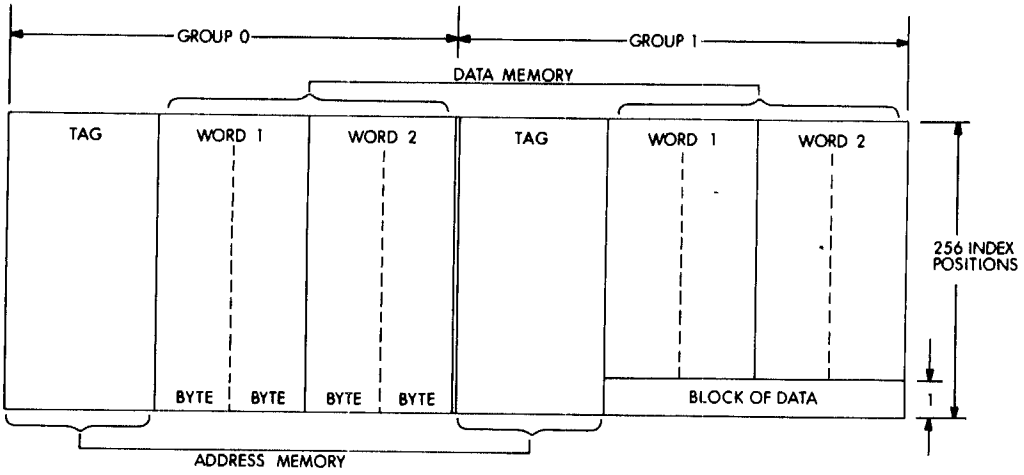


Figure 10-10 Cache Memory (2,048 bytes)

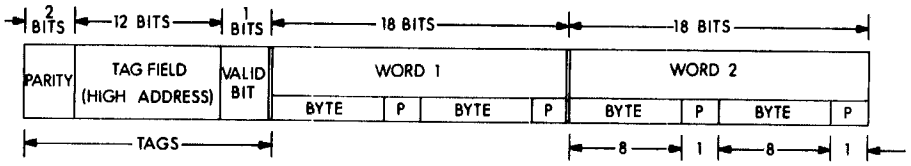


Figure 10-11 Block of Data plus Tags

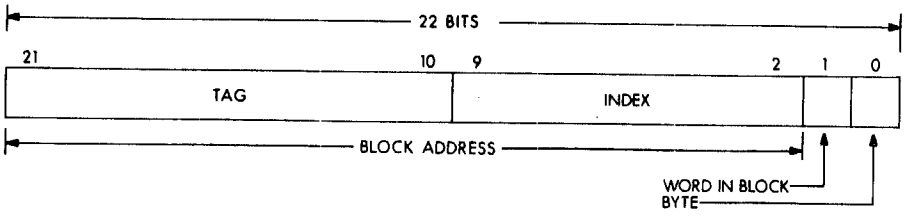


Figure 10-12 Physical Address

The elements of the cache must store not only the data, but also the address identification. Since the index position itself implies part of the address, only the high address field (called tag field) must be stored. The combination of the tag plus index gives the address of the two-word block in main memory. The lowest two bits in the physical address select the particular word in the block, and the byte (if needed).

There are two places in the cache where any block of data can go; a particular index position in either Group 0 to Group 1. Random selection determines into which group the information is placed, overwriting the previous data. Another bit is needed within the cache to determine if the block has been loaded with data. When power is first applied, the cache data are invalid, and the valid bit for each data block is cleared. When a particular block location is updated, the associated valid bit is set to indicate good data.

Figure 10-11 shows the organization for a single block of data within a set. Note that data has byte parity, and that the non-data part, called "tags," contains a 12-bit high order address field plus a valid bit and two parity bits.

General Operation

The system always looks for data in the fast cache memory first. If it is there (a hit), execution proceeds at the fastest rate. If the information is not there (a miss), and the operation was a read, a two-word block of data is transferred from main memory to the cache. If there is a miss while trying to write, cache is not updated. Main memory and the cache are both updated on write hits.

The operation of hits or misses is summarized in Table 10-1.

Table 10-1 Operation on Hit or Miss

	What Happens In	
	CACHE	MAIN MEMORY
READ hit miss	no change updated	no change no change
WRITE hit miss	updated no change	updated updated

When power is first applied (Power-Up), all of the valid bits are cleared. If power is suddenly lost, cache data may become invalid, but main memory, with non-volatile core or battery-backed-up MOS, will have a correct copy of all the data.

With a typical program, writes occur only 10% of the time. Reads occur 90% of the time. Read hits will average 80% to 95% of all cycles with a typical program.

PARITY

System Reliability

Parity is used extensively in the main memory of the PDP-11/70 to ensure the integrity of data storage and transfer, and to enhance the reliability of system operation. All of memory (cache and main memory) has byte parity. Parity is generated and checked on all transfers between core or MOS and cache, again between cache and the CPU, between high-speed mass storage devices and their controllers, and again between the controllers and main memory. A software routine can be used to log the occurrence of parity errors, to handle recovery from errors, and to provide information on system reliability and performance.

Parity in the System

Main memory stores one parity bit for each 8-bit byte in core, or an equivalent function in check bits for ECC MOS memory. Refer to Figure 10-13. The cache also stores byte parity for data, and it stores two parity bits for the address and control information (tag storage) associated with each 2-word block of data.

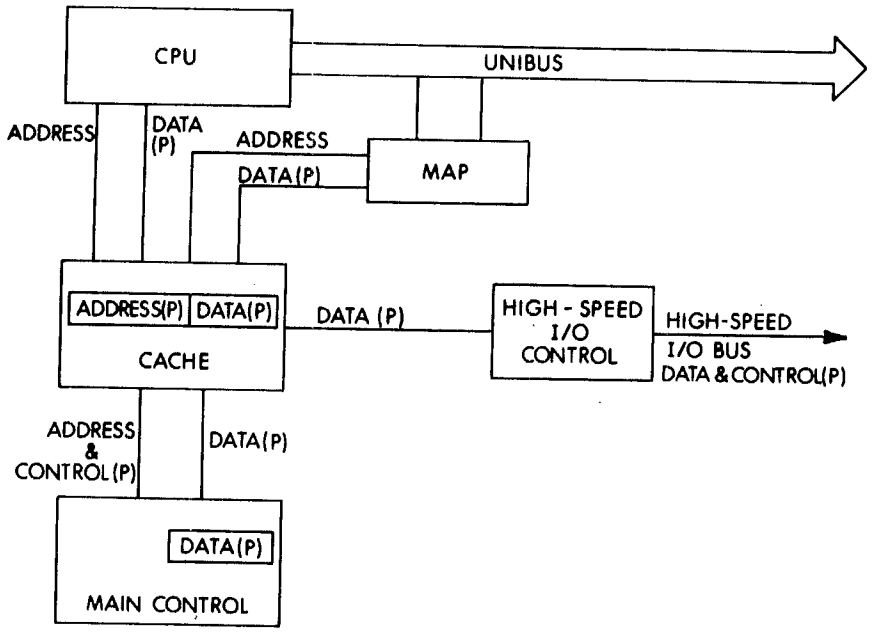


Figure 10-13 Parity (P) in the PDP-11/70 System

The bus between main memory and the cache contains parity on the data and address and control lines. The high-speed I/O controllers

check and generate parity for data transfers to main memory, and they have the capability of handling address errors that are flagged by the control in the cache memory.

System Handling of Parity Errors

Extensive capabilities have been designed into the PDP-11/70 to allow recovery from parity errors, and to allow operation in a degraded mode if a section of the memory system is not operating properly. This type of operation is possible under program control by using the built-in control registers.

If part or all of the cache memory is malfunctioning, it is possible to bypass half or all of the cache. Misses can be forced within the cache, such that all read data is brought from main memory. Operation will be slower, but the system will yield correct results. If part of main memory is not working, the Memory Management unit can be used to map around it. If data found in the cache does not have correct parity, the memory system automatically tries the copy in main memory, to allow program execution to proceed.

Details of how to perform this programming are explained in the next section on the CPU and memory control registers.

Aborts and Traps

Two actions can take place after detection of a parity error. The cycle can be aborted. Then the computer transfers control through the vector at location 114 to an error handling routine. The other action is that the instruction is completed, but then the computer traps (also through location 114). In the first case it was not possible to complete the cycle, whereas in the second case, it was. This second type of parity error usually (but not always) causes the trap before the next instruction is fetched. Refer to Table 10-2.

Table 10-2 Response to Parity Errors

PARITY ERROR DETECTED	CONDITION FOR ABORT	CONDITION FOR TRAP
CPU cycle, data error, read from main memory	Error in requested word.	Error in the other word.
UNIBUS cycle,* data error, read from main memory		Error in either word.

PARITY ERROR DETECTED	CONDITION FOR ABORT	CONDITION FOR TRAP
CPU cycle, address error, reference to main memory	All reads and writes.	
UNIBUS cycle address error reference to main memory		All reads and writes
CPU or UNIBUS cycle, data or address error, reference to cache	All reads.	
High-speed I/O cycle, data or address error, ref to main memory	(no CPU aborts or traps occur; high-speed I/O controllers handle their parity errors).	

NOTE

When a parity error is detected on data going to the UNIBUS, the parity error signal is asserted.

System Response to Parity Errors

Data are read from main memory to the cache in 2-word blocks. If the read cycle was caused by the CPU, and a parity error is detected in the requested word, an abort occurs. If it was in the other word, a trap occurs. On UNIBUS cycles, a trap is caused if there is a read error in either word.

When an address parity error is detected on any read or write to main memory, an abort is caused for both CPU and UNIBUS cycles.

When any fast data memory or address memory parity error is detected on any read from the cache, a trap occurs. On a fast data memory parity error, the CPU will try to get the data from main memory, and also overwrite the same cache location with the new (correct) word just fetched. On an address memory parity error, the CPU will go to main memory for the data, and will correct (overwrite) the tag storage in the cache.

Data transfers for the high-speed mass storage devices take place with main memory. No data is stored in the cache. Parity errors are

handled by the device controllers; no CPU aborts or traps occur, and no cache status registers are affected.

Table 10-2 summarizes the system response.

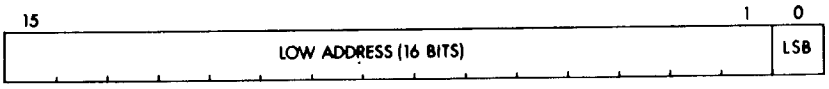
CACHE REGISTERS

The registers described in this section provide information about parity errors, memory status and CPU status. These hardware registers have program addresses in the top 4K words of physical address space (Peripheral Page).

Register	Address
Low Error Address	17 777 740
High Error Address	17 777 742
Memory System Error	17 777 744
Control	17 777 746
Maintenance	17 777 750
Hit/Miss	17 777 752

Some bit positions of the registers are not used (not implemented with hardware) and are indicated by cross-hatching. These bits are always read as zeros by the program. Most of the bits can be read or written under program control. The above six registers are located on the cache control board of the 11/70.

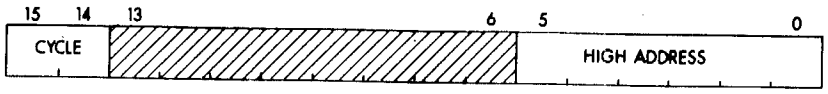
Low Error Address Register 17 777 740



This register contains the lowest 16 bits of the 22-bit address of the first error. The least significant bit is bit 0. The high order bits are contained in the High Error Address Register.

All the bits are read-only. The bits are undetermined after a Power-Up. They are not affected by a Console Start or RESET instruction.

High Error Address Register 17 777 742



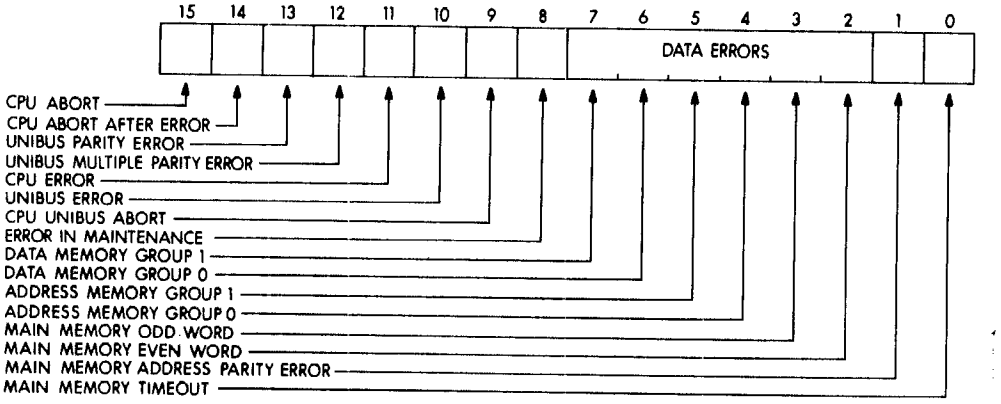
Bit: 15-14 **Name:** Cycle Type
Function: These bits are used to encode the type of memory cycle which was being requested when the parity error occurred.

Bit 15	Bit 14	Cycle Type
0	0	Data In (read)
0	1	Data In Pause
1	0	Data Out
1	1	Data Out Byte

Bit: 5-0 **Name:** Address
Function: These bits contain the highest 6 bits of the 22-bit address of the first error. Register Bit 5 corresponds to the physical address Bit 22.

All the bits are read-only. The bits are undetermined after a Power-Up. They are not affected by a Console Start or RESET instruction.

Memory System Error Register 17 77 744



Bit: 15 **Name:** CPU Abort
Function: Set if an error occurs which caused the cache to abort a processor cycle.

Bit: 14 **Name:** CPU Abort After Error
Function: Set if an abort occurs with the Error Address Register locked by a previous error.

Bit: 13 **Name:** UNIBUS Parity Error
Function: Set if an error occurs which resulted in the UNIBUS Map asserting the parity error signal on the UNIBUS.

Bit: 12 **Name:** UNIBUS Multiple Parity Error
Function: Set if an error occurs which caused the parity error to be asserted on the UNIBUS with the Error Address Register locked by a previous error.

Bit: 11 **Name:** CPU Error
Function: Set if any memory error occurs during a cache CPU cycle.

Bit: 10 **Name:** UNIBUS Error
Function: Set if any memory errors occur during a cache cycle from the UNIBUS.

Bit: 9 **Name:** CPU UNIBUS Abort
Function: Set if the processor traps to vector 114 because of UNIBUS parity error on a DATI or DATIP memory cycle.

Bit: 8 **Name:** Error in Maintenance
Function: Set if an error occurs when any bit in the Maintenance Register is set. The Maintenance Register will then be cleared.

Bit: 7-6 **Name:** Data Memory
Function: These bits are set if a parity error is detected in the fast data memory in the cache. Bit 7 is set if there is an error in Group 1; bit 6 for Group 0.

Bit: 5-4 **Name:** Address Memory
Function: These bits are set if a parity error is detected in the address memory in the cache. Bit 5 is set if there is an error in Group 1; bit 4 for Group 0.

Bit: 3-2 **Name:** Main Memory
Function: These bits are set if a parity error is detected on data from main memory. Bit 3 is set if there is an error in either byte of the odd word: bit 2 for the even word. (Main memory always transfers two words at a time.) An abort occurs if the error is in the word needed by a CPU reference. A trap occurs if the error is in the other word, or if it is a UNIBUS reference.

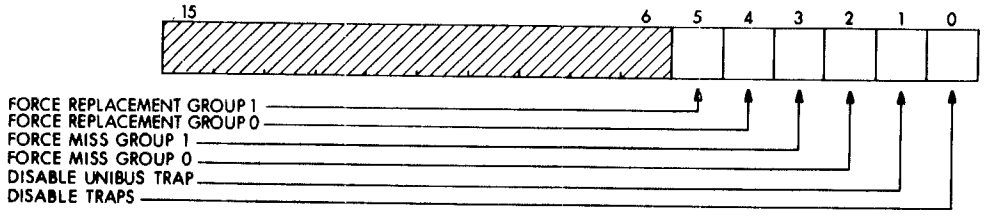
Bit: 1 **Name:** Main Memory Address Parity Error
Function: Set if there is a parity error detected on the address and control lines on the main memory bus.

Bit: 0 **Name:** Main Memory Timeout
Function: Set if there is no response from main memory. For CPU cycles, this error causes an abort. When a UNIBUS device requests a non-existent location, this bit will set, cause a timeout on the UNIBUS, and then cause the CPU to trap to vector 114.

The bits are cleared on Power-Up or by Console Start. They are unaffected by a RESET instruction.

When writing to the Memory System Error Register, a bit is unchanged if a 0 is written to that bit, and it is cleared if a 1 is written to that bit. Thus, the register is cleared by writing the same data back to the register. This guarantees that if additional error bits were set between the read and the write, they will not be inadvertently cleared.

Control Register 17 777 746



Bit: 5-4 **Name:** Force Replacement
Function: Setting these bits forces data replacement within a Group in the cache by main memory data on a read miss. Bit 5 selects Group 1 for replacement; bit 4 selects Group 0.

Bit: 3-2 **Name:** Force Miss
Function: Setting these bits forces misses on reads to the cache. Bit 3 forces misses on Group 1; bit 2 forces misses on Group 0. Setting both bits forces all cycles to main memory.

Bit: 1 **Name:** Disable UNIBUS Trap
Function: Set to disable traps to vector 114 when the parity error signal is placed on the UNIBUS.

Bit: 0 **Name:** Disable Traps
Function: Set to disable traps from non-fatal errors.

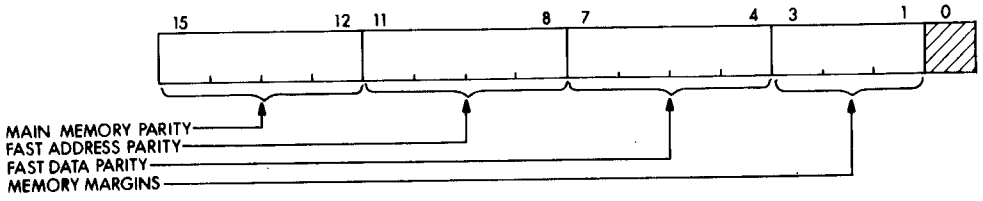
Bits 5 through 0 are read/write. The bits are cleared on Power Up or by Console Start.

The PDP-11/70 can run in a degraded mode if problems are detected in the cache. If Group 0 of the cache is malfunctioning, it is possible to force all operations through Group 1. If bits 2 and 5 of the Control Register are set, and bits 3 and 4 are clear, the CPU will not be able to read data from Group 0, and all main memory data replacements will occur within Group 1. In this manner, half the cache will be operating. But system throughput will not decrease by 50%, since the statistics of read hit probability will still provide reasonably fast operation.

If Group 1 is malfunctioning, bits 3 and 4 should be set, and bits 2 and 5 cleared, such that only Group 0 is operating. If all of the cache is malfunctioning, bits 2 and 3 should be set. The cache will be by-passed, and all references will be to main memory.

Bits 1 and 0 can be set to disable trapping; more memory cycles will be performed, but overall system operation will produce correct results.

Maintenance Register 17 777 750



Bit: 15-12 **Name:** Main Memory Parity
Function: Setting these bits causes the four parity bits to be 1's. There is 1 bit per byte; there are 4 bytes in the data block.

Bit Set	Byte
15	odd word, high byte
14	odd word, low byte
13	even word, high byte
12	even word, low byte

Bit: 11-8 **Name:** Fast Address Parity
Function: Setting these bits causes the four parity bits for fast address memory to be wrong. Bits 11 and 10 affect Group 1; bits 9 and 8 affect Group 0.

Bit: 7-4 **Name:** Fast Data Parity
Function: Setting these bits causes the four parity bits to be 1's.

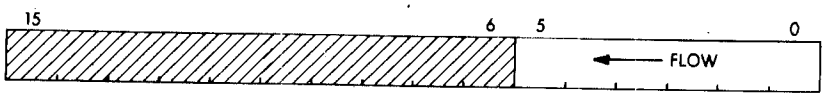
Bit Set	Byte
7	Group 1, high byte
6	Group 1, low byte
5	Group 0, high byte
4	Group 0, low byte

Bit: 3-1 **Name:** Memory Margins
Function: These bits are encoded to do maintenance checks on main memory.

Bit 3	Bit 2	Bit 1	
0	0	0	Normal operation
0	0	1	Check wrong address parity
0	1	0	Early strobe margin
0	1	1	Late strobe margin
1	0	0	Low current margin
1	0	1	High current margin
1	1	0	(reserved)
1	1	1	(reserved)

All of main memory is margined simultaneously.

Hit/Miss Register 17 777 752



This register indicates whether the six most recent references by the CPU were hits or misses. A 1 indicates a read hit; a 0 indicates a read miss or a write. The lower numbered bits are for the more recent cycles.

All the bits are read-only. The bits are undetermined after a Power-Up. They are not affected by a RESET instruction.

HIGH-SPEED CONTROLLERS

Mounting Space

The PDP-11/70 CPU assembly provides dedicated, prewired space for up to four high-speed I/O controllers. Refer to Figure 10-14. DC power for the controllers is derived from the cabinet power supply.

Interfacing

Each group of mass storage peripherals communicates with its high-speed controller through a separate high-speed I/O bus. This I/O bus consists of a set of 56 signals for data, control, status, and parity. High transfer rate is achieved by using synchronous block transfer of data simultaneously with asynchronous control information. The controller contains an 8-word data buffer.

Data are transferred in a Direct Memory Access (DMA) mode. An internal 32-bit wide data bus transfers 4 bytes in parallel between memory and the high-speed controllers. The Priority Arbitration logic within the cache memory controls the timing of data transfers; but the cache itself is not used for storage. Data transfers are between main

memory and the mass storage peripheral. The cache is not affected, except that on a write hit from the I/O bus to memory, the valid bit is cleared for that particular 2-word block within the cache. In this way, the affected areas of the cache are flagged as having incorrect data, but main memory always contains the correct, updated information.

The UNIBUS plays a subordinate role with respect to the high-speed controllers. The UNIBUS is used:

a. to supply control and status information
b. to generate an interrupt request (by the controller)

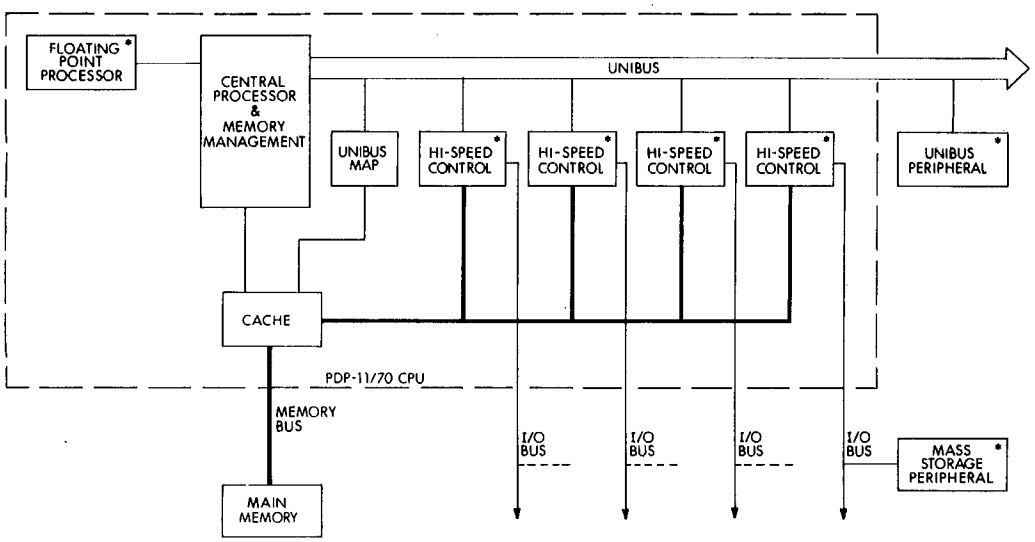


Figure 10-14 PDP-11/70 Block Diagram

The UNIBUS is not used for data transfer.

The registers within the controller (which can be read and written directly) are addressed from the UNIBUS. In a typical DMA transfer, the registers would first be loaded with the following data:

a. number of words to be transferred
b. starting address in memory for data transfers
c. control information specifying the device and type of operation

Increased Data Transfer Rate

The architecture of the PDP-11/70 allows overlapping of some operations, providing faster program execution speed. CPU and UNIBUS read hits with the cache memory are overlapped with mass storage device reads from main memory. It is possible to overlap the read cycles of several mass storage devices.

Parity

Parity is generated and checked in the system for data and address and control information, to ensure the integrity of the information transferred. The RHCS3 register in the controller is used to indicate the occurrence of parity errors during memory transfers.

REGISTERS

The controller contains six local registers, plus part of one more which is shared with the mass-storage device. Other registers needed by the particular mass storage system and device are contained in the device itself. Appendix B contains information about the mass storage device registers.

Controller Registers

RHCS1	Control and Status 1 (partial)
RHWC	Word Count
RHBA	Bus Address (Main Memory Bus)
RHBAE	Bus Address Extension (Main Memory Bus)
RHCS2	Control and Status 2
RHCS3	Control and Status 3
RHDB	Data Buffer (Maintenance)

CONTROLLER REGISTERS

Control and Status 1 Register (RHCS1)

This register is used by the controller and the mass storage device to store the device commands and hold operational status. Register bits 0 through 5, 11, and 12 are dedicated for use by the drive and are physically located in each drive attached to the controller. When reading or writing this register, the selected drive (indicated by bits 2 through 0 in the RHCS2 register) will respond in those bits' positions.

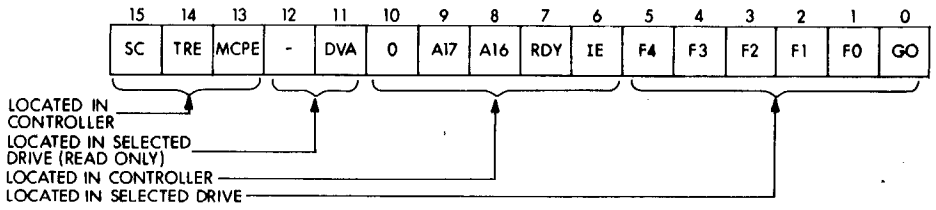
When the program reads, writes a word, or writes the low byte of this register, a register cycle will be initiated to the selected drive over the high-speed I/O bus. If the unit selected does not exist or respond, an NED (non-existing drive) error will result. The program may, however, write the upper byte of this register without regard to the unit selected and without affecting any drive.

Register bits 0 through 5 indicate the command to be performed and are actually stored in the selected drive. The controller will always interrogate the command code being passed to the drive by the program and will prepare for the appropriate memory cycle required by

data transfer operations. Data transfer command codes are designated by 51_8 through 77_8 (always odd, since the GO bit must be asserted to execute the function) and will cause the controller to become busy (RDY negated) until the completion of the operation. When the controller is busy, no further data transfer commands may be issued (see PGE bit 10 in RHCS2). Commands that are not data transfer commands, however, may be issued at any time and to any drive which is not busy.

While a data transfer is in progress, unit select bits U(02:00) in RHCS2 may be changed by the program in order to issue a non-data-transfer command to another drive. This will not affect the data transfer.

When a non-data-transfer command code is written into RHCS1 while a data transfer is taking place, only the even (low) byte of RHCS1 should be written. This will prevent the program from unintentionally changing the A16 and A17 status bits if the transfer is completed just before the register is written. (While the RDY bit is negated, the controller prevents program modification of these control bits even when the write is done to the odd byte).



Control and Status 1 Bit Usage

Bit: 15 **Name:** SC
Special Condition
Function: Read-only. Set by TRE, Attention, or MCPE. Cleared by UNIBUS INIT, Controller Clear, or by removing the Attention condition. SC = TRE + ATTN + MCPE. Attention occurs when any drive has a) an error condition, b) a change in status, or c) completed a function requiring action by the program (other than data transfer).

Bit: 14 **Name:** TRE
Transfer Error
Function: Read/Write. Set by DLT, WCE, PE, NED, NEM, PGE, MXF, MDPE, or a drive error during a data transfer. Cleared by UNIBUS INIT, controller clear, error clear (the action of writing a 1 in the TRE bit), or by loading a data transfer command with GO set. TRE = DLT + WCE + PE + NED + NEM + PGE + MXF + MDPE + (EXCP-EBL)

Bit: 13 **Name:** MCPE
Mass I/O Bus Control Parity Error
Function: Read-only. Set by a parity error on the control section of the I/O bus when reading a remote register (located in the drive). Cleared by UNIBUS INIT, Controller Clear, error clear, or by loading a data transfer command with GO set. Parity errors which occur on the control bus when writing a drive register are detected by the drive. Parity checking occurs at the completion of the register cycle (an MCPE when reading the RHCS1 register would not be indicated on the same cycle).

Bit: 12 **Name:** Reserved for use by the Drive
Function: Read-only. Always read as 0 if not implemented by the selected drive.

Bit: 11 **Name:** DVA
Drive Available
Function: Read-only. Implemented by the drive. Set when the selected drive in available to the controller. Used in dual-port drive applications. Always a 1 in single port drives.

Bit: 10 **Name:** Not used
Function: Always read as 0.

Bit: 9
8 **Name:** A17
A16
Bus Address Extension Bits
Function: Read/Write. Upper address extension bits of the BA register. Cleared by UNIBUS INIT, Controller Clear, or by writing 0's in these bit positions. These bits cannot be modified by writing to the RHCS1 register while the controller is busy (RDY negated). Incremented by a carry from the RHBA register during data transfers to/from memory. These bits can also be set/cleared through the RHBAE register.

Bit: 7 **Name:** RDY
Ready
Function: Read-only. Indicates controller status. When set, the controller will accept any command. When cleared, the controller is performing a data transfer command and will allow only non-data transfer commands to be executed. The assertion of RDY (transfer complete or TRE) will cause an interrupt if IE = 1.

Bit: 6 **Name:** IE
Interrupt Enable
Function: Read/Write. Control bit which can be set under program control. When IE = 1, an interrupt may occur due to RDY, Attention, or

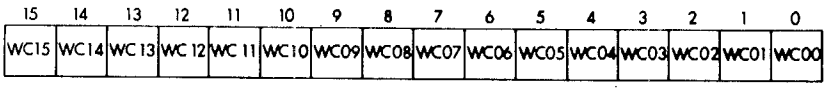
MCPE being asserted. Cleared by UNIBUS INIT, Controller Clear, or automatically cleared when an interrupt is recognized by the CPU. A program-controlled interrupt may occur by writing 1s into IE and RDY at the same time. This bit can be set/cleared through the RHCS3 register.

Bit: 5-0 **Name:** F4-F0 and GO
Function: Read/Write. F4-F0 are function (command) code control bits which determine the action to be performed by the controller and/or drive. The GO bit must be set in order to execute the command. The GO bit is reset by the drive at the end of the operation. The function code bits are stored in the selected drive. Only data transfer commands (defined as F4·(F3 + F2)·GO will cause the controller to become busy (RDY negated). All other command codes are ignored by the controller.

Function Code Table

F4	F3	F2	F1	F0	
0	0	0	0	0	Reserved for drive related commands. No controller action taken.
through					
1	0	0	1	1	
1	0	1	0	0	Write Check commands. Memory data compared with drive data in controller. Memory address increments.
1	0	1	0	1	
1	0	1	1	0	
1	0	1	1	1	Write Check command. Memory address decrements.
1	1	0	0	0	Write commands. Memory data written into drive. Memory address increments.
1	1	0	0	1	
1	1	0	1	0	
1	1	0	1	1	Write command. Memory address decrements.
1	1	1	0	0	Read commands. Drive data written into Memory. Memory address increments.
1	1	1	0	1	
1	1	1	1	0	
1	1	1	1	1	Read command. Memory address decrements.

Word Count Register (RHWC)
This register is loaded by the program with the 2's complement of the number of words to be transferred. During a data transfer, it is incremented by 1 each time a word is transmitted to or from memory.

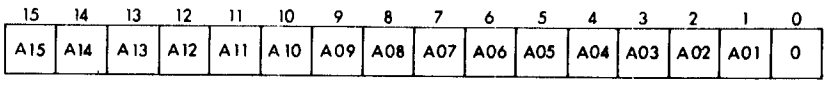


Word Count Register Bit Usage

Bit: WC(15:00) **Name:** Word Count
Function: Read/Write. Set by the program to specify the number of words to be transferred (2's complement form). This register is cleared only by writing 0's into it. Incremented for each word transferred to/from memory.

Bus Address Register (RHBA)
This register is loaded by the program to specify the lower 16 bits of the starting memory address to which data transfers will take place. The RHBA and RHBAE registers combine to form the complete 22 bit memory address.

During a data transfer, this register is incremented (decremented for specific function codes) by 2 each time a word is transmitted to or from memory. If the BAI (Bus Address Increment Inhibit) bit (bit 3 of RHCS2) is set, the incrementing (or decrementing) of the RHBA register is inhibited and all transfers take place to or from the starting memory address.



Bus Address Register Bit Usage

Bit: 15-1 **Name:** A (15:01)
Bus Address
Function: Read/Write. Loaded by the program to specify the starting memory address of a data transfer operation. Cleared by UNIBUS INIT or Controller Clear. The RHBA register is incremented (or decremented) by 2 whenever a word is transmitted to or from memory.

Bit: 0 **Name:** Not Used
Function: Always read as a 0

Bus Address Extension Register (RHBAE)
The RHBAE register contains the upper 6 bits of the memory address and combine with the lower 16 bits located in RHBA to form the complete 22 bit address. This register should be loaded by the program with the RHBA register to specify the starting memory address of a data transfer operation. The 6-bit field is incremented (decremented for specific function codes) each time a carry (borrow) occurs from the RHBA register during memory transfers.

Address bits A16 and A17 can also be set or cleared through the RHCS1 register. If an address extension field is written into RHBAE, the program should ensure that A16 and A17 are not altered when a command is loaded into RHCS1. This can be accomplished by either loading the command with a write low byte instruction to RHCS1 or by ensuring the proper value appears in the A16 and A17 bit positions of RHCS1.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	A21	A20	A19	A18	A17	A16

Bus Address Extension Register Bit Usage

Bit: 15-6 **Name:** Not Used
Function: Always read as 0

Bit: 5-0 **Name:** A(21:16)
Bus Address
Function: Read/Write. Loaded by the program to specify the starting memory address of a data transfer operation. Cleared by UNIBUS INIT or Controller Clear. The RHBAE register is incremented (or decremented) each time a carry (borrow) out of RHBA occurs. A16 and A17 can also be set or cleared through the RHCS1 register.

Control and Status 2 Register (RHCS2)
This register indicates the status of the controller and contains the drive unit number U(2:0). The unit number specified in bits 2 through 0 of this register indicates which drive is responding when registers located in a drive are addressed.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DLT	WCE	PE	NED	NEM	PGE	MXF	MDPE	OR	IR	CLR	PAT	BAI	U2	U1	U0

Control and Status 2 Register Bit Usage

Bit: 15 **Name:** DLT
Data Late
Function: Read-only. Set when the controller is unable to supply a data word during a write operation or accept a data word during a read or write-check operation at the time the drive demands a transfer. Cleared by UNIBUS INIT, Controller Clear, error clear, or loading a data transfer command with GO set. DLT causes TRE. Buffering is eight words deep in the controller and a DLT error indicates a severely overloaded system.

Bit: 14 **Name:** WCE
Write Check Error
Function: Read-only. Set when the controller is performing a write-check operation and a word on the drive does not match the corresponding word in memory. Cleared by UNIBUS INIT, Controller Clear, error clear, or loading a data transfer command with GO set. WCE causes TRE. If a mismatch is detected during a write-check command execution, the transfer terminates and the WCE bit is set. The mismatched data word from the drive is displayed in the data buffer (RHDB).

Bit: 13 **Name:** PE
Parity Error
Function: Read-only. Set if a parity error occurred between memory and the controller during a memory transfer. Cleared by UNIBUS INIT Controller Clear, error clear, or loading a data transfer command with GO set.

PE = APE + DPEOW + DPEEW

Bit: 12 **Name:** NED
Non-Existent Drive
Function: Read-only

Set when the program reads or writes a register in a drive selected by U(02:00) which does not exist or is powered down. (The drive fails to assert TRA within 1.5 μs after assertion of DEM.) Cleared by UNIBUS INIT, Controller Clear, error clear, or loading a data transfer command with GO set. NED causes TRE.μ**Bit:**
11 **Name:** NEM
Non-Existent Memory
Function: Read-only. Set when the controller is performing a DMA transfer and the memory address specified in RHBA is non-existent. Cleared by UNIBUS INIT, Controller Clear, error clear, or loading a data transfer command with GO set. NEM causes TRE to set.

Bit: 10 **Name:** PGE
Program Error

Function: Read-only. Set when the program attempts to initiate a data transfer operation while the controller is currently performing one. Cleared by UNIBUS INIT, Controller Clear, error clear, or loading a data transfer command with GO set. PGE causes TRE to set. The data transfer command code is inhibited from being written into the drive.

Bit: 9 **Name:** MXF
Missed Transfer
Function: Read-only. Set if the drive does not respond to a data transfer command within 650 μsec. Cleared by UNIBUS INIT, Controller Clear, error clear, or loading a data transfer command with GO set. MXF causes TRE to set. This error occurs if a data transfer command is loaded into a drive which has ERR set, or if the drive fails to initiate the command for any reason (such as parity error or illegal function).

Bit: 8 **Name:** DPE
Mass I/O Bus Data Parity Error
Function: Read-only. Set when a parity error occurs on the data section of the I/O bus while doing a read or write-check operation. Cleared by UNIBUS INIT, Controller Clear, error clear, or loading a data transfer command with GO set. MDPE causes TRE. Parity errors on the data bus during write operations are detected by the drive.

Bit: 7 **Name:** OR
Output Ready
Function: Read-only.Set when a word is present in RHDB and can be read by the program, Cleared by UNIBUS INIT, Controller Clear, or by reading DB. Serves as a status indicator for diagnostic check of the data buffer.

Bit: 06 **Name:** IR
Input Ready
Function: Read-only. Set when a word may be written in the RHDB register by the program. Cleared when the data buffer is full (contains eight words). Serves as a status indicator for diagnostic check of the data buffer.

Bit: 5 **Name:** CLR
Controller Clear
Function: Write-only. When a 1 is written into this bit, the controller and all drives are initialized. UNIBUS INIT also causes Controller Clear to occur.

Bit: 4 **Name:** PAT
Parity Test
Function: Read/Write. While PAT is set, the controller generates even parity on both the Control and Data sections of the I/O bus. When

clear, odd parity is generated. Cleared by UNIBUS INIT or Controller Clear. While PAT is set, the controller checks for even parity received on the Data Bus but not on the Control Bus.

Bit: 3 **Name:** BAI
UNIBUS Address Increment Inhibit
Function: Read/Write. When BAI is set, the controller will not increment the BA register during a data transfer. This bit cannot be modified while the controller is doing a data transfer (RDY gated). Cleared by UNIBUS INIT or Controller Clear. When set during a data transfer, all data words are read from or written into the same memory location.

Bit: 2-0 **Name:** U(2:0)
Unit Select (2:0)
Function: Read/Write. These bits are written by the program to select a drive. Cleared by UNIBUS INIT or Controller Clear. The unit select bits can be changed by the program during data transfer operations without interfering with the transfer.

Control and Status 3 (RHCS3)
The RHCS3 register contains parity error information associated with the memory bus. Bit position 13 of the RHCS2 (PE) indicates that a parity error occurred during the memory transfer. Bits 15 through 13 of RHCS3 further localize the error for diagnostic maintenance. In addition, bits 3 through 0 provide the diagnostic program with the ability to invert the sense of parity check and thereby verify correct operation of the parity circuits.

An Interrupt Enable bit in the RHCS3 register allows the program to enable interrupts without writing into a drive register as previously described. This bit also appears in the RHCS1 register for program compatibility and can be set or cleared by writing into either register.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
APE	DPE OW	DPE EW	WCE OW	WCE EW	DBL	0	0	0	IE	0	0	IPCK 3	IPCK 2	IPCK 1	IPCK 0

Control and Status 3 Bit Usage

Bit: 15 **Name:** APE
Address Parity Error
Function: Read-only. Set if the address parity error line indicates that the memory detected a parity error on address and control information during a memory transfer. Cleared by UNIBUS INIT, Controller Clear, error clear, or loading a data transfer command with GO

set. APE causes PE, bit 13 of RHCS2. When an APE error occurs, the RHBA cnd RHBAE registers contain the address +4 of the double word address at which the error occurred during a double word operation or the address +2 during a single word operation.

Bit: 14, 13 **Name:** DPE, OW, EW Data Parity Error Odd Word Even Word
Function: Read-only. Set if a parity error is detected on data from memory when the control is performing a write or write check command. Cleared by UNIBUS INIT, Controller Clear, error clear, or loading a data transfer command with GO set. DPE causes PE, bit 13 of RHCS2. When a DPE error occurs, the RHBA and RHBAE registers contain the address +4 of the double word address at which the error occurred during a double word operation, or the address +2 during a single word operation.

Bit: 12, 11 **Name:** CE
OW, EW
Write Check Error Odd word, Even word
Function: Read-only. Set when data fails to compare between memory and the drive. Cleared by UNIBUS INIT, Controller Clear, error clear, or loading a data transfer command with the GO bit set. Causes WCE, bit 14 of RHCS2. The word read from the drive which did not compare is locked in the data buffer and can be examined by reading the RHDB register.

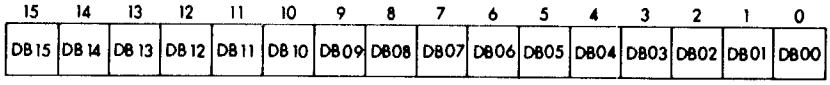
Bit: 10 **Name:** DBL
Double word
Function: Read-only. Set if the last memory transfer was a double word operation. Cleared by UNIBUS INIT, Controller Clear or loading a data transfer command with GO set.

Bit: 9-7 **Name:** Not Used
Function: Always read as 0

Bit: 6 **Name:** IE
Interrupt Enable
Function: Read/Write. IE is a control bit which can be set under program control. When IE = 1, an interrupt may occur due to RDY or SC being asserted. Cleared by UNIBUS INIT, Controller Clear, or automatically cleared when an interrupt is recognized by the CPU. When a 0 is written into IE by the program, any pending interrupts are cancelled.

This bit can also be set or cleared by writing into the RHCS1 register. If written through the RHCS3 register, write operation is not performed into a drive register simultaneously.

Bit: 15-0 **Name:** DB(15:00)
Data Buffer
Function: Read/Write. When read, the contents of OBUF (internal register) are delivered. Upon completion of the read, the next sequential word in the buffer will be clocked into OBUF. Used by the program for diagnostic purposes. When the register is written into, IR is cleared until the DB is ready to accept a new word. When the register is read, it will cause OR to be cleared until a new word is ready. During a Write Check Error condition the data word read from the disk which did not compare with the corresponding word in memory is frozen in RHDB for examination by the program.



Data Buffer Bit Usage

Bit: 5-4 **Name:** Not Used
Function: Always read as 0

Bit: 3-0 **Name:** IPCK (3:0)
Invert Parity Check (3:0)
Function: Read/Write. These bits are written by the program to control the data parity detection logic. When set, inverse parity is checked with data during memory transfers of write and write check operations. Parity control is provided for each byte in double word addresses.
IPCK 0—Even Word, Even Byte

IPCK 1—Even Word, Odd Byte

IPCK 2—Odd Word, Even Byte

IPCK 3—Odd Word, Odd Byte

Data Buffer Register (RHDB)
This register provides a maintenance tool to check the data buffer in the controller. A total of eight words is accepted before the data buffer becomes full. Successive reads from DB read out words in the same order in which they were entered into the data buffer.

The IR (input ready) and OR (output ready) status indicators in the RHCS2 register are provided so that the programmer can determine when words can be read from or written into the RHDB. IR should be asserted before attempting a write into DB; OR should be asserted before attempting a read from DB.

The RHDB register can be read and written only as an entire word. Any attempt to write a byte will cause an entire word to be written. Reading the DB register is a "destructive read-out" operation. The top data word in the data buffer is removed by the action of reading DB, and a new data word (if present) replaces it a short time later. Conversely, the action of writing the DB register does not destroy the contents of DB. It merely causes one more data word to be inserted into the data buffer, if it was not full.

CONSOLE OPERATION

The PDP-11/70 console allows direct control of the computer system. It contains a power switch for the CPU, which is also usually used as the Master Switch for the system. The console is used for starting, stopping, resetting, and debugging. Lights and switches provide the facilities for monitoring operation, system control, and maintenance. Debugging and detailed tracing of operations can be accomplished by having the computer execute single instructions or single cycles. Contents of all locations can be examined, and data can be entered manually from the console switches.

GENERAL

The PDP-11/70 Operator's Console provides the following facilities:

a) Power Switch (with a key lock)

b) ADDRESS Register display (22 bits)

c) DATA Register display (16 bits), plus Parity Bit Low Byte, & Parity Bit High Byte

d) Switch Register (22 switches)

e) Error Lights
   - ADRS ERR (Address Error)
   - PAR ERR (Parity Error)

f) Processor State Lights (7 indicators)
   - RUN
   - PAUSE
   - MASTER
   - USER
   - SUPERVISOR
   - KERNEL
   - DATA

g) Mapping Lights
   - 16 BIT
   - 18 BIT
   - 22 BIT

h) ADDRESS Display Select Switch (8 positions)
   - USER I
   - USER D
   - SUPER I (Virtual)
   - SUPER D
   - KERNEL I
   - KERNEL D
   - PROG PHY (Program Physical)
   - CONS PHY (Console Physical)

i) DATA Display Select Switch (4 positions)
   - DATA PATHS
   - BUS REGISTER
   - μADRS FPP/CPU
   - DISPLAY REGISTER

j) Lamp Test Switch

k) Control Switches
   - LOAD ADRS
   - EXAM (Examine)
   - DEP (Deposit)
   - CONT (Continue)
   - ENABLE/HALT
   - S INST/S BUS CYCLE (Single Instruction/Single Bus Cycle)
   - START

STARTING AND STOPPING

Starting

Once power is on, execution can be started by placing the ENABLE/HALT switch in the ENABLE position, putting the starting address in the Switch Register, and depressing the LOAD ADRS switch. Verify in the Address Display Lights that the address was entered correctly, then depress the START switch. The computer system will be cleared and will then start running. Once execution has begun, depressing the START switch again has no effect.

If the system needs to be initialized but execution is not wanted, the START switch should be depressed while the HALT/ENABLE switch is in the HALT position.

Stopping

Set the ENABLE/HALT switch to the HALT position. The computer will stop execution, but the contents of all memory locations will be retained. The switch can then be set to the ENABLE position with no effect on the system.

NOTE

NPRs are still serviced after halt from the console if S BUS CYCLE is disabled.

Continuing

After the computer has been stopped, execution can be resumed from the point at which it was halted by using the CONT (Continue) Switch. The function of the CONT Switch depends on the position of the ENABLE/HALT Switch:

ENABLE (up)	CPU resumes normal execution.
HALT (down)	The mode is used for debugging purposes and forces execution of a single instruction or a single bus cycle.

REFERENCING MEMORY

Unmapped References

When performing unmapped memory references from the console, the Address Select Switch must be set to CONS PHY. This means that the 22-bit address entered in the Switch Register should be the physical address desired. To examine a memory location, depress the LOAD ADRS switch and then the EXAM switch. The address referenced will appear in the Address Display Lights. The DATA Select switch should be selecting DATA PATHS, and the contents of that location are displayed in the Data Display Lights. To deposit information into a memory location, depress the LOAD ADRS switch, then enter the desired data in the Switch Register and raise the DEP switch. The DATA Select switch should be in the DATA PATHS position, and the deposited information will appear in the DATA Display Lights.

Mapped References

Sometimes, when software is running with Memory Management enabled, the physical addresses generated are not known. This makes examining and depositing memory locations more difficult. For this reason, the six positions KERNEL I through USER D of the ADDRESS Select switch are provided. When doing a memory reference, the low order 16 bits of the Switch Register are considered to be a Virtual Address and are relocated by Memory Management using the set of PAR/PDR's indicated by the ADDRESS Select switch.

To examine a memory location, depress the LOAD ADRS switch and the EXAM switch. The DATA Select switch should be selecting DATA PATHS, and the contents of that location are displayed in the DATA Display Lights. To deposit information into a memory location, de-

press the LOAD ADRS switch, then enter the desired data in the Switch Register and raise the DEP switch. The Data Select Switch should be in the DATA PATHS position, and the deposited information will appear in the DATA Display Lights.

The PROG PHY (Program Physical) position of the ADDRESS Select switch is used as a debugging tool. After an examine or deposit has been performed on a virtual address, changing the ADDRESS Select switch to select PROG PHY will display the Physical Address generated by Memory Management in the Address Display Lights. Using the PROG PHY position in any other way will produce meaningless results.

NOTE

An EXAM or DEP operation which causes an addressing error (ADRS ERR or PAR ERR) will be aborted and must be corrected by performing a new LOAD ADRS operation with a valid address.

STEP OPERATIONS

Performing more than one EXAM operation in a row or more than one DEP operation in a row results in a STEP operation. Depressing the EXAM switch after previous examination of a location displays the contents of the next location in memory. Raising the DEP switch after a previous deposit into a memory location causes the current contents of the Switch Register to be deposited into the next location in memory.

In each case, the Address Display is updated by 2 to hold the value of the now current address. This allows consecutive EXAM operations and consecutive DEP operations without the use of the LOAD ADRS switch. An EXAM-STEP or DEP-STEP operation will not cross a 32K word memory block boundary.

NOTE

The EXAM and DEP switches are coupled to enable an EXAM—DEP—EXAM sequence to be carried out on a location without having to do extra LOAD ADRS operations. The following example deposits values into consecutive memory locations.

Operation (Activate Switch)	**Location shown in ADDRESS Display**
LOAD ADRS	X
EXAM	X
DEP	X
EXAM	X

Operation (Activate Switch)	Location shown in ADDRESS Display
EXAM (result is EXAM—STEP)	X+2
DEP	X+2
EXAM	X+2

GENERAL REGISTERS

The General Registers can be examined and deposited using the EXAM and DEP Switches provided the previous LOAD ADRS operation loaded the Address Display with a "register address."

Address	Register
17 777 700	Register 0 (Set 0)
.	.
.	.
.	.
.	.
17 777 705	Register 5 (Set 0)
17 777 706	Register 6, Kernel Mode
17 777 707	Program Counter
17 777 710	Register 0 (Set 1)
.	.
.	.
.	.
.	.
17 777 715	Register 5 (Set 1)
17 777 716	Register 6, Supervisor Mode
17 777 717	Register 6, User Mode

Examining and depositing into General Register Addresses is independent of the ADDRESS Select switch. It is not possible to be mapped to a General Register.

EXAM-STEP and DEP-STEP operations can be performed on the General Registers, similar to that for memory locations, except that:

a) ADDRESS Display is incremented by 1 (instead of 2)

b) The STEP after address 17 777 717 is 17 777 700, such that the addresses are looped.

c) It is not possible to STEP up to the first General Register (17 777 700) from 17 777 676

SINGLE INSTRUCTION/SINGLE BUS CYCLE

Once the machine is halted, a useful debugging tool is being able to execute code, a small segment at a time. The S INST/S BUS CYCLE (Single Instruction/Single Bus Cycle) switch provides that capability. The ENABLE/HALT switch must be in the HALT position. To start execution of a segment depress the CONT switch. How much is executed is a function of the S INST/S BUS CYCLE switch.

Position	
S INST	Depressing the CONT Switch will result in the execution of one instruction. This means that the machine state can be determined after each instruction. Examining and depositing into memory locations is a method of accomplishing this. The contents of the DATA Display Lights are not necessarily meaningful.
S BUS CYCLE	For this mode to have any meaning, the DATA Select switch should be selecting the BUS REG (Bus Register). Depressing the CONT Switch will execute until the end of the next bus cycle. The Address Display Lights will then contain the address of the location at which the bus cycle was performing. (Virtual or Physical, depending on the position of the ADDRESS Select switch). The DATA Display Lights, on a read operation, will contain the data that was read (this could be an instruction or data). During a write operation, the lights will contain the data just written (except during a stack operation or Floating Point Instruction). Examine and deposit operations cannot be used in this mode. Depressing the LOAD ADRS, EXAM, or DEP switch will not cause anything to happen. If an examine or deposit operation is desired, the S INST/S BUS CYCLE switch should be changed to select

S INST and the CONT switch should be depressed once. (This will cause execution until the end of the current instruction). The system will then be ready to perform an examine or deposit.

FUNCTIONS OF SWITCHES & INDICATORS

Power Switch

OFF	Power to the processor is OFF.
POWER	Power to the processor is ON, and all console switches function normally.
LOCK	Power to the processor is ON, but the seven control switches LOAD ADRS through START are disabled. All other switches are functional.

Control Switches

When a LOAD ADRS switch is depressed, the contents of the Switch Register are loaded into the ADDRESS Display. The address displayed in the Address Display Lights is a function of the position of the ADDRESS Select switch.

EXAM (Examine)

Depressing the EXAM switch causes the contents of the current location specified in the Address Display to be displayed in the DATA Display Register when the DATA Select switch is in the DATA PATHS position. The address in the Address Display will be mapped or unmapped depending on the position of the ADDRESS Select switch. The location displayed in the Address Display Lights is also a function of that switch.

DEP (Deposit)

Raising the DEP switch causes the current contents of the Switch Register to be deposited into the address specified by the current contents of the Address Display.

The address in the Address Display will be mapped or unmapped depending on the position of the ADDRESS Select switch. The location displayed in the Address Display Lights is also a function of that switch.

CONT (Continue)

Depressing the CONT switch causes the CPU to resume execution. The CONT switch has no effect when the CPU is in RUN state.

ENABLE/HALT
The ENABLE/HALT switch is a two position switch used to stop machine execution and to enable the system to run.

S INST/S BUS CYCLE (Single Instruction/Single Bus Cycle)
The S INST/S BUS CYCLE switch affects only the operation of the CONT switch. It controls whether the machine stops after instructions or bus cycles. This switch has no effect on any switches when the ENABLE/HALT switch is set to ENABLE.

START
The functions of the START switch depend on the setting of the ENABLE/HALT switch as follows:

ENABLE	Starts execution
HALT	Clears the computer system

Switch Register
The switches are used to manually load data or an address into the processor, as determined by the control switches and the ADDRESS Select switch.

Note that bits 0 to 15 of the current setting of the Switch Register may be read under program control from a read-only register at address 17 777 570.

Lamp Test
The Lamp Test switch (which is not labeled) is located between the Switch Register and the LOAD ADRS switch. It is used for maintenance purposes. When the Lamp Test switch is raised, all console indicator lights should go on. An indicator which does not light is defective and should be replaced.

Address Select Switch

VIRTUAL (6-position for User, Supervisor, & Kernel)	Uses a 16-bit Virtual Address where bits 16 to 21 are always OFF
CONS PHY (Console Physical)	Uses a 22-bit Physical Address to perform console operations (e.g., LOAD ADRS, EXAM, & DEP).
PROG PHY (Program Physical)	Displays the 22-bit Physical Address of the current bus cycle that was generated by the Memory Management Unit.

Address Display
The ADDRESS Display lights are used to show the address of data being examined or just deposited. The address is interpreted as a Virtual or Physical Address as determined by the ADDRESS Select switch.

Data Select Switch

DATA PATHS	The normal display mode, shows examined or deposited data.
BUS REG	The internal CPU register used for bus cycles.
μADRS FPP/CPU	The ROM address, FPP control microprogram (bits 15 to 8) and the CPU control microprogram (bits 7 to 0).
DISPLAY REGISTER	The contents of the Display Register. This has an address of 17 777 570.

Data Display
The Data Display lights are used to show the 16-bit word data just examined or deposited, or other data within the CPU. The PARITY HIGH & LOW lights indicate the parity bit for the respective bytes on read operations; on write operations the bits are off. The interpretation of the data is determined by the DATA Select switch.

Status Indicator Lights
ERROR INDICATORS

PAR ERR	Lights to indicate a parity error during a reference to memory.
ADRS ERR	Lights to indicate any of the following addressing errors: a) Reference to non-existent memory b) Access control violation c) Reference to unassigned memory pages

PROCESSOR STATE

RUN	The CPU is executing program instructions. If the instruction being executed is a WAIT instruction, the RUN light will be on. The CPU will proceed from the WAIT on receipt of an external interrupt, or on console intervention.

PAUSE	The CPU is inactive because the current instruction execution has been completed as far as possible without more data from the UNIBUS or memory, or the CPU is waiting to regain control of the the UNIBUS (UNIBUS mastership).
MASTER	The CPU is in control of the UNIBUS (UNIBUS Master only when it needs the UNIBUS). The CPU relinquishes control of the UNIBUS during DMA and NPR data transfers.
MODE	
USER	The CPU is executing program instructions in User mode.
SUPER (Supervisor)	The CPU is executing program instructions in Supervisor mode.
KERNEL	The CPU is executing program instructions in Kernel mode.
DATA	If on, the last memory reference was to D address space in the current CPU mode. If off, the last memory reference was to I address space in the current mode.
ADDRESS	
16 bit	Lights when the CPU is using 16-bit mapping.
18 bit	Lights when the CPU is using 18-bit mapping.
22 bit	Lights when the CPU is using 22-bit mapping.

M9301-YC, -YH/M9312 BOOTSTRAP LOADER

Features

- Contains bootstrap routines for a wide range of storage media
- Allows bootstrapping of any drive unit on a particular controller
- Runs diagnostic programs to test the basic CPU, Cache, and Main Memory
- Allows booting to selected physical memory segments in 32K increments

- Switch-selectable default loading device

Description

The M9312 and M9301-YC, -YH are dedicated diagnostic bootstrap loaders for use with the PDP-11/70. They contain a ROM organized as 512 16-bit words which are separated into hardware verification programs and bootstrap routines. They are double-height extended modules which occupy rows E and F of slot one in the PDP-11/70 CPU.

DIAGNOSTICS (M9312)

The M9312 provides basic diagnostic tests for the CPU, memory, and cache when used with PDP-11/60 and PDP-11/70 computers. All diagnostic tests reside in ROM (read-only memory) locations 765000 through 765776 (console emulator routine is eliminated.) These diagnostics test the basic CPU including the branches, the registers, all addressing modes, and many of the instructions in the PDP-11 repertoire. Memory from virtual address 1000 to the highest available address up to 28K will also be checked. After main memory has been verified, with the cache off, the cache memory will be tested to verify that hits occur properly. Main memory will be scanned again to ensure that the cache is working properly throughout the 28K of memory to be used in the boot operation. If one of the cache memory tests fails, the operator can attempt to boot the system anyway by pressing CONTINUE. This will cause the program to force misses in both groups of the cache before going to the bootstrap section of the program. The following is a list of M9312 diagnostic tests.

TEST 1	This test verifies the unconditional branch.
TEST 2	Test CLR, MODE 0, and BMI, BVS, BHI, BLT, BLOS
TEST 3	Test DEC, MODE 0, and BPL, BEQ, BGE, BLE
TEST 4	Test ROR, MODE 0, and BVC, BHIS, BNE
TEST 5	Test register data path.
TEST 6	Test ROL, BCC, BLT
TEST 7	Test ADD, INC, COM, and BCS, BLE
TEST 10	Test ROR, DEC, BIS, ADD, and BLO
TEST 11	Test COM, BIC, and BGT, BLE
TEST 12	Test SWAB, CMP, BIT, and BNE, BGT
TEST 13	Test MOVB, SOB, CLR, TST and BPL, BNG
TEST 14	Test JSR, RTS, RTI, and JMP

TEST 15	Test main memory from virtual 001000 to last address. Cache memory diagnostic tests.
TEST 16	Test cache data memory.
TEST 17	Test memory with the data cache on.

DIAGNOSTICS (M9301-YC, -YH)

The diagnostic portion of the program will test the basic CPU, including the branches, the registers, all addressing modes, and most of the instructions in the PDP-11 repertoire. It will then set the stack pointer to kernel D-space PAR 7. It will also turn on, if requested, memory management and the UNIBUS map, and will check memory from virtual address 1000 to 157776. After main memory has been verified, with the cache off, the cache memory will be tested to verify that hits occur properly. Main memory will be scanned again to ensure that the cache is working properly throughout the 28K of memory to be used in the boot operation.

If one of the cache memory tests fails, the operator can attempt to boot the system anyway by pressing CONTINUE. This will cause the program to force misses in both groups of the cache before going to the bootstrap section of the program.

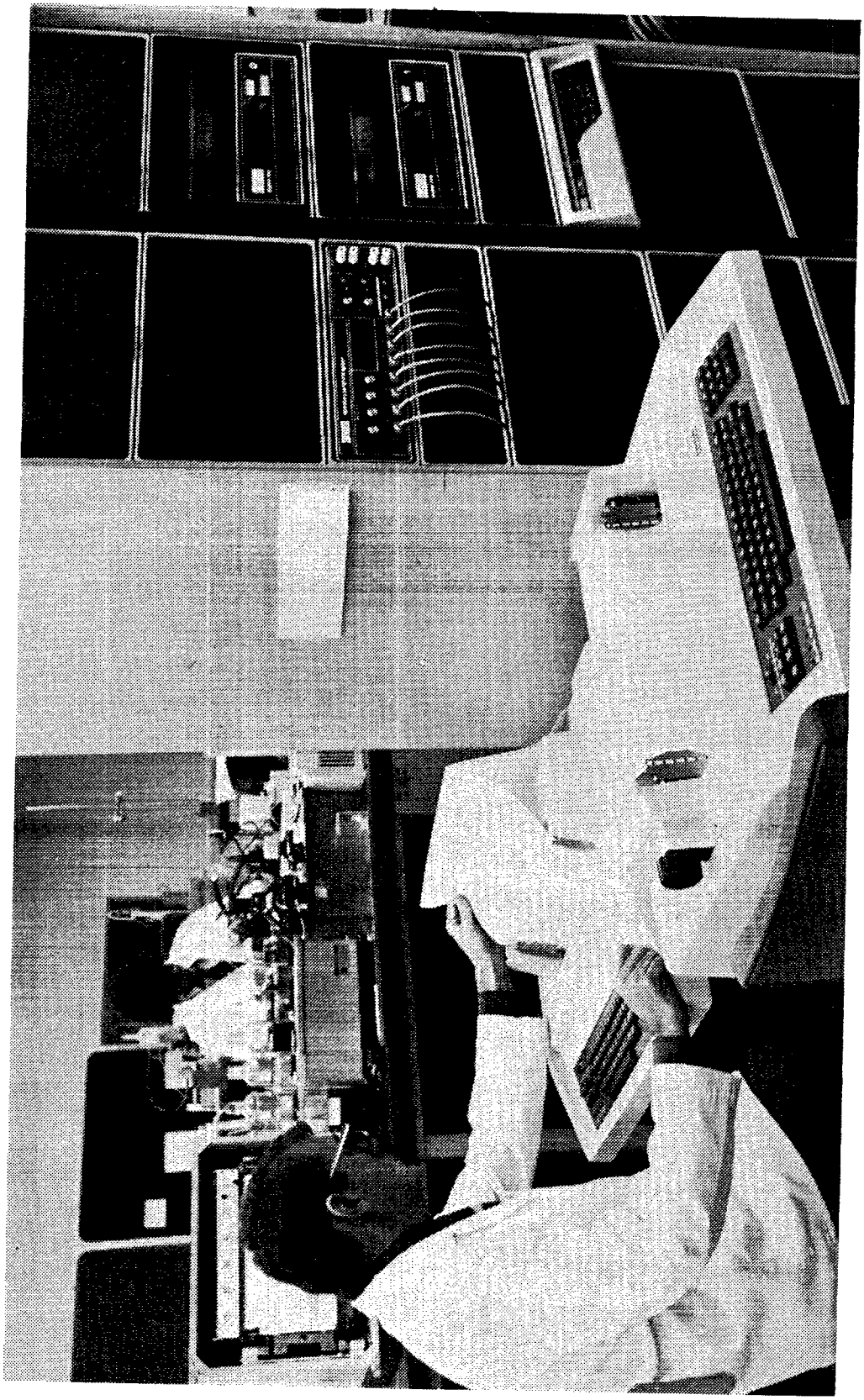
A listing of the M9301-YC, -YH diagnostic tests follows.

TEST 1	This test verifies the unconditional branch
TEST 2	Test CLR, MODE 0, and BMI, BVS, BHI, BLOS
TEST 3	Test DEC, MODE 0, and BPL, BEQ, BGE, BGT, BLE
TEST 4	Test ROR, MODE 0, and BVC, BHIS, BHI, BNE
TEST 5	Test BHI, BLT, and BLOS
TEST 6	Test BLE and BGT
TEST 7	Test register data path and modes 2, 3, 6
TEST 10	Test ROL, BCC, BLT, and MODE 6
TEST 11	Test ADD, INC, COM, and BCS, BLE
TEST 12	Test ROR, BIS, ADD, and BLO, BGE
TEST 13	Test DEC and BLOS, BLT
TEST 14	Test COM, BIC, and BGT, BGE, BLE
TEST 15	Test ADC, CMP, BIT, and BNE, BGT, BEQ
TEST 16	Test MOVB, SOB, CLR, TST and BPL, BNE
TEST 17	Test ASR, ASL

TEST 20	Test ASH, and SWAB
TEST 21	Test 16 Kernel PARs
TEST 22	Test and load KIPDRs
TEST 23	Test JSR, RTS, RTI, and JMP
TEST 24	Load and turn on memory management and the UNIBUS map
TEST 25	Test main memory from virtual 1000 to 28K
TEST 26	Test cache data memory
TEST 27	Test virtual 28K with cache on

ERROR RECOVERY

If the processor halts in one of the two cache tests, the error is recoverable. By pressing CONTINUE, the program will either attempt to finish the test (if at either 17 773 644 or 17 773 736) or force misses in both groups of the cache and attempt to boot the system monitor with the cache fully disabled (if at 17 773 654, 17 773 746, or 17 773 764). The run time for this program is approximately three seconds.



CHAPTER 11
FLOATING POINT PROCESSORS

The floating point processor is an option available for all members of the PDP-11 family except the 11/03 and 11/04. A floating point processor (FPP) is much faster and more effective for high speed numerical data handling than software floating point routines. Users who are programming in FORTRAN, BASIC, and APL find that the FPP gives them the speed and capability that they require for data and number manipulation.

There are four FPPs available for the PDP-11 family: the FP11-A, used with the PDP-11/34A; the FP11-C, used with the PDP-11/70; the FP11-E, used with the PDP-11/60; and the FP11-F, used with the PDP-11/44.

FPPs perform all floating point arithmetic operations and convert data between integer and floating point formats.

Features of the floating point processors are:

- 17-digit precision in 64-bit mode, 8 in 32-bit mode
- overlapped operation with the central processor (FP11-C and FP11-E)
- high speed operation
- single and double precision (32- or 64-bit) floating point modes
- flexible addressing modes
- six 64-bit floating point accumulators
- error recovery aids

ARCHITECTURE

The floating point processors contain scratch registers, a floating exception address pointer (FEA), a program counter, a set of status and error registers, and six general purpose accumulators, AC0-AC5.

The accumulators are 32 or 64 bits long, depending on the instruction and on the FPP status. In a 32-bit instruction, only the left-most 32 bits are used.

The six floating point accumulators are used in numeric calculations and in inter-accumulator data transfers. The first four accumulators (AC0-AC3) are also used for all data transfers between the FPP and the general registers, or memory.

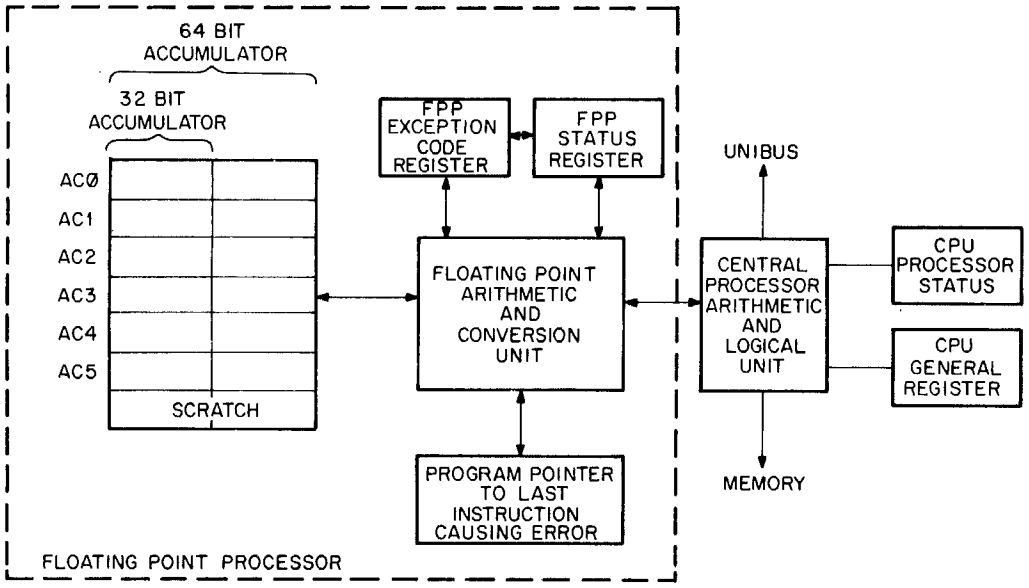


Figure 11-1 Floating Point Processor

OPERATION

A floating point processor functions as an integral part of the central processor. It operates using similar address modes, and using the same memory management facilities provided by the memory management option. FPP instructions can reference the floating point accumulators, the central processor's general registers, or any location in memory.

The FP11-C and the FP11-E overlap operation with the central processor. When an FPP instruction is fetched from memory, the FPP will execute that instruction in parallel with the CPU as the CPU continues *its* instruction sequence. The CPU is delayed a very short period of time during the FPP instruction fetch operation, and then is free to proceed independently of the FPP. The interaction between the two processors is automatic, permitting a program to take full advantage of the parallel operation of the two processors, by the intermixing of FPP and CPU instructions. This is all accomplished by the hardware of the processors. When an FPP instruction is encountered in a program, the CPU first initiates floating point handshaking and calculates the address of the operand. It then checks the status of the FPP. If the FPP is busy, the CPU waits until it receives a done signal before continuing execution of the program. For example:

```
LDD(R3)+,AC3        ;Pick up constant operand and
                    ;place it in AC3
```

```
ADDLP:   LDD(R3)+,AC0      ;Load AC0 with next value
                           ;in table
         MUL AC3,AC0       ;and multiply by constant
                           ;in AC3
         ADDD AC0,AC1      ;and add the result into AC1
         SOB R5,ADDLP      ;check to see whether done
         STCDI AC1@R4      ;done, convert double
                           ;to integer and store.
```

In this example, the FPP executes the first three instructions. After the ADD is fetched into the FPP, the CPU will execute the SOB, calculate the effective address of the STCDI instruction, and then wait for the FPP to be done with the ADDD before continuing past the STCDI instruction. Autoincrement and autodecrement addressing automatically adds or subtracts the correct amount to the contents of the register, depending on the modes represented by the instruction.

FLOATING POINT DATA FORMATS

A floating point number is defined as having the form (2**K)f, where K is an integer and f is a fraction. For a non-vanishing number, K and f are uniquely determined by imposing the condition $1/2 \leq f < 1$. The fractional part, f, of the number is said to be normalized. For the number zero, f must be assigned the value 0, and the value of K is indeterminate.

The FPP data formats are derived from this mathematical representation for floating point numbers. Two types of floating point data are provided: single precision, or floating mode, where the word is 32 bits long; and double precision, or double mode, where the word is 64 bits long. Sign magnitude notation is used.

Non-Vanishing Floating Point Numbers

The fractional part f is assumed normalized, so that its most significant bit must be 1. This 1 is the hidden bit; it is not stored in the data word, but the hardware restores it before carrying out arithmetic operations. The floating and double modes reserve 23 and 55 bits respectively for f, which with the hidden bit imply effective word lengths of 24 bits and 56 bits for precise arithmetic operations.

Eight bits are reserved for the storage of the exponent K in excess 128 (200_8) notation (i.e., as $K+200_8$). Thus exponents from -128 to $+127$ can be represented by 0 to 377_8, or 0 to 255_{10}. For reasons given below, a biased EXP of 0 (true exponent of -200_8 is reserved for floating point zero. Thus exponents are restricted to the range -127 to $+127$ inclusive (-117_8 to 177_8) or, in excess 200_8 notation, 1 to 377_8. The remaining bit of the floating point word is the sign bit.

Floating Point Zero

Because of the hidden bit, the fractional part is not available to distinguish between zero and non-vanishing numbers whose fractional part is exactly 1/2. Therefore, the FPP reserves a biased exponent of 0 for this purpose. Any floating point number with biased exponent of 0 either traps or is treated as if it were an exact 0 in arithmetic operations. An exact zero is represented by a word in which the bits are all 0s. An arithmetic operation in which the resulting true exponent exceeds 177_8 is regarded as producing a floating overflow; if the true exponent is less than -177_8 the operation is regarded as producing a floating underflow. A biased exponent of 0 can thus arise from arithmetic operations as a special case of underflow (true exponent = 0). Recall that only eight bits are reserved for the biased exponent. The fractional part of the results obtained from such overflows and underflows is correct.

The Undefined Variable

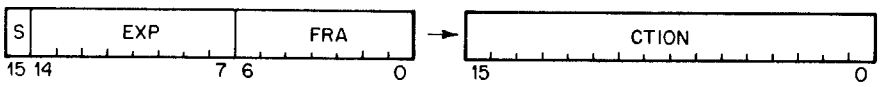
The undefined variable is any bit pattern with a sign bit of one and a biased exponent of zero. The term undefined variable is used to indicate that these bit patterns are not assigned a corresponding floating point arithmetic value. An undefined variable is frequently referred to as "−0" elsewhere in this chapter.

The FPP design assures that the undefined variable will not be stored as the result of any floating point operation in a program run with the overflow and underflow interrupts disabled. This is achieved by storing an exact zero on overflow or underflow, if the corresponding interrupt is disabled. This feature, together with an ability to detect a reference to the undefined variable, is intended to provide the user with a debugging aid. If a −0 is generated, it is not a result of a previous floating point arithmetic instruction.

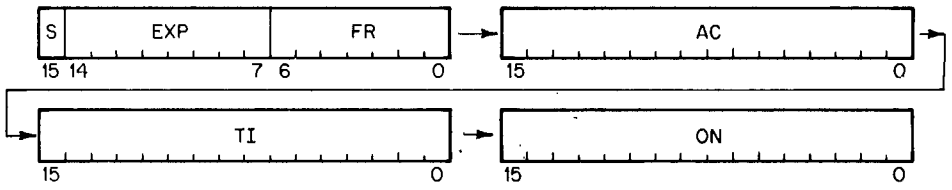
FLOATING POINT DATA

Floating point data is stored in words of memory as illustrated below.

F Format, single precision



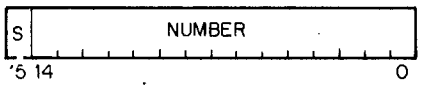
D Format, double precision



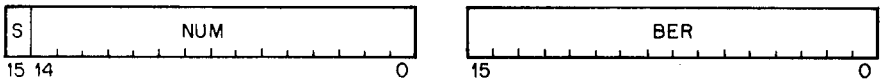
S = Sign of fraction
EXP = Exponent in excess 200 notation, restricted to 1 to 377 octal for non-vanishing numbers.
FRACTION = 23 bits in F Format, 55 bits in D Format, + one hidden bit (normalization). The binary radix point is to the left.

The FPP provides for conversion of floating point to integer format and vice-versa. The processor recognizes single precision integer (I) and double precision integer long (L) numbers, which are stored in standard 2's complement form:

I Format



L Format



where

S = Sign of number
NUMBER = 15 bits in I Format, 31 bits in L Format.

FLOATING POINT UNIT STATUS REGISTER (FPS REGISTER)

This register provides mode and interrupt control for the floating point unit, and conditions resulting from the execution of the previous instruction.

Four bits of the FPS register control the modes of operation:

- Single/Double: Floating point numbers can be either single or double precision.

- Long/Short: Integer numbers can be 16 bits or 32 bits.
- Chop/Round: The result of a floating point operation can be either chopped or rounded. The term chop is used instead of truncate in order to avoid confusion with truncation of series used in approximations for function subroutines.
- Normal/Maintenance: A special maintenance mode is available on the FP11-C and FP11-E.

The FPS register contains an error flag and four condition codes (5 bits): carry, overflow, zero, and negative, which are equivalent to the CPU condition codes.

The floating point processor recognizes seven floating point exceptions:

- detection of the presence of the undefined variable in memory
- floating overflow
- floating underflow
- failure of floating to integer conversion
- maintenance trap
- attempt to divide by zero
- illegal floating OP code

For the first five of these exceptions, bits in the FPS register are available to enable or disable interrupts individually. An interrupt on the occurrence of either of the last two exceptions can be disabled only by setting a bit which disables interrupts on all seven of the exceptions as a group.

Of the fourteen bits described above, five, the error flag and condition codes, are set by the FPP as part of the output of a floating point instruction. Any of the mode and interrupt control bits (except the FP11-C and FP11-E, FMM bit) may be set by the user; the LDFS instruction is available for this purpose. These fourteen bits are stored in the FPS register as follows:

Bit	**Name**
15	Floating Error (FER)

Description

The FER bit is set by the FPP if:

1. Division by zero occurs.
2. Illegal OP code occurs.
3. Any one of the remaining occurs and the corressponding interrupt is enabled.

Note that the above action is independent of whether the FID bit (next item) is set or clear.

Note also that the FPP never resets the FER bit. Once the FER bit is set by the FPP, it can be cleared only by an LDFPS instruction or by the RESET instruction. This means that the FER bit is up-to-date only if the most recent floating point instruction produced a floating point exception.

Bit	**Name**
14	Interrupt Disable (FID)

Description

If the FID bit is set, all floating point interrupts are disabled. Note that if an individual interrupt is simultaneously enabled, only the interrupt is inhibited; all other actions associated with the individual interrupt enabled take place.

NOTE

The FID bit is primarily a maintenance feature. Normally, it should be clear. In particular, it must be clear if you wish to assure that storage of −0 by the FPP is always accompanied by an interrupt.

Through the rest of this chapter, it is assumed that the FID bit is clear in all discussions involving overflow, underflow, occurrence of −0, and integer conversion errors.

Bit	**Name**
13	Not Used

Bit	**Name**
12	Not Used

Bit	**Name**
11	Interrupt on Undefined Variable (FIUV)

Description

An interrupt occurs if FIUV is set and a −0 is obtained from memory as an operand of ADD, SUB, MUL, DIV, CMP, MOD, NEG, ABS, TST, or any LOAD instruction. The interrupt occurs before execution except on NEG and ABS instructions. For these instructions, the interrupt occurs after execution. When FIUV is reset, −0 can be loaded and used in any FPP operation. Note that the interrupt is not activated by the presence of −0 in an AC operand of an arithmetic instruction. In particular, trap on −0 never occurs in mode 0.

The FPP will not store a result of −0 without the simultaneous occurrence of an interrupt.

Bit	Name
10	Interrupt on Underflow (FIU)

Description
When the FIU bit is set, floating underflow will cause an interrupt. The fractional part of the result of the operation causing the interrupt will be corrected. The biased exponent will be too large by 400_8, except for the special case of 0, which is correct. An exception is discussed in the detailed description of the LDEXP instruction.

If the FIU bit is reset and if underflow occurs, no interrupt occurs and the result is set to exact 0.

Bit	Name
9	Interrupt on Overflow (FIV)

Description
When the FIV bit is set, floating overflow will cause an interrupt. The fractional part of the result of the operation causing the overflow will be correct. The biased exponent will be too small by 400_8.

If the FIV bit is reset, and overflow occurs, there is no interrupt. The FPP returns exact 0. Special cases of overflow are discussed in the detailed descriptions of the MOD and LDEXP instructions.

Bit	Name
8	Interrupt on Integer Conversion Error (FIC)

Description
When the FIC bit is set, and a conversion to integer instruction fails, an interrupt will occur. If the interrupt occurs, the destination is set to 0, and all other registers are left untouched.

If the FIC bit is reset, the result of the operation will be the same as explained above, but no interrupt will occur.

The conversion instruction fails if it generates an integer with more bits than can fit in the short or long integer word specified by the FL bit (see bit 6 below).

Bit	Name
7	Floating Double Precision Mode (FL)

Description
Determines the precision that is used for floating point calculations. When set, double precision is selected; when reset, single precision is used.

Bit	Name
6	Floating Long Integer Mode (FL)

Description
Active in conversion between integer and floating point format. When set, the integer format selected is double precision 2's complement (i.e., 32 bits). When reset, the integer format is assumed to be single precision 2's complement (i.e., 16 bits).

Bit	Name
5	Floating Chop Mode (FT)

Description
When bit FT is set, the result of any arithmetic operation is chopped (or truncated).

When reset, the result is rounded.

Bit	Name
4	Floating Maintenance Mode (FMM) (FP11-C and FP11-E)

Description
This mode is a maintenance feature. Refer to the maintenance manual for the details of its operation. The FMM bit can be set only in kernel mode.

Bit	Name
3	Floating Negative (FN)

Description
FN is set if the result of the last operation was negative, otherwise it is reset.

Bit	Name
2	Floating Zero (FZ)

Description
FZ is set if the result of the last operation was zero; otherwise it is reset.

Bit	Name
1	Floating Overflow (FV)

Description
FV is set if the last operation resulted in an exponent overflow; otherwise it is reset.

Bit	Name
0	Floating Carry (FC)

Description
FC is set if the last operation resulted in a carry of the most significant bit. This can occur only in floating or double to integer conversions.

FLOATING EXCEPTION CODE AND ADDRESS REGISTERS

One interrupt vector is assigned to take care of all floating point exceptions (location 244). The seven possible errors are coded in the 4-bit FEC (Floating Exception Code) register as follows:

2	Floating OP code error
4	Floating divide by zero
6	Floating (or double) to integer conversion error
8	Floating overflow
11	Floating underflow
12	Floating undefined variable
14	Maintenance trap

The address of the instruction producing the exception is stored in the FEA (Floating Exeception Address) register.

The FEC and FEA registers are updated when one of the following occurs:

- divide by zero
- illegal OP code
- any of the other five exceptions with the corresponding interrupt enabled

If one of the five exceptions occurs with the corresponding interrupt disabled, the FEC and FEA are not updated. Inhibition of interrupts by the FID bit does not inhibit updating of the FEC and FEA, if an exception occurs. The FEC and FEA are not updated if no exception occurs. This means that the STST (store status) instruction will return current information only if the most recent floating point instruction produced an exception. Unlike the FPS register, no instructions are provided for storage into the FEC and FEA registers.

FLOATING POINT PROCESSOR INSTRUCTION ADDRESSING

Floating point processor instructions use the same type of addressing as do the central processor instructions. A source or destination operand is specified by designating one of eight addressing modes and one of eight central processor general registers to be used in the specified mode. The modes of addressing are the same as those of the central processor except for mode 0. In mode 0 the operand is located in the designated floating point processor accumulator, rather than in a central processor general register. The modes of addressing are:

0 = Direct Accumulator
1 = Deferred
2 = Autoincrement

3 = Autoincrement deferred
4 = Autodecrement
5 = Autodecrement deferred
6 = Indexed
7 = Indexed deferred

Autoincrement and autodecrement operate on increments and decrements of 4 for F Format and 10 for D Format.

In mode 0, you can use all six FPP accumulators (AC0-AC5) as your source or destination. In all other modes, which involve transfer of data from memory or the general register, you are restricted to the first four FPP accumulators (AC0-AC3).

In immediate addressing (mode 2, R7) only 16 bits are loaded or stored.

ACCURACY

This section contains some general comments on the accuracy of the FPP. The descriptions of the individual instructions include their accuracy. An instruction or operation is regarded as exact if the result is identical to an infinite precision calculation involving the same operands. All arithemetic instructions treat an operand whose biased exponent is 0 as an exact 0 (unless FIUV is enabled and the operand is −0, in which case an interrupt occurs). For all arithmetic operations, except DIV, a zero operand implies that the instruction is exact. The same statement applies to DIV if the zero operand is the dividend, but if it is the divisor, division is undefined and an interrupt occurs.

For non-vanishing floating point operands, the fractional part is binary normalized. It contains 24 bits for floating mode and 56 bits for double mode. The internal hardware registers contain 60 bits for processing the fractional parts of the operands, of which the high order bit is reserved for arithmetic overflow. There are, internally, 35 guard bits for floating mode and 3 guard bits for double mode arithmetic operations. For ADD, SUB, MUL, and DIV, two guard bits are necessary and sufficient to guarantee return of a chopped or rounded result identical to the corresponding infinite precision operation, chopped or rounded to the specified word length. Thus, with two guard bits, a chopped result has an error bound of one least significant bit (LSB), a rounded result has an error bound of 1/2 LSB. To obtain the corresponding statements on accuracy for a radix other than 2, replace references to *bit* in the two preceding sentences with the word *digit.* These error bounds are realized for most instructions. For the addition of operands of opposite sign or for the subtraction of operands of the same sign in rounded double precision, the error bound is 3/4 LSB (FP11-C

and FP11-E), or 33/64 (FP11-A, and FP11-F) which is slightly larger than the 1/2 LSB error bound for all other rounded operations.

The error bound for the FP11-C differs from the FP11-A, since the FP11-C and FP11-E carry three guard bits while the FP11-A and FP11-F carry seven guard bits.

In the rest of this chapter, an arithmetic result is called exact if no non-vanishing bits would be lost by chopping. The first bit lost in chopping is referred to as the rounding bit. The value of a rounded result is related to the chopped result as follows:

- If the rounding bit is one, the rounded result is the chopped result incremented by an LSB (least significant bit).
- If the rounding bit is zero, the rounded and chopped results are identical.

It follows that:

- If the result is exact
  rounded value = chopped value = exact value
- If the result is not exact, its magnitude
  - is always decreased by chopping
  - is decreased by rounding if the rounding bit is zero
  - is increased by rounding if the rounding bit is one

Occurrence of floating point overflow and underflow is an error condition. The result of the calculation cannot be correctly stored because the exponent is too big to fit into the eight bits reserved for it. However, the internal hardware produces the correct answer. For the case of underflow, replacement of the correct answer by zero is a reasonable resolution of the problem for many applications. This is done on the FPP if the underflow interrupt is disabled. The error incurred by this action is an absolute rather than a relative error. It is bounded (in absolute value) by 2^{-128}. There is no such simple resolution for the case of overflow. The action taken, if the overflow interrupt is disabled, is described under FIV (bit 9).

The FIV and FIU bits (of the floating point status word) provide you with an opportunity to implement your own fix-up of an overflow or underflow condition. If such a condition occurs and the corresponding interrupt is enabled, the hardware stores the fractional part and the low eight bits of the biased exponent. The interrupt will take place and you can identify the cause by examination of the FV (floating overflow) bit or the FEC (floating exception) register. You can readily verify that (for the standard arithmetic operations ADD, SUB, MUL, and DIV) the biased exponent returned by the hardware bears the following relation to the correct exponent generated by the hardware:

- on overflow: it is too small by 400_8
- on underflow: if the biased exponent is 0, it is correct. If it is not 0, it is too large by 400_8

Thus, with the interrupt enabled, enough information is available to determine the correct answer. You may, for example, rescale your variables (via STEXP and LDEXP) to continue your calculation. Note that the accuracy of the fractional part is unaffected by the occurrence of underflow or overflow.

FLOATING POINT INSTRUCTIONS

Each instruction that references a floating point number can operate on either floating or double precision numbers, depending on the state of FD mode bit. Similarly, there is a mode bit FL that determines whether 32-bit integers (FL = 1) or 16-bit integers (FL=0) are used in conversion between integer and floating point representation. FSRC and FDST use floating point addressing modes; SRC and DST use CPU addressing modes.

In the descriptions of the floating point instructions, the operations of the FP11-A, FP11-E, FP11-F, and FP11-C are identical, except where explicitly stated otherwise.

Floating Point Instruction Format

Mnemonic	Description
OC	Op Code = 17
FOC	Floating Op Code
AC	Accumulator
FSRC, FDST	Use FPP Address Modes
SRC, DST	Use CPU Address Modes
f	Fraction
XL	Largest fraction that can be represented: 1-2**(−24), FD=0, single precision 1-2**(−56), FD=1, double precision
XLL	Smallest number that is not identically zero = 2**(−128)−2** (−127))*J(½)
XUL	Largest number that can be represented = 2**(127)*XL
JL	Largest integer that can be represented: 2**(15)−1 if FL=0 2**(31)−1 if FL=1

Mnemonic	Description
ABS (address)	Absolute value of (address)
EXP (address)	Biased exponent of (address)
<	Less than
≤	Less than or equal
>	Greater than
≥	Greater than or equal
LSB	Least significant bit

Mnemonic/ Name	Code	Operation	Condition Codes
ABSF ABSD Make Absolute Floating/Double	1706FDST	If (FDST) < 0 FDST ← − (FDST). If EXP (FDST) = 0, FDST ← exact 0. For all other cases, FDST ← (FDST).	FC ← 0. FV ← 0. FZ ← 1 if EXP(FDST) = 0, else FZ ← 0. FN ← 0

Description: Set the contents of FDST to its absolute value.

Interrupts: If FIUV is set; trap on −0 occurs after execution.

Overflow and underflow cannot occur.

Accuracy: These instructions are exact.

Special Comments: If a −0 is present in memory and the FIUV bit is enabled, then the FP11-E and integral floating point unit store exact 0 in memory (zero exponent, zero fraction, and positive sign). The condition code reflects an exact 0 (FZ ← 1).

Mnemonic/ Name	Code	Operation	Condition Codes
ADDD Add Floating/Double	172ACFS-RO	Let SUM = (AC) + (FSRC): If underflow occurs and FIU is not enabled, AC ← exact 0.	FC ← 0. FV ← 1 if overflow occurs, else FV ← 0. FZ ← 1 if (AC) = 0, else FZ ← 0.

Mnemonic/ Name	Code	Operation	Condition Codes
		If overflow occurs and FIV is not enabled, AC ← exact 0. For all other cases, AC ← SUM.	FN ← 1 if (AC) < 0, else FN ← 0.

Description:

Add the contents of FSRC to the contents of AC. The addition is carried out in single or double precision and is rounded or chopped in accordance with the values of the FD and FT bits in the FPS register. The result is stored in AC except for:

- overflow with interrupt disabled
- underflow with interrupt disabled

For these exceptional cases, an exact 0 is stored in AC.

Interrupts:

If FIUV is enabled, trap on −0 in FSRC occurs before execution.

If overflow or underflow occurs and if the corresponding interrupt is enabled, the trap occurs with the faulty result in AC. The fractional parts are correctly stored. The exponent part is too large by 400_8 for underflow, except for the special case of 0, which is correct.

Accuracy:

Errors due to overflow and underflow are described above. If neither occurs, then for oppositely signed operands with exponent differences of 0 or 1, the answer returned is exact if a loss of significance of one or more bits occurs. Note that these are the only cases for which loss of significance of more than one bit can occur. For all other cases, the result is inexact with error bounds of:

- 1 LSB in chopping mode with either single or double precision
- 3/4 LSB (FP11-C and E) or 33/64 LSB (FP11-A and FP11-F) in rounding mode with double precision

Special Comment: The undefined variable −0 can occur only in conjunction with overflow or underflow. It will be stored in AC only if the corresponding interrupt is enabled.

Mnemonic/ Name	Code	Operation	Condition Codes
CFCC Copy Floating Condition Codes	170000	C ← FC V ← FV Z ← FZ N ← FN	

Description: Copy FPP condition codes into the CPU's condition codes.

Mnemonic/ Name	Code	Operation	Condition Codes
CLRF CLRD Clear Floating/Double	1704FDST	FDST ← exact 0	FC ← 0 FV ← 0 FZ ← 1 FN ← 0

Description: Set FDST to 0. Set FZ condition code and clear other condition code bits.

Interrupts: No interrupts will occur. Neither overflow nor underflow can occur.

Accuracy: These instructions are exact.

Mnemonic/ Name	Code	Operation	Condition Codes
CMPF CMPD Compare Floating/ Double	173 (AC+4) FSRC	(FSRC) (AC)	FC ← 0 FV ← 0 FZ ← 1 If (FSRC) − (AC) = 0, else FZ ←0 FN ← 1 If (FSRC) − (AC) < 0, else FN ←0

Description: Compare the contents of FSRC with the accumulator. Set the appropriate floating point condition codes. FSRJC and accumulator are left unchanged (see special comment below).

Interrupts: If FIUV is enabled, trap on −0 occurs before execution.

Accuracy: These instructions are exact.

Special Comment: An operand which has a biased exponent of zero is treated as if it were true zero. If both operands have biased exponents of zero, the accumulator gets a true zero and, hence, may be modified.

Mnemonic/ Name	Code	Operation	Condition Codes
DIVF DIVD Divide Floating/Double	174(AC + 4)FSRC	If EXP(FSRC) = 0, AC ← (AC): instruction is aborted. If EXP(AC) = 0, AC ← exact 0. For all other cases, let QUOT = (AC)/(FSRC): If underflow occurs and FIU is not enabled AC ← exact 0. For all remaining cases, AC ← QUOT.	FC ← 0 FV ← 1 if overflow occurs, else FV ← 0 FZ ← 1 if EXP(AC) = 0, else FZ ← 0 FN ← 1 if (AC) < 0, else FN ← 0

Description: If either operand has a biased exponent of 0, it is treated as an exact 0. For FSRC this would imply division by zero; in this case the instruction is aborted, the FEC register is set to 4, and an interrupt occurs. Otherwise the quotient is developed to single or double precision with enough guard bits for correct rounding. The quotient is rounded or chopped in accordance with the values of the FD and FT bits in the FPS register. The result is stored in AC except for:

- overflow with interrupt disabled
- underflow with interrupt disabled

For these exceptional cases, an exact 0 is stored in the accumulator.

Interrupts: If FIUV is enabled, trap on −0 in FSRC occurs before execution.

If EXP(FSRC) = 0, interrupt traps on attempt to divide by 0.

If overflow or underflow occurs and if the corresponding interrupt is enabled, the trap occurs with the faulty result in AC. The fractional parts are correctly stored. The exponent part is too large by 400_8 for underflow, except for the special case of 0, which is correct.

Accuracy: Errors due to overflow, underflow, and division by 0 are described above. If none of these occurs, the error in the quotient will be bounded by 1 LSB in chopping mode and by ½ LSB in rounding mode.

Special Comments: The undefined variable −0 can occur only in conjunction with overflow or underflow. It will be stored in AC only if the corresponding interrupt is enabled.

Mnemonic/ Name	Code	Operation	Condition Codes
LDCDF LDCFD Load and Convert from Double to Floating or from Floating to Double	177(AC+4) FSRC	If EXP(FSRC) = 0, AC ← exact 0 If FD = 1 , FT = 0, FIV = 0 and rounding causes overflow, AC ← exact 0. In all other cases AC ← C_{xy} (FSRC), where C_{xy} specifies conversion from floating mode x to floating mode y. x = D, y = F if FD = 0 (single) x = F, y = D if FD = 1 (double)	FC ← 0 FV ← 1 if conversion produces overflow, else FV ← 0 FZ ← 1 if (AC) = 0, else FZ ← 0 FN ← 1 if (AC) < 0, else FN ← 0

Description: If the current mode is floating mode (FD = 0), the source is assumed to be a double precision number and is converted to single precision. If the floating chop bit (FT) is set, the number is chopped, otherwise the number is rounded.

If the current mode is double mode (FD = 1), the source is assumed to be a single-precision number,

and is loaded left-justified in the AC. The lower half of the AC is cleared.

Interrupts: If FIUV is enabled, trap on −0 occurs before execution.

Overflow cannot occur for LDCFD.

A trap occurs if FIV is enabled, and if rounding with LDCDF causes overflow; AC ← overflowed result of conversion. This result must be +0 or −0.

Underflow cannot occur.

Accuracy: LDCFD is an exact instruction. Except for overflow, described above, LDCDF incurs an error bounded by one LSB in chopping mode, and by ½ LSB in rounding mode.

Special Comment: If (FSRC) = −0, the FZ and FN bits are both set regardless of the condition of FIUV.

Mnemonic/ Name	Code	Operation	Condition Codes
LDCIF, LDCID LDCLF, LDCLD Load and Convert Integer or Long Integer to Floating or Double Precision	177ACSRC	AC ← C_{jx} (SRC), where C_{jx} specifies conversion from integer mode j to floating mode x; j = 1 if FL = 0 j = L if FL = 1 x = F if FD = 0 x = D if FD = 1	FC ← 0 FV ← 0 FZ ← 1 If (AC) = 0, else FZ ← 0 FN ← 1 If (AC) < 0, else FN ← 0

Description: Conversion is performed on the contents of SRC from a 2's complement integer with precision j to a floating point number of precision x. Note that j and x are determined by the state of the mode bits FL and FD: j = I or L, and x = F or D.

If a 32-bit integer is specified (L mode) and (SRC) has an addressing mode of 0, or immediate addressing mode is specified, the 16 bits of the source register are left-justified and the remaining 16 bits loaded with zeros before conversion.

In the case of LDCLF, the fractional part of the floating point representation is chopped or rounded to 24 bits for FT = 1 and 0 respectively.

Interrupts: Note: SRC is not floating point, so trap on −0 cannot occur.

Overflow and underflow cannot occur.

Accuracy: LDCIF, LDCID, and LDCD are exact instructions. The error incurred by LDCLF is bounded by 1 LSB in chopping mode, and by ½ LSB in rounding mode.

Mnemonic/ Name	**Code**	**Operation**	**Condition Codes**
LDEXP Load Exponent	176(AC+4) SRC	**Note:**177 and 200, appearing below, are octal numbers. If −200<SRC<200, EXP(AC) ←(SRC) + 200 and the rest of AC is unchanged. If SRC>177 and FIV is enabled, EXP(AC) ←(SRC) <6:0> on FP11-C, EXP(AC) ← ((SRC)+ 200) <7:0> on FP11-A, FP11-E, FP11-F. If SRC>177 and FIV is disabled, AC ← exact 0. If SRC <−177 and FIU is disabled, AC ← exact 0. If SRC <−177 and FIU is enabled, EXP(AC) ←(SRC) <6:0> on FP11-C, EXP(AC) ← ((SRC) + 200) <7:0> on FP11-A, FP11-E, FP11-F.	FC ← 0. FV ← 1 if SRC>177, else FV ← 0. FZ ← 1 if EXP(AC)=0, else FZ ← 0. FN ← 1 if (AC)<0, else FN ← 0.

Description Change AC so that its unbiased exponent = (SRC). That is, convert (SRC) from 2's complement to excess 200 notation, and insert in the EXP field of AC. This is a meaningful operation only if ABS(SRC) ≤ 177.

If SRC <−177, result is treated as overflow. If SRC<177, result is treated as underflow. Note that the FP11-C, FP11-F and FP11-A do not treat these abnormal conditions in exactly the same way.

Interrupts: No trap on −0 in AC occurs, even if FIUV enabled. If SRC > 177 and FIV enabled, trap on overflow will occur.
If SRC < −177 and FIU enabled, trap on underflow will occur.

The answers returned by the FP11-C, FP11-E, FP11-F, and FP11-A differ for overflow and underflow conditions.

Accuracy: Errors due to overflow and underflow are described above. If EXP(AC)=0 and SRC ≠ −200, (AC) changes from a floating point number treated as 0 by all floating arithmetic operations to a non-zero number. This is because the insertion of the "hidden" bit in the hardware implementation of arithmetic instructions is triggered by a non-vanishing value of EXP.

Mnemonic/ Name	**Code**	**Operation**	**Condition Codes**
LDF LDD Load Floating/Double	172(AC+4) FSRC	AC ← (FSRC)	FC ← 0 FV ← 0 FZ ← 1 if (AC) = 0, else FZ ← 0 FN ← 1 if (AC) < 0, else FN ← 0

Description: Load single or double precision number into accumulator.

Interrupts: If FIUV is enabled, trap on −0 occurs before AC is loaded. Neither overflow nor underflow can occur.

Accuracy: These instructions are exact and permit use of −0 in a subsequent floating point instruction if FIUV is not

enabled and (FSRC) = −0. If (FSRC) = −0, the FZ and FN bits are both set, regardless of the condition of FIUV.

Mnemonic/ Name	Code	Operation	Condition Codes
LDFPS Load FPP's Program Status	1701SRC	FPS ← (SRC)	

Description: Load FPP's status from SRC.

Special Comment: On the FP11-C, bits 13 and 12 are ignored. Bit 4 can be set if the CPU is in kernel mode.

On the FP11-A and FP11-F, the FPS is loaded with the source. The user is cautioned not to use bits 12 and 13 (in FP11-C, FP11-F, FP11-E, and the FP11-A) or bit 4 (in the FP11-A and FP11-F) for a special purpose since these bits are not recoverable by the STFPS instruction.

Mnemonic/ Name	Code	Operation	Condition Codes
MODF MODD Multiply and Integerize Floating/ Double	171(AC + 4) FSRC	See below	FC ← 0 FV ← 1 if PROD overflows, else FV ← 0 FZ ← 1 if (AC) = 0, else FZ ← 0 FN ← if (AC) < 0, else FN ← 0

Description and Operation: This instruction generates the product of its two floating point operands, separates the product into integer and fractional parts and then stores one or both parts as floating point numbers.

Let PROD = (AC)*(FSRC) so that in:
Floating point: ABS(PROD) = (2**K)*f where ½.LE.f.LT.1 and EXP(PROD) = $(200+K)_8$ Fixed Point binary: PROD = N + g, with
N = INT(PROD)=the integer part of PROD
and

g = PROD − INT(PROD) = the fractional part of PROD with $0 \leq g < 1$
Both N and g have the same sign as PROD.
They are returned as follows:

If AC is an even-numbered accumulator (0 or 2), N is stored in AC + 1 (1 or 3), and g is stored in AC.

If AC is an odd-numbered accumulator, N is not stored, and g is stored in AC.

The two statements above can be combined as follows: N is returned to ACv1 and g is returned to AC, where v means .OR.

Five special cases occur, as indicated in the following formal description with L = 56 for Double Mode:

1. If PROD overflows and FIV enabled:

   ACv1 ← N, chopped to L bits, AC ← exact 0.

   Note that EXP(N) is too small by 400_8, and that 0 can get stored in ACv1.

   If FIV is not enabled: ACv1 ← exact 0, AC ← exact 0, and −0 will never be stored.

2. If 2**L≤ABS(PROD) and no overflow:

   ACv1 ← N, chopped to L bits, AC ← exact 0.

   The sign and EXP of N are correct, but low order bit information, such as parity, is lost.

3. If 1≤ABS(PROD)<2**L:

   ACv1 ← N, AC ← g

   The integer part N is exact. The fractional part g is normalized, and chopped or rounded in accordance with FT. Rounding may cause return of ± unity for the fractional part. For L = 24, the error in g is bounded by 1 LSB in chopping mode and by ½ LSB in rounding mode. For L= 56, the error in g increases from the above limits as ABS(N) increases above 3 because only 59 bits of PROD are generated:

   if 2**p≤ABS(N)<2**(p + 1), with p > 2,

   the low order p − 2 bits of g may be in error.

4. If ABS (PROD)<1 and no underflow:

   ACv1 ← exact 0, AC ← g

There is no error in the integer part. The error in the fractional part is bounded by 1 LSB in chopping mode and ½ LSB in rounding mode. Rounding may cause a return of ± unity for the fractional part.

5. If PROD underflows and FIU enabled:

   ACv1 ← exact 0 AC ← g

   Errors are as in Case 4, except that EXP(AC) will be too large by 400_8 (except if EXP = 0, it is correct). Interrupt will occur and −0 can be stored in AC.

   IF FIU is not enabled, ACv1 ← exact 0 and AC ← exact 0. For this case, the error in the fractional part is less than 2**(−128).

Interrupts: If FIUV is enabled, trap on −0 in FSRC will occur before execution.

Overflow and underflow are discussed above.

Accuracy: Discussed above.

Applications:

1. Binary to decimal conversion of a proper fraction: the following algorithm, using MOD, will generate decimal digits D(1), D(2) ... from left to right:

```
Initialize:     I ← 0
                X ← number to be
                converted:
                ABS(X) < 1

While X ≠ 0 do
Begin PROD      ← X*10;
                I ← I + 1;
                D(I) ← INT(PROD);
                X ← PROD − INT(PROD);
                END;
```

   This algorithm is exact; it is case 3 in the description: the number of non-vanishing bits in the fractional part of PROD never exceeds L, and hence neither chopping nor rounding can introduce error.

2. To reduce the argument of a trigonometric function.

ARG*2/PI = N + g. The low two bits of N identify the quadrant, and g is the argument reduced to the first quadrant. The accuracy of N + g is limited to L bits because of the factor 2/PI. The accuracy of the reduced argument thus depends on the size of N.

3. To evaluate the exponential function e**x, obtain

   x*(log e base 2) = N + g.
   Then e**x = (2**N)* (e**(g*ln 2))

   The reduced argument is g*ln2<1 and the factor 2**N is an exact power of 2, which may be scaled in at the end via STEXP, ADD N to EXP and LDEXP. The accuracy of N + g is limited to L bits because of the factor (log e base 2). The accuracy of the reduced argument thus depends on the size of N.

Mnemonic/ Name	Code	Operation	Condition Codes
MULF MULD Multiply Floating/Double	171AC FSRC	Let PROD = (AC)* (FSRC) If underflow occurs and FIU is not enabled, AC ← exact 0. If overflow occurs and FIV is not enabled, AC ← exact 0. For all other cases AC ← PROD	FC ← 0. FV ← 1 if overflow occurs, else FV ← 0 FZ ← 1 if (AC) = 0, else FZ ← 0 FN ← 1 if (AC) < 0, else FN ← 0

Description:

If the biased exponent of either operand is zero, (AC) ← exact 0. For all other cases PROD is generated to 48 bits for floating mode and 59 bits for double mode. The product is rounded or chopped for FT = 0 and 1, respectively, and is stored in AC except for

- overflow with interrupt disabled
- underflow with interrupt disabled

For these exceptional cases, an exact 0 is stored in accumulator.

Interrupts: If FIUV is enabled, trap on −0 occurs before execution.

If overflow or underflow occurs and if the corresponding interrupt is enabled, the trap occurs with the faulty results in AC. The fractional parts are correctly stored. The exponent part is too small by 400_8 for underflow, except for the special case of 0, which is correct.

Accuracy: Errors due to overflow and underflow are described above. If neither occurs, the error incurred is bounded by 1 LSB in chopping mode and ½ LSB in rounding mode.

Special Comment: The undefined variable −0 can occur only in conjunction with overflow or underflow. It will be stored in AC only if corresponding interrupt is enabled.

Mnemonic/ Name	Code	Operation	Condition Codes
NEGF NEGD Negate Floating/Double	1707FDST	FDST ← −(FDST) if EXP(FDST) ≠0, else FDST ← exact 0.	FC ← 0. FV ← 0. FZ ← 1 If EXP(FDST) = 0, else FZ ← 0. FN ← 1 If (FDST) < 0, else FN ←0.

Description: Negate single or double precision number, store result in same location. (FDST)

Interrupts: If FIUV is enabled, trap on −0 occurs after execution.

Neither overflow nor underflow can occur.

Accuracy: These instructions are exact.

Special Comment: If a −0 is present in memory and the FIUV bit is enabled, then the FP11-E and the integral floating point unit store exact 0 in memory (zero exponent, zero fraction, and positive sign). The condition code reflects an exact 0 (FZ ← 1).

Mnemonic/ Name	Code	Operation	Condition Codes
SETF Set Floating Mode	170001	FD ← 0	

Description: Set the FPP in single precision mode.

Mnemonic/ Name	Code	Operation	Condition Codes
SETD Set Floating Double Mode	170011	FD ← 1	

Description: Set the FPP in double precision mode.

Mnemonic/ Name	Code	Operation	Condition Codes
SETI Set Integer Mode	170002	FL ← 0	

Description: Set the FPP for integer data.

Mnemonic/ Name	Code	Operation	Condition Codes
SETL Set Long Integer Mode	170012	FL ← 1	

Description Set the FPP for long integer data.

Mnemonic/ Name	Code	Operation	Condition Codes
STCFD STCDF Store and Convert from Floating to	176AC-FDST	If EXP(AC) = 0, FDST ← 0 and rounding causes overflow, FDST ← exact 0.	FC ← 0. FV ← 1 If conversion produces overflow else FV ← 0.

Mnemonic/ Name	Code	Operation	Condition Codes
Double or from Double to Floating		In all other cases, FDST ← C_{xy} (AC), where C_{xy} specifies conversion from floating mode x to floating mode y: x = F and y = D if FD = 0, x = D and y = F if FD = 1.	FZ ← 1 If (AC) = 0, else FZ ← 0. FN ← 1 If (AC) < 0, else FN ← 0.

Description: If the current mode is single precision, the accumulator is stored left-justified in FDST and the lower half is cleared. If the current mode is double precision, the contents of the accumulator are converted to single precision, chopped or rounded depending on the state of FT, and stored in FDST.

Interrupts: Trap on −0 will not occur even if FIUV is enabled because FSRC is an accumulator.

Underflow cannot occur.

Overflow cannot occur for STCFD.

A trap occurs if FIV is enabled, and if rounding with STCDF causes overflow; FDST ← overflowed result of conversion. This result must be +0 or −0.

Accuracy: STCFD is an exact instruction. Except for overflow, described above, STCDF incurs an error bounded by 1 LSB in chopping mode and ½ LSB in rounding mode.

Mnemonic/ Name	Code	Operation	Condition Codes
STCFI STCFL STCDI STCDL Store and Convert from Floating or Double to In-	175(AC + 4)DST	DST ← C_{xj} (AC) if −JL − 1 < C_{xj} (AC) < JL + 1, else DST ← 0, where C_{xj} specifies conversion from floating mode x to integer mode j:	C ← FC ← 0 if − JL − 1 < C_{xj} (AC) < JL + 1, else FC ← 1. V ← FV ← 0. Z ← FZ ← 1 if (DST) = 0, else FZ ← 0.

Mnemonic/ Name	Code	Operation	Condition Codes
teger or Long Integer		j = I if FL = 0, j = L if FL = 1, x = F if FD = 0, x = D if FD = 1. JL is the largest integer: 2**15 – 1 for FL = 0 2** 31 – 1 for FL = 1	N ← FN ← 1 if (DST) < 0, else FN ← 0.

Description: Conversion is performed from a floating point representation of the data in the accumulator to an integer representation.

If the conversion is to a 32-bit word (L mode) and an address mode of 0, or immediate addressing mode, is specified, only the most significant 16 bits are stored in the destination register.

If the operation is out of the integer range selected by FL, FC is set to 1 and the contents of the DST are set to 0.

Numbers to be converted are always chopped (rather than rounded) before conversion. This is true even when the chop mode bit, FT, is cleared in the floating point status register.

Interrupts: These instructions do not interrupt if FIUV is enabled, because the −0, if present, is in AC, not in memory.

If FIC is enabled, trap on conversion failure will occur.

Accuracy: These instructions store the integer part of the floating point operand, which may not be the integer most closely approximating the operand. They are exact if the integer part is within the range implied by FL.

Mnemonic/ Name	Code	Operation	Condition Codes
STEXP Store Exponent	175ACDST	DST ← EXP(AC)− 200_8	C ← FC ← 0. V ← FV ← 0. Z ← FZ ← 1 if (DST) = 0, else

Mnemonic/ Name	Code	Operation	Condition Codes
			FZ ← 0. N ← FN ← 1 if (DST) < 0, else FN ← 0.

Description: Convert accumulator's exponent from excess 200 octal notation to 2's complement, and store result in DST.

Interrupts: This instruction will not trap on −0.

Overflow and underflow cannot occur.

Accuracy: This instruction is always exact.

Mnemonic/ Name	Code	Operation	Condition Codes
STF STD Store Floating/ Double	174AC- FDST	FDST ← (AC)	FC ← FC FV ← FV FZ ← FZ FN ← FN

Description: Store single or double precision number from accumulator.

Interrupts: These instructions do not interrupt if FIUV enabled, because the −0, if present, is in AC, not in memory. Neither overflow nor underflow can occur.

Accuracy: These instructions are exact.

Special Comment: These instructions permit storage of a −0 in memory from AC. Note, however, that the FPP can store a −0 in an AC only if it occurs in conjunction with overflow or underflow, and if the corresponding interrupt is enabled. Thus, the user has an opportunity to clear the −0, if he wishes.

Mnemonic/ Name	Code	Operation	Condition Codes
STFPS Store FPP's Program Status	1702DST	DST ← (FPS)	

Description: Store FPP's status in DST.

Special Comment: On the FP11-C, FP11-E, and FP11-A, bits 13 and 12 are loaded with zeros. All other bits (with the exception of bit 4 in the FP11-A) represent the corresponding bits in the FPS. The FP11-A has no maintenance mode so bit 4 is loaded with zero.

Mnemonic/ Name	Code	Operation	Condition Codes
STST Store FPP's Status	1703DST	DST ← (FEC) DST + 2 ← (FEA)	

Description: Store the FEC and then the FPP's exception address pointer in DST and DST + 2.

NOTES:

1. If destination mode specifies a general register or immediate addressing, only the FEC is saved.
2. The information in these registers is current only if the most recently executed floating point instructions caused a floating point exception.

Mnemonic/ Name	Code	Operation	Condition Codes
SUBF SUBD Subtract Floating/ Double	173AC- FSRC	Let DIFF = (AC) − (FSRC): If underflow occurs and FIU is not enabled, AC ← exact 0. If overflow occurs and FIV is not enabled, AC ← exact 0. For all other cases, AC ← DIFF.	FC ← 0. FV ← 1 if overflow occurs, else FV ← 0. FZ ← 1 if (AC) = 0, else FZ ← 0. FN ← 1 if (AC) < 0, else FN ← 0.

Description:

Subtract the contents of FSRC from the contents of AC. The subtraction is carried out in single or double precision and is rounded or chopped in accordance with the values of the FD and FT bits in the FPS register. The result is stored in AC except for:

- overflow with interrupt disabled
- underflow with interrupt disabled

For these exceptional cases, an exact 0 is stored in AC.

Interrupts:

If FIUV is enabled, trap on −0 in FSRC occurs before execution.

If overflow or underflow occurs and if the corresponding interrupt is enabled, the trap occurs with the faulty results in AC. The fractional parts are correctly stored. The exponent part is too small by 400_8 for overflow. It is too large by 400_8 for underflow, except for the special case of 0, which is correct.

Accuracy:

Errors due to overflow and underflow are described above. If neither occurs, then for like-signed operands with exponent difference of 0 or 1, the answer returned is exact if a loss of significance of more than one bit can occur. Note that these are the only cases for which loss of significance of more than one bit can occur. For all other cases the result is inexact with error bounds of:

1 LSB in chopping mode with either single or double precision.

½ LSB in rounding mode with single precision.

3/4 LSB (FP11-C and FP11-E) and 33/64 LSB (FP11-A and FP11-F) in rounding mode with double precision.

Special Comment:

The undefined variable −0 can occur only in conjunction with overflow or underflow. It will be stored in the AC only if the corresponding interrupt is enabled.

Mnemonic/ Name	Code	Operation	Condition Codes
TSTF TSTD Test Floating/Double	1705FDST	FDST ← (FDST)	FC ← 0. FV ← 0. FZ ← 1 if EXP(FDST) = 0, else FZ ← 0. FN ← 1 if (FDST) < 0, else FN ← 0.

Description: Set the floating point processor's condition codes according to the contents of FDST.

Interrupts: If FIUV is set, trap on −0 occurs after execution.

Accuracy: These instructions are exact.

Special Comment: This instruction does not write to the destination.

FLOATING POINT PROCESSOR TIMING

The timing and the processes for determining the timing of the floating point instruction vary with each processor. The following sections explain specifically the instruction time and the calculation methods for FP11-A, FP11-C, FP11-F, and FP11-E.

The following table summarizes the floating point execution time of the FP11-A, FP11-E, FP11-F, and FP11-C.

Table 11-1 Comparison of Floating Point Processor Instruction Timing (sec)

Operation (register-to-register)	11/34A FP11-A	11/70 FP11-C	11/60 FP11-E	11/44 FP11-F
Single Precision				
Add/Subtract	8.91	1.65	1.02	8.91
Multiply	16.2	3.27	1.53	16.2
Divide	16.2	4.29	7.00	16.2
Double Precision				
Add/Subtract	8.91	1.68	1.02	8.91
Multiply	25.36	5.43	3.74	25.36
Divide	35.36	6.73	12.75	35.36

FLOATING POINT INSTRUCTION TIMING: FP11-A

Instruction Execution Time

The execution time of an FP11-A floating point instruction is dependent on the following conditions:

- type of instruction
- type of addressing mode specified
- type of memory
- memory management facility enabled or disabled

Additionally, the execution time of certain instructions, such as ADD, is dependent on the data.

Table 11-2 provides the basic instruction times for mode 0. Tables 11-3 through 11-7 show the additional time required for instructions other than mode 0. For example, to calculate the execution time of a MULF (single-precision multiply) for mode 3 (autoincrement deferred) with the result to be rounded:

1. Refer to Table 11-2 which gives MULF, mode 0, execution time of 13.4 μsec.
2. Refer to Note 1 as specified in the notes column of Table 11-2. Note 1 specifies an additional 0.84 μsec is to be added if rounding mode is specified. This yields 14.24 μsec.
3. The modes 1-7 column of Table 11-2 refers to Table 11-3 to determine the additional time required for mode 1 through 7 instructions. In this example, mode 3 specifies an additional 3 μsec for single precision yielding 17.34 μsec.

All timing information is in microseconds unless otherwise noted. Times are typical; processor timing can vary $\pm$10%.

NOTE

Add .13 μsec for each memory cycle if MS11-JP MOS memory is utilized. Add .12 μsec for each DATI memory cycle if memory management is enabled.

Table 11-2 FP11-A Instruction Execution Times

Instr.	Mode 0 (Reg. to Reg.)	Notes	Modes 1 thru 7
LDF	4.0		
LDD	4.0		
LDCFD	5.8	1	
LDCDF	5.8	1	
CMPF	5.5		
CMPD	5.5		
DIVF	13.3	1	Use Table 11-3
DIVD	20.6	1	to determine
ADDF	7.5	1,2	memory-to-register times
ADDD	7.5	1,2	for these instructions
SUBF	7.9	1,2	
SUBD	7.9	1,2	
MULF	13.4	1	
MULD	20.7	1	
MODF	17.4	1,3	
MODD	24.7	1,3	
STF	2.4		
STD	2.4		Use Table 11-4
STCDF	5.2		to determine
STCFD	5.2		memory-to-register times
CLRF	2.6		for these instructions
CLRD	2.6		
ABSF	3.5		
ABSD	3.5		Use Table 11-5
NEGF	3.6		to determine
NEGD	3.6		memory-to-memory times
TSTF	3.6		for these instructions
TSTD	3.6		
LDFPS	2.5		
LDEXP	4.4		Use Table 11-6

Instr.	Mode 0 (Reg. to Reg.)	Notes	Modes 1 thru 7
LDCIF	7.5	1,4	to determine
LDCID	7.5	1,4	memory-to-register times
LDCLF	7.5	1,4	for these instructions
LDCLD	7.5	1,4	
STFPS	2.8		
STST	2.6		Use Table 11-7
STEXP	3.4		to determine
LSTCFI	4.5	5	register-to-memory times
STCDI	4.5	5	for these instructions
STCFL	4.5	5	
STCDL	4.5	5	
The following instructions do not reference memory			
CFCC	2.0		
SETF	2.2		
SETD	2.2		Execution times
SETI	2.2		are as shown
SETL	2.2		

Table 11-3 Floating Source Fetch Time

	Memory Cycles		Time(μs)	
Addressing Mode	Single Precision	Double Precision	Single Precision	Double Precision
1	2	4	2.00	4.20
2	2	4	2.20	4.40
2 Immediate	1	1	1.00	1.00
3	3	5	3.00	5.20
4	2	4	2.20	4.40
5	3	5	3.00	5.20
6	3	5	3.20	5.40
7	4	6	4.20	6.40

Table 11-4 Floating Destination Store Time

	Memory Cycles		Time(μs)	
Addressing Mode	Single Precision	Double Precision	Single Precision	Double Precision
1	2	4	1.38	2.94
2	2	4	1.56	3.12
2 Immediate	1	1	0.60	0.60
3	3	5	2.38	3.94
4	2	4	1.56	3.12
5	3	5	2.38	3.94
6	3	5	2.56	4.12
7	4	6	3.56	5.12

Table 11-5 Floating Destination Fetch And Store Time

	Memory Cycles		Time(μs)	
Addressing Mode	Single Precision	Double Precision	Single Precision	Double Precision
1	2	2	1.42	1.42
2	2	2	1.60	1.60
2 Immediate	2	2	1.60	1.60
3	3	3	2.42	2.42
4	2	2	1.60	1.60
5	3	3	2.60	2.60
6	3	3	2.60	2.60
7	4	4	3.60	3.60

Table 11-6 Source Fetch Time

	Memory Cycles		Time(μs)	
Addressing Mode	Single Precision	Double Precision	Single Precision	Double Precision
1	1	2	1.00	1.18
2	1	2	1.18	1.36
2 Immediate	1	1	1.18	1.18
3	2	3	2.00	2.18
4	1	2	1.18	1.36
5	2	3	2.00	2.18
6	2	3	2.18	2.36
7	3	4	3.18	3.36

Table 11-7 Destination Store Time

	Memory Cycles		Time(μs)	
Addressing Mode	Single Precision	Double Precision	Single Precision	Double Precision
1	1	2	0.60	1.38
2	1	2	0.96	1.68
2 Immediate	1	1	0.96	0.96
3	2	3	1.60	2.38
4	1	2	0.96	1.68
5	2	3	1.60	2.38
6	2	3	1.78	2.56
7	3	4	2.78	3.56

NOTES:

- Add 0.84 μsec when in rounding mode (FT = 0).
- Add 0.24 μsec per shift to align binary points and 0.24 μsec per shift for normalization. The number of alignment shifts is equal to the exponent difference for exponent differences bounded as follows:

  $1 \leq |EXP\ (AC) - EXP\ (FSRC)| \leq 24$ single precision
  $1 \leq |EXP\ (AC) - EXP\ (FSRC)| \leq 56$ double precision

The number of shifts required for normalization is equivalent to the number of leading zeros of the result.

- Add .24 μsec times the exponent of the product if the exponent of the product is:

  $1 \leq$ EXP (PRODUCT) ≤ 24 single precision
  $1 \leq$ EXP (PRODUCT) ≤ 56 double precision

  Add 0.24 μsec per shift for normalization of the fractional result. The number of shifts required for normalization is equivalent to the number of leading zeros in the fractional result.
- Add 0.24 μsec per shift for normalization of the integer being converted to a floating point number. For positive integers, the number of shifts required to normalize is equivalent to the number of leading zeros; for negative integers, the number of shifts required for normalization is equivalent to the number of leading ones.
- Add 0.24 μsec per shift to convert the fraction and exponent to integer form, where the number of shifts is equivalent to 16 minus the exponent when converting to short integer or 32 minus the exponent when converting to long integer for exponents bounded as follows:

  $1 \leq$ EXP (AC) ≤ 15 short integer
  $1 \leq$ EXP (AC) ≤ 31 long integer

FLOATING POINT INSTRUCTION TIMING: FP11-C

Floating point instruction times are calculated in a manner similar to the calculation of CPU instruction timing. Since the FP11-C is a separate processor operating in parallel with the main processor, however, the calculation of floating point instruction times must take this parallel processing or overlap into account. The following is a description of the method used to calculate the effective floating point instruction execution times.

TERM	DEFINITION
Instruction Decode Preinteraction Time	CPU time required to decode a floating point instruction opcode and to store the general register referred to in the floating point instruction in a temporary floating point register (FPR). This time is fixed at 450 ns.

TERM	DEFINITION
Address Calculation Time	CPU time required to calculate the address of the operand. This time is dependent on the addressing mode specified. Refer to Table 11-8.
Wait Time	CPU time spent waiting for completion by the floating point processor of a previous floating point instruction, in the case of load class instructions. For store class instructions, the wait time is the sum of time during which the floating point completes a previous floating point instruction and floating point execution time for the store class instruction. Wait time is calculated as follows:
(Load Class Instructions)	Wait time = [floating point execution time (previous FP instruction)] − [disengage and fetch time (previous FP instruction)] − [CPU execution time for interposing non-floating point instruction] − [preinteraction time] — [address calculation time]. If the result is ≤ 0, the wait time is zero.
(Store Class Instructions)	Wait time = [floating point execution time (previous floating point instruction)] − [CPU execution time for interposing non-FP instruction] − [disengage and fetch time (previous FP instruction)] − [preinteraction] + [floating point execution time] — [address calculation time]. If the result is ≤ 0, the wait time is zero.
Resync Time	If the CPU must wait for the floating point processor (i.e., wait time = 0), an additional 450 ns must be added to the effective execu-

TERM	DEFINITION
	tion time of the instruction. If wait time = 0, then resync time =0.
Interaction Time	CPU time required actually to initiate floating point processor operation.
Argument Transfer Time	CPU time required to fetch and transfer to the floating point processor the required operand. This time is 300 ns × the number of 16-bit words read from memory (load class floating point instructions), or 1200 ns × the number of 16-bit words written to memory (store class instructions).
Disengage and Fetch Time	CPU time required to fetch the next instruction from memory. This time is 300 ns.
Floating Point Execution Time	Time required by the floating point processor to complete a floating point instruction once it has received all arguments (load class instructions). Execution times are contained in Tables 11-2 through 11-7.
Effective Execution Time	Total CPU time required to execute a floating point instruction. Effective Execution Time = Preinteraction + Address Calculation + Wait Time + Resync Time + Interaction Time + Argument Transfer + Disengage and Fetch.

Table 11-8 Address Calculation Times

Mode	Address Calculation Time nsec
0	0
1	300
2	300
3	600
4	300
5	750
6	600
7	1050

Table 11-9 FP11-C Execution Times

Instruction	Minimum nsec	Maximum nsec	Typical
LDF	360	360	
LDD	360	360	
ADDF	900	2520	950
ADDD	900	4140	980
SUBF	900	1980	1130
SUBD	900	4140	1160
MULF	1800	3440	2520
MULD	3060	6220	4680
DIVF	1920	6720	3540
DIVD	3120	14400	6000
MODF	2880	5990	
MODD	3780	9770	
LDCFD	420	420	
LDCDF	540	540	
STF*	0		
STD*	0		
CMPF	540	1080	
CMPD	540	1080	
STCFD*	720	720	720
STCDF*	540	720	540
LDCIF	1260	1440	1440
LDCID	1260	1440	1440

Instruction	Minimum nsec	Maximum nsec	Typical
LDCLF	1260	1980	
LDCLD	1260	1980	
LDEXP	540	900	
STCFI*	1260	1620	
STCFL*	1260	2160	
STCDI*	1260	1620	
STCDL*	1260	2160	
STEXP*	360	360	
	MO	Not MO	
CLRD	180	2150	
CLRD	180	14350	
NEGF	360	2400	
NEGD	360	2400	
ABSF	360	2400	
ABSD	360	2400	
TSTF	180	180	
TSTD	180	180	
LDFPS	180	0	
STFPS*	0		
STST*	0		
CFCC	0		
SETF	180		
SETD	180		
SETI	180		
SETL	180		

* Store Class Instructions

Load class instructions are those which do not deposit results in a memory location.

Execution of a load class floating point instruction by the floating point occurs in parallel with CPU operation and can be overlapped. Figure 11-2 gives a simplified picture of how a load class floating point instruction is executed.

Store class instructions are those which store a result from the floating point into a memory location. Execution of a store class instruction by the floating point processor must occur before the result can be stored, hence parallel processing cannot occur for store class floating point instructions.

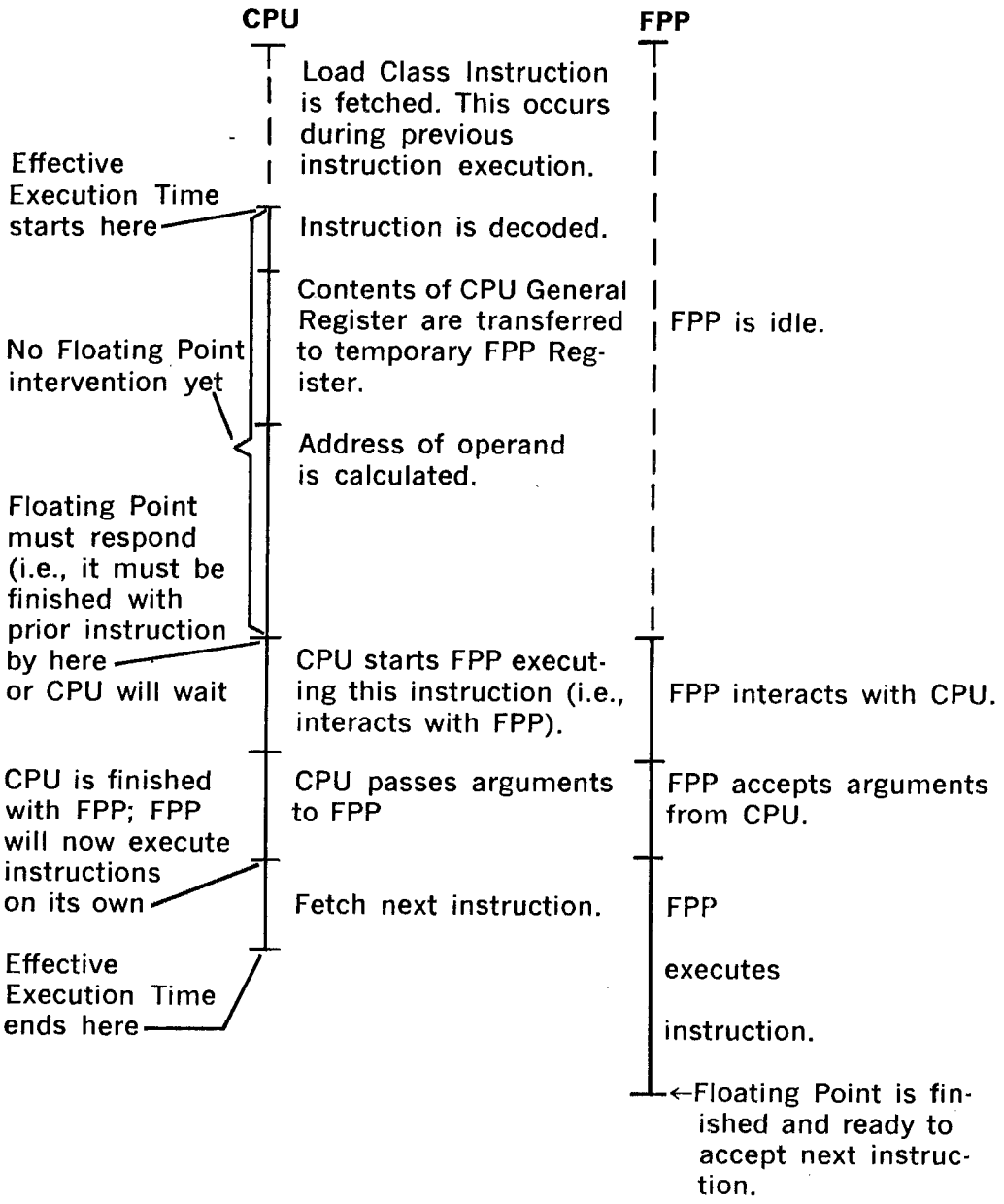


Figure 11-2 Load Class Floating Point Instruction

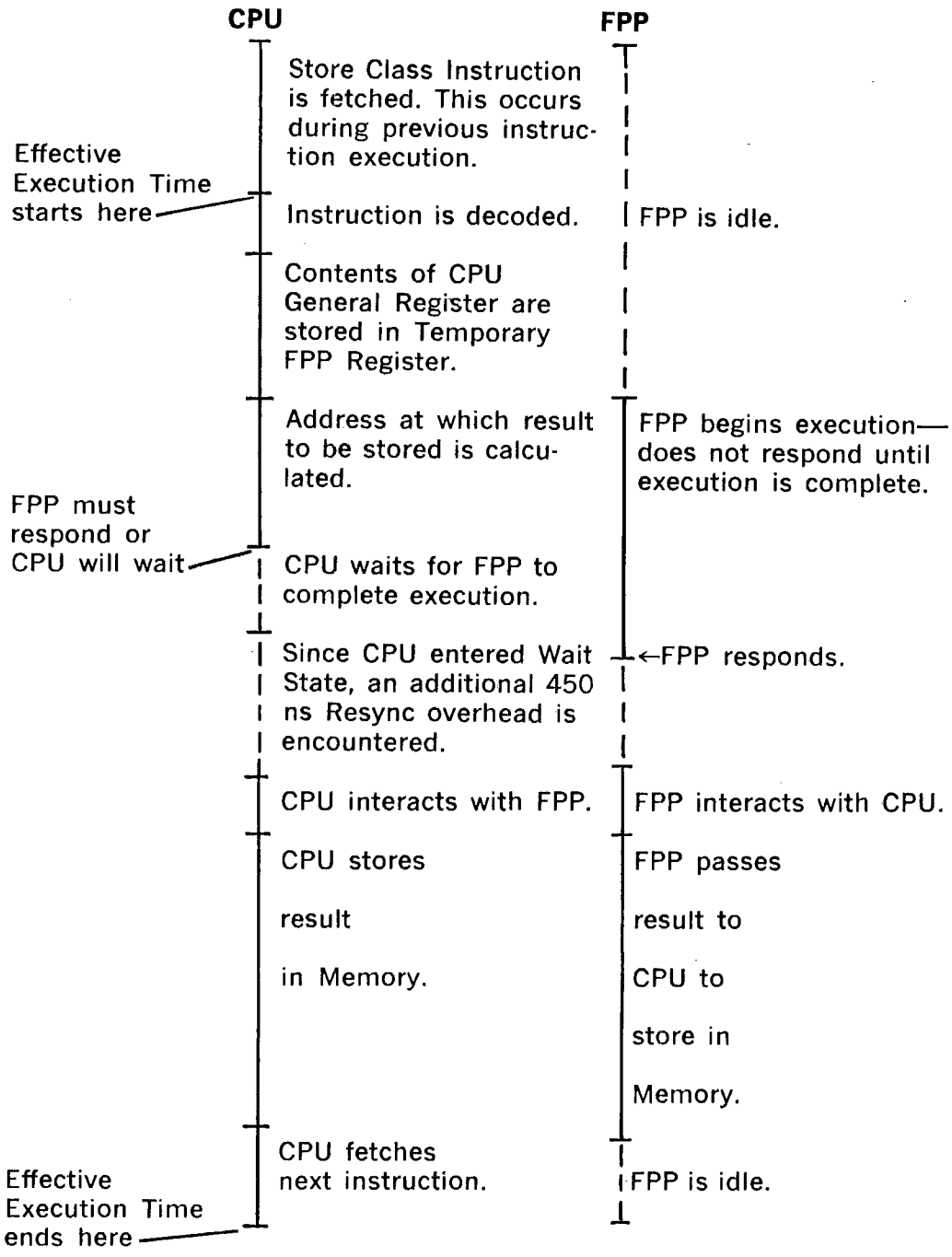


Figure 11-3 Store Class Floating Instruction

Figures 11-2 and 11-3 show how timing associated with a typical load class and store class instruction is derived.
Figures 11-4 and 11-5 show how effective execution times for actual floating point instructions in a program are calculated. Note that effective execution times are dependent on previous floating point instructions.
Referencing Figure 11-4, a sample calculation of effective time would be:
For MULF (R0), AC1, effective execution time is the summation of the following:

Preinteraction Time	450 ns
Address Calculation Time (Mode 1 from Table 11-8)	300 ns
Wait Time (Since FPP is idle, Wait = 0)	0 ns
Resync Time (Since Wait = 0, Resync = 0)	0 ns
Interaction Time	300 ns
Argument Transfer Time (Transfer 2 words @ 300 ns/word)	600 ns
Disengage and Fetch Time	300 ns
Effective Execution Time	1950

For LDF X(R3),ACLO (Ref. Figure 11-4), first we calculate Wait Time:

Wait Time = [Floating Point Execution (previous FP instruction)(MULF)]	1800 ns
− [Disengage and Fetch Time (previous FPT instruction)]	− 300 ns
− [Execution time of interposing nonFPT instruction (SOB)]	− 750 ns
− [Preinteraction Time]	− 450 ns
− [Address Calculation (Mode 6 from Table 11-8)]	− 600 ns
	− 300 ns

Since calculation resulted in a negative number, Wait Time = 0.

...so effective execution time is the summation of the following:

Preinteraction Time	450 ns
Address Calculation Time (Mode 6 from Table 11-8)	600 ns
Wait Time (From above calculation)	0 ns
Resync Time (Since Wait Time = 0, Resync = 0)	0 ns
Interaction Time	300 ns
Argument Transfer Time (2 words @ 300 ns/word)	600 ns
Disengage and Fetch Time	300 ns
Effective Execution Time	2250 ns

FLOATING POINT INSTRUCTION TIMING: FP11-E

Floating point instruction times are calculated similarly to the calculation of CPU instruction timing. However, since the FP11-E is a separate processor, and its execution can proceed in parallel with the PDP-11/60, calculation of floating point instruction times must take this independent processing into account.

The following information describes the method used to calculate effective instruction execution times.

NOTE

Resync and interaction times present in the FP11-C are not considered, since handshaking synchronization overhead has been eliminated by the use of decoding and instruction fetch logic. In the FP11-E, the fetching of floating point instructions is initiated by the CPU, but is received simultaneously by both processors.

In addition to instruction fetch and address calculation, the CPU converts fixed to floating point notation and, in some instances, fully executes the instruction, for example, LDFPS.

TERM	DEFINITION
Instruction decode	CPU time required to decode a floating point instruction op code. This time is fixed at 340 nsec.
Address calculation time	CPU time required to calculate the address of the operand. This time is dependent on the addressing mode specified. Refer to Tables 11-11 and 11-12.
Wait time	CPU time spent waiting for completion by the floating point processor of a previous floating point instruction in the case of a load class of instruction. For store class instructions, the wait time is the summation of time during which the floating point processor completes a previous floating point instruction and floating point execution time for store class instruction. Wait time is calculated as follows:

TERM	DEFINITION
(Load Class Instructions)	Wait time = [floating point execution (previous FP instructions)] − [disengage and fetch time] − [CPU execution time for interposing non-floating point instruction] − [Instruction fetch time] − [Address calculation time]. If the result is ≤ 0, the wait time is 0.
(Store Class Instructions)	Wait Time = [Floating point execution time (previous FP instruction)] − [Disengage and fetch time] − [CPU execution time for interposing non-floating point instruction] − [Instruction fetch time] + [Floating point execution time] − [Address calculation time]. If the result is ≤ 0, the wait time is 0.
Argument transfer time	CPU time required to fetch and transfer operands. This time is 340 nsec × the number of 16-bit words read from memory or 1170 nsec × the number of 16-bit words written into memory. Add 1.075 μsec for a word received from memory (MM-11D memory only) that is a miss.
Shared execution time	CPU time spent in the execution of integer convert routines or any one of the instructions in category 5. Refer to Table 11-13.
Disengage and fetch time	Time required to fetch the next instruction from memory. This time is fixed at 340 nsec for a cache hit. Add 1.075 μsec for a cache miss (MM-11D).
Floating point execution time	Time required by the floating point processor to complete a floating point instruction once it

TERM	DEFINITION
	has received all operands (load class). Refer to Table 11-14.
Effective execution time	Total CPU time required to execute a floating point instruction.
	Effective execution time = instruction decode + address calculation + wait time + argument transfer time + shared execution time + disengage and fetch.

Table 11-10 Floating Point Instructions

Category	Instruction	
LOAD CLASS	LDF,LDD	
	ADF, ADD	
	SUBF, SUBD	
	MULF, MULD	
	DIVF, DIVD	
	MODF, MODD	
	LDCF, LDCD	
	CMPF, CMPD	
	LDCIF, LDCID	
	LDCLF, LDCLD	
LOAD CLASS (INTEGER CONVERT)	LDEXP	
STORE CLASS	STF, STD	
	STCDF	
	STCFD	
STORE CLASS (INTEGER CONVERT)	STCFI	
	STCFL	
	STCDI	
	STCDL	
	STEXP	
NULL (CPU EXECUTES)	CLRF, CLRD	Not MO
	NEGF, NEGD	Not MO
	ABSF, ABSD	Not MO
	TSTF, TSTD	Not MO

Category	Instruction
	LDFPS
	STFPS
	STST
	CFCC
	SETF, SETD
	SETI, SETL

Table 11-11 Address Calculation (Floating/Double)

Mode	Time (nsec)	Read Memory Cycle
0	0	0
1	510	0
2	510	0
3	850	1
4	850	0
5	1360	1
6	850	1
7	1360	2

Table 11-12 Address Calculation (integer)

Mode	Time (nsec)	Read Memory Cycle
0	340	0
1	340	0
2	340	0
3	850	1
4	510	0
5	1020	1
6	850	1
7	1360	2

Table 11-13 Shared Execution Time

	Instruction	Time (nsec)
1.	CLRF	2210
	CLRD (Not MO)	2720
2.	NEGF	3060
	NEGD (Not MO)	3400
3.	ABSF	3060
	ABSD (Not MO)	3400
4.	TSTF	3060
	TSTD (Not MO)	3400
5.	LDFPS	2040
	STFPS	1360
	STST	2550
6.	CFCC	1020
	SETD	1190
	SETI	1360
	SETD	1190
	SETL	1360
7.	STEXP	2210
8.	LDEXP	1700

Table 11-14 FP11-E Execution Times (nsec)

Instruction	MO	M6	Not (MO or M6)
1. LDF	170	0	0
2. LDD	170	0	340

	MO			Not MO		
	Min.	Max.	Typical	Min.	Max.	Typical
3.ADDF	340	1700	510	680	2040	850
4.ADDD	340	2890	680	1020	3570	1360
5.SUBF	340	1700	510	680	2040	850
6.SUBD	340	2890	680	1020	3570	1360
7.MULF	850	850	850	1020	1020	1020
8.MULD	3060	3060	3060	3570	3570	3570
9.DIVF	6120	6460		6290	6800	
10.DIVD	11900	12410		12240	12580	
11.MODF	3040	4250		3210	4420	
12.MODD	5610	8500		6120	9010	
13.LDCFD*	1700	1700		2040	2040	
14.LDCDF*	2040	2040		2720	2730	
15.STF	170	170		510	510	
16.STD	170	170		510	510	
17.CMPF	170	850		340	1020	
18.CMPD	170	850		680	1360	
19.STCFD	680	850		1700	2210	
20.STCDF	680	1020		1700	2550	
21.LDCIF	7310	9860		7140	9520	
22.LDCID	7310	9690		6970	9350	
23.LDCLF	7480	10030		8500	13770	
24.LDCLD	7310	9860		8330	13600	
25.LDEXP	680	680		680	680	
26.STCFI*	5270	7650		4930	7310	
27.STCFL*	5270	10370		6800	11900	
28.STCDI*	5270	7650		4930	7310	
29.STCDL*	5270	10370		6800	11900	
30.STEXP	0	0				
31.CLRF	170			0	0	
32.CLRD	170			0	0	
33.NEGF	340			0	0	
34.NEGD	340			0	0	
35.ABSF	340			0		

Instruction	MO	M6	Not (MO or M6)
1. LDF	170	0	0
2. LDD	170	0	340

	MO			Not MO		
	Min.	Max.	Typical	Min.	Max.	Typical
36.ABSD	340			0		
37.TSTF	170			0		
38.TSTD	170			0		
39.LDFPS	0			0		
40.STFPS	0			0		
41.STST	0			0		
42.CFCC	0			0		
43.SETF	0			0		
44.SETD	0			0		
45.SETI	0			0		
46.SETL	0			0		

* Requires CPU shared code execution. For Mode 0 address calculation, add 4 cycles.

Table 11-15 Load Class of Instructions

CPU	FP11-E
Load class instruction is fetched. This occurs during previous instruction execution.	
Instruction is decoded.	
Address of operands is calculated.	FP11-E decodes instruction and goes into idle state.
CPU passes operands to the FP11-E.	FP11-E receives operands from CPU.
Disengage and fetch next instruction.	FP11-E executes instruction.
Load class (integer convert) of instructions is fetched. This occurs during previous instruction.	

CPU	FP11-E
Instruction is decoded.	FP11-E decodes instruction, goes into idle state.
Address of operands is calculated and fetched from memory.	
Integer conversion by CPU	
CPU passes result to FP11-E.	FP11-E receives result from CPU.
Disengage and fetch next instruction.	FP11-E stores results.

Table 11-16 Store Class of Instructions

CPU	FP11-E
Store class of instructions is fetched. This occurs during previous instruction.	FP11-E is idle.
Instruction is decoded.	FP11-E decodes instruction.
Address of operands is calculated.	FP11-E starts instruction execution.
CPU waits for FP11-E to complete execution.	
CPU receives result from the FP11-E and stores it in memory.	FP11-E passes result to be stored in memory.
CPU fetches next instruction.	FP11-E is idle.

Table 11-17 Store Class of Instructions (Integer Convert)

CPU	FP11-E
Store class (integer convert) is fetched. This occurs during previous instructions.	FP11-E is idle.
Instruction is decoded.	FP11-E decodes instruction.
CPU receives floating point number from FP11-E.	FP11-E passes floating point number.
Integer conversion performed by CPU.	FP11-E is idle.

CPU does address calculation
and stores result in memory.

Tables 11-8 and 11-9 show how effective execution times for actual floating point instructions in a program are calculated. Note that the effective execution times are dependent on previous floating point instructions. Note also that all memory references are considered to be cache hits.

A sample calculation of effective time would be:

For MULF (R0), AC1:

Instruction Fetch	340 nsec
Address Calculation Time (Mode 1 from Table 11-11)	510 nsec
Wait Time (Since FPP is idle, Wait = 0)	0 nsec
Argument Transfer Time (Transfer 2 words @ 340 nsec/word)	680 nsec
Disengage and Fetch Time	340 nsec
Effective Execution Time	1870 nsec

For LDF X (R3), AC0 (Ref. Figure 11-5):

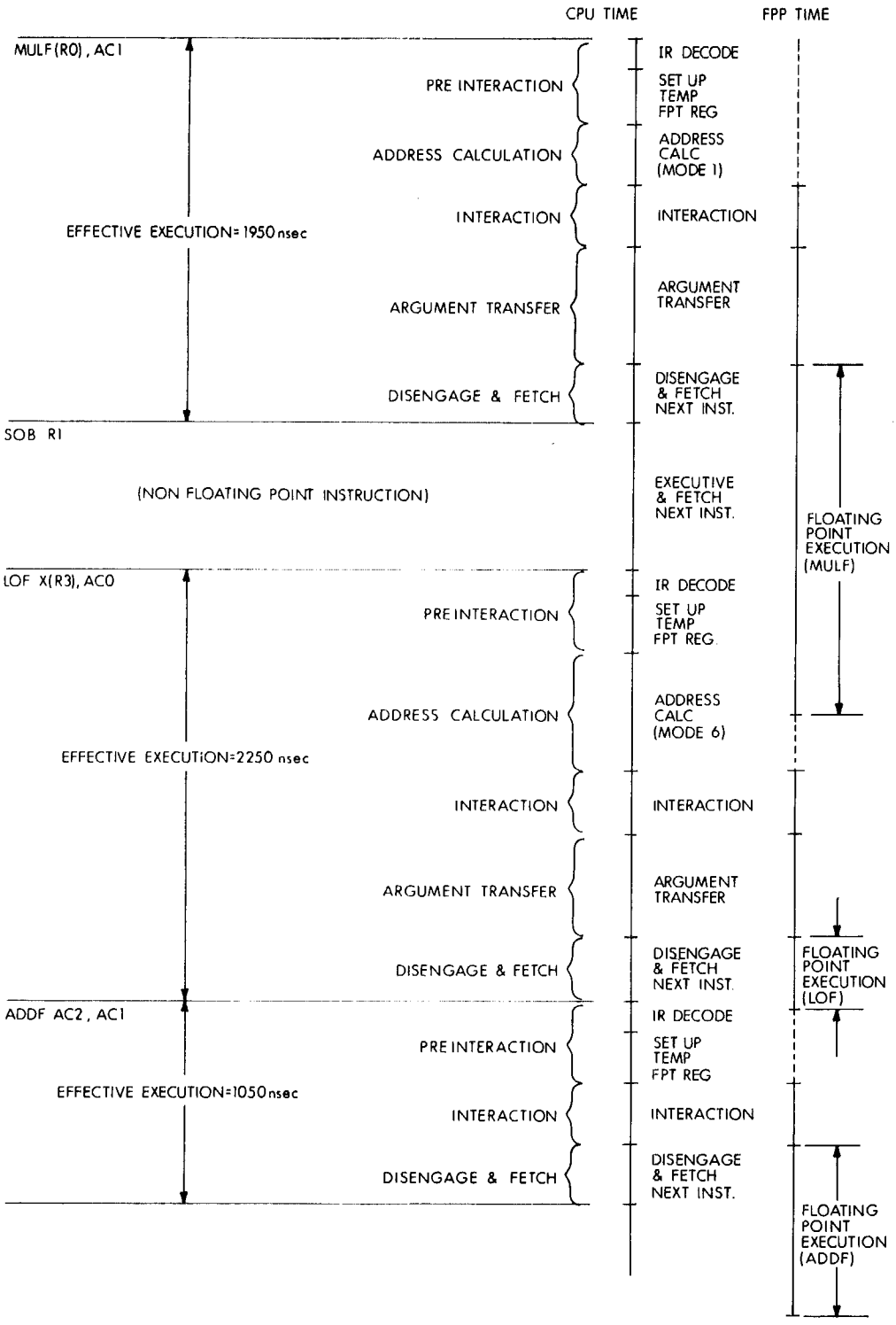
First, calculate Wait Time:

Wait Time = [Floating Point Execution (previous FP instruction) (MULF)]	1020 nsec
− [Disengage and Fetch Time (previous FPT instruction)]	− 340 nsec
− [Execution Time of interposing nonFPT instruction (SOB)	−2400 nsec
− [Instruction Fetch]	− 340 nsec
− [Address Calculation (Mode 6 from Table 11-11)]	− 850 nsec
	−2910 nsec

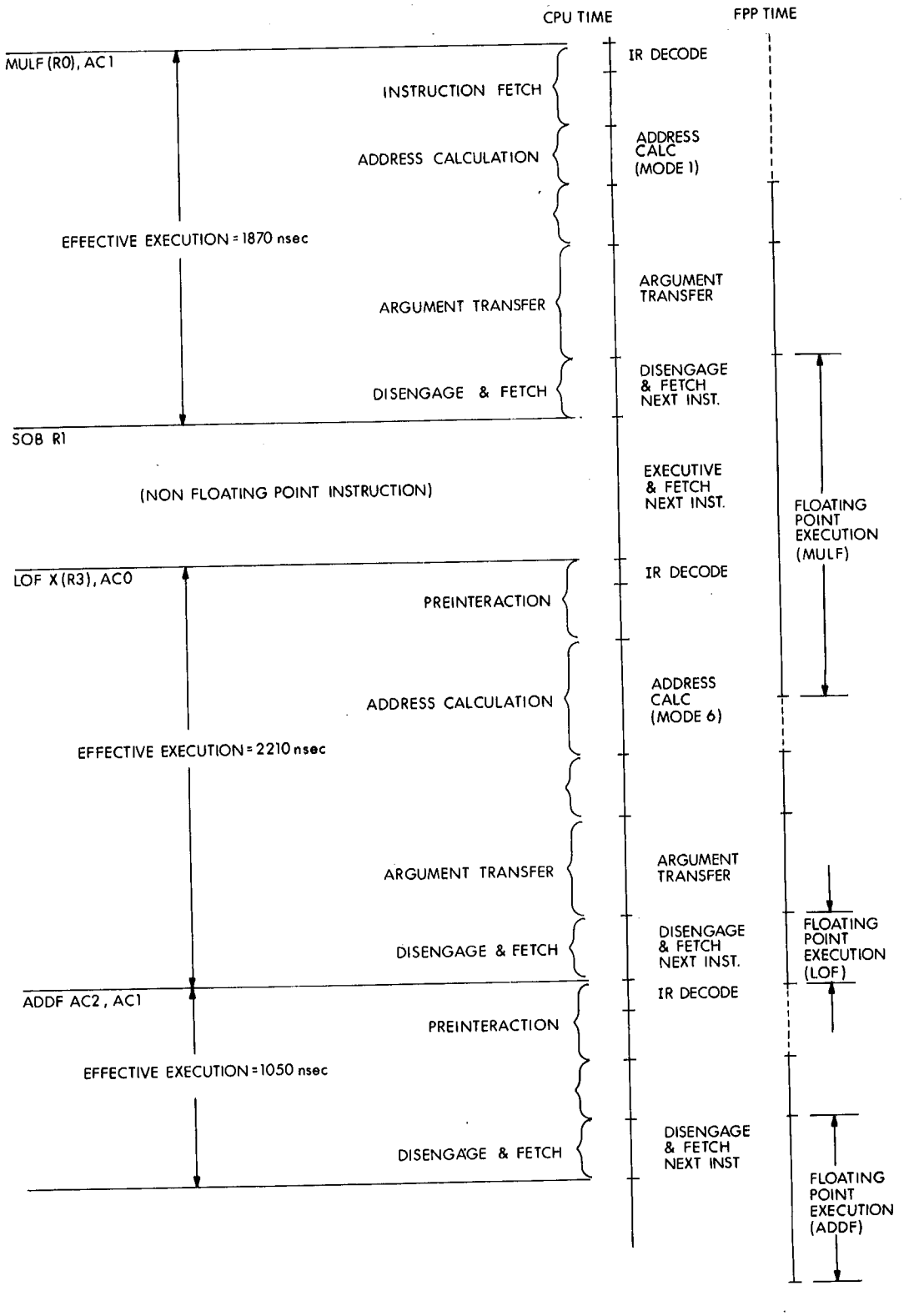
Since calculation resulted in a negative number, Wait Time = 0.

...so Effective Execution Time is the summation of the following:

Instruction Fetch	340 nsec
Address Calculation Time (Mode 6 from Table 11-11)	850 nsec
Wait Time (from above calculation)	0 nsec
Argument Transfer Time (2 words @ 340 nsec/word)	680 nsec
Disengage and Fetch Time	340 nsec
Effective Execution Time	2210 nsec



Calculation of Effective Execution Times for Load Class Instructions (FP11-C)



Calculation of Effective Execution Times for Load Class Instructions (FP11-E)

FLOATING POINT INSTRUCTION TIMING: FP11-F

Instruction Execution Time

The execution time of an FP11-F floating point instruction is dependent on the following conditions:

- type of instruction
- type of addressing mode specified
- type of memory
- memory management facility enabled or disabled

Additionally, the execution time of certain instructions, such as ADD, is dependent on the data.

Table 11-18 provides the basic instruction times for mode 0. Tables 11-19 through 11-23 show the additional time required for instructions other than mode 0. For example, to calculate the execution time of a MULF (single-precision multiply) for mode 3 (autoincrement deferred) with the result to be rounded:

1. Refer to Table 11-18 which gives MULF, mode 0, execution time of 13.4 μsec.
2. Refer to Note 1 as specified in the notes column of Table 11-18. Note 1 specifies an additional 0.84 μsec is to be added if rounding mode is specified. This yields 14.24 μsec.
3. The Modes 1 through 7 column of Table 11-18 refers to Table 11-19 to determine the additional time required for mode 1 through 7 instructions. In this example, mode 3 specifies an additional 3 μsec for single precision yielding 17.24 μsec.

All timing information is in microseconds unless otherwise noted. Times are typical; processor timing can vary ±10%. All instructions assume 100% cache hits.

NOTE

Add .09 μsec for each DATI memory cycle if memory management is enabled.
Add .630 μsec for each DATI memory cycle if a cache miss is encountered.

Table 11-18 FP11-F Instruction Execution Times

Instr.	Mode 0 (Reg. to Reg.)	Notes	Modes 1 thru 7
LDF	3.0		
LDD	3.0		
LDCFD	4.8	1	
LDCDF	4.8	1	
CMPF	4.5		
CMPD	4.5		
DIVF	12.3	1	Use Table 11-19
DIVD	19.6	1	to determine
ADDF	6.5	1,2	memory-to-register times
ADDD	6.5	1,2	for these instructions
SUBF	6.9	1,2	
SUBD	6.9	1,2	
MULF	12.4	1	
MULD	19.7	1	
MODF	16.4	1,3	
MODD	23.7	1,3	
STF	1.4		
STD	1.4		Use Table 11-20
STCDF	4.2		to determine
STCFD	4.2		memory-to-register times
CLRF	1.6		for these instructions
CLRD	1.6		
ABSF	2.5		
ABSD	2.5		Use Table 11-21
NEGF	2.6		to determine
NEGD	2.6		memory-to-memory times
TSTF	2.6		for these instructions
TSTD	2.6		
LDFPS	1.5		
LDEXP	3.4		Use Table 11-22
LDCIF	6.5	1,4	to determine
LDCID	6.5	1,4	memory-to-register times
LDCLF	6.5	1,4	for these instructions
LDCLD	6.5	1,4	

Instr.	Mode 0 (Reg. to Reg.)	Notes	Modes 1 thru 7
STFPS	1.8		
STST	1.6		Use Table 11-23
STEXP	2.4		to determine
STCFI	3.5	5	register-to-memory times
STCDI	3.5	5	for these instructions
STCFL	3.5	5	
STCDL	3.5	5	
The following instructions do not reference memory			
CFCC	1.0		
SETF	1.2		
SETD	1.2		Execution times
SETI	1.2		are as shown.
SETL	1.2		

Table 11-19 Floating Source Fetch Time

	Memory Cycles		Time (μs)	
Addressing Mode	Single Precision	Double Precision	Single Precision	Double Precision
1	2	4	0.60	1.4
2	2	4	0.80	1.6
2 Immediate	1	1	0.30	0.3
3	3	5	0.90	1.7
4	2	4	0.80	1.6
5	3	5	0.90	1.7
6	3	5	1.10	1.9
7	4	6	1.40	2.2

Table 11-20 Floating Destination Store Time

	Memory Cycles		Time (μs)	
Addressing Mode	Single Precision	Double Precision	Single Precision	Double Precision
1	2	4	1.38	2.94
2	2	4	1.56	3.12
2 Immediate	1	1	0.60	0.60
3	3	5	1.68	3.24
4	2	4	1.56	3.12
5	3	5	1.68	3.24
6	3	5	1.86	3.42
7	4	6	2.16	3.72

Table 11-21 Floating Destination Fetch And Store Time

	Memory Cycles		Time (μs)	
Addressing Mode	Single Precision	Double Precision	Single Precision	Double Precision
1	2	2	0.72	0.72
2	2	2	0.90	0.90
2 Immediate	2	2	0.80	0.80
3	3	3	1.02	1.02
4	2	2	0.90	0.90
5	3	3	1.20	1.20
6	3	3	1.20	1.20
7	4	4	1.50	1.50

Table 11-22 Source Fetch Time

	Memory Cycles		Time (μs)	
Addressing Mode	Short Integer	Long Integer	Short Integer	Long Integer
1	1	2	0.30	0.70
2	1	2	0.48	1.28
2 Immediate	1	1	0.48	0.48
3	2	3	0.60	1.0
4	1	2	0.48	1.28
5	2	3	0.60	1.0
6	2	3	0.78	1.18
7	3	4	1.08	1.48

Table 11-23 Destination Store Time

	Memory Cycles		Time (μs)	
Addressing Mode	Short Integer	Long Integer	Short Integer	Long Integer
1	1	2	0.60	1.38
2	1	2	0.96	1.68
2 Immediate	1	1	0.96	0.96
3	2	3	0.90	1.68
4	1	2	0.96	1.68
5	2	3	0.90	1.68
6	2	3	1.08	1.86
7	3	4	1.38	2.16

NOTES:

1. Add 0.84 μsec when in rounding mode (FT = 0).
2. Add 0.24 μsec per shift to align binary points and 0.24 μsec per shift for normalization. The number of alignment shifts is equal to the exponent difference for exponent differences bounded as follows:

   $1 \leq EXP(AC) - EXP(FSRC) \leq 24$, single precision
   $1 \leq EXP(AC) - EXP(FSRC) \leq 56$, double precision

The number of shifts required for normalization is equivalent to the number of leading zeros of the result.

3. Add 0.24 μsec times the exponent of the product if the exponent of the product is:

   $1 \leq$ EXP(PRODUCT) ≤ 24, single precision
   $1 \leq$ EXP(PRODUCT) ≤ 56, double precision

   Add 0.24 μsec per shift for normalization of the fractional result. The number of shifts required for normalization is equivalent to the number of leading zeros in the fractional result.

4. Add 0.24 μsec per shift for normalization of the integer being converted to a floating point number. For positive integers, the number of shifts required to normalize is equivalent to the number of leading zeros; for negative integers, the number of shifts required for normalization is equivalent to the number of leading ones.

5. Add 0.24 μsec per shift to convert the fraction and exponent to integer form, where the number of shifts is equivalent to 16 minus the exponent when converting to short integer, or 32 minus the exponent when converting to long integer for exponents bounded as follows:

   $1 \leq$ EXP(AC) ≤ 15, short integer
   $1 \leq$ EXP(AC) ≤ 31, long integer

CHAPTER 12
COMMERCIAL INSTRUCTION SET

Commercial Instruction Set

The PDP-11 Commercial Instruction set (CIS11) consists of the following extended instruction groups:

07602X	Commercial Load 2 Descriptors
07603X	Character String Move
07604X	Character String Search
07605X	Numeric String
07606X	Commercial Load 3 Descriptors
07607X	Packed String
07613X	Character String Move (in-line)
07614X	Character String Search (in-line)
07615X	Numeric String (in-line)
07617X	Packed String (in-line)

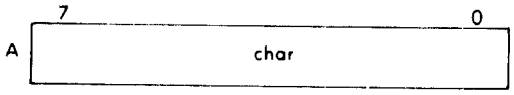
These include instructions which operate on character strings and on decimal numbers. Each generic type of instruction is provided in two forms. The essential difference between the two forms is the manner in which operands are delivered to the instruction. The first form is the "register" form, where operands are implicitly obtained from the general registers. The second form is the "in-line" form, where operands or word address pointers to operands follow the opcode word in the instruction stream. The mnemonic for the in-line form is the mnemonic for the register form suffixed with the letter "I". The condition codes are set identically for both forms. The in-line forms minimize register modification.

Instructions are also provided which efficiently load operands into the general registers.

Character Data Types

There are three different character data types. The "character" is a single byte, and is an abbreviated string of length 1. The "character string" is a contiguous group of bytes in memory. The third is a "character set."

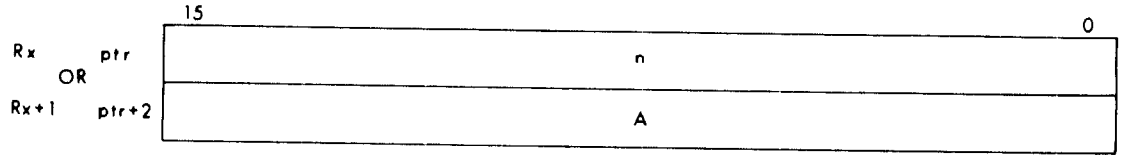
The character is an 8-bit byte:



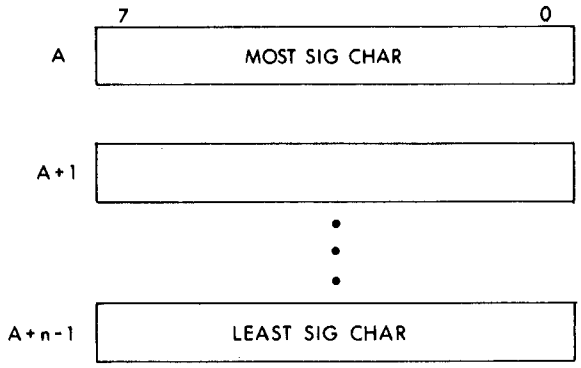
The character is used as an operand by CIS11 instructions. When it appears in a general register, the character is in the low order half; the high order half of the register must be zero. When it appears in the instruction stream, the character is in the low order half of a word; the high order half of the word must be zero. If the high order half of a word which contains a character is non-zero, the effect of the instruction which uses it will be unpredictable.

A character string is a contiguous sequence of bytes in memory that begins and ends on a byte boundary. It is addressed by its most significant character (lowest address). The highest address is the least significant character. It is specified by a two word descriptor with the attributes of length and lowest address. The length is an unsigned binary integer which represents the number of characters in the string and may range from 0 to 65,535. A character string with zero length is said to be vacant: its address is ignored. A character string with non-zero length is said to be occupied.

The character string descriptor is used as an operand by CIS11 instructions. It appears in two consecutive general registers, or in two consecutive words in memory pointed to by a word in the instruction stream. The following figure shows the descriptor for a character string of length "n" starting at address "A" in memory:



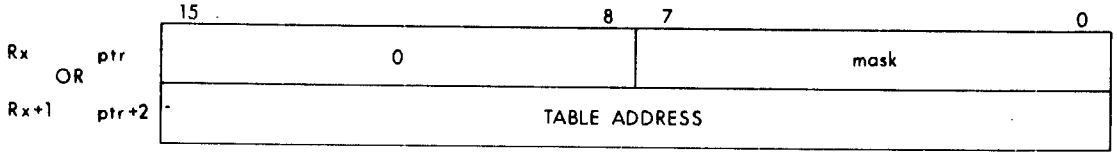
The following figure shows the character string in memory:



A "character set" is a subset of the 256 possible characters that can be encoded in a byte. It is specified by a descriptor which consists of the address of a 256-byte table and an 8-bit mask. The address is of the zeroeth byte in the table. Each byte in the table specifies up to eight orthogonal character subsets of which the corresponding character is a member. The mask selects which combinations of these orthogonal subsets comprise the entire character set. In effect, each bit in the mask corresponds to one of eight orthogonal subsets that may be encoded by the table. The mask specifies the union of the selected subsets into the character set. Typical sets would be: upper case, lower case, non-zero digits, end of line, etc.

Operationally, a character (char) is considered to be in the character set if the evaluation of (M[table.adr+char] AND mask) is not equal to zero. The character is not in the character set if the evaluation is zero. Each byte in the table indicates which combination of up to eight orthogonal character subsets (i.e., one for each of the eight bit vectors 00000001_2, 00000010_2, 00000100_2, 00001000_2, 00010000_2, 00100000_2, 01000000_2 and 10000000_2) the corresponding character is a member. The mask specifies which union of the eight orthogonal character subsets comprise the total character set. For example, if the eight bit vector 00000001_2 appearing in the table corresponds to the character subset of all upper case alphabetic characters, 00000010_2 appearing in the table corresponds to the character subset of all lower case alphabetic characters, and 00000100_2 appearing in the table corresponds to the decimal digits, then using the mask 00000011_2 with this table specifies the character set of all alphabetic characters, and using the mask 00000111_2 specifies the character set of all alphanumeric characters.

The character set descriptor is used as an operand by CIS11 intructions. It appears in two consecutive general registers, or in two consecutive words in memory pointed to by a word in the instruction stream. If the high order half of the first descriptor word is non-zero, the effect of an instruction which uses a character set will be unpredictable.



Character String Instructions

The character string operations conveniently provide most of the common, as well as time-consuming, functions that are encountered in commercial data and text processing applications.

Instructions are provided to move and to search character strings:

Character String Move Instructions

MOVC(I)	move character
MOVRC(I)	move reverse justified character
MOVTC(I)	move translated character

Character String Search Instructions

LOCC(I)	locate character
SKPC(I)	skip character
SCANC(I)	scan character
SPANC(I)	span character
CMPC(I)	compare character
MATC(I)	match character

The character string move instructions use character string descriptors as operands. These descriptors specify a source and a destination character string. The contents of the source are moved to the destination with alignment at either the most significant character

as in MOVC(I) and MOVTC(I), or the least signficant character as in MOVRC(I). If the source is longer than the destination, characters are truncated from the side opposite that of the alignment; if the destination is longer than the source, the destination is completed with fill characters on the side opposite that of the alignment. The MOVTC(I) instructions move a translated source string to a destination string.

The character string search instructions use a character string descriptor as one operand. The other operand is either a character, a character string descriptor, or a character set descriptor. These instructions are used to examine the source string to find the presence or absence of characters. The source string is processed from most significant to least significant character.

Conceptually, these instructions may be divided into three classes:

1. Character String Searches — CMPC(I) compares two character strings. The condition codes are set according to the comparison of the corresponding most significant unequal characters. MATC(I) finds an object string within a source string. This is the "instring" function that languages and text processing systems provide.
2. Character Searches — LOCC(I) finds the first occurrence of a given character in a string. SKPC(I) skips to the first non-occurrence of a given character in a string.
3. Character Set Searches — In these instructions, a string is examined until a member of a character set is either found as a SCANC(I), or not found as in SPANC(I). This aids the search for one of several delimiters such as "/", ",", CR, LF, FF, etc, or the passing of combinations of characters such as blanks, tabs, etc. LOCC(I) and SKPC(I) are optimizations of SCANC(I) and SPANC(I) in which the set consists of a single character.

The setting of condition codes reflects the results of the character string operations. For character string moves, the condition codes indicate whether the source and destination strings were of equal length, the source was shorter than the destination such that fill characters were used, or the source was longer than the destination such that characters were truncated. This is accomplished by setting the condition codes on the result of arithmetically comparing the initial source and destination lengths. For CMPC(I), the condition codes are the result of arithmetically comparing the most significant corresponding pair of unequal characters. For the other search instructions, they show whether or not the operand strings were completely examined.

The condition codes for some character string search instructions may be interpreted according to the notion of success or failure. Suc-

cess is the accomplishment of the instruction's task; failure is the inability to accomplish the task. Since the condition codes are set based on the results of the instruction, there is an indirect correspondence between these settings and success or failure. This correspondence is invariant within an instruction, but it is not the same for all search instructions. Therefore, different branch instructions must be used to test the operation of each instruction. They are summarized in the following table:

Instruction	Success	Failure
LOCC(I)	BNE	BEQ
SCANC(I)	BNE	BEQ
CMPC(I)	BEQ	BNE
MATC(I)	BNE	BEQ

The "register form" of character string instructions implicity find operands in the general registers. These operands include character, character string descriptor, character set descriptor, and translation table address. If an instruction does not use a register, its contents will be undisturbed. R0-R1 generally contain a source character string descriptor; R2-R3 generally contain a second source character string descriptor, or the destination string descriptor. The low order half of R4 is used as an explicit character. R4-R5 is used to contain a character set descriptor. R5 contains the starting address of a 256-byte table which is used for character translation.

When move instructions terminate, R0 contains the number of unmoved source characters, and R1, R2, and R3 are cleared. For search instructions, the registers are updated to represent descriptors for the resulting strings.

The "in-line form" of character string instructions find operands, or pointers to operands, in the instruction stream immediately following the opcode word. Operands which appear directly in the instruction stream include characters and translation table addresses. Descriptors are represented in the instruction stream by a single word whose contents are interpreted as a word address pointer to the two-word descriptor. These descriptors specify character strings and character sets. Some instructions return a character string descriptor in R0-R1.

In general, all character string instructions are unaffected by the overlapping of source or destination strings. The result of the move instructions is equivalent to having read the entire source string before storing characters in the destination. If the destination string of the MOVTC(I) instructions overlaps the translation table, the characters stored in the destination string will be unpredictable.

Decimal String Data Types

Two classes of decimal string data types—numeric strings and packed strings—are defined. Both have similar arithmetic and operational properties; they primarily differ in the representation of signs and the placement of digits in memory.

The numeric string data types are signed zoned, unsigned zoned, trailing overpunch, leading overpunched, trailing separate and leading separate. The packed string data types are signed packed and unsigned packed. Instructions which operate on numeric strings permit each numeric string operand to be separately specified; similarly, packed string instructions permit each packed string operand to be separately specified. Thus, within each of the two classes of decimal strings, the operands of an instruction may be of any data type within the appropriate class.

Decimal strings exist in memory as contiguous bytes which begin and end on a byte boundary. They represent numbers consisting of 0 to 31_{10} digits, in either sign-magnitude or absolute-value form. Sign-magnitude strings (SIGNED) may be positive or negative; absolute-value strings (UNSIGNED) represent the absolute value of the magnitude. Decimal numbers are whole integer values with an implied decimal radix point immediately beyond the least significant digit; they may be conceptually extended with zero digits beyond the most significant digit.

A 4-bit binary coded decimal representation is used for most digits in decimal strings. A four bit half byte is called a "nibble" and may be used to contain a binary bit pattern which represents the value of a decimal digit. The following table shows the binary nibble contents associated with each decimal digit:

digit	nibble	digit	nibble
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

Each decimal string data type may have several representations. These representations permit a certain latitude when accepting source operands. Decimal String data types have a PREFERRED representation, which is a valid source representation and which is used to construct the destination string. Additional ALTERNATE representations are provided for some decimal data types when accepting source operands.

Decimal strings used as source operands will not be checked for validity. Instructions will produce unpredictable results if a decimal string used as a source operand contains an invalid digit encoding, invalid sign designator, or, in the case of overpunched numbers, an invalid sign/digit encoding.

When used as a source, decimal strings with zero magnitude are unique, regardless of sign. Thus, both positive and negative zero have identical interpretations.

Conceptually, decimal string instructions first determine the correct result, and then store the decimal string representation of the correct result in the destination string. A result of zero magnitude is considered to be positively signed. If the destination string can contain more digits than are significant in the result, the excess most significant destination string digits have zero digits stored in them. If the destination string cannot contain all significant digits of the result, the excess most significant result digits are not stored; the instruction will indicate decimal overflow. Note that negative zero is stored in the destination string as a side effect of decimal overflow where the sign of the result is negative and the destination is not large enough to contain any non-zero digits of the result.

If the destination string has zero length, no result digits will be stored. The sign of the result will be stored in separate and packed strings, but not in zoned and overpunched strings. Decimal overflow will indicate a non-zero result.

Decimal String Descriptors

Decimal strings are represented by a two-word descriptor. The descriptor contains the length, data type, and address of the string. It appears in two consecutive general registers (register form of instructions), or in two consecutive words in memory pointed to by a word in the instruction stream (in-line form of instructions). The unused bits are reserved by the architecture and must be 0. The effect of an instruction using a descriptor will be unpredictable if any non-zero reserved field in the descriptor contains non-zero values or a reserved data type encoding is used. The design of the numeric and packed string descriptors are identical:

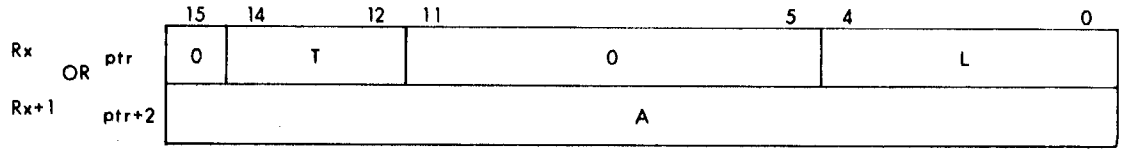
First Word

length <4:0>	Number of digits specified as an unsigned binary integer.
data type <14:12>	Specifies which decimal data type representation is used.

Second Word

address <15:0>	Specifies the address of the byte which contains the most significant digit of the decimal string.

The following figure shows the descriptor for a decimal string of data type "T" whose length is "L" digits and whose most significant digit is at address "A":



The encodings (in binary) for the NUMERIC string data type field are:

000	signed zoned
001	unsigned zoned
010	trailing overpunch
011	leading overpunch
100	trailing separate
101	leading separate
110	—reserved by the architecture
111	—reserved by the architecture

The encodings (in binary) for the PACKED string data type field are:

000	—reserved by the architecture
001	—reserved by the architecture
010	—reserved by the architecture
011	—reserved by the architecture
100	—reserved by the architecture
101	—reserved by the architecture
110	signed packed
111	unsigned packed

Packed Strings

Packed strings can store two decimal digits in each byte. The least significant (highest addressed) byte contains the sign of the number in bits <3:0> and the least significant digit in bits <7:4>.

Signed Packed Strings — The preferred positive sign designator is 1100_2; alternate positive sign designators are 1010_2, 1110_2 and 1111_2. The preferrred negative sign designator is 1101_2; the alternate negative sign designator is 1011_2. Source strings will properly accept both the preferred and alternate designators; destination strings will be stored with the preferred designator.

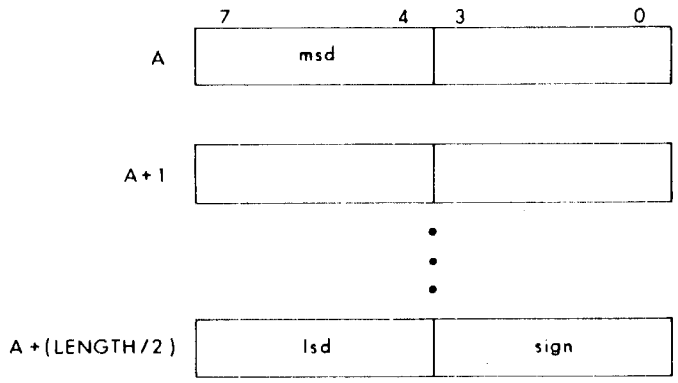
Unsigned Packed Strings — The unsigned sign designator is 1111_2.

PACKED SIGN NIBBLE:

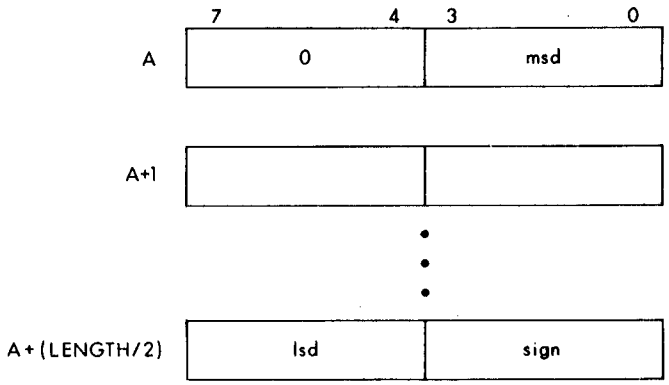
sign nibble	preferred designator	alternate designators
positive	1100_2	1010_2 1110_2 1111_2
negative	1101_2	1011_2
unsigned	1111_2	

For other than the least significant byte, bytes contain two consecutive digits—the one of lower significance in bits <3:0> and the one of higher significance in bits <7:4>. For numbers whose length is odd, the most significant digit is in bits <7:4> of the lowest addressed bytes. Numbers with an even length have their most significant digit in bits <3:0> of the lowest addressed byte; bits <7:4> of this byte must be zero for source strings, and are cleared to 0000_2 for destination strings. Numbers with a length of one occupy a single byte and contain their digit in bits <7:4>. The number of bytes which represent a packed string is [length/2]+1 (integer division where the fractional portion of the quotient is discarded).

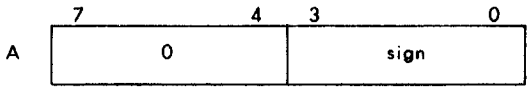
The following is packed string with an odd number of digits:



The following is a packed string with an even number of digits:



A zero length packed string occupies a single byte of storage; bits <7:4> of this byte must be zero for source strings, and are cleared to 0000_2 for destination strings. Bits <3:0> must be a valid sign for source strings, and are used to store the sign of the result for destination strings. When used as a source, zero length strings represent operands with zero magnitude. When used as a destination, they can only reflect a result of zero magnitude without indicating overflow. The following is a zero length packed string:



A valid packed string is characterized by:

1. A length from 0 to 31_{10} digits.
2. Every digit nibble is in the range 0000_2 to 1001_2.
3. For even length sources, bits <7:4> of the lowest addressed byte are 0000_2.
4. Signed Packed Strings—sign nibble is either 1010_2, 1011_2, 1100_2, 1101_2, 1110_2 or 1111_2.
5. Unsigned Packed Strings — sign nibble is 1111_2.

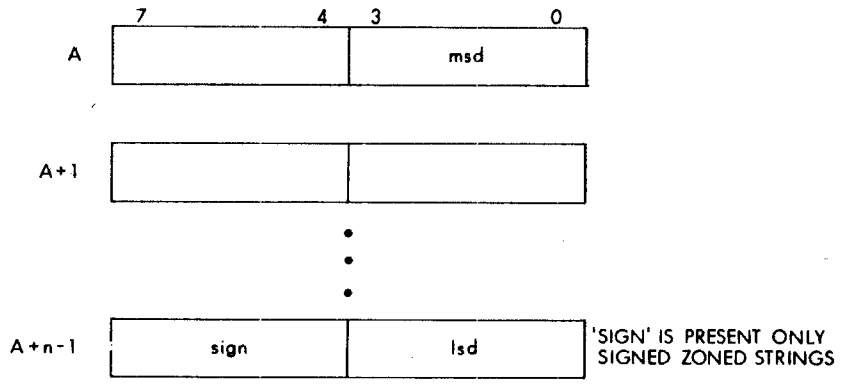
Zoned Strings

Zoned strings represent one decimal digit in each byte. Each byte is divided into two portions—the high order nibble (bits <7:4>) and the low order nibble (bits <3:0>). The low order nibble contains the value of the corresponding decimal digit.

Signed Zoned Strings — When used as a source string, the high order nibble of the least significant byte contains the sign of the number; the high order nibbles of all other bytes are ignored. Destination strings are stored with the sign in the high order nibble of the least significant byte, and 0011_2 in the high order nibble of all other bytes. 0011_2 in the high order nibble corresponds to the ASCII encoding for numeric digits. The positive sign designator is 0011_2; the negative sign designator is 0111_2.

Unsigned Zoned Strings — When used as a source string, the high order nibbles of all bytes are ignored. Destination strings are stored with 0011_2 in the high order nibble of all bytes.

The number of bytes needed to contain a zoned string is identical to the length of the decimal number.



A zero length zoned string does not occupy memory; the address portion of its descriptor is ignored. When used as a source, zero length strings provide operands with zero magnitude; when used as a destination, they can only accurately reflect a result of zero magnitude (the sign of the operation is lost). An attempt to store a non-zero result will be indicated by setting overflow.

A valid zoned string is characterized by:

1. A length from 0 to 31_{10} digits.
2. The low order nibbles of each byte are in the range 0000_2 to 1001_2.
3. Signed Zoned Strings—The high order nibble of the least significant byte is either 0011_2 or 0111_2.

Overpunch Strings

Overpunch strings represent one decimal digit in each byte. Trailing overpunch strings combine the encoding of the sign and the least significant digit; leading overpunch strings combine the encoding of the sign and the most significant digit. Bytes other than the byte in which the sign is encoded are divided into two portions—the high order nibble (bits <7:4>) and the low order nibble (bits <3:0>). The low order nibble contains the value of the corresponding decimal digit. When used as a source string, the high order nibble of all bytes which do not contain the sign are ignored. Destination strings are stored with 0011_2 in the high order nibble of all bytes which do not contain the sign. 0011_2 in the high order nibble corresponds to the ASCII encoding for numeric digits.

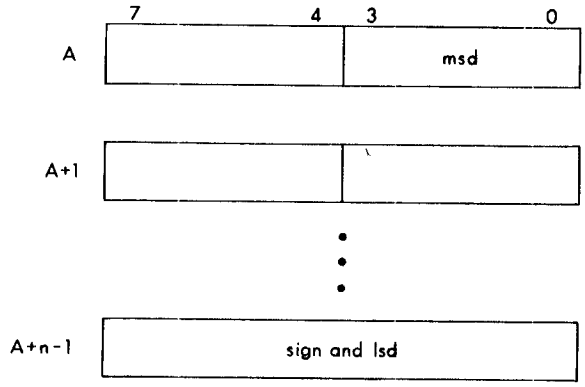
The following table shows the sign of the decimal string and the value of the digit which is encoded in the sign byte. Source strings will properly accept both the preferred and alternate designators; destination strings will store the preferred designator. The preferred designators correspond to the ASCII graphics "A" to "R," "{," and "}." The alternate designators correspond to the ASCII graphics "0" to "9," "[," "?," "]," "!" and ":"

OVERPUNCH SIGN/DIGIT BYTE:

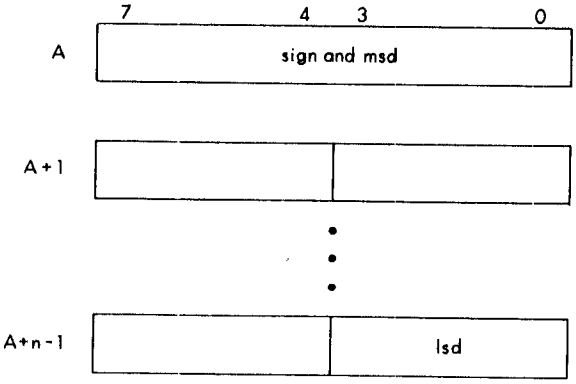
overpunch sign/digit	**preferred designator**	**alternate designators**
+0	01111011_2	00110000_2, 01011011_2, 00111111_2
+1	01000001_2	00110001_2
+2	01000010_2	00110010_2
+3	01000011_2	00110011_2
+4	01000100_2	00110100_2
+5	01000101_2	00110101_2
+6	01000110_2	00110110_2
+7	01000111_2	00110111_2
+8	01001000_2	00111000_2
+9	01001001_2	00111001_2
−0	01111101_2	01011101_2, 00100001_2, 00111010_2
−1	01001010_2	
−2	01001011_2	
−3	01001100_2	
−4	01001101_2	
−5	01001110_2	
−6	01001111_2	
−7	01010000_2	
−8	01010001_2	
−9	01010010_2	

The number of bytes needed to contain an overpunch string is identical to the length of the decimal number.

The following is a trailing overpunch string:



The following is a leading overpunch string:



A zero length overpunch string does not occupy memory; the address portion of its descriptor is ignored. When used as a source, zero length strings provide operands with zero magnitude; when used as a destination, they can only accurately reflect a result of zero magnitude (the sign of the operation is lost). An attempt to store a non-zero result will be indicated by setting overflow.

A valid overpunch string is characterized by:

1. A length from 0 to 31_{10} digits.
2. The low order nibble of each digit byte is in the range 0000_2 to 1001_2.
3. The encoded sign/digit byte contains values from the above table of preferred and alternate overpunch sign/digit values.

Separate Strings

Separate strings represent one decimal digit in each byte. Trailing separate strings encode the sign in a byte immediately beyond the least significant digit; leading separate strings encode the sign in a byte immediately beyond the most significant digit. Bytes other than the byte in which the sign is encoded are divided into two portions—the high order nibble (bits <7:4>) and the low order nibble (bits <3:0>). The low order nibble contains the value of the corresponding decimal digit.

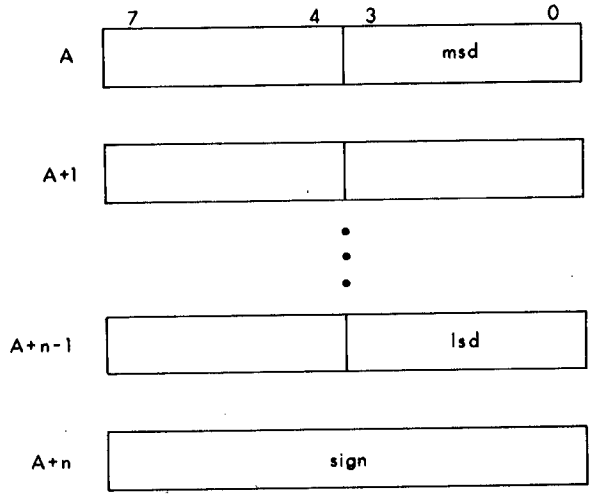
When used as a source string, the high order nibbles of all digit bytes are ignored. Destination strings are stored with 0011_2 in the high order nibble of all digit bytes. 0011_2 in the high order nibble corresponds to the ASCII encoding for numeric digits. The preferred positive sign designator is 00101011_2 and the alternate positive sign designator is 00100000_2. The negative sign designator is 00101101_2. These designators correspond to the ASCII encoding for "+," "space," and "−."

SEPARATE SIGN BYTE:

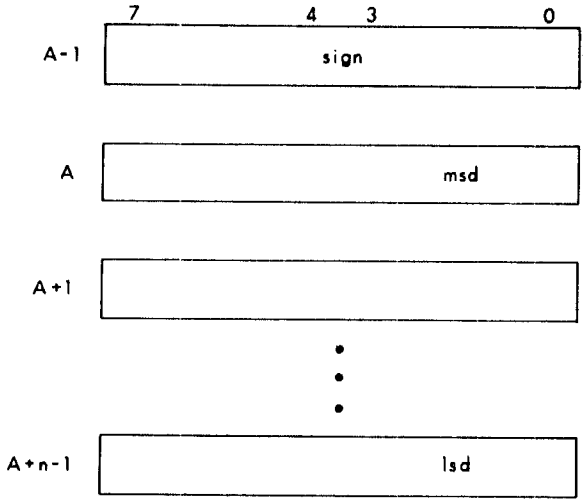
sign byte	**preferred designator**	**alternate designator**
positive	00101011_2	00100000_2
negative	00101101_2	

The number of bytes needed to contain a leading or trailing separate string is identical to (length+1).

The following is a trailing separate string:



The following is a leading separate string:



A zero length separate string occupies a single byte of memory which contains the sign. When used as a source, zero length strings provide operands with zero magnitude; when used as a destination, they can only reflect a result of zero magnitude without indicating overflow; the sign of the result is stored.

The following is a zero length trailing separate string:

The following is a zero length leading separate string:

A valid separate string is characterized by:

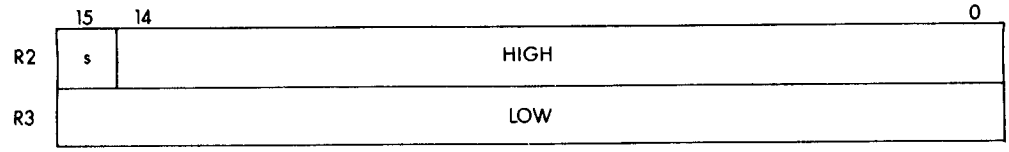
1. A length from 0 to 31_{10} digits.
2. The low order nibble of each digit byte is in the range 0000_2 to 1001_2.

3. The sign byte is either 00100000_2, 00101011_2 or 00101101_2.

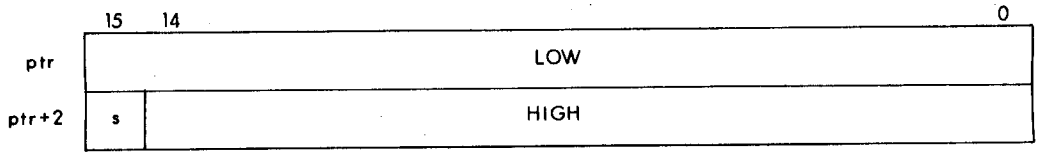
Long Integer

Long integers are 32-bit binary 2's complement numbers organized as two words in consecutive registers or in memory—no descriptor is used. One word contains the high order 15 bits. The sign is in bit<15>; bit<14> is the most significant. The other word contains the low order 16 bits with bit<0> the least significant. The range of numbers that can be represented is −2,147,483,648 to +2,147,483,647.

The register form of decimal convert instructions use a restricted form of long integer with the number in the general register pair R2-R3:



The in-line form of decimal convert instructions reference the long integer by a word address pointer which is part of the instruction stream:



Note that these two representations of long integers differ. There is no single representation of long integer among EAE, EIS, FPP and software. The "register form" was selected to be compatible with EIS; the "in-line form" was selected to be compatible with current standard software usage.

Decimal String Instructions

The decimal string instruction groups aid manipulation of decimal data. Several numeric (byte) and packed decimal data types are supported. Instructions are provided for basic arithmetic operations, as well as for compare, shift, and convert functions.

Instructions

Each arithmetic, shift and compare instruction operates on a single class of data type. Both numeric and packed string instructions are

provided for most operations. Convert instructions have a source operand of one data type and a destination operand of another data type. Decimal string instructions specify to which class each of their decimal string operands belong. The data type supplied as part of each operand's descriptor may be any valid data type of the class. This permits a general mixing of data types within numeric and packed classes.

The data types on which an instruction operates are designated by the last letter(s) of the opcode mnemonic. "N" denotes numeric strings, "P" denotes packed strings, and "L" denotes long binary integers.

The arithmetic instructions are ADDN(I), ADDP(I), SUBN(I), SUBP(I), MULP(I) and DIVP(I). ASHN(I) and ASHP(I) shift a decimal string by a specified number of digit positions (either direction) with optional rounding, and store the result in the destination string. Thus, they effectively multiply or divide by a power of ten. If the shift count is zero, these shift instructions can be used simply to move decimal strings (destinations are stored with preferred representation). Move negated may be accomplished by using SUBN(I) or SUBP(I). Arithmetic comparison instructions, CMPN(I) and CMPP(I), are provided to examine the relative difference between two decimal strings.

CVTNL(I) and CVTPL(I) convert a decimal string to a long (32-bit) 2's complement integer. CVTLN(I) and CVTLP(I) convert a long integer to a decimal string. CVTNP(I) and CVTPN(I) convert between numeric and packed decimal strings.

The instructions are:

Numeric String Instructions

ADDN(I)	add numeric
SUBN(I)	subtract numeric
ASHN(I)	arithmetic shift numeric
CMPN(I)	compare numeric

Packed String Instructions

ADDP(I)	add packed
SUBP(I)	subtract packed
MULP(I)	multiply packed
DIVP(I)	divide packed
ASHP(I)	arithmetic shift packed
CMPP(I)	compare packed

Convert Instructions

CVTNL	convert numeric to long
CVTLN	convert long to numeric
CVTPL	convert packed to long
CVTLP	convert long to packed
CVTNP	convert numeric to packed
CVTPN	convert packed to numeric

Condition Codes

For instructions which store a value in a destination string, the N and Z bits reflect the value stored. The N bit indicates a negative destination; the Z bit indicates a destination having zero magnitude. A destination string with zero magnitude is considered to be positive (even if a negative zero was stored as a consequence of decimal overflow). Thus, the setting of N and Z are mutually exclusive.

The V bit will indicate whether the destination string accurately represents the result of the instruction. It is also set if division by zero was attempted. If the V bit is set, the destination string will represent the least significant portion of the result (truncated). If the V bit is cleared, the destination represents the true result.

For DIVP(I), C indicates division by zero. Otherwise, C is always cleared.

For comparisions using the CMPN(I) and CMPP(I) instructions, the N and Z bits reflect the signed relationship between the source strings. The signed branch instructions can test the result. V and C are cleared.

For instructions which return a long integer value, N reflects the sign of the 2's complement integer, and Z indicates whether it was zero. V indicates whether the long integer could not contain all significant digits and sign of the result. CVTNL(I) and CVTPL(I) also use C to represent a borrow from a more significant portion of the long binary result. Otherwise, C is cleared.

Operand Delivery

The "register form" of decimal string instructions implicitly find their operands in the general registers. These operands include decimal string descriptors, long binary integers, and shift descriptor words. If an instruction does not use a register, its contents will be undisturbed. R0-R1 generally contain the first source descriptor, R2-R3 generally contain the second source descriptor, and R4-R5 generally contain the destination descriptor. ASHN and ASHP use R4 to contain a shift descriptor word. CVTLN, CVTLP, CVTNL and CVTPL use R0-R1 to

contain a decimal string descriptor, and R2-R3 for the long integer. When an instruction is completed, the source descriptor registers are cleared.

The "in-line form" of decimal string instructions find their operands, or pointers to descriptors, in the instruction stream immediately following the opcode word. Operands which appear directly in the instruction stream are shift descriptor words. Operands which are represented in the instruction stream by a pointer containing the word address of the descriptor are decimal string descriptors and long binary integers. No in-line form of decimal string instructions modify R0-R6.

Data Overlap

The operation of decimal string instructions is unaffected by any overlap of the source operands provided that each source operand is a valid representation of the specified data type.

The overlap of the destination string and any of the source strings will, in general, produce unpredictable results. However, ADDN(I), ADDP(I), SUBN(I) and SUBP(I) will permit the destination string to overlap either or both source strings only if all corresponding digits of the strings are in coincident bytes in memory. This facilitates two-address arithmetic.

Commercial Load Descriptor Instructions

The commercial load descriptor instructions augment the character and decimal string instructions by efficiently loading the general registers with string descriptors. Two forms of instructions are provided. The L2Dr instructions load two string descriptors into the general registers. The first descriptor is loaded into R0-R1 and the second descriptor is loaded into R2-R3. This instruction supports equal length character string move, equal length character string compare, character string matching, and decimal string compare.

The second form, the L3Dr instructions, take three descriptors. The first is loaded into R0-R1, the second into R2-R3, and the third into R4-R5. The instruction supports 3-address arithmetic.

The condition codes are not affected.

Words containing the addresses of the descriptors (two for L2Dr and three for L3Dr) are in consecutive locations in memory. The descriptor addresses are found by applying the addressing mode @(Rr)+ once for each descriptor. The value of r is encoded as the low order three bits of the instruction's opcode. If $0 \leq r \leq 5$, then r can be thought of as the base address of a small table in memory, where each entry in the table contains the address of a descriptor. If r =6, then the instructions

effectively pop the addresses of descriptors off of the stack. If r=7, then the descriptor addresses are contiguous with the instruction's opcode word.

The string descriptors are two words long. The address of the descriptor is that of the low order word. It is loaded into the corresponding even register. The high order word of the descriptor is loaded into the corresponding odd register. Note that although these instructions are described in terms of string descriptors, they are applicable for other instances where two consecutive words in memory referenced by a pointer are to be copied into even-odd general register pairs.

The instructions are:

Commercial Load Descriptor Instructions

L2D0	load 2 descriptors using @(R0)+
L2D1	load 2 descriptors using @(R1)+
L2D2	load 2 descriptors using @(R2)+
L2D3	load 2 descriptors using @(R3)+
L2D4	load 2 descriptors using @(R4)+
L2D5	load 2 descriptors using @(R5)+
L2D6	load 2 descriptors using @(R6)+
L2D7	load 2 descriptors using @(R7)+
L3D0	load 3 descriptors using @(R0)+
L3D1	load 3 descriptors using @(R1)+
L3D2	load 3 descriptors using @(R2)+
L3D3	load 3 descriptors using @(R3)+
L3D4	load 3 descriptors using @(R4)+
L3D5	load 3 descriptors using @(R5)+
L3D6	load 3 descriptors using @(R6)+
L3D7	load 3 descriptors using @(R7)+

EXTENDED INSTRUCTION OVERVIEW

Opcode Utilization and Availability

Opcodes in the following ranges are reserved and are not available for usage:

000010_8- 000077_8
007000_8- 007777_8
107000_8- 107777_8
170006_8
170010_8
170013_8- 170077_8

In general, extended PDP-11 instructions will use opcodes in the range 076000_8- 076777_8>

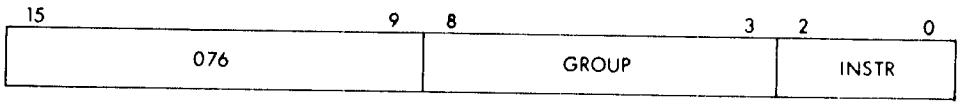
The extended opcode space is divided into 64 groups of eight instructions each. Groups are treated as integral entities. A group is declared "closed" when all eight instructions in it have been defined or when no further instructions are admissible into it. Otherwise a group is considered "open," and future instructions may be added to it. The opcode groups are specified later in this chapter.

PDP-11 extended instructions can be defined (a) to operate on implicitly specified operands or (b) to require explicit operand specifiers in the instruction stream, or both. Explicit operand specifiers may use either (i) a general operand-specifier format or (ii) an opcode-specific operand-specifier format.

If an extended instruction uses only implicit operands, only the opcode will appear in the instruction-stream (see section "Operands for Extended Instructions").

If an extended instruction uses explicit operands, the opcode word is followed in the instruction stream by as many operand specifiers and operands as the specification of the instruction requires. As in traditional PDP-11 instructions, explicit general operand specifiers using modes 6 or 7 or using R7 in modes 2 or 3 will also require additional words in the instruction stream (see section "Operands for Extended Instructions").

The extended instruction opcode word is structured as follows:



Bits <8:3> contain the group code. Bits <2:0> specify the instruction within the group.

Extended Instruction Groups

The extended instruction groups are defined in the following table, where X represents the set of eight instructions in the group.

Code	Group	Status
07600X		open
07601X		open
07602X	Commercial Load 2 Descriptors	closed

Code	Group	Status
07603X	Character String Move	closed
07604X	Character String Search	closed
07605X	Numeric String	closed
07606X	Commercial Load 3 Descriptors	closed
07607X	Packed String	closed
07610X		open
07611X		open
07612X		open
07613X	Character String Move (in-line)	closed
07614X	Character String Search (in-line)	closed
07615X	Numeric String (in-line)	closed
07616X		open
07617X	Packed String (in-line)	closed
07620X		open
07621X		open
07622X		open
07623X		open
07624X		open
07624X		open
07625X		open
07626X		open
07627X		open
07630X		open
07631X		open
07632X		open
07633X		open
07634X		open
07635X		open
07636X		open
07637X		open
07640X		open
07641X		open
07642X		open
07643X		open
07644X		open
07645X		open
07646X		open
07647X		open
07650X		open
07651X		open
07652X		open
07653X		open

Code	Group	Status
07654X		open
07655X		open
07656X		open
07657X		open
07660X	Processor-Specific #0	open
07661X	Processor-Specific #1	open
07662X	Processor-Specific #2	open
07663X	Processor-Specific #3	open
07664X	Processor-Specific #4	open
07665X	Processor-Specific #5	open
07666X	Processor-Specific #6	open
07667X	Processor-Specific #7	open
07670X	CSS/Customer #0	open
07671X	CSS/Customer #1	open
07672X	CSS/Customer #2	open
07673X	CSS/Customer #3	open
07674X	CSS/Customer #4	open
07675X	CSS/Customer #5	open
07676X	CSS/Customer #6	open
07677X	CSS/Customer #7	open

The extended instruction groups fall into three major categories:

1. The group 07600X - 07657X is for instructions which will be of general use across the range of PDP-11 processors. The opcodes in this range will be characterized as (a) uniquely and immutably defined and (b) reasonable for implementation on all processor models of the PDP-11 family.
2. The groups 07660X - 07667X are for instructions which will be used only on specific processors of the PDP-11 family. These, too, will be uniquely and immutably defined, but each opcode will be restrictively assigned to a specific processor model and may not be implemented on other processors.
3. The groups 07670X - 07677X will neither be uniquely nor immutably defined but will be left available for customer usage.

Operands for Extended Instructions

Operands for extended instructions may be implicitly or explicitly specified. Explicit operands are specified, either in a general or in an opcode-specific manner, through information expressed directly in the instruction stream. R7 is conceptually incremented by two as each word which contains an operand-specifier or operand in the instruction stream is fetched.

Implicitly specified operands do not appear in the instruction stream. If an instruction uses an implicitly specified operand, the definition of that instruction will specify the exact location and form of such an operand.

Implicitly specified operands may be defined to be located:

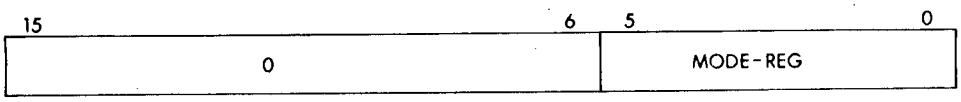
1. In the general-purpose registers
2. In defined machine registers
3. On the R6 stack
4. In defined locations in the virtual address space
5. In defined locations in the physical address space

The definition of an instruction may specify that operands immediately follow it in the instruction stream. The format and interpretation of such operands can be specified in an opcode-specific manner and will so be defined in the description of the instruction.

When an instruction uses explicit general operand specifiers, the operand specifiers shall immediately follow the extended opcode in the instruction stream. As many operand specifiers as the instruction requires follow in consecutive order.

Instructions which use a single general operand will use the single-operand-specifier format. Instructions which require two consecutive explicit operands will use the double-operand-specifier format. Instructions which use more than two consecutive explicit operands will specify the operands in a succession of double-operand-specifiers, and the last operand, when there are an odd number of operands, will be specified in the single-operand format.

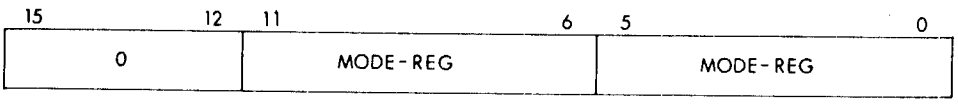
The single-operand specifier consists of a word in the following format:



Bits <15:6> must be 0. If not, a trap through vector 4 (invalid instruction specifier) will be taken.

Bits <5:0> specify the operand in the traditional PDP-11 mode-register format.

The double-operand specifier consists of a word in the following format:



Bits <15:12> must be 0, else a trap through vector 4 (invalid instruction specifier) will be taken.

Bits <11:6> specify the first of the two operands, and bits <5:0> specify the second. Each operand is specified in the traditional PDP-11 mode-register format.

Additional operand words are required in the instruction stream for as many general operand specifiers as use mode 6 or 7 (with any register) or as utilize modes 2 or 3 with register 7. These additional operand words immediately follow the operand-specifier word which calls for them.

For example, a hypothetical instruction

ZAP #A, (R1)+, B(R4), C, D

requiring explicit general operands would appear in the instruction stream as the following eight words:

opcode zzz for ZAP	076zzz
specifiers for operands 1 & 2	002721
value of literal A	aaaaaa
specifiers for operands 3 & 4	006467
value of index B	bbbbbb
displacement off PC for address of C	cccccc
specifier for operand 5	000067
displacement off PC for address of D	dddddd

SUSPENDABLE INSTRUCTIONS

The intent of defining instruction suspendability is to establish a means for providing reasonable interrupt latency and does not presume to endow extended instructions with an ability to recover from trap conditions from which sequences of basic instructions cannot recover.

Suspension-events refer primarily to events which occur asynchronously to the instruction's execution; these are specifically the interrupts generated by I/O peripheral devices, power-fail traps, and float-

ing point processor exceptions. Secondarily, suspension-events can refer also to those synchronous trap events which occur only for information notification purposes and do not imply that the integrity of the instruction's execution is in jeopardy. Such suspension events include "yellow zone" traps.

Each extended instruction is classified either as "non-suspendable" or as "potentially suspendable."

As explained below, two implementation choices are possible for non-suspendable instructions, and three are possible for potentially suspendable instructions. The following diagram can serve as a guide to subsequent portions of this section.

Architecture	**Implementation**
A) Non-Suspendable	1) non-interruptible
	2) restartable
B) Potentially Suspendable	1) non-interruptible
	2) restartable
	3) suspendable

A non-suspendable instruction has no architectural mechanism to allow it to be suspended, while a suspension-event is serviced, and then subsequently to be resumed.

A non-suspendable instruction may be implemented either as "non-interruptible" or as "restartable."

If an instruction is implemented as non-interruptible, then once its execution has commenced, the processor will defer service of all suspension-events until after the completion of the instruction.

If an instruction is implemented as restartable, then the instruction may be aborted to allow the processor to service suspension-events. The programmer visible state will be restored to that which existed immediately prior to the instruction execution. Upon the processor's return from servicing the suspension-event, the instruction will be started afresh.

Potentially suspendable instructions have a defined architectural mechanism, (PS<8> as described below), by which they can be suspended in mid-execution to allow the processor to service suspension-events and then subsequently to be resumed from the point where they had been suspended.

A potentially suspendable instruction may be implemented either as "non-interruptible," as "restartable," or as "suspendable."

The presence of suspension-events may cause certain extended instructions to be suspended on some processors. If the instruction is suspended, PS<8> will be set, R7 will be backed up to address the opcode word, and the suspension-event will be serviced. When the instruction is resumed, PS<8> indicates that execution of the instruction has previously begun.

In order to make these instructions suspendable on all processors, the instruction state is part of the user state which is saved by interrupt handling routines. This includes the general registers, condition codes and memory. This state is processor dependent when suspended. Software should not attempt to interpret or modify this state; it must only be saved and restored. Up to 64_{10} words of internal instruction state may also have been pushed onto the stack. This state must not be modified by software. The instruction will remove this state from the stack when it is resumed.

If PS<8> is set prior to executing a potentially suspendable instruction, the effect of the instruction is unpredictable.

At the normal completion of an potentially suspendable instruction, PS<8> will be cleared.

In order to promote uniform nomenclature, the name of the bit PS<8> will be "Instruction Suspension" with the corresponding mnemonic "IS".

All suspendable instructions will use PS<8> to indicate instruction suspension. If, when a potentially suspendable instruction is executed, PS<8> is clear, it means that the instruction is being commenced; if it is set, it means that the instruction is being resumed. PS<8> will be cleared when:

1. A suspended instruction successfully completes.
2. The processor powers-up.
3. A new PS is fetched from vector location with PS<8> clear.
4. RTI or RTT is executed with new PS<8> clear.
5. It is explicitly cleared by an instruction.

PS<8> will be set when:

1. A potentially suspendable instruction is interrupted and wishes to be suspended.
2. A new PS is fetched from vector location with PS<8> set.
3. RTI or RTT is executed with PS<8> set.
4. It is explicitly set by an instruction.

The setting of this bit will have no effect on instructions which are not potentially suspendable; such instructions will not implicitly modify this bit.

When an instruction is suspended, the following state may contain information vital to the resumption of the instruction. The information must be preserved and restored prior to restarting the suspended instruction. This information may vary from one execution of the instruction to another.

1. General registers R0 through R5.
2. Condition code bits (PS<3:0>).
3. Up to 64_{10} words on the stack of the context in which the suspended instruction was executing.
4. Any destinations used by the instruction.

Stack Utilization

Extended instructions may use the R6 stack for temporary "scratch" state storage.

The maximum number of additional words which an extended instrucion may claim on the R6 stack is 64_{10}. The reason for imposing a limit is to ensure that system software can adequately provide for worst-case stack allocation requirements. In addition to the above restriction, the normal PDP-11 stack-limit mechanism remains in effect for extended instructions just as it does for any other instruction.

If an extended instruction is interrupted, R6 must have been updated to encompass any additional stack storage still required for completion of the instruction.

All extended instructions will support dynamic stack allocation facilities used by some software systems. This means that memory management traps which result from over-extending the stack area must be survivable. If insufficient stack space exists, the instruction must terminate by a memory management abort in such a way that if additional stack space were allocated, the instruction could be successfully restarted.

Unpredictable Conditions

"Unpredictable" means that the outcome is indeterminate and non-repeatable. Either the result of an instruction or the effect of an instruction can be unpredictable. When the results of an instruction are unpredictable, the condition codes and destination operands (but not their descriptors) will contain unpredictable values; destinations may not even contain valid results. When the effect of an instruction is unpredictable, the entire user or process state, and not only the por-

tion typically used by the instruction, will be unpredictable. In a machine with multiple modes and address spaces, an unpredictable operation in a less privileged mode will not affect the state of a more privileged mode, nor will it result in accesses to memory from user mode which are outside the mapped limits of the user's program.

Note that architectural constraints exist on unpredictable effects. In particular, an unpredictable effect which manifests itself as a trap must meet all the requirements for the particular trap.

Implementors are encouraged to select the manifestations of unpredictable results and effects to be such that their occurrence is visible to software at the earliest possible time.

Multiprogramming Integrity

Machine implementations shall ensure that, under all initial settings of registers and memory, extended instructions shall not violate any bound implicit in multiprogrammed operation. Specifically, the following are to be avoided:

1. A less-privileged program escaping into a higher-privileged mode.
2. A program escaping beyond its address-mapping limits.
3. A non-interruptible or non-terminating sequence.
4. Excessive interrupt latency.

ADDN/ADDP/ADDNI/ADDPI

Purpose: Add Decimal

Operation: dst ← src2 + src1

Condition Codes:
- N: set if dst < 0; cleared otherwise
- Z: set if dst = 0; cleared otherwise
- V: set if dst cannot contain in all significant digits of the result; cleared otherwise
- C: cleared

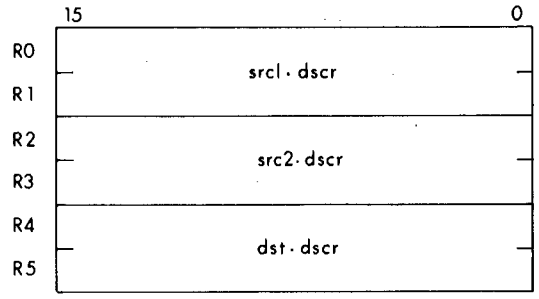
Opcodes:

ADDN	076050
ADDP	076070
ADDNI	076150
ADDPI	076170

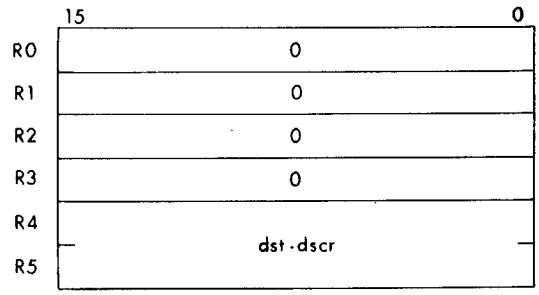
Description: Src1 is added to src2, and the result is stored in the destination string. The condition codes reflect the value stored in destination string, and whether all significant digits were stored.

Register Form—ADDN and ADDP

When the instruction starts, the operands must have been placed in the general registers. The first source descriptor is placed in R0-R1, the second source descriptor is placed in R2-R3, and the destination descriptor is placed in R4-R5:



When the instruction is completed, the source descriptor registers are cleared:



In-line Form—ADDNI and ADDPI

Each word address pointer which follows the opcode word in the instruction stream refers to a two-word decimal string descriptor. R0-R6 are unchanged when the instruction is completed.

Notes:

1. The operation of these instructions is unaffected by any overlap of the source strings provided that each source string is a valid representation of the specified data type.
2. Source strings may overlap the destination string only if all corresponding digits of the strings are in coincident bytes in memory.

ASHN/ASHP/ASHNI/ASHPI

Purpose: Arithmetic Shift Decimal

Operation: dst ← src * (10 ** shift count)

Condition Codes:
- N: set if dst <0; cleared otherwise
- Z: set if dst = 0; cleared otherwise
- V: set if dst cannot contain all significant digits of the result; cleared otherwise
- C: cleared

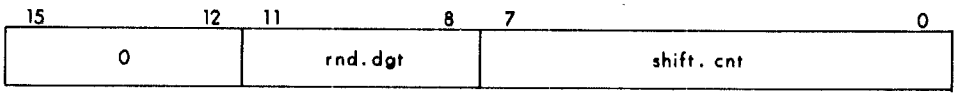
Opcodes:

ASHN	076056
ASHP	076076
ASHNI	076156
ASHPI	076176

Description: The decimal number specified by the source descriptor is arithmetically shifted and stored in the area specified by the destination descriptor. The shifted result is aligned with the least significant digit position in the destination string. The shift count is a 2's complement byte whose value ranges from -128_{10} to $+127_{10}$. If the shift count is positive, a shift in the direction of least to most significant digits is performed. A negative shift count performs a shift from most to least significant digit. Thus, the shift count is the power of ten by which the source is multiplied; negative powers of ten effectively divide. Zero digits are supplied for vacated digit positions. A zero shift count will move the source to the destination. The condition codes reflect the value stored in the destination string, and whether all significant digits were stored.

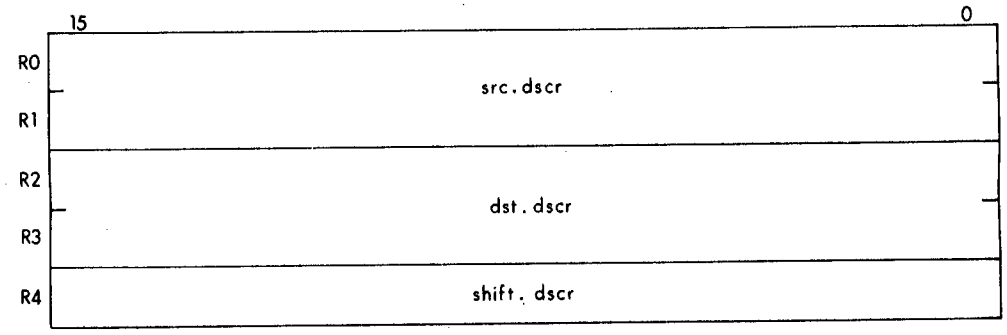
A negative shift count invokes a rounding operation. The result is constructed by shifting the source the specified number of digit positions. The rounding digit is then added to the most significant digit which was shifted out. If this sum is less than 10_{10}, the shifted result is stored in the destination string. If the sum is 10_{10} or greater, the magnitude of the shifted result is increased by 1 and then stored in the destination string. If no rounding is desired, the rounding digit should be zero.

The shift count and rounding digit are represented in a single word referred to as the shift descriptor. Bits <15:12> of this word must be zero.

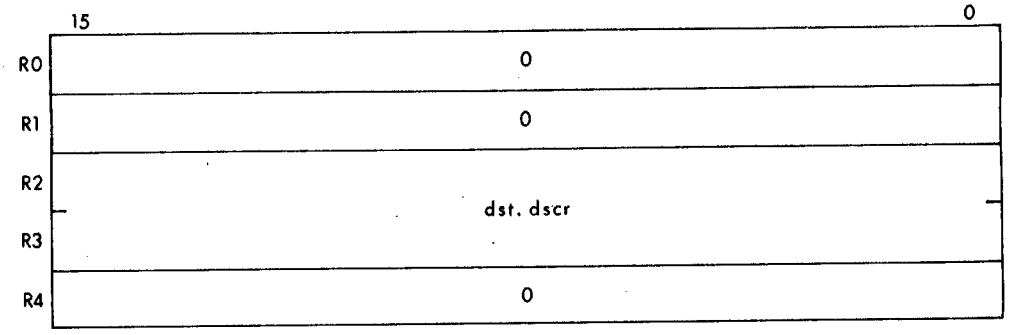


Register Form—ASHN and ASHP

When the instruction starts, the operands must have been placed in the general registers. The source descriptor is placed in R0-R1, the destination descriptor is placed in R2-R3, and the shift descriptor is placed in R4.



When the instruction is completed, the source descriptor registers and shift descriptor register are cleared.



In-line Form—ASHNI and ASHPI

The words which follow the opcode word in the instruction stream are a word address pointer to a two-word decimal string source descriptor, a word address pointer to a two-word decimal string destination descriptor, and a shift descriptor word. R0-R6 are unchanged when the instruction is completed.

Notes:

1. If bits <15:12> of the shift descriptor word are not zero, the effect of the instruction is unpredictable.
2. If bits <11:8> of the shift descriptor are not a valid decimal digit, the results of the instruction are unpredictable.
3. Any overlap of the source and destination strings will produce unpredictable results.

CMPC/CMPCI

Purpose: Compare Character

Operation: src1 is compared with src2 (src1-src2)

Condition Codes: The condition codes are based on the arithmetic comparison of the most significant pair of unequal src1 and src2 characters (src1.byte-src2.byte)

- N: set if result < 0; cleared otherwise
- Z: set if result = 0: cleared otherwise
- V: set if there was arithmetic overflow, that is, src1.byte<7> and src2.byte<7> were different, and src2.byte<7> was the same as bit <7> of (src1.byte-src2.byte); cleared otherwise
- C: cleared if there was a carry from the most significant bit of the result; set otherwise.

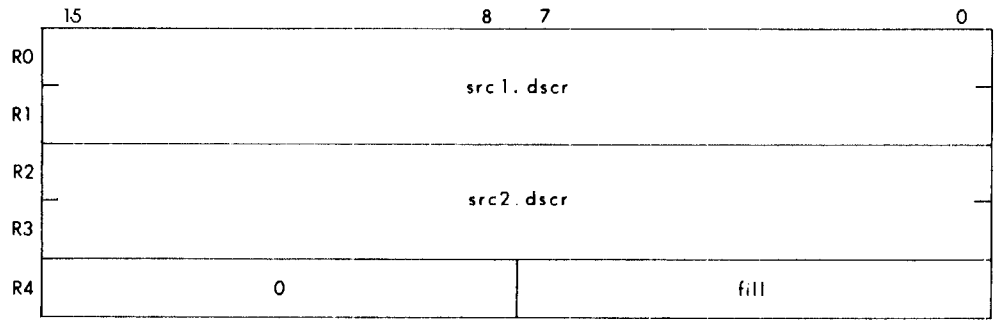
Opcodes:

CMPC	076044
CMPCI	076144

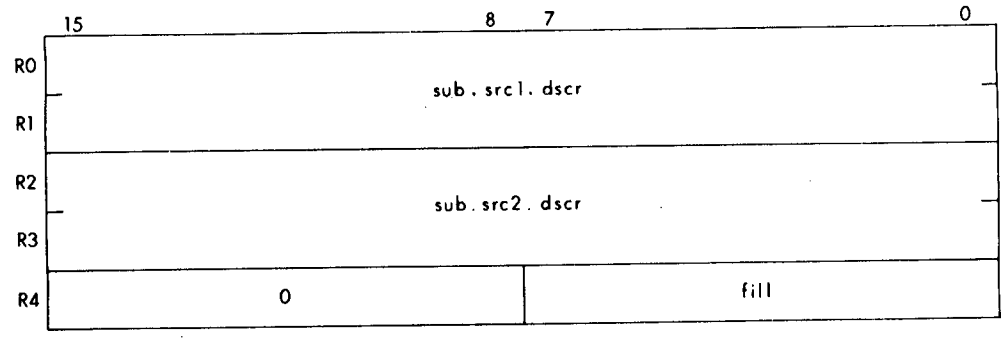
Description: Each character of src1 is compared with the corresponding character of src2 by examining the character strings from most significant to least significant characters. If the character strings are of unequal length, the shorter character string is conceptually extended to the length of the longer character string with fill characters beyond its least significant character. The instruction terminates when the first corresponding unequal characters are found or when both character strings are exhausted. The condition codes reflect the last comparison, permitting the unsigned branch instructions to test the result.

Register Form—CMPC

When the instruction starts, the operands must have been placed in the general registers. The first source character string descriptor is placed in R0-R1, the second source character string descriptor is placed in R2-R3, the fill character is placed in R4<7:0>, and R4<15:8> must be zero.



The instruction terminates with sub-string descriptors in R0-R1 and R2-R3 which represent the portion of each source character string beginning with the most significant corresponding unequal characters. R0-R1 contain a descriptor for the unequal portion of the original src1 string; R2-R3 contain a descriptor for the unequal portion of the original src2 string. A vacant character string descriptor indicates that the entire source character string was equal to the corresponding portion of the other source character string, including extension by the fill character; its address is one greater than that of the least significant character of the character string.



In-line Form—CMPCI The words which follow the opcode word in the instruction stream are a word address pointer to a two-word character string src1 descriptor, a word address pointer to a two-word character string src2 descriptor, and a word whose low order half contains the fill character and whose high order half must be zero. R0-R6 are unchanged when the instruction is completed.

Notes:

1. The operation of this instruction is unaffected by any overlap of the source character strings.
2. If the src1 character string is vacant, the fill character will be compared with src2. If the src2 character string is vacant, the fill character will be compared with src1. If both character strings are vacant, the condition codes will indicate equality.
3. CMPC—If an initial source character string descriptor is vacant, the resulting sub-string descriptor is the same as the original character string descriptor.
4. A test for success is BEQ; a test for failure is BNE.
5. When the instruction terminates, the condition codes will be set as if a CMPB instruction operated on the most significant unequal characters. If both strings are initially vacant or are identical, the condition codes will be set as if the last characters to be compared were identical. This results in equality with N cleared, Z set, V cleared, and C cleared.
6. Both CMPC and CMPCI update the condition codes. CMPC returns sub-string descriptors.

CMPN/CMPP/CMPNI/CMPPI

Purpose: Compare Decimal

Operation: src1 is compared with src2 (src1-src2)

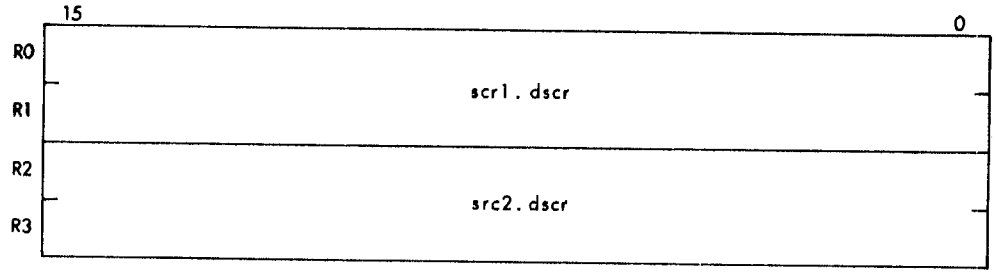
Condition Codes:
- N: set if src1 < src2; cleared otherwise
- Z: set if src1 = src2; cleared otherwise
- V: cleared
- C: cleared

Opcodes:

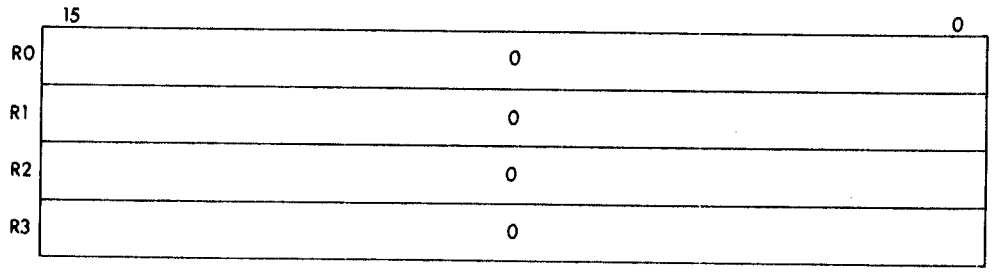
CMPN	076052
CMPP	076072
CMPNI	076152
CMPPI	076172

Description: Src1 is arithmetically compared with src2. The condition codes reflect the comparison. The signed branch instruction can be used to test the result.

Register Form—CMPN and CMPP
When the instruction starts, the operands must have been placed in the general registers. The first source descriptor is placed in R0-R1, and the second source descriptor is placed in R2-R3.



When the instruction is completed, the source descriptor registers are cleared.



In-line Form—CMPNI and CMPPI
Each word address pointer which follows the opcode word in the instruction stream refers to a two-word decimal string descriptor. R0-R6 are unchanged when the instruction is completed.

Notes:

1. The operation of these instructions is unaffected by any overlap of the source strings provided that each source string is a valid representation of the specified data type.

CVTLN/CVTLP/CVTLNI/CVTLPI

Purpose: Convert Long to Decimal

Operation: decimal string ← long integer

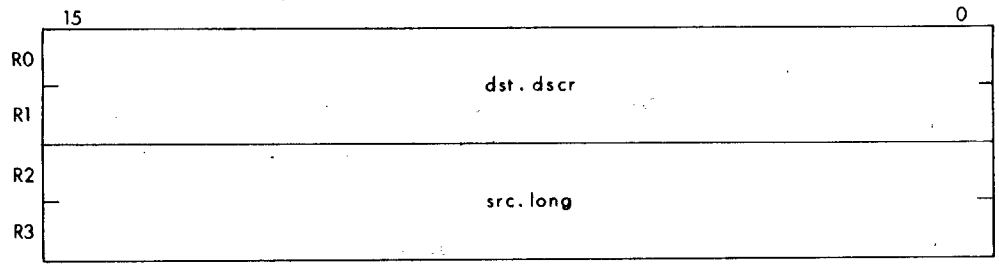
Condition Codes:
- N: set if dst < 0; cleared otherwise
- Z: set if dst = 0; cleared otherwise
- V: set if dst cannot contain all significant digits of the result; cleared otherwise
- C: cleared

Opcodes:

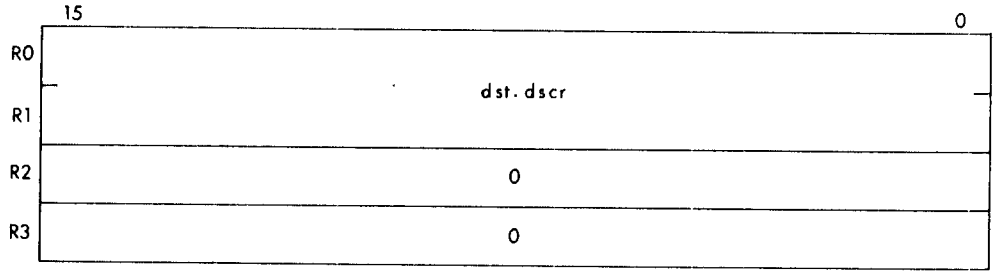
CVTLN	076057
CVTLP	076077
CVTLNI	076157
CVTLPI	076177

Description: The source long integer is converted to a decimal string. The condition codes reflect the result stored in the destination decimal string, and whether all significant digits were stored.

Register Form—CVTLN and CVTLP
When the instruction starts, the operands must have been placed in the general registers. The destination descriptor is placed in R0-R1, and the source long integer is placed in R2-R3.



When the instruction is completed, the source long integer registers are cleared.



In-line Form—CVTLNI and CVTLPI

The words which follow the opcode word in the instruction stream are a word address pointer to a two-word decimal string descriptor, and a word address pointer to a two-word long integer source. R0-R6 are unchanged when the instruction is completed.

Notes:

1. Register forms use a long integer oriented with the sign and high order portion in R2, and the low order portion in R3.
2. In-line forms use a long integer oriented with the low order portion in src.long, and the sign and high order portion in src.long + 2.

CVTNL/CVTPL/CVTNLI/CVTPLI

Purpose: Convert Decimal to Long

Operation: long integer ← decimal string

Condition Codes: The condition codes are based on the long integer destination and on the sign of the source decimal string.

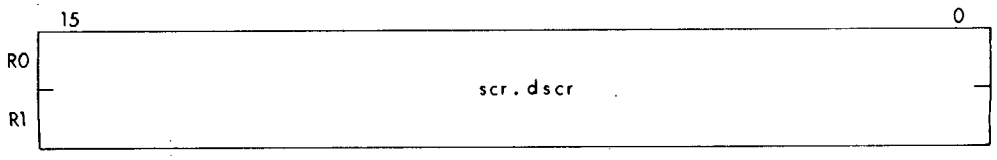
- N: set if long.integer < 0; cleared otherwise
- Z: set if long.integer = 0; cleared otherwise
- V: set if long.integer dst cannot correctly represent the 2's complement form of the result; cleared otherwise
- C: set if src < 0 and long.integer#0; cleared otherwise

Opcodes:

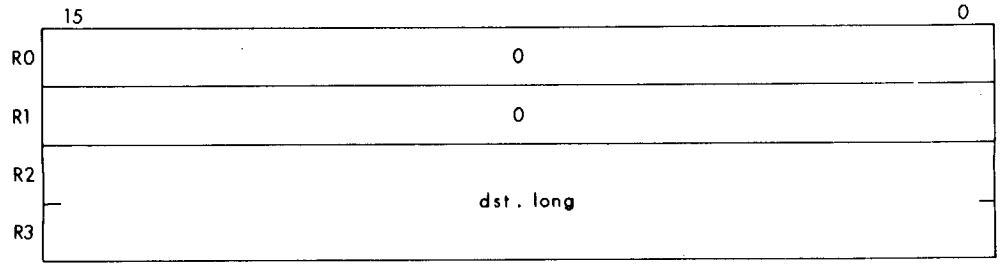
CVTNL	076053
CVTPL	076073
CVTNLI	076153
CVTPLI	076173

Description: The source decimal string is converted to a long integer. The condition codes reflect the result of the operation, and whether significant digits were not converted.

Register Form—CVTNL and CVTPL
When the instruction starts, the operands must have been placed in the general registers. The source decimal string descriptor is placed in R0-R1.



When the instruction is completed, the source decimal string descriptors are cleared, and the destination long integer is returned in R2-R3.



In-line Form—CVTNLI and CVTPLI
The words which follow the opcode word in the instruction stream are a word address pointer to a two-word decimal string source descriptor, and a word address pointer to a two-word long integer destination. R0-R6 are unchanged when the instruction is completed.

Notes:

1. Register forms use a long integer oriented with the sign and high order portion in R2, and the low order portion in R3.
2. In-line forms use a long integer oriented with the low order portion in dst.long, and the sign and high order portion in dst.long + 2.
3. If the V bit is set, the contents of the long integer destination are the least significant 32 bits of the result.
4. A source whose value is +2**31 can be represented as a 32-bit binary integer. However, since the destination is a 2's complement long integer, the resulting condition codes will be: N set, Z cleared, V set, and C cleared.

CVTNP/CVTPN/CVTNPI/CVTPNI

Purpose: Convert Decimal

Operation:

CVTNP/CVTNPI	packed string ← numeric string
CVTPN/CVTPNI	numeric string ← packed string

Condition Codes:

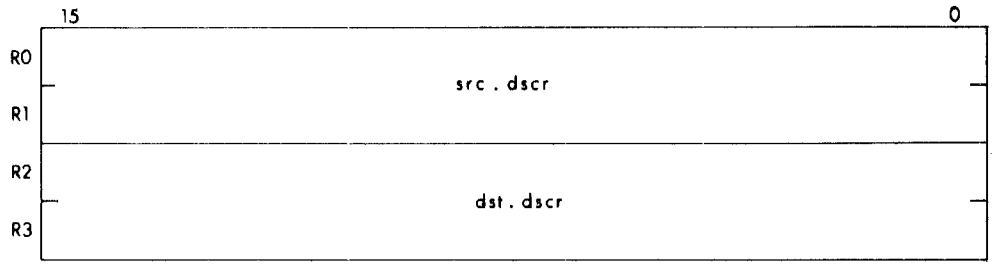
N: set if dst < 0; cleared otherwise
Z: set if dst = 0; cleared otherwise
V: set if dst cannot contain all significant digits of the result; cleared otherwise
C: cleared

Opcodes:

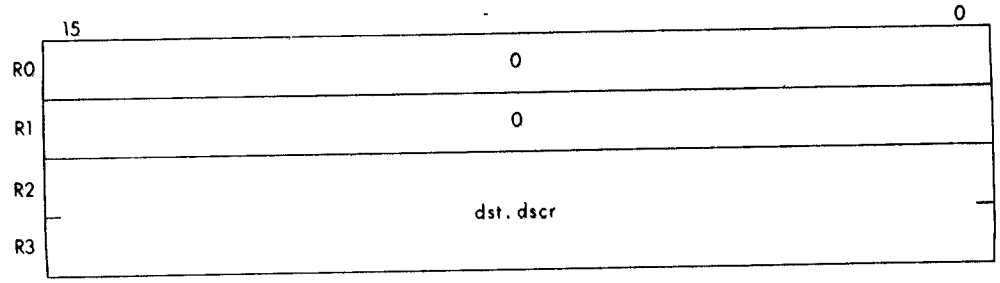
CVTNP	076055
CVTPN	076054
CVTNPI	076155
CVTPNI	076154

Description: These instructions convert between numeric and packed decimal strings. The source decimal string is converted and moved to the destination string. The condition codes reflect the result of the operation, and whether all significant digits were stored.

Register Form—CVTNP and CVTPN
When the instruction starts, the operands must have been placed in the general registers. The source descriptor is placed in R0-R1 and the destination descriptor is placed in R2-R3.



When the instruction is completed, the source descriptor registers are cleared.



In-line Form—CVTNPI and CVTPNI
Each word address pointer which follows the opcode word in the instruction stream refers to a two-word decimal string descriptor. R0-R6 are unchanged when the instruction is completed.

Notes:

1. The results of the instruction are unpredictable if the source and destination strings overlap.
2. These instructions use both a numeric and a packed decimal string descriptor.

DIVP/DIVPI

Purpose: Divide Decimal

Operation: dst ← src2/src1

Condition Codes:
- N: set if dst < 0; cleared otherwise
- Z: set if dst = 0; cleared otherwise
- V: set if dst cannot contain all significant digits of the result or if src1 = 0; cleared otherwise
- C: set if src1 = 0; cleared otherwise

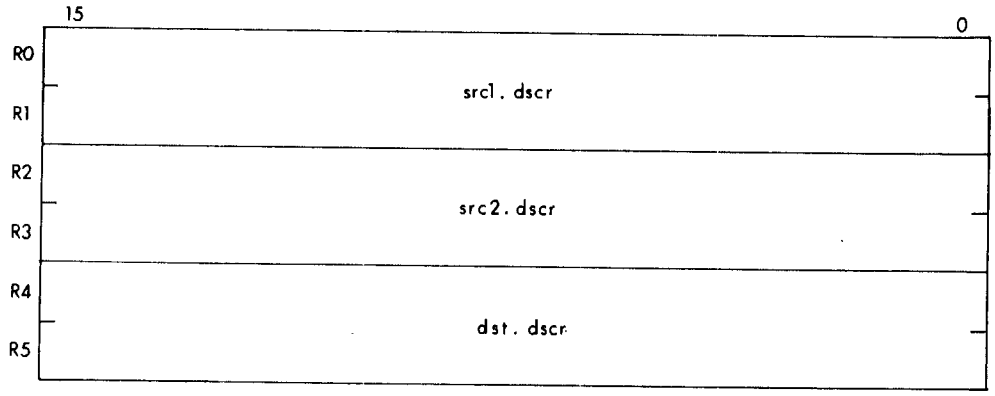
Opcodes:

DIVP	076075
DIVPI	076175

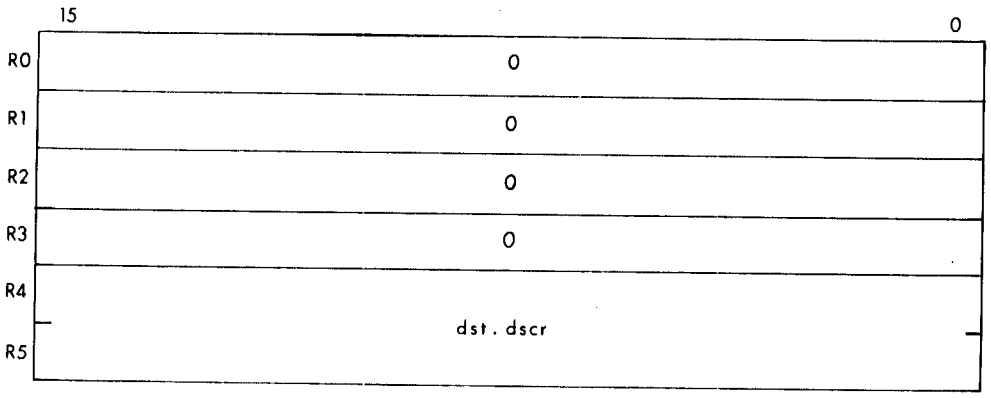
Description: Src2 is divided by src1, and the quotient (fraction truncated) is stored in the destination string. The condition codes reflect the value stored in the destination string, and whether all significant digits were stored.

Register Form—DIVP

When the instruction starts, the operands must have been placed in the general registers. The first source descriptor is placed in R0-R1, the second source descriptor is placed in R2-R3, and the destination descriptor is placed in R4-R5.



When the instruction is completed, the source descriptor registers are cleared.



In-line Form—DIVPI

Each word address pointer which follows the opcode word in the instruction stream refers to a two-word decimal string descriptor. R0-R6 are unchanged when the instruction is completed.

Notes:

1. The operation of these instructions is unaffected by any overlap of the source strings provided that each source string is a valid representation of the specified data type.
2. The results of the instruction are unpredictable if the source and destination strings overlap.

3. Division by zero will set the V and C bits. The destination string, and the N and Z condition code bits will be unpredictable.
4. No numeric string divide instruction is provided.

LOCC/LOCCI

Purpose: Locate Character

Operation: Search source character string for a character.

Condition Codes: The condition codes are based on the final contents of R0.

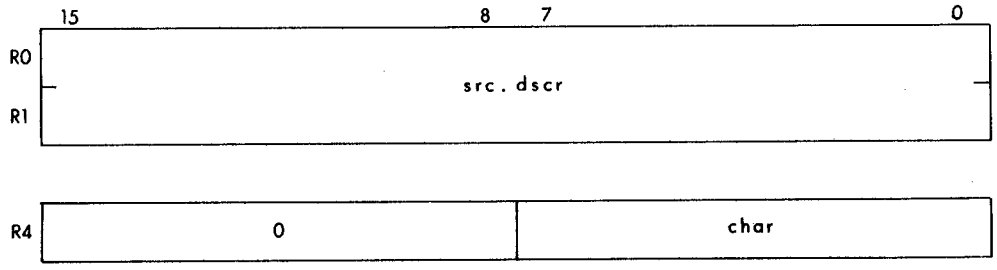
N: set if R0<15> set; cleared otherwise
Z: set if R0 = 0; cleared otherwise
V: cleared
C: cleared

Opcodes:

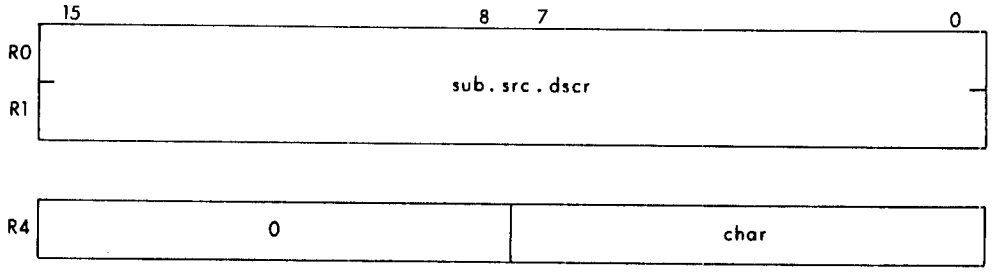
LOCC	076040
LOCCI	076140

Description: The source character string is searched from most significant to least significant character until the first occurrence of the search character. A character string descriptor is returned in R0-R1 which represents the portion of the source character string beginning with the located character. If the source character string contains only characters not equal to the search character, the instructions return a vacant character string descriptor with an address one greater than that of the least significant character of the source character string. The condition codes reflect the resulting value in R0.

Register Form—LOCC
When the instruction starts, the operands must have been placed in the general registers. The source character string descriptor is placed in R0-R1, the search character is placed in R4<7:0>, and R4<15:8> must be zero.

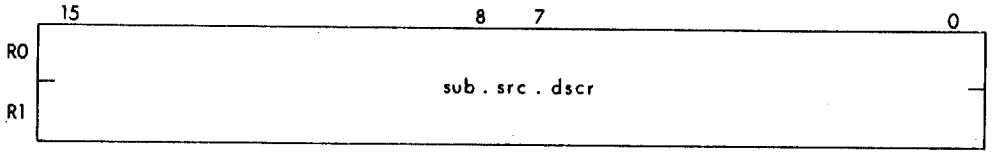


When the instruction is completed, R0-R1 contain a character set descriptor which represents the substring of the source character string beginning with the located character.



In-line Form—LOCCI

The words which follow the opcode word in the instruction stream are a word address pointer to a two-word character string source descriptor, and a word whose low order half contains the search character and whose high order half must be zero. When the instruction is completed, R0-R1 contain a character string descriptor which represents the substring of the source character string beginning with the located character. R2-R6 are unchanged.



Notes:

1. If the initial source character string descriptor is vacant, the instruction terminates with the condition codes indicating no match was found. The original source character string descriptor is returned in R0-R1.
2. A test for success is BNE; a test for failure is BEQ.
3. The condition codes will be set as if this instruction were followed by TST R0.

L2DR

Purpose: Load 2 Descriptors

Operation: Load word pairs into R0-R1 and R2-R3.

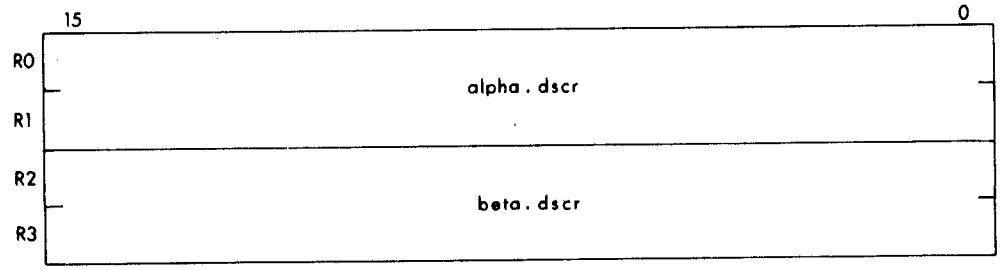
Condition Codes:
N: not affected
Z: not affected
V: not affected
C: not affected

Opcodes: L2DR 07602r

Description: This instruction augments the character and decimal string instructions by efficiently loading string descriptors into the general registers.

A descriptor "alpha" is loaded into R0-R1; a second descriptor "beta" is loaded into R2-R3. The address of the descriptors is determined by the addressing mode @(Rr)+ where r is the low order three bits of the opcode word. The address of the descriptor "alpha" is derived by applying this addressing mode once; the address of the descriptor "beta" is derived by applying this addressing mode a second time. The addressing mode auto-increments the indicated register by two. The addressing mode computation is not affected by the descriptors which are loaded into the general registers. The words which contain the addresses of the descriptors are in consecutive words in memory; the descriptions themselves may be anywhere in memory. The condition codes are not affected.

When the instruction is completed, the "alpha" descriptor is in R0-R1 and the "beta" descriptor is in R2-R3.



L3DR

Purpose: Load 3 Descriptors

Operation: Load word pairs into R0-R1, R2-R3, and R4-R5.

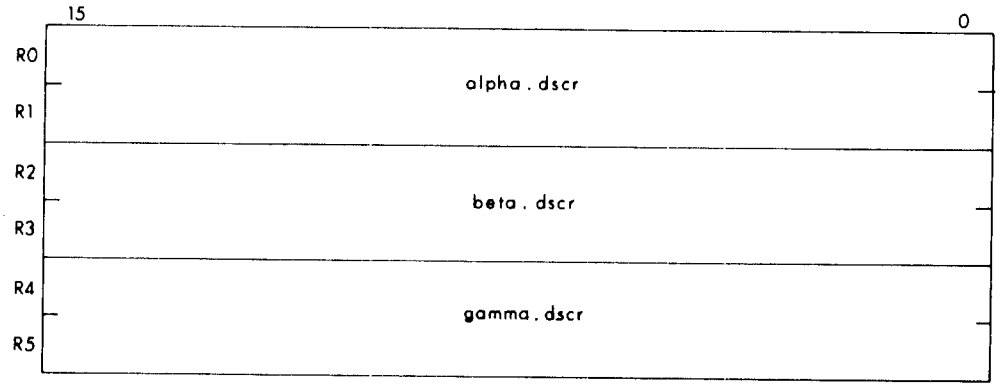
Condition Codes:
N: not affected
Z: not affected
V: not affected
C: not affected

Opcodes: L3DR 07606r

Description: This instruction augments the character and decimal string instructions by efficiently loading string descriptors into the general registers.

A descriptor "alpha" is loaded into R0-R1; a second descriptor "beta" is loaded into R2-R3; a third descriptor "gamma" is loaded into R4-R5. The address of the descriptors is determined by the addressing mode @(Rr)+ where r is the low order three bits of the opcode word. The address of the descriptor "alpha" is derived by applying this addressing mode once. The address of the descriptor "beta" is derived by applying this addressing mode a second time. The address of the descriptor "gamma" is derived by applying this addressing mode a third time. The addressing mode autoincrements the indicated register by two. The addressing mode computation is not affected by the descriptors which are loaded into the general registers. The words which contain the addresses of the descriptors are in consecutive words in memory; the descriptors themselves may be anywhere in memory. The condition codes are not affected.

When the instruction is completed, the "alpha" descriptor is in R0-R1, the "beta" descriptor is in R2-R3 and the "gamma" descriptor is in R4-R5.



MATC/MATCI

Purpose: Match Character

Operation: Search source character string for object character string.

Condition Codes: The condition codes are based on the final contents of R0.

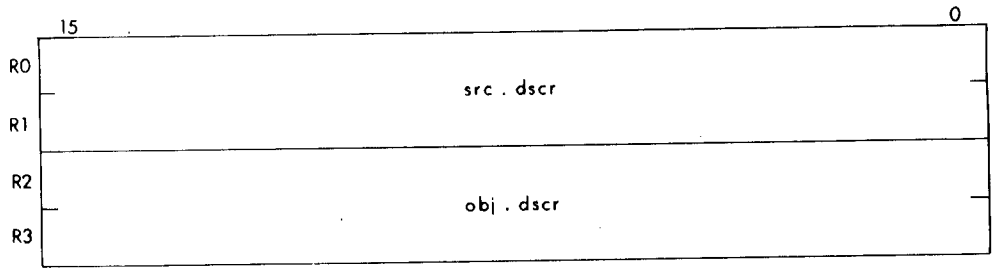
N: set if R0<15> set; cleared otherwise
Z: set if R0 = 0; cleared otherwise
V: cleared
C: cleared

Opcodes:

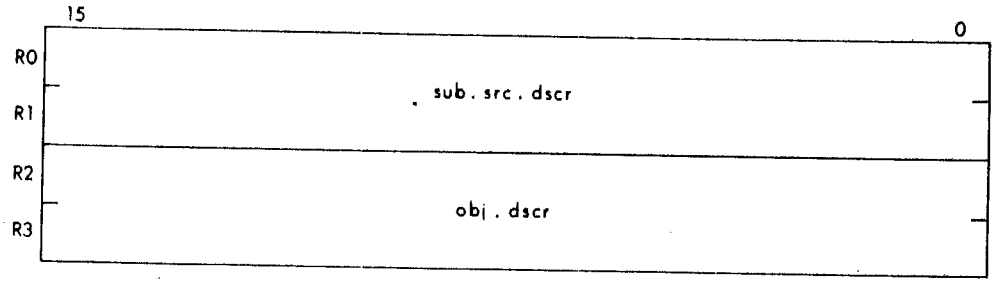
MATC	076045
MATCI	076145

Description: The source character string is searched from most significant to least significant character for the first occurrence of the entire object character string. A character string descriptor is returned in R0-R1 which represents the portion of the original source character string from the most significant character which completely matches the object character string to the end of the source character string. If the object character string did not completely match any portion of the source character string, the character descriptor returned in R0-R1 is vacant with an address one greater than the least significant character in the source string. The condition codes reflect the resulting value in R0. If the Z bit is cleared, the entire object was successfully matched with the source character string; if the Z bit is set, the match failed.

Register Form—MATC
When the instruction starts, the operands must have been placed in the general registers. The source character string descriptor is placed in R0-R1, and the object character string descriptor is placed in R2-R3.

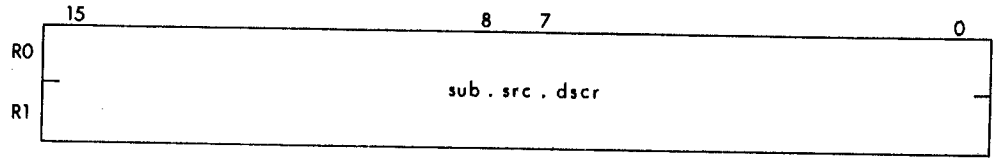


The instruction terminates with a character sub-string descriptor returned in R0-R1 which represents the portion of the original source character string beginning with the most significant character to competely match the object character string.



In-line Form—MATCI

The words which follow the opcode word in the instruction stream are a word address pointer to a two-word character string source descriptor, and a word address pointer to a two-word character string object descriptor. The instruction terminates with a character sub-string descriptor returned in R0-R1 which represents the portion of the original source character string beginning with the most significant character to completely match the object character string. R2-R6 are unchanged when the instruction is completed.



Notes:

1. The operation of this instruction is unaffected by any overlap of the source and object character strings.
2. A vacant object character string matches any non-vacant source character string. A vacant source character string will not match any object character string. If the initial source character string descriptor is vacant, the instruction terminates with the condition codes indicating no match was found. The original source character string descriptor is returned in R0-R1.
3. If the length of the object character string is greater than that of the source character string, no match is found; R0-R1 and the condition codes will be updated.
4. A test for success is BNE; a test for failure is BEQ.
5. The condition codes will be set as if this instruction were followed by TST R0.

MOVC/MOVCI

Purpose: Move Character

Operation: dst ← src

Condition Codes: The condition codes are based on the arithmetic comparison of the initial character string lengths (result=src.len-dst.len).

- N: set if result < 0; cleared otherwise
- Z: set if result = 0; cleared otherwise
- V: set if there was arithmetic overflow, that is, src.len<15> and dst.len<15> were different, and dst.len<15> was the same as bit <15> of (src.len−dst.len); cleared otherwise
- C: cleared if there was a carry from the most significant bit of the result; set otherwise

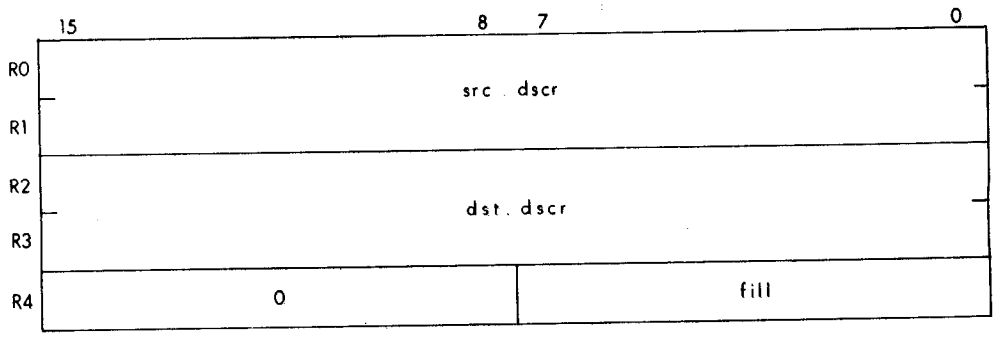
Opcodes:

MOVC	076030
MOVCI	076130

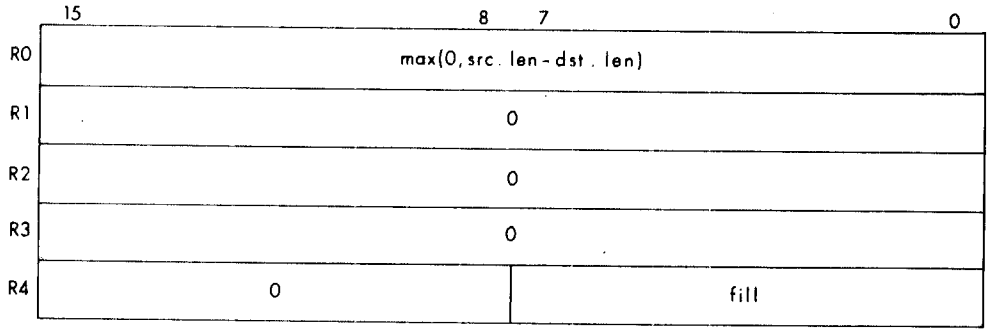
Description: The character string specified by the source descriptor is moved into the area specified by the destination descriptor. It is aligned by the most significant character. The condition codes reflect an arithmetic comparison of the original source and destination lengths. If the source string is shorter than the destination string, the fill character is used to complete the least significant part of the destination string. This is indicated by the C bit set. If the source string is longer than the destination string, the least significant characters of the source string are not moved. This is indicated by the Z and C bits cleared. If the source and destination strings are of equal length, all characters are moved with neither truncation nor filling. This is indicated by the Z bit set. The unsigned branch instructions may test the result of the instruction.

Register Form—MOVC

When the instruction starts, the operands must have been placed in the general registers. The source character string descriptor is placed in R0-R1, the destination character string descriptor is placed in R2-R3, the fill character is placed in R4<7:0>, and R4<15:8> must be zero.



When the instruction is completed, R0 contains the number of unmoved source string characters, and R1 through R3 are cleared.



In-line Form—MOVCI

The words which follow the opcode word in the instruction stream are a word address pointer to a two-word character string source descriptor, a word address pointer to a two-word character string destination descriptor, and a word whose low-order half contains the fill character and whose high-order half must be zero. R0-R6 are unchanged when the instruction is completed.

Notes:

1. The operation of this instruction is unaffected by any overlap of the source and destination strings. The result is equivalent to having read the entire source string before storing characters in the destination.
2. If the source string is vacant, the fill character will be propagated through the destination string. If the destination string is vacant, no characters will be moved. The condition codes will be updated. MOVC will update the general registers.
3. MOVC — When the instruction terminates, R0 is zero only if Z or C are set.
4. The condition codes will be set as if this instruction were preceded by CMP src.len, dst.len.

MOVRC/MOVRCI

Purpose: Move Reverse Justified Character

Operation: dst ← reverse justified src

Condition Codes: The condition codes are based on the arithmetic comparison of the initial character string lengths (result = src.len−dst.len).

- N: set if result < 0; cleared otherwise
- Z: set if result = 0; cleared otherwise
- V: set if there was arithmetic overflow, that is, src.len<15> and dst.len<15> were different, and dst.len<15> was the same as bit <15> of (src.len−dst.len); cleared otherwise

C: cleared if there was a carry from the most significant bit of the result; set otherwise

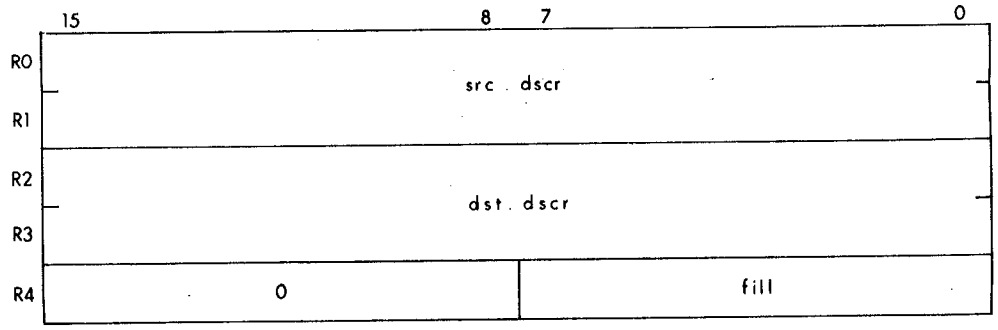
Opcodes:

MOVRC	076031
MOVRCI	076131

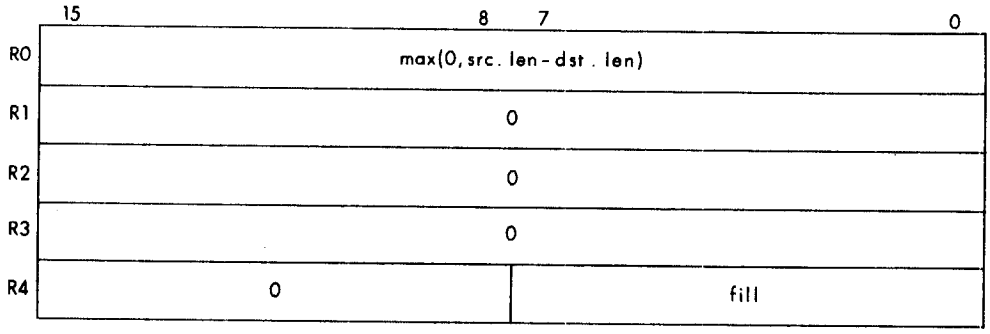
Description: The character string specified by the source descriptor is moved into the area specified by the destination descriptor. It is aligned by the least significant character. The condition codes reflect an arithmetic comparison of the original source and destination lengths. If the source string is shorter than the destination string, the fill character is used to complete the most significant part of the destination string. This is indicated by the C bit set. If the source string is longer than the destination string, the most significant characters of the source string are not moved. This is indicated by the Z and C bits cleared. If the source and destination strings are of equal length, all characters are moved with neither truncation nor filling. This is indicated by the Z bit set. The unsigned branch instructions may test the result of the instruction.

Register Form—MOVRC

When the instruction starts, the operand must have been placed in the general registers. The source character string descriptor is placed in R0-R1, the destination character string descriptor is placed in R2-R3, the fill character is placed in R4<7:0>, and R4<15:8> must be zero.



When the instruction is completed, R0 contains the number of unmoved source string characters, and R1 through R3 are cleared.



In-line Form—MOVRCI

The words which follow the opcode word in the instruction stream are a word address pointer to a two-word character string source descriptor, a word address pointer to a two-word character string destination descriptor, and a word whose low order half contains the fill character and whose high order half must be zero. R0-R6 are unchanged when the instruction is completed.

Notes:

1. The operation of this instruction is unaffected by any overlap of the source and destination strings. The result is equivalent to having read the entire source string before storing characters in the destination.
2. If the source string is vacant, the fill character will be propagated through the destination string. If the destination string is vacant, no characters will be moved. Condition codes will be updated. MOVRC will update the general registers.
3. MOVRC — When the instruction terminates, R0 is zero only if Z or C are set.
4. The condition codes will be set as if this instruction were preceded by CMP src.len, dst.len.

MOVTC/MOVTCI

Purpose: Move Translated Character

Operation: dst ← translated src

Condition Codes: The condition codes are based on the arithmetic comparison of the initial character string lengths (result = src.len−dst.len).

N: set if result < 0; cleared otherwise

Z: set if result = 0; cleared otherwise

V: set if there was arithmetic overflow, that is, src.len<15> and dst.len<15> were different, and dst.len<15> was the same as bit <15> of (src.len−dst.len); cleared otherwise

C: cleared if there was a carry from the most significant bit of the result; set otherwise

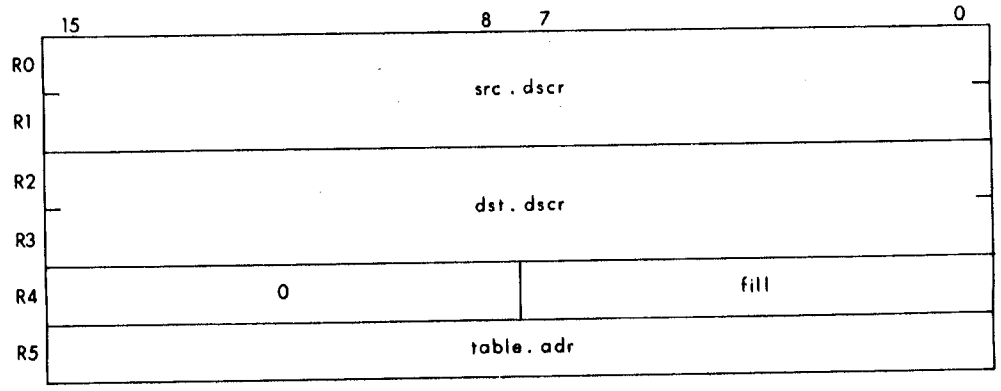
Opcodes:

MOVTC	076032
MOVTCI	076132

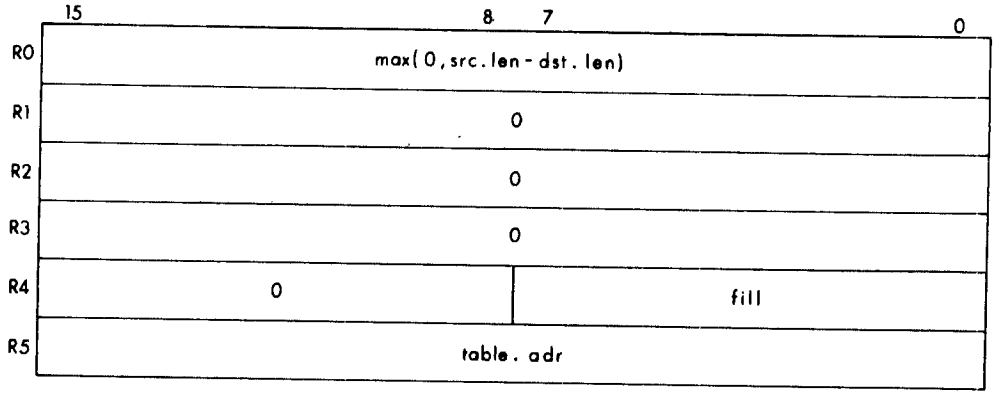
Description: The character string specified by the source descriptor is translated and moved into the area specified by the destination descriptor. It is aligned by the most significant character. Translation is accomplished by using each source character as an 8-bit positive integer index into a 256-byte table, the address of which is an operand of the instruction. The byte at the indexed location in the table is stored in the destination string. The condition codes reflect an arithmetic comparison of the original source and destination lengths.

If the source string is shorter than the destination string, the untranslated fill character is used to complete the least significant part of the destination string. This is indicated by the C bit set. If the source string is longer than the destination string, the least significant characters of the source string are not moved. This is indicated by the Z and C bits cleared. If the source and destination strings are of equal length, all characters are translated and moved with neither truncation nor filling. This is indicated by the Z bit set. The unsigned branch instructions may test the result of the instruction.

Register Form—MOVTC
When the instruction starts, the operands must have been placed in the general registers. The source character string descriptor is placed in R0-R1, the destination character string descriptor is placed in R2-R3, the fill character is placed in R4<7:0>, R4<15:8> must be zero, and the translation table address is placed in R5.



When the instruction is completed, R0 contains the number of unmoved source string characters, and R1 through R3 are cleared.



In-line Form—MOVTCI

The words which follow the opcode word in the instruction stream are a word address pointer to a two-word character string source descriptor, a word address pointer to a two-word character string destination descriptor, a word whose low-order half contains the fill character and whose high-order half must be zero, and a word containing the address of the translation table. R0-R6 are unchanged when the instruction is completed.

Notes:

1. The operation of this instruction is unaffected by any overlap of the source and destination strings. The result is equivalent to having read the entire source string before storing characters in the destination.
2. If the destination string overlaps the translation table in any way, the results of the instruction will be unpredictable.
3. If the source string is vacant, the untranslated fill character will be propagated through the destination string. If the destination string is vacant, no characters will be moved. Condition codes will be updated. MOVTC will update the general registers.
4. MOVTC — When the instruction terminates, R0 is zero only if Z or C are set.
5. The condition codes will be set as if this instruction were preceded by CMP src.len, dst.len.
6. The effect of the instruction is unpredictable if the entire 256-byte translation table is not in readable memory.

MULP/MULPI

Purpose: Multiply Decimal

Operation: dst ← src2 * src1

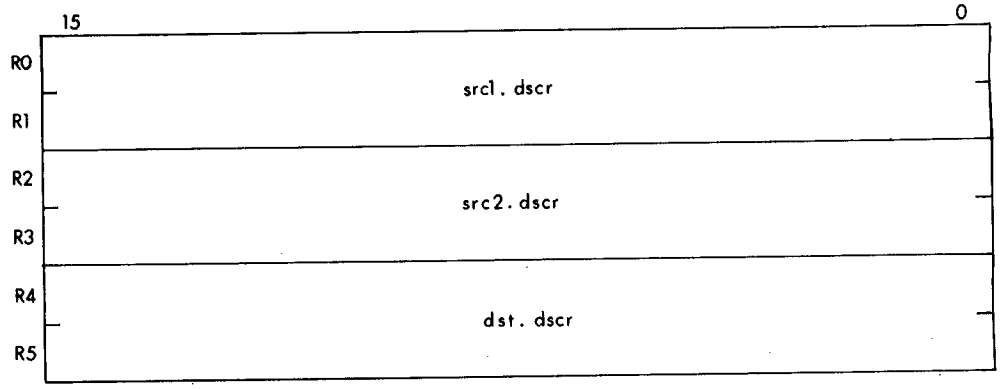
Condition Codes:
- N: set if dst < 0; cleared otherwise
- Z: set if dst = 0; cleared otherwise
- V: set if dst cannot contain all significant digits of the result; cleared otherwise
- C: cleared

Opcodes:

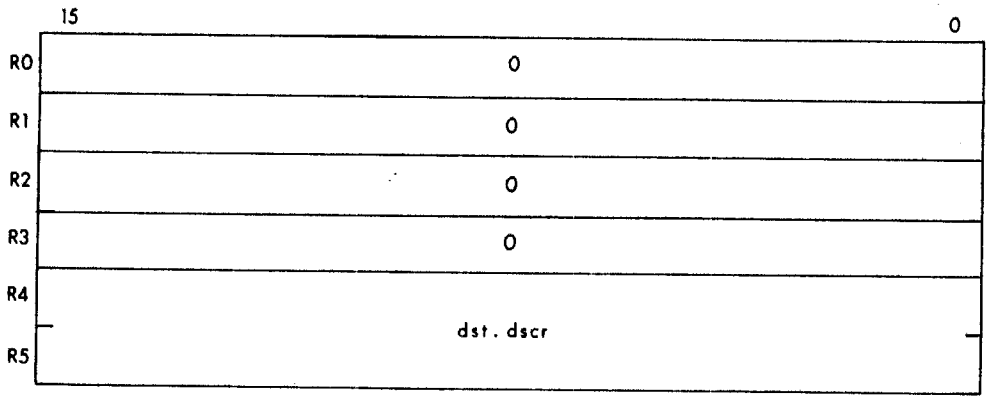
MULP	076074
MULPI	076174

Description: Src1 and src2 are multiplied, and the result is stored in the destination string. The condition codes reflect the value stored in the destination string, and whether all significant digits were stored.

Register Form—MULP
When the instruction starts, the operands must have been placed in the general registers. The first source descriptor is placed in R0-R1, the second source descriptor is placed in R2-R3, and the destination descriptor is placed in R4-R5.



When the instruction is completed, the source descriptor registers are cleared.



In-line Form—MULPI

Each word address pointer which follows the opcode word in the instruction stream refers to a two-word decimal string descriptor. R0-R6 are unchanged when the instruction is completed.

Notes:

1. The operation of these instructions is unaffected by any overlap of the source strings provided that each source string is a valid representation of the specified data type.
2. The results of the instruction are unpredictable if the source and destination strings overlap.
3. No numeric string multiply instruction is provided.

SCANC/SCANCI

Purpose: Scan Character

Operation: Search source character string for a member of the character set.

Condition Codes: The condition codes are based on the final contents of R0.

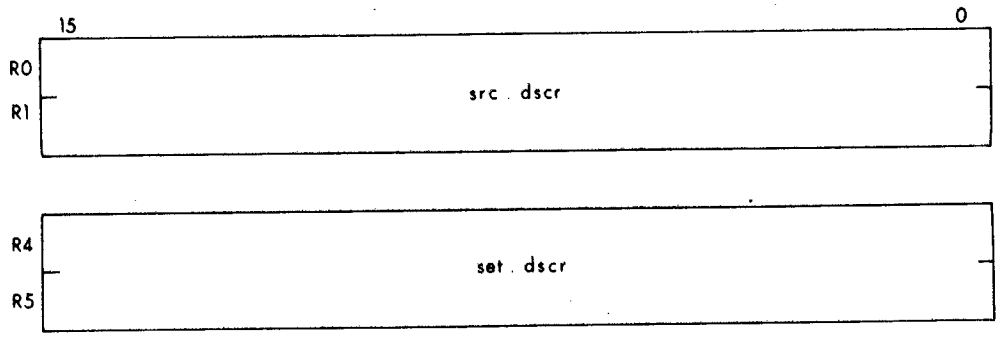
- N: set if R0<15> set; cleared otherwise
- Z: set if R0 = 0; cleared otherwise
- V: cleared
- C: cleared

Opcodes:

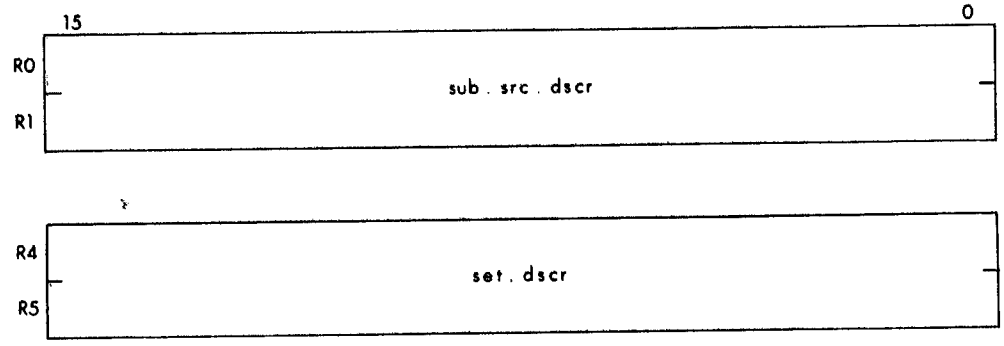
SCANC	076042
SCANCI	076142

Description: The source character string is searched from most significant to least significant character until the first occurrence of a character which is a member of the character set. A character string descriptor is returned in R0-R1 which represents the portion of the source character string beginning with the located member of the character set. If the source character string contains only characters which are not in the character set, the instructions return a vacant character string descriptor with an address one greater than that of the least significant character of the source character string. The condition codes reflect the resulting value in R0.

Register Form—SCANC
When the instruction starts, the operands must have been placed in the general registers. The source character string descriptor is placed in R0-R1, and the character set descriptor is placed in R4-R5.

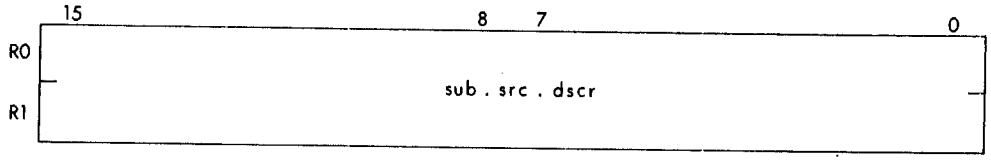


When the instruction is completed, R0-R1 contain a character string descriptor which represents the substring of the source character string beginning with the most significant character which is a member of the character set.



In-line Form—SCANCI
The words which follow the opcode word in the instruction stream are a word address pointer to a two-word character string source descriptor, and a word

address pointer to a two-word character set descriptor. When the instruction is completed, R0-R1 contain a character string descriptor which represents the sub-string of the source character string beginning with the most significant character which is a member of the character set. R2-R6 are unchanged.



Notes:

1. If the initial source character string descriptor is vacant, the instruction terminates with the condition codes indicating that no characters in the set were found. The original source character string descriptor is returned in R0-R1.
2. The source character string and character set table may overlap in any way.
3. A test for success is BNE; a test for failure is BEQ.
4. The condition codes will be set as if this instruction were followed by TST R0.
5. The effect of the instruction is unpredictable if the entire 256-byte character set table is not in readable memory.

SKPC/SKPCI

Purpose: Skip Character

Operation: Search source character string until a character other than the search character is found.

Condition Codes: The condition codes are based on the final contents of R0.

- N: set if R0<15> set; cleared otherwise
- Z: set if R0 = 0; cleared otherwise
- V: cleared
- C: cleared

Opcodes:

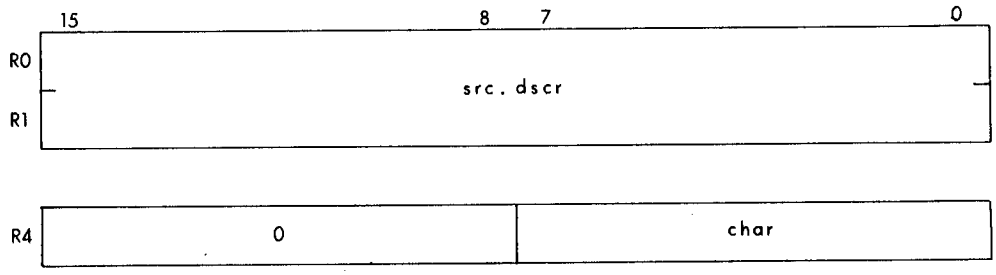
SKPC	076041
SKPCI	076141

Description: The source characer string is searched from most significant to least significant character until the first occurrence of a character which is not the search character. A character string descriptor is returned in R0-R1

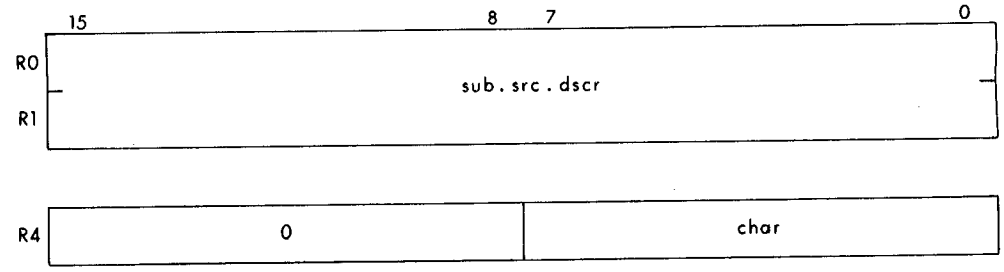
which represents the portion of the source character string beginning which the most significant character which was not equal to the search character. If the source character string contains only characters equal to the search character, the instruction returns a vacant character string descriptor with an address one greater than that of the least significant character of the source character string. The condition codes reflect the resulting value in R0.

Register Form—SKPC

When the instruction starts, the operands must have been placed in the general registers. The source character string decriptor is placed in R0-R1, the search character is placed in R4<7:0>, and R4<15:8> must be zero.

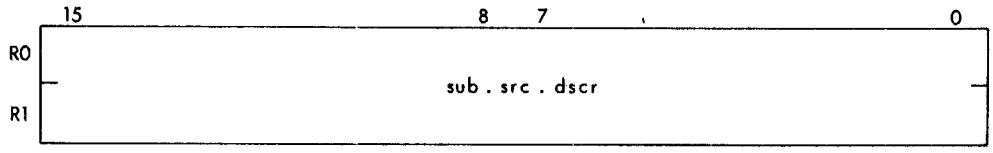


When the instruction is completed, R0-R1 contain a character string descriptor which represents the substring of the source character string beginning with the most significant character which was not equal to the search character.



In-line Form—SKPCI

The words which follow the opcode word in the instruction stream are a word address pointer to a two-word character string source descriptor, and a word whose low-order half contains the search character and whose high-order half must be zero. When the instruction is completed, R0-R1 contain a character string descriptor which represents the substring of the source character string beginning with the most significant character which was not equal to the search character. R2-R6 are unchanged.



Notes:

1. If the initial source character string descriptor is vacant, the instruction terminates with the condition codes indicating the character string only contained search characters. The original source character string descriptor is returned in R0-R1.
2. The condition codes will be set as if this instruction were followed by TST R0.

SPANC/SPANCI

Purpose: Span Character

Operation: Search source character string for a character which is not a member of the character set.

Condition Codes: The condition codes are based on the final contents of R0.

N: set if R0<15> set; cleared otherwise

Z: set if R0 = 0; cleared otherwise

V: cleared

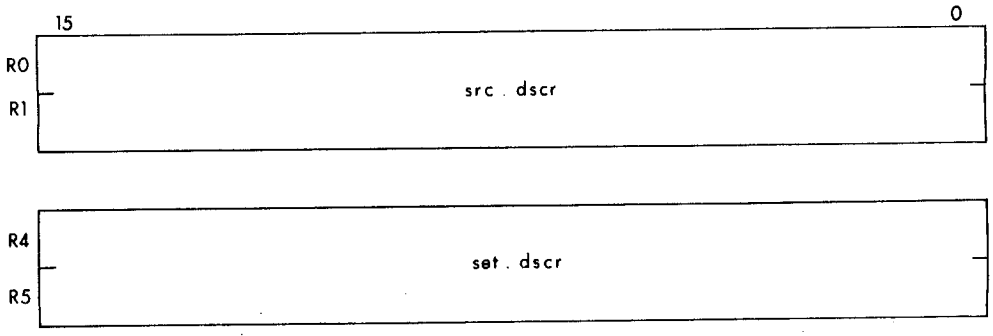
C: cleared

Opcodes:		
	SPANC	076043
	SPANCI	076143

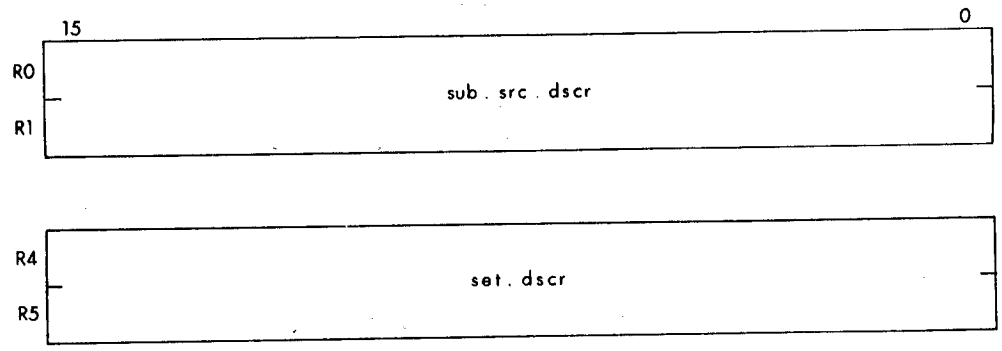
Description: The source character string is searched from most significant to least significant character until the first occurrence of character which is not a member of the character set. A character string descriptor is returned in R0-R1 which represents the portion of the source character string beginning with

the character which is not a member of the character set. If the source character string contains only characters which are in the character set, the instruction returns a vacant character string descriptor with an address one greater than that of the least significant character of the source character string. The condition codes reflect the resulting value in R0.

Register Form—SPANC
When the instruction starts, the operands must have been placed in the general registers. The source character string descriptor is placed in R0-R1, and the character set descriptor is placed in R4-R5.

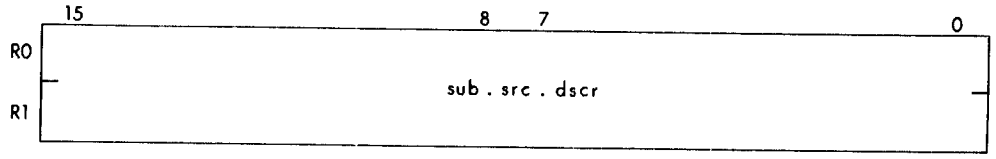


When the instruction is completed, R0-R1 contain a character string descriptor which represents the substring of the source character string beginning with the most significant character which is not a member of the character set.



In-line Form—SPANCI

The words which follow the opcode word in the instruction stream are a word address pointer to a two-word character string source descriptor, and a word address pointer to a two-word character set descriptor. When the instruction is completed, R0-R1 contain a character string descriptor which represents the substring of the source character string beginning with the most significant character which is not a member of the character set. R2-R6 are unchanged.



Notes:

1. If the initial source character string descriptor is vacant, the instruction terminates with the condition codes indicating that only characters in the set were found. The original source character string descriptor is returned in R0-R1.
2. The source character string and character set table may overlap in any way.
3. The condition codes will be set as if this instruction were followed by TST R0.
4. The effect of the instruction is unpredictable if the entire 256-byte character set table is not in readable memory.

SUBN/SUBP/SUBNI/SUBPI

Purpose: Subtract Decimal

Operation: dst ← src2−src1

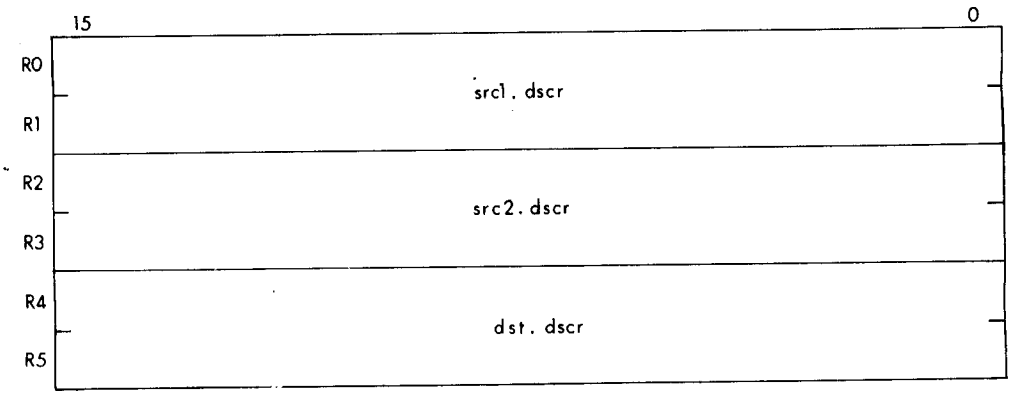
Condition Codes:
- N: set if dst < 0; cleared otherwise
- Z: set if dst = 0; cleared otherwise
- V: set if dst cannot contain all significant digits of the result; cleared otherwise
- C: cleared

Opcodes:

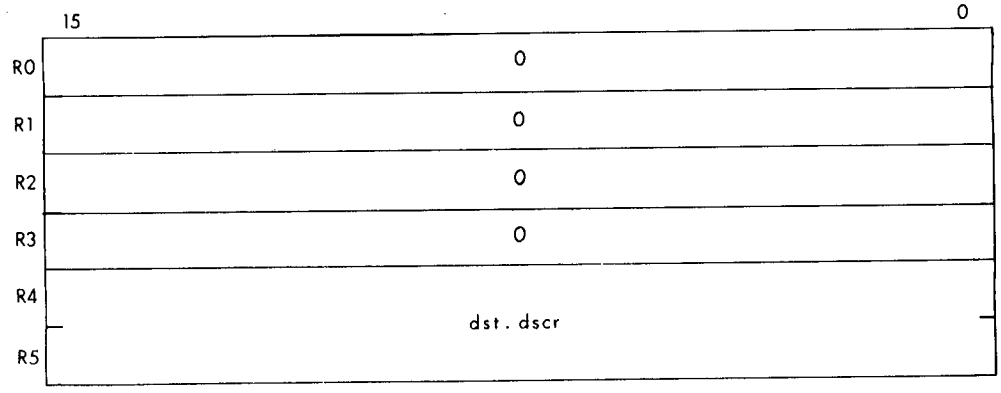
SUBN	076051
SUBP	076071
SUBNI	076151
SUBPI	076171

Description: Src1 is subtracted from src2, and the result is stored in the destination string. The condition codes reflect the value stored in the destination string, and whether all significant digits were stored.

Register Form—SUBN and SUBP
When the instruction starts, the operands must have been placed in the general registers. The first source descriptor is placed in R0-R1, the second source descriptor is placed in R2-R3, and the destination descriptor is placed in R4-R5.



When the instruction is completed, the source descriptor registers are cleared.



In-line Form—SUBNI and SUBPI
Each word address pointer which follows the opcode word in the instruction stream refers to a two-word decimal string descriptor. R0-R6 are unchanged when the instruction is completed.

Notes:

1. The operation of these instructions is unaffected by any overlap of the source strings provided that each source string is a valid representation of the specified data type.
2. Source strings may overlap the destination string only if all corresponding digits of the strings are in coincident bytes in memory.

APPENDIX A

UNIBUS ADDRESSES

I/O PAGE ADDRESSES

Device	Address	Size in Words	Number of Devices
AA11	776750	8	1
AA11	776400	8	4
AD01	776770	4	1
ADF11	770460	8	1
AFC11	772570	4	1
AR11	770400	8	1
BM792-YA	773000	32	1
BM792-YB	773100	32	1
BM792-YC	773200	32	1
BM792-YH	773300	32	1
BM873-YA	773000	128	1
BM873-YB	773000	256	1
BM873-YC	773000	256	1
CD11	777160	4	1
CM11	777160	4	1
CR11	777160	4	1
DC11	774000	4	32
DC14-D	777360	8	1
DL11-A	777560	4	1
DL11-A	776500	4	15
DL11-B	777560	4	1
DL11-B	776500	4	15
DL11-C	775610	4	31
DL11-D	775610	4	31
DL11-E	775610	4	31
DL11-W	777546	1	1
DL11-W	777560	4	1
DL11-W	776500	4	15
DM11	775000	4	16
DM11-BB	770500	4	16
DN11-AA	775200	4	16
DN11-DA	775200	1	64
DP11	77440	4	32
DR11-A(1)	772470	4	1
DR11-A(2)	772460	4	1

Device	Address	Size in Words	Number of Devices
DR11-A(3)	772450	4	1
DR11-A(4)	772440	4	1
DR11-B(1)	772410	4	1
DR11-B(2)	772430	4	1
DR11-B(3)	772450	4	1
DR11-B(4)	772470	4	1
DR11-C(1)	772470	4	1
DR11-C(2)	772460	4	1
DR11-C(3)	772450	4	1
DR11-C(4)	772440	4	1
DS11	775400	67	1
DT11	777420	1	8
DV11	775000	16	4
DX11	776200	16	2
FP11	772160	8	1
GT40	772000	4	4
ICR/ICS11	771000	256	1
KE11	777300	8	2
KG11	770700	4	8
KL11	776500	4	15
KL11	777560	4	1
KT11	772200	64	1
KT11-SR3	772516	1	1
KW11-L	777546	1	1
KW11-P	772540	4	1
KW11-W	772400	4	1
LP11	777510	4	1
LP20	775400	32	2
LPS11	770400	16	1
LS11	777510	4	1
LV11	777510	4	1
M792	773000	32	8
M9301-XX	765000	256	1
M9301-XX	773000	256	1
MM11-LP	772100	1	16
MR11-DB	773100	64	1
MS11-K	772100	1	16
MS11-LP	772100	1	16
NCV11	772760	8	1
OST	772500	6	1
PA611	772600	32	1
PA611	772700	32	1

Device	Address	Size in Words	Number of Devices
PC11	777550	4	1
PDP-11/04	777570	68	1
PDP-11/05	777570	68	1
PDP-11/10	777570	68	1
PDP-11/15	777570	68	1
PDP-11/20	777570	68	1
PDP-11/34A	777570	68	1
PDP-11/35	777570	68	1
PDP-11/40	777570	68	1
PDP-11/44	777570	68	1
PDP-11/45	777570	68	1
PDP-11/55	777570	68	1
PDP-11/60	777570	68	1
PDP-11/70	777570	68	1
PR11	777550	4	1
RC11	777440	8	1
RF11	777460	8	1
RJP04	776700	22	1
RJS04	772040	16	1
RJ611	777440	16	1
RK11	777400	8	1
RL11	774400	4	2
RP11	776700	16	1
RS/RP/TJ	776300	32	1
RX02	777170	4	1
RX11	777170	4	1
TA11	777500	4	1
TC11	777340	8	1
Testers	770000	32	1
TJU16	772440	16	1
TM11	772520	8	1
TS04	772520	8	1
UDC-Units	771000	1	256
UDC11	771774	2	1
Unibus-Map	770200	64	1
VT48	772000	16	1
VTV01	772600	56	1
XY11	777530	4	1
M9312	773024	256	1
M9312	773224	256	1

INTERRUPT AND TRAP VECTORS

000	(reserved)
004	Illegal instructions, Bus Errors, Stack Limit, Illegal Internal Address, Microbreak. Microbreak.
010	Reserved instructions
014	BPT, breakpoint trap (Trace)
020	IOT, input/output trap
024	Power Fail
030	EMT, emulator trap
034	TRAP instruction
040	System software
044	System software
050	System software
054	System software
060	Console Terminal, keyboard/reader
064	Console Terminal, printer/punch
070	PC11, paper tape reader
074	PC11, paper tape punch
100	KW11-L, line clock
104	KW11-P, programmable clock
110	
114	Memory system errors (Cache, UNIBUS Memory, UCS Parity)
120	XY Plotter
124	DR11-B DMA interface; (DA11-B)
130	ADO1, A/D subsystem
134	AFC11, analog subsystem
140	AA11, display
144	AA11, light pen
150	
154	
160	
164	
170	User reserved
174	User reserved
200	LP11/LS11, line printer
204	RS04/RF11, fixed head disk
210	RC11, disk
214	TC11, DECtape
220	RK11, disk
224	TU16/TM11, magnetic tape
230	CD11/CM11/CR11, card reader
234	UDC11, digital control subsystem; ICS/ICR11
240	PIRQ, Program Interrupt Request (11/55,11/45)

244 Floating Point Error
250 Memory Management
254 RP04/RP11 disk pack
260 TA11, cassette
264 RX11, floppy disk

270 User reserved
274 User reserved

300 (start of floating vectors)

FLOATING VECTORS

There is a floating vector convention used for communications (and other) devices that interface with the PDP-11. These vector addresses are assigned in order starting at 300 and proceeding upwards to 777. The following Table shows the assigned sequence. It can be seen that the first vector address, 300, is assigned to the first DC11 in the system. If another DC11 is used, it would then be assigned vector address 310, etc. When the vector addresses have been assigned for all the DC11's (up to a maximum of 32), addresses are then assigned consecutively to each unit of the next highest-ranked device (KL11 or DP11 or DM11, etc.), then to the other devices in accordance with the priority ranking.

Priority Ranking for Floating Vectors

(starting at 300 and proceeding upwards)

Rank	Device	Vector Size (in octal)	Max No.
1	DC11	$(10)_8$	32
2	KL11, DL11-A, DL11-B	10	16
3	DP11	10	32
4	DM11-A	10	16
5	DN11	4	16
6	DM11-BB (DH11-AD or DV11)	4	16
7	DR11-A	10*	32
8	DR11-C	10*	32
9	PA611 Reader	4*	16
10	PA611 Punch	4*	16
11	DT11	10*	8
12	DX11	10*	4
13	DL11-C, DL11-D, DL11-E	10	31
14	DJ11	10	16
15	DH11	10	16
16	GT40	10	1
17	LPS11	30*	1
18	DQ11	10	16
19	KW11-W	10	1
20	DU11	10	16
21	DUP11	10	
22	DV11	10	

*—The first vector for the first device of this type must always be on a $(10)_8$ boundary.

FLOATING ADDRESSES

There is a floating address convention used for communications (and other) devices interfacing with the PDP-11. These addresses are assigned in order starting at 760 010 and proceeding upwards to 763 776. Floating addresses are assigned in the following sequence:

Rank	Device
1	DJ11
2	DH11
3	DQ11
4	DU11

DEVICE ADDRESSES

777 776	Processor Status word (PS)	
777 774	Stack Limit (SL)	
777 772	Program Interrupt Request (PIR)	
777 770	Microprogram Break	
777 766	CPU Error	
777 764	System I/D	
777 762	Upper Size } System Size	
777 760	Lower Size }	
777 756		
777 754		
777 752	Hit/Miss	
777 750	Maintenance	
777 746	Cache Control	
777 744	Memory System Error	
777 742	High Error Address	
777 740	Low Error Address	
777 717	User	R6 (SP)
777 716	Supervisor	R6 (SP)
777 715	General registers, Set 1	R5
777 714		R4
777 713		R3
777 712		R2
777 711		R1
777 710		R0
777 707		R7 (PC)
777 706	Kernel	R6 (SP)
777 705	General registers, Set 0	R5
777 704		R4
777 703		R3
777 702		R2
777 701		R1
777 700		R0

777 676	User Data PAR, reg 0-7	
777 660		
777 656	User Instruction PAR, reg 0-7	
777 640		
777 636	User Data PDR, reg 0-7	
777 620		
777 616	User Instruction PDR, reg 0-7	
777 600		
777 576		(MMR2)
777 574	Memory Mgt regs,	(MMR1)
777 572		(MMR0)
777 570	Console Switch & Display Register	
777 566		printer/punch data
777 564	Console Terminal,	printer/punch status
777 562		keyboard/reader data
777 560		keyboard/reader status
777 556		punch data (PPB)
777 554	PC11/PR11,	punch status (PPS)
777 552		reader data (PRB)
777 550		reader status (PRS)
777 546	KW11-L, clock status (LKS)	
777 544	KU116-AA, UCS	Data
777 542		Address
777 540		Status
777 516		printer data
777 514	LP11/LS11/LV11,	printer status
777 512		
777 510		
777 506		
777 504		
777 502	TA11,	cassette data (TADB)
777 500		cassette status (TACS)
777 476	RK06,	Maintenance Register 3, (RKMR3)
777 474		Maintenance Register 2, (RKMR2)
777 472		ECC Pattern Register (RKECPT)
*777 470		ECC Position Register (RKECPS)
*777 466		Maintenance Register 1 (RKMR1)
*777 464		Data Buffer (RKDB)
*777 462		Unused
*777 460		Desired Cylinder (RKDC)
*777 456		Attention Summary/Offset (RKAS/OF)
*777 454		Error (RKER)
*777 452		Drive Status (RKDS)

*Also used by RF 11

**777 450		Control and Status 2 (RKCS2)
**777 446		Disk Address (RKDA)
**777 444		Bus Address (RKBA)
*777 442		Word Count (RKWC)
**777 440		Control and Status 1 (RKCS1)
777 436		#8
777 434		#7
777 432		#6
777 430	DT11, bus switch	#5
777 426		#4
777 424		#3
777 422		#2
777 420		#1
777 416		disk data (RKDB)
777 414		maintenance
777 412		disk address (RKDA)
777 410	RK11,	bus address (RKBA)
777 406		word count (RKWC)
777 404		disk status (RKCS)
777 402		errorr (RKER)
777 400		drive status (RKDS)
777 376	DC14-D	
777 360		
777 356		
777 354		
777 352		
777 350		DECtape data (TCDT)
777 346	TC11,	bus address (TCBA)
777 344		word count (TCWC)
777 342		command (TCCM)
777 340		DECtape status (TCST)
777 336	KE11-A, EAE #2	
777 320		
777 316		arithmetic shift
777 314		logical shift
777 312		normalize
777 310	KE11-A, EAE #1,	step count/status register
777 306		multiply
777 304		multiplier quotient
777 302		accumulator
777 300		divide
777 166		data (CDDB)
777 164	CR11/ data (CRB2) comp	cur adrs (CDBA)
777 162	CM11, data (CRB1)	CD11, col count (CDCC)
777 160	status (CRS)	status (CDST)
776 776		
776 774		
776 772	AD01,	A/D data (ADDB)
776 770		A/D status (ADCS)

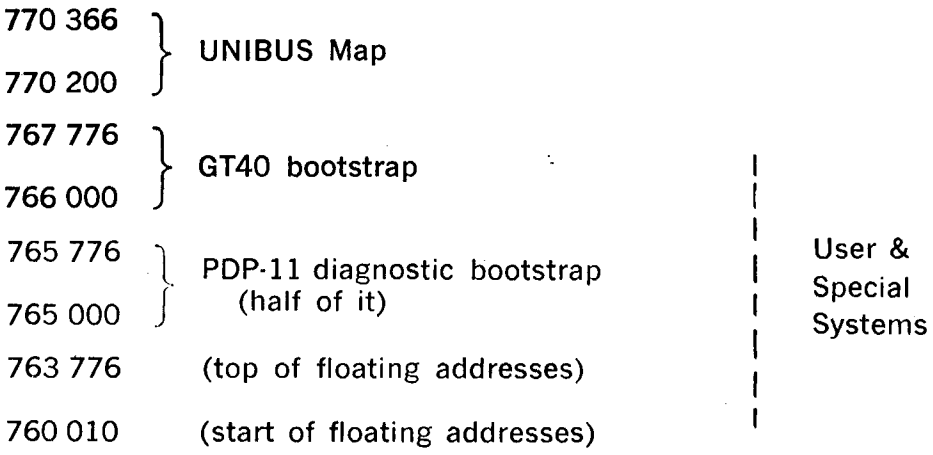
**Also used by RC 11

776 766		register 4 (DAC4)			
776 764		register 3 (DAC3)			
776 762		register 2 (DAC2)			
776 760	AA11 #1,	register 1 (DAC1)			
776 756		D/A status (CSR)			
776 754					
776 752		cont & status #3 (RPCS3)			
776 750		bus adrs ext (RPBAE)			
776 746		ECC pattern (RPEC2)			
776 744		ECC position (RPEC1)			
776 742		error #3 (RPER3)			
776 740		error #2 (RPER2)			
776 736		cur cylinder (RPCC)			silo memory (SILO)
776 734		desired cyl (RPDC)			cyl adrs (SUCA)
776 732		offset (RPOF)			maint 3 (RPM3)
776 730		serial number (RPSN)			maint 2 (RPM2)
776 726		drive type (RPDT)			maint 1 (RPM1)
776 724		maintenance (RPMR)			disk adrs (RPDA)
776 722		data buffer (RPDB)			cyl adrs (RPCA)
776 720	RP04,	look ahead (RPLA)		RP11,	bus adrs (RPBA)
776 716		attn summary (RPAS)			word count (RPWC)
776 714		error #1 (RPER1)			disk status (RPCS)
776 712		drive status (RPDS)			error (RPER)
776 710		cont & status #2 (RPCS2)			disk status (RPDS)
776 706		sector/track adrs (RPDA)			
776 704		UNIBUS address (RPBA)			
776 702		word count (RPWC)			
776 700		cont & status #1 (RPCS1)			
776 676	KL11,	#16			
776 500	DL11-A, -B,	#1			
776 476		#5			
776 400	AA11,	#2			
776 276	DX11				
776 200					
776 176		#31			
775 610	DL11-C, -D, -E,	#1			
775 576		#4			
775 400	DS11,	#1			

775 376	DN11, #16	
775 200	DN11, #1	
775 176	DM11, #16	DV11, #1-4
775 000	DM11, #1	DV11, #1-4
774 776	DP11, #1	
774 400	DP11, #32	
774 376	DC11, #32	
774 000	DC11, #1	
773 766	BM792, BM873 ROM PDP-11 diagnostic bootstrap (half of it)	
773 000		
772 776	PA611 typeset punch	
772 700		
772-676	PA611 typeset reader	
772 600		
772 576	AFC11, maintenance (AFMR)	
772 574	AFC11, MX channel/gain (AFCG)	
772 572	AFC11, flying cap data (AFBR)	
772 570	AFC11, flying cap status (AFCS)	
772 556	XY11 plotter	
772 550	XY11 plotter	
772 546		
772 544	KW11-P, counter	
772 542	KW11-P, count set	
772 540	KW11-P, clock status	
772 536		
772 534		
772 532	TM11, read lines (MTRD)	
772 530	TM11, tape data (MTD)	
772 526	TM11, memory address (MTCMA)	
772 524	TM11, byte record counter (MTBRC)	
772 522	TM11, command (MTC)	
772 500	TM11, tape status (MTS)	
772 516	Memory Mgt reg (MMR3)	
772 476	TM11, cont & status #3 (MTCS3)	
772 474	TM11, bus adrs ext (MTBAE)	
772 472	TM11, tape control (MTTC)	
772 470	TM11, serial number (MTSN)	

772 466		drive type (MTDT)
772 464		maintenance (MTMR)
772 462		data buffer (MTDB)
772 460		check character (MTCK)
772 456	TU16,	attention summary (MTAS)
772 454		error (MTER)
772 452		drive status (MTDS)
772 450		cont & status #2 (MTCS2)
772 446		frame count (MTFC)
772 444		UNIBUS address (MTBA)
772 442		word count (MTWC)
772 440		cont & status #1 (MTCS1)
772 436 } 772 430	DR11-B #2	
772 416		data (DRDB)
772 414	DR11-B #1,	status (DRST)
772 412		bus address (DRBA)
772 410		word count (DRWC)
772 376 } 772 360	Kernel Data PAR, reg 0-7	
772 356 } 772 340	Kernel Instruction PAR, reg 0-7	
772 336 } 772 320	Kernel Data PDR, reg 0-7	
772 316 } 772 300	Kernel Instruction PDR, reg 0-7	
772 276 } 772 260	Supervisor Data PAR, reg 0-7	
772 256 } 772 240	Supervisor Instruction PAR, reg 0-7	
772 236 } 772 220	Supervisor Data Descriptor PDR, reg 0-7	
772 216 } 772 200	Supervisor Instruction Descriptor PDR, reg 0-7	
772 100 } 772 136	UNIBUS Memory Parity	

```
772 072          cont & status #3 (RSCS3)
772 070          bus adrs ext (RSBAE)
772 066          drive type (RSDT)
772 064          maintenance (RSMR)
772 062          data buffer (RSDB)
772 060          look ahead (RSLA)
772 056          attention summary (RSAS)
772 054   RS04,  error (RSER)
772 052          drive status (RSDS)
772 050          control & status #2 (RSCS2)
772 046   RS04,  desired disk adrs (RSDA)
772 044          UNIBUS address (RSBA)
772 042          word count (RSWC)
772 040          control & status #1 (RSCS1)

772 016 }
        } GT40 #2
772 010 }

772 006          Y axis
772 004          X axis
772 002   GT40 #1 status
772 000          program counter

771 776          status (UDCS) |
771 774   UDC11,  scan (UDSR)  | ICS/ICR11
771 772                        |
771 770

771 776 }
        } UDC functional I/O modules
771 000 }

770 776 }        #8
        } KG11,
770 700 }        #1

770 676 }            #16
        } DM11-BB,
770 500 }            #1

770 436              DMA
770 434
770 432
770 430
770 426
770 424
770 422          ext DAC
770 420          D/A YR
770 416          D/A XR
770 414          D/A SR
770 412   LPS11,  D I/O output
770 410          D I/O input
770 406          CKBR
770 404          CKSR
770 402          ADBR
770 400          ADSR
```



NOTE

All presently unused UNIBUS addresses are reserved by Digital.

APPENDIX B

INSTRUCTION TIMING

PDP-11/04 CENTRAL PROCESSOR

INSTRUCTION EXECUTION TIME

The execution time for an instruction depends on the instruction itself and the modes of addressing used. In the most general case, the Instruction Execution Time is the sum of a Basic Time, a Source Address Time, and a Destination Address Time.

$$\text{Instr Time} = \text{Basic Time} + \text{SRC Time} + \text{DST Time}$$

Double Operand instructions require all 3 of these Times, Single Operand instructions require a Basic Time and a DST Time, and with all other instructions the Basic Time is the Instr Time.

All Timing information is in microseconds, unless otherwise noted. Times are typical; processor timing can vary ±10%.

BASIC TIMES

Double Operand Instruction	Basic Time (μ sec) MOS	Parity MOS
ADD, SUB, BIC, BIS	3.17	3.33
CMP, BIT	2.91	3.07
MOV	2.91	3.07
Single Operand		
CLR, COM, INC, DEC, NEG, ADC, SBS	2.65	2.81
ROR, ROL, ASR, ASL	2.91	3.07
TST	2.39	2.55
SWAB	2.91	3.07
All Branches (branch true)	2.65	2.81
All Branches (branch false)	1.87	2.03
Jump Instructions		
JMP	0.91	0.88
JSR	3.27	3.27
Control, Trap, and Miscellaneous Instructions		
RTS	4.11	4.43
RTI, RTT	5.31	5.79
Set N,Z,V,C	2.39	2.55
Clear N,Z,V,C	2.39	2.55
HALT	1.46	1.62
WAIT	2.13	2.29
RESET	100 ms	100 ms
IOT, EMT, TRAP, BPT	7.95	8.49

ADDRESSING TIMES

ADDRESSING FORMAT **Time (μsec)**

Mode	Description	Symbolic	SRC Time*		DST Time**	
			MOS	Parity MOS	MOS	Parity MOS
0	REGISTER	R	0	0	0	0
1	REGISTER DEFERRED	@R or (R)	0.94	1.10	1.48	1.67
2	AUTO-INCREMENT	(R)+	1.20	1.36	1.76	1.95
3	AUTO-INCREMENT DEFERRED	@(R)+	2.66	2.98	3.20	3.55
4	AUTO-DECREMENT	−(R)	1.20	1.36	1.76	1.95
5	AUTO-DECREMENT DEFERRED	@−(R)	2.66	2.98	3.20	3.55
6	INDEX	X(R)	2.92	3.24	3.46	3.81
7	INDEX DEFERRED	@X(R)	4.38	4.86	4.92	5.43

* For Source time, add the following for odd byte addressing: 0.52 (μsec)

** For Destination time, modify as follows:

a) Add for odd byte addressing with a non-modifying instruction: 0.52 (μsec)

b) Add for odd byte addressing with a modifying instruction modes 1-7: 1.04 (μsec)

c) Subtract for all non-modifying instructions except Mode 0:
MOS: 0.54 Parity MOS: 0.57 (μsec)

d) Add for MOVE instructions Mode 1-7: 0.26 (μsec)

e) Subtract for JMP and JSR instructions, modes 3, 5, 6, 7: 0.52 (μsec)

B.2 PDP-11/34A CENTRAL PROCESSOR

INSTRUCTION EXECUTION TIME

The execution time for an instruction depends on the instruction itself, the modes of addressing used, and the type of memory being referenced. In the most general case, the Instruction Execution Time is the sum of a Source Address Time, a Destination Address Time, and an Execute, Fetch Time.

$$\text{Instr Time} = \text{SRC Time} + \text{DST Time} + \text{EF Time}$$

Some of the instructions require only some of these times, and are so noted. All Timing information is in microseconds, unless otherwise noted. Times are typical; processor timing can vary ± 10%.

BASIC INSTRUCTION SET TIMING

Double Operand

$$\text{Instr Time} = \text{SRC Time} + \text{DST Time} + \text{EF Time}$$

Single Operand

$$\text{Instr Time} = \text{DST Time} + \text{EF Time}$$

Branch, Jump, Control, Trap, & Misc

$$\text{Instr Time} = \text{EF Time}$$

NOTES

1) The times specified apply to both word and byte instructions whether odd or even byte.
2) Timing is given without regard for NPR or BR servicing.
3) If the memory management is enabled execution times increase by 0.12 μsec for each memory cycle used.
4) All timing is based on memory with the following performance characteristics:

Memory	Access Time	Cycle Time
Core (MM11-DP)	.510 μsec	1.0 μsec
MOS (MS11-JP)	.635	.775

I. SOURCE ADDRESS TIME

Instruction	Source Mode	Memory Cycles	Core (MM11-DP)	MOS (MS11-JP)
Double Operand	0	0	0.00 μsec	0.00 μsec
	1	1	1.13	1.26
	2	1	1.33	1.46
	3	2	2.37	2.62
	4	1	1.28	1.41
	5	2	2.57	2.82
	6	2	2.57	2.82
	7	3	3.80	4.18

II. DESTINATION TIME

Instruction	Destination Mode	Memory Cycles	Core	MOS
Modifying Single Operand and Modifying Double Operand (Except MOV, SWAB, ROR, ROL ASR ASL)	0	0	0.00	0.00
	1	2	1.62	1.74
	2	2	1.77	1.89
	3	3	2.90	3.15
	4	2	1.77	1.89
	5	3	3.00	3.25
	6	3	3.10	3.35
	7	4	4.29	4.66
MOV	0	0	0.00	0.00
	1	1	0.93	0.93
	2	1	0.93	0.93
	3	2	2.17	2.29
	4	1	1.13	1.13
	5	2	2.22	2.34
	6	2	2.37	2.49
	7	3	3.50	3.75
MTPS	0	0	0.00	0.00
	1	1	0.95	0.95
	2	1	1.13	1.26
	3	2	2.26	2.51
	4	1	1.13	1.26
	5	2	2.26	2.51
	6	2	2.44	2.69
	7	3	3.57	4.20

	Destination Mode	Memory Cycles	Core	MOS
MFPS	0	0	0.00	0.00
	1	1	0.64	0.64
	2	1	0.64	0.64
	3	2	1.95	2.08
	4	1	0.82	0.82
	5	2	1.95	2.08
	6	2	2.13	2.26
	7	3	3.26	3.51

III. EXECUTE, FETCH TIME

DOUBLE OPERAND

Instruction	Memory Cycles	Core	MOS
ADD, SUB, CMP, BIT, BIC, BIS, XOR	1	2.03	2.16
MOV	1	1.83	1.96

SINGLE OPERAND

CLR, COM, INC, DEC, ADC, SBC, TST	1	1.83	1.96
SWAB, NEG	1	2.03	2.16
ROR, ROL, ASR, ASL	1	2.18	2.31
MTPS	2	2.99	3.12
MFPS	2	1.99	2.12

EIS INSTRUCTIONS (use with DST times)

MUL	1	*8.82	*8.95
DIV (overflow)	1	2.78	2.91
		12.48	12.61
ASH	1	**4.18	**4.31
ASHC	1	**4.18	**4.31

MEMORY MANAGEMENT INSTRUCTIONS

MFPI (D)	2	3.07	3.14
MTPI (D)	2	3.37	3.34

* Add 200ns for each bit transition in serial data from LSB to MSB
** Add 200ns per shift

Instruction	Destination Mode	Memory Cycles	Core	MOS
SWAB, ROR, ROL, ASR, ASL	0	0	0.00	0.00
	1	2	1.42	1.54
	2	2	1.57	1.69
	3	3	2.70	2.95
	4	2	1.62	1.74
	5	3	2.80	3.05
	6	3	2.90	3.15
	7	4	4.09	4.46
Non-Modifying Single Operand and Double Operand	0	0	0.00	0.00
	1	1	1.13	1.26
	2	1	1.28	1.41
	3	2	2.42	2.67
	4	1	1.33	1.46
	5	2	2.52	2.77
	6	2	2.62	2.87
	7	3	3.80	4.18
MFPI (D) MTPI (D)	0	0	0.00	0.00
	1	1	0.98	1.24
	2	1	1.32	1.44
	3	2	2.20	2.45
	4	1	1.18	1.44
	5	2	2.20	2.45
	6	2	2.40	2.65
	7	3	3.59	3.96

BRANCH INSTRUCTIONS

Instruction	Memory Cycles	Core	MOS
BR, BNE, BEQ, (Branch) BPL, BMI, BVC, BVS, BCC, BCS, BGE, BLT, BGT, BLE, BHI, BLOS, BHIS, BLO	1	2.18	2.31
(No Branch)	1	1.63	1.76
SOB (Branch)	1	2.38	2.51
(No Branch)	1	1.98	2.11

JUMP INSTRUCTIONS

	Destination Mode	Memory Cycles	Core	MOS
JMP	1	1	1.83	1.96
	2	1	2.18	2.31
	3	2	3.12	3.37
	4	1	2.03	2.16
	5	2	3.07	3.32
	6	2	3.07	3.32
	7	3	4.25	4.78
JSR	1	2	3.32	3.44
	2	2	3.47	3.59
	3	3	4.40	4.65
	4	2	3.32	3.44
	5	3	4.40	4.65
	6	3	4.60	4.85
	7	4	5.69	6.06

Instruction	Memory Cycles	Core	MOS
RTS	2	3.32	3.57
MARK	2	4.27	4.52
RTI, RTT	3	4.60	4.98
Set or Clear C,V,N,Z	1	2.03	2.16
HALT	1	1.68	1.81
WAIT	1	1.68	1.81
RESET	1	100 msec	100 msec
IOT, EMT, TRAP, BPT	5	7.32	7.7

LATENCY

Interrupts (BR requests) are acknowledged at the end of the current instruction. For a typical instruction, with an instruction execution time of 4 μsec, the average time to request acknowledgement would be 2 μsec.

Interrupt service time, which is the time from BR acknowledgement to the first subroutine instruction, is 7.32 μsec, max. for core, and 7.7 μsec for MOS.

NPR (DMA) latency, which is the time from request to bus mastership for the first NPR device, is 2.5 μsec, max.

NOTES

1. Add 0.84 μseconds when in rounding mode (FT = 0).
2. Add 0.24 μseconds per shift to align binary points and 0.24 μseconds per shift for normalization. The number of alignment shifts is equal to the exponent difference for exponent differences bounded as follows:

   $1 \leq |EXP\ (AC) — EXP\ (FSRC)| \leq 24$ single precision
   $1 \leq |EXP\ (AC) — EXP\ (FSRC)| \leq 56$ double precision

   The number of shifts required for normalization is equivalent to the number of leading zeroes of the result.
3. Add .24 μseconds times the exponent of the product if the exponent of the product is:

   $1 \leq EXP\ (PRODUCT) \leq 24$ single-precision
   $1 \leq EXP\ (PRODUCT) \leq 56$ double-precision

   Add 0.24 μseconds per shift for normalization of the fractional result. The number of shifts required for normalization is equivalent to the number of leading zeroes in the fractional result.
4. Add 0.24 μseconds per shift for normalization of the integer being converted to a floating point number. For positive integers, the number of shifts required to normalize is equivalent to the number of leading zeroes; for negative integers, the number of shifts required for normalization is equivalent to the number of leading ones.
5. Add 0.24 μseconds per shift to convert the fraction and exponent to integer form, where the number of shifts is equivalent to 16 minus the exponent when converting to short integer or 32 minus the exponent when converting to long integer for exponents bounded as follows:

   $1 \leq EXP\ (AC) \leq 15$ short integer
   $1 \leq EXP\ (AC) \leq 31$ long integer

B.3 PDP-11/44 CENTRAL PROCESSOR

Timing for the instructions assumes the following conditions:

1. Times specified are typical and may vary by ±10%. They apply to both byte and word instructions, whether odd or even byte.
2. Timing is given without regard to NPR or BR servicing and assumes that no service states are used except where explicitly forced by the microstructures.
3. Cache times assume 100% hits. Non-cache times assume 0% hits.
4. If memory management is used, add 0.09 μsec per memory to the instruction time.
5. The memory timing is assumed to be the following:

MS11-M	DATI (P)	490 ns taa
	DATO (B)	230 ns taa

6. All times are expressed in μsec.

MOV, CMP, BIT, BIS, BIC, ADD, SUB Except any SMO/DMO Combinations

REGISTER TO REGISTER INSTRUCTION TIMES

INST	UNCACHED TIME	CACHED TIME	# MEM CYCLE
MOV (0,0)	1.23	.60	1
ADD, BIS, BIC (0,0)		1.41	.78
1CMP, BIT, SUB			

For the following instructions, use the time indicated directly.

To figure time, add SRC time from the first table to DST time from the second table for the appropriate instruction.

SRC MODE TIMES FOR ALL INSTRUCTIONS LISTED (INCLUDING FETCH)

SRC MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	1.23	.64	1
1	1.92	.66	2
2	2.10	.84	2
3	2.97	1.08	3
4	2.10	.84	2
5	2.97	1.08	3
6	3.15	1.26	3
7	4.02	1.50	4

MOV DST MODE TIMES

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	.18	.18	0
1	.77	.77	0

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
2	.77	.77	0
3	1.82	1.19	1
4	.95	.95	0
5	1.82	1.19	1
6	2.00	1.37	1
7	2.87	1.60	2

ADD, BIS, BIC

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	.36	.36	0
1	1.46	.83	1
2	1.64	1.01	1
3	2.51	1.25	2
4	1.64	1.01	1
5	2.51	1.25	2
6	2.69	1.43	2
7	3.56	1.67	3

CMP, BIT

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	.36	.36	0
1	1.05	.42	1

INST	UNCACHED TIME	CACHED TIME	# MEM CYCLE
2	1.23	.60	1
3	2.10	.84	2
4	1.23	.60	1
5	2.10	.84	2
6	2.28	1.02	2
7	3.15	1.26	3

SUB

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	.18	.18	0
1	1.46	.83	1
2	1.64	1.01	1
3	2.51	1.25	2
4	1.64	1.01	1
5	2.51	1.25	2
6	2.69	1.43	2
7	3.56	1.67	3

XOR, NEG

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	1.59	.96	1
1	2.69	1.43	2

SRC MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
2	2.89	1.61	2
3	3.74	1.85	3
4	2.87	1.61	2
5	3.74	1.85	3
6	3.92	2.03	3
7	4.79	2.27	4

CLR, COM, INC, DEC, SBL, ADL, SXT

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	1.23	.60	1
1	2.51	1.25	2
2	2.69	1.43	2
3	3.56	1.67	3
4	2.69	1.43	2
5	3.56	1.67	3
6	3.74	1.85	3
7	4.61	2.09	4

TST

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	1.23	.60	1

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
1	2.60	.84	2
2	2.28	1.02	2
3	3.15	1.26	3
4	2.28	1.02	2
5	3.15	1.26	3
6	3.33	1.44	3
7	4.20	1.68	4

ROL, ROR, ASR, ASL

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	1.59	.96	1
1	2.69	1.43	2
2	2.87	1.61	2
3	3.74	1.85	3
4	2.87	1.61	2
5	3.74	1.85	3
6	3.92	2.03	3
7	4.79	2.27	4

SWAB

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	1.41	.78	1

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
1	2.51	1.25	2
2	2.69	1.43	2
3	3.56	1.67	3
4	2.69	1.43	2
5	3.56	1.67	3
6	3.74	1.85	3
7	4.61	2.09	4

MFPI (D)

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	2.18	1.55	1
1	3.23	1.97	2
2	3.41	2.15	2
3	4.10	2.21	3
4	3.41	2.15	2
5	4.10	2.21	3
6	4.28	2.39	4
7	5.15	2.64	4

MTPI (D)

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	2.64	1.38	2

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
1	3.59	2.26	2
2	3.27	2.51	2
3	4.46	2.57	3
4	3.27	2.51	2
5	4.46	2.57	3
6	4.64	2.75	3
7	5.51	2.99	4

JMP

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
1	1.23	.60	1
2	1.59	.96	1
3	2.28	1.02	2
4	1.41	.78	1
5	2.28	1.02	2
6	2.28	1.02	2
7	3.33	1.44	3

JSR

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
1	2.47	1.91	1
2	2.65	2.09	1

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
3	3.34	2.15	2
4	2.47	1.91	1
5	3.34	2.15	2
6	3.52	2.40	2
7	4.39	2.57	3

CALL TO SUPERVISOR MODE

DST MODE	UNCACHED TIME	CACHED TIME	# MEM CYCLE
0	7.46	6.20	2
1	8.15	6.44	3
2	8.33	6.45	3
3	9.20	6.68	4
4	8.33	6.44	3
5	9.20	6.68	4
6	9.38	6.86	4
7	10.25	7.10	5

BRANCHES

BNE, ETC. TYPE	UNCACHEDCASHED		# MEM CYCLE
FAILED	1.05	.42	1
PASSED	1.59	.96	1
SOB NO BRANCH (1)		1.41	.78
	1.77	1.14	1

TRAP, SUBROUTINES

		UNCACHED	CACHED	# MEM CYCLE
TRAP INST.		5.68	3.93	3
RTS		2.46	1.20	2
RTI, RTT	K	3.61	1.92	3
	K	4.35	2.46	3

MISCELLANEOUS

	UNCACHED	CACHED	# MEM
SET, CLR CC's	1.41	.78	(1)
WAIT (LOOP)	1.53	.90	(1)
EXIT	5.56	3.67	(3)
RESET	1.23	.60	1
	1.50	In Kernel Mode	
MARK	3.36	2.10	2
MFPT	1.41	.78	1
SPL	2.85	2.22	1

EIS

ASH DM	0	3.93	1	3.30	ADD 180 ns.
	1	4.62	2	3.36	FOR TRANS.
	2	4.80	2	3.54	FOR RIGHT SHIFTS
	3	5.67	2	3.78	SUBTRACT 0.6 ns.
	4	4.80	2	3.36	
	5	5.60	3	3.78	
	6	5.78	3	3.89	
	7	6.65	4	4.13	

ASHC DM	0	3.51	1	2.88	ADD 180 ns.
	1	4.20	2	2.94	FOR TRANS.
	2	4.38	2	3.12	
	3	5.25	3	3.36	
	4	4.38	2	3.12	
	5	5.25	3	3.36	
	6	5.43	3	3.54	
	7	6.30	4	3.78	
MUL DM	0	6.63	1	6.00	ADD 180 ns. PER
	1	7.32	2	6.06	BIT TRANSITION
	2	7.50	2	6.24	
	3	8.37	3	6.48	
	4	7.50	2	6.24	
	5	8.37	3	6.48	
	6	8.55	3	6.66	
	7	9.42	4	6.90	
DIV DM	0	11.01	1	10.28	
	1	11.07	2	10.44	
	2	11.88	2	10.62	
	3	12.75	3	10.86	
	4	11.88	1	10.62	
	5	12.75	3	10.86	
	6	12.93	3	11.04	
	7	13.08	4	11.28	

B.4 PDP-11/60 CENTRAL PROCESSOR

The execution time for an instruction depends on the instruction itself, the modes of addressing used, and the type of memory being referenced. In the most general case, the Instruction Execution Time is the sum of a Source Address Time, a Destination Address Time, and an Execute, Fetch Time.

$$\text{Instr Time} = \text{SRC Time} + \text{DST Time} + \text{EF Time}$$

Some of the instructions require only some of these times and are so noted. Times are typical and are based upon the MM11-WP memory as backing store. The simplified presentation of the timing data has occasionally resulted in a larger time for an instruction being noted. All times may vary +10% due to clock and bus tolerances.

BASIC INSTRUCTION SET TIMING

Double Operand

all instructions,
except MOV: Instr Time = SRC Time + DST Time + EF Time
MOV: Instr Time = SRC Time + EF Time (word only)

Single Operand

all instructions: Instr Time = DST Time + EF Time or
Instr Time = SRC Time + EF Time

Branch, Jump, Control, Trap & Misc

all instructions: Instr Time = EF Time

EIS (MUL, DIV, ASH, ASHC)

all instructions: Instr Time = DST Time + EF Time

Floating Point

all instructions:
except ABSF, ABSD,
NEGF, and NEGD: Instr Time = SRC Time + EF Time
ABSF, ABSD,
NEGF and NEGD: Instr Time = DST Time + EF Time

Using the Chart Times

To compute a particular instruction time, first find the instruction "EF" Time. Select the proper EF Time for the SRC and DST modes. Observe all "NOTES" to the EF Time by adding the correct amount to basic EF number.

Next, note whether the particular instruction requires the inclusion of SRC and DST Times; if so, add the appropriate amounts to correct EF number.

Chart Times

The times given in the chart are for Cache "hits"; that is, all the read cycles are assumed to be in the Cache. The number of read cycles in each subset of the instruction is also included so that timing can be calculated for a specific case of hits and misses, or timing can be calculated based on an average hit rate.

a) Specific hits and misses
   Add 1.1 μsec for each read cycle which is a miss instead of a hit.

b) Average hit rate
   If P_H is the percent of reads that are hits, add $1.1 \times (1 - P_H) \times$ (number of read cycles) to the instruction timing.

For example, an ADD A,B instruction using Mode 6 (indexed) address modes:

1) All Hits:

SRC time	= 0.85	μsec	2 read cycles
DST time	= 0.85	μsec	2 read cycles
EF time	= 2.2	μsec	1 read cycle
TOTAL	= 3.9	μsec	5 read cycles

2) 4 Hits, 1 Miss
   Total = 3.9 + 1.1
   = 5.0 μsec.

3) Read hit rate of 87%
   Total = 3.9 + (1.1)(1 − .87)(5)
   = 4.6 μsec.

NOTES

1. The times specified generally apply to Word instructions. In most cases Even Byte instructions have the same time, with some Odd Byte instructions taking longer. All exceptions are noted.
2. Timing is given without regard for NPR or BR serving.
3. Times are not affected if Memory Management is enabled.
4. All times are in microseconds, except where noted.

Source Address Time

Instruction	Source Mode	SRC Time	Read Memory Cycle
Double Operand	0	.00	0
	1	.51	1
	2	.51	1
	3	1.0	2
	4	.68	1
	5	1.2	2
	6	.85	2
	7	1.4	3

Destination Address Time

Instruction	DST Mode	DST Time (A)	Read Memory Cycle
Single Operand and Double Operand (except MOV, MTPI, MTPD, JMP, JRS)	0	.00	0
	1	.51	1
	2	.51	1
	3	1.0	2
	4	.68	1
	5	1.2	2
	6	.85	2
	7	1.4	3

NOTE (A): Add .17 μsec for odd byte instructions, except DST Mode 0.

Execute, Fetch Time

(Double Operand)

Instruction (Use with SRC Time and DST Time)	EF Time (SRC Mode 0) (DST Mode 0)	Read Mem. CYC	EF Time (SRC Mode 1-7) (DST Mode 0)	Read Mem. CYC	EF Time (SRC Mode 0-7) (DST Mode 1-7)	Read Mem. CYC
ADD, SUB, BIC, BIS	.34	1	1.0	1	2.2	1
CMP, BIT	.34	1	1.0	1	1.0	1
XOR	.34	1	—	—	1.0	1
MOVB	.34	1	.51	1	.51	1

Instruction (Use with SRC Time)	DST Mode	DST Register	EF Time (SRC Mode = 0)	EF Time (SRC Mode 1-7)	Read Memory Cycle
MOV	0	0-7	.34	.51	1
	1	0-7	1.0	1.0	1
	2	0-7	1.0	1.0	1
	3	0-7	1.4	1.4	2
	4	0-7	1.2	1.0	1
	5	0-7	1.5	1.5	2
	6	0-7	1.2	1.4	2
	7	0-7	1.7	1.9	3

Execute, Fetch Time

(Single Operand)

Instruction (Use with DST Time)	EF Time (DST Mode = 0)	Read Memory Cycle	EF Time (DST Mode 1-7)	Read Memory Cycle
TST	.34	1	.68	1
CLR, COM, INC, DEC, ADC, ROL, ASL	.34	1	1.9	1
NEG, SBC, ROR, ASR	1.2	1	2.4	1

Instruction	EF Time	Read Memory Cycle	
MFPI, MFPD	6.1	1	Use with SRC Times

Instruction	DST Mode	Instruction Time	Read Memory Cycle
MTPI, MTPD	0	3.6	1
	1	6.1	2
	2	6.3	2
	3	6.6	3
	4	6.3	2
	5	6.8	3
	6	6.6	3
	7	7.1	4

Branch Instructions

Instruction	Instruction Time	Read Memory Cycle
BR, BNE, BEQ, BPL, BMI, BVC, BVS, BCC, BGS, BGE, BLT, BGT, BLE, BHI, BLOS, BHIS, BLO	.85	1
SOB	2.0	1

JUMP Instructions

Instruction	DST Mode	Instruction Time	Read Memory Cycle
JMP	1	1.2	1
	2	1.4	1
	3	1.5	2
	4	1.4	1
	5	1.7	2
	6	1.4	2
	7	1.9	3

Instruction	DST Mode	Instruction Time	Read Memory Cycle
JSR	1	2.5	1
	2	2.7	1
	3	2.9	2
	4	2.7	1
	5	3.2	2
	6	2.9	2
	7	3.6	3

Miscellaneous Instructions

Instruction	Instruction Time	Read Memory Cycle
RTS	1.5	2
MARK	2.4	2
RTI	2.4	3
RTT	3.1	3
SET N, V, Z, C	1.5	1
CLR N, V, Z, C		
RESET	10 msec	1
IOT, EMT, BPT, TRAP	4.6	3

EIS Instructions MUL, DIV, ASH, ASHC

Source Address Time

Source Mode	Time (μsec)	Read Memory Cycle
0	.340	0
1	.640	1
2	.640	1
3	1.19	2
4	.85	1
5	1.36	2
6	1.19	2
7	1.70	3

Add 1.1 μsec for each read cycle which is a miss

EF Time

Instruction	EF Time (All Modes)	Read Memory Cycle
DIV	7.65 μsec	1
MUL	6.12 μsec	1
*ASH	3.57 μsec	1
*ASHC	4.25 μsec	1

*Add .17 μsec for each shift

B.5 PDP-11/70 CENTRAL PROCESSOR

The execution time for an instruction depends on the instruction itself, the modes of addressing used, and the type of memory being referenced. In the most general case, the Instruction Execution Time is the sum of a Source Address Time, and an Execute, Fetch Time.

$$\text{Instr Time} = \text{SRC Time} + \text{DST Time} + \text{EF Time}$$

Some of the instructions require only some of these times, and are so noted. Times are typical; processor timing, with core memory, may vary +15% to −10%.

BASIC INSTRUCTION SET TIMING

Double Operand

all instructions,
except MOV: Instr Time = SRC Time + DST Time
(but including MOVB) + EF Time
MOV Instruction: Instr Time = SRC Time + EF Time
(word only)

Single Operand

all instructions: Instr Time = DST Time + EF Time or
Instr Time = SRC Time + EF Time

Branch, Jump, Control, Trap & Misc

all instructions: Instr Time = EF Time

USING THE CHART TIMES

To compute a particular instruction time, first find the instruction "EF" Time. Select the proper EF Time for the SRC and DST modes. Observe all "NOTES" to the EF Time by adding the correct amount to basic EF number.

Next, note whether the particular instruction requires the inclusion of SRC and DST Times, if so, add the appropriate amounts to correct EF number.

USING THE CHART TIMES

The times given in the chart for Cache "hits"; that is, all the read cycles are assumed to be in the Cache. The number of read cycles in each subset of the instruction is also included so that timing can be calculated for a specific case of hits and misses, or timing can be calculated based on an average hit rate.

a) Specific hits and misses
Add 1.02 μsec for each read cycle which is a miss instead of a hit.

b) Average hit rate
If P_H is the percent of reads that are hits, add $1.02 \times (1 - P_H) \times$ (Number of read cycles) to the instruction timing.

For example, an ADD A,B instruction using Mode 6 (indexed) address modes:

1) All Hits:

SRC time	= 0.60 μsec	2 read cycles
DST time	= 0.60 μsec	2 read cycles
EF time	= 1.35 μsec	1 read cycle
TOTAL	= 2.55 μsec	5 read cycles

2) 4 Hits, 1 Miss
Total = 2.55 + 1.02
= 3.57 μsec

3) Read hit rate of 90%
Total = 2.55 + (1.02) (.1) (5)
= 3.06 μsec

NOTES

1. The times specified generally apply to Word instructions. In most cases Even Byte instructions have the same time, with some Odd Byte instructions taking longer. All exceptions are noted.
2. Timing is given without regard for NRP or BR serving. Core memory is assumed to be iocated within the first 128K memory unit.

3. Times are not affected if Memory Management is enabled.
4. All times are in microseconds.

SOURCE ADDRESS TIME

Instruction	Source Mode	SRC Time	Read Memory Cycles
Double Operand	0	.00	0
	1	.30	1
	2	.30	1
	3	.75	2
	4	.45	1
	5	.90	2
	6	.60	2
	7	1.05	3

DESTINATION ADDRESS TIME

Instruction	DST Mode	DST Time (A)	Read Memory Cycles
Single Operand and Double Operand (except MOV, MTPI, MTPD, JMP, JRS	0	.00	0
	1	.30	1
	2	.30	1
	3	.75	2
	4	.45	1
	5	.90	2
	6	.60	2
	7	1.05	3

NOTE (A): Add .15 μsec for odd byte instructions, except DST Mode 0.

EXECUTE, FETCH TIME

Double Operand

Instruction (Use with SRC Time and DST Time)	EF Time (SRC Mode 0) (DST Mode 0)	Read Mem Cyc	EF Time (SRC Mode 1-7) (DST Mode 0)	Read Mem Cyc	EF Time (SRC Mode 0-7) (DST Mode 1-7)	Read Mem Cyc
ADD, SUB, BIC, BIS MOVB	.30 (D)	1	.45 (D)	2	1.20 (C)	1
CMP, BIT	.30 (D)	1	.45 (D)	1	.45 (C)	1
XOR	.30 (D)	1	.30 (D)	1	1.20	1

NOTE (C): Add 0.15 μsec if SRC is R1 to R7 and DST is R6 or R7.
NOTE (D): Add 0.3 μsec if DST is R7.

Instruction (Use with SRC Time)	DST Mode	DST Register	EF Time (SRC Mode = 0)	EF Time (SRC Mode = 1-7)	Read Memory Cycles
MOV	0	0-6	.30	.45	1
	0	7	.60	.75	1
	1	0-7	1.20	1.20	1
	2	0-7	1.20	1.20	1
	3	0-7	1.65	1.65	2
	4	0-7	1.35	1.35	1
	5	0-7	1.80	1.80	2
	6	0-7	1.50	1.65	2
	7	0-7	1.95	2.10	3

Single Operand

Instruction (Use with DST Time)	EF TIME (DST Mode = 0)	Memory Cycles	EF Time (DST Mode 1 to 7)	Read Memory Cycles
CLR, COM, INC, DEC, ADC, SBC, ROL, ASL, SWAB, SXT	.30 (J)	1	1.20	1
NEG	.75	1	1.50	1
TST	.30 (J)	1	.45	1
ROR, ASR	.30 (J)	1	1.20 (H)	1
ASH, ASHC	.75 (I)	1	.90 (I)	1

NOTE (H): Add 0.15 μsec if odd byte.
NOTE (I): Add 0.15 μsec per shift.
NOTE (J): Add 0.30 μsec if DST is R7.

Instruction (Use with SRC Times)	EF Time	Read Memory Cycles
MUL	3.30	1
DIV		
by zero	.90	1
shortest	7.05	1
longest	8.55	1

Instruction	EF Time	Read Memory Cycles	
MFPI	1.50	1	use with SRC times
MFPD	1.50	1	

Instruction	DST Mode	Instruction Time	Read Memory Cycles
MTPI	0	.90	1
MTPD	1	1.65	2
	2	1.65	2
	3	2.10	3
	4	1.80	2
	5	2.25	3
	6	2.10	3
	7	2.55	4

Branch Instructions

Instruction	Instr Time (Branch)	Instr Time (No Branch)	Read Memory Cycles
BR, BNE, BEQ, BPL, BMI, BVC, BVS, BCC, BCS, BGE, BLT, BGT, BLE, BHI, BLOS, BHIS, BLO	.60	.30	1
SOB	.60	.75	1

Jump Instructions

Instruction	DST Mode	Instr Time	Read Memory Cycles
JMP	1	.90	1
	2	.90	1
	3	1.20	2
	4	.90	1
	5	1.35	2
	6	1.05	2
	7	1.50	3
JSR	1	1.95	1
	2	1.95	1
	3	2.25	2
	4	1.95	1
	5	2.40	2
	6	2.10	2
	7	2.55	3

Control, Trap & Miscellaneous Instructions

Instruction	Instr Time	Read Memory Cycles
RTS	1.05	2
MARK	.90	2
RTI, RTT	1.50	3
SET N, Z, V. C CLR, N, Z, V, C	.60	1
HALT	1.05	0
WAIT	.45	0
WAIT Loop for a BR is .3 μsec.		
RESET	10ms	1
IOT, EMT, TRAP, BRT	3.30	3
SPL	.60	1
INTERRUPT	2.31	2
First Device		

EFFECTIVE MEMORY CYCLE TIME

The overall effective cycle time of the CPU can be calculated from the following formula:

$$TC_E = P_R \times [(P_H \times TC_H) + (1 - P_H)\, TC_M] + (1 - P_R)\, TC_W$$

Where TC_E = Effective cycle time
TC_H = Cycle time for a read hit = 0.30 μsec
TC_M = Cycle time for a read miss = 1.32 μsec
TC_W = Cycle time for a write = 0.75 μsec
P_R = Percent of cycles that are reads
P_H = Percent of reads that are hits

Thus, for an average PDP-11/70 program which has a read rate of 91% and a read hit rate of 93%, the effective cycle time is:

$TC_E = .91 \times [(.93 \times .30) + (.07 \times 1.32)] + (.09 \times .75) = .41$ μsec

APPENDIX C
CONVERSION TABLE

Decimal	Octal	Binary
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	10	1000
9	11	1001
10	12	1010
11	13	1011
12	14	1100
13	15	1101
14	16	1110
15	17	1111
16	20	10000
17	21	10001
18	22	10010
19	23	10011
20	24	10100
21	25	10101
22	26	10110
23	27	10111
24	30	11000
25	31	11001
26	32	11010
27	33	11011
28	34	11100
29	35	11101
30	36	11110
31	37	11111
32	40	100000
33	41	100001
34	42	100010
35	43	100011

Decimal	Octal	Binary
36	44	100100
37	45	100101
38	46	100110
39	47	100111
40	50	101000
41	51	101001
42	52	101010
43	53	101011
44	54	101100
45	55	101101
46	56	101110
47	57	101111
48	60	110000
49	61	110001
50	62	110010
51	63	110011
52	64	110100
53	65	110101
54	66	110110
55	67	110111
56	70	111000
57	71	111001
58	72	111010
59	73	111011
60	74	111100
61	75	111101
62	76	111110
63	77	111111
64	100	1000000
65	101	1000001
66	102	1000010
67	103	1000011
68	104	1000100
69	105	1000101
70	106	1000110
71	107	1000111
72	110	1001000
73	111	1001001
74	112	1001010
75	113	1001011
76	114	1001100
77	115	1001101

Decimal	Octal	Binary
78	116	1001110
79	117	1001111
80	120	1010000
81	121	1010001
82	122	1010010
83	123	1010011
84	124	1010100
85	125	1010101
86	126	1010110
87	127	1010111
88	130	1011000
89	131	1011001
90	132	1011010
91	133	1011011
92	134	1011100
93	135	1011101
94	136	1011110
95	137	1011111
96	140	1100000
97	141	1100001
98	142	1100010
99	143	1100011
100	144	1100100
101	145	1100101
102	146	1100110
103	147	1100111
104	150	1101000
105	151	1101001
106	152	1101010
107	153	1101011
108	154	1101100
109	155	1101101
110	156	1101110
111	157	1101111
112	160	1110000
113	161	1110001
114	162	1110010
115	163	1110011
116	164	1110100
117	165	1110101
118	166	1110110
119	167	1110111

Decimal	Octal	Binary
120	170	1111000
121	171	1111001
122	172	1111010
123	173	1111011
124	174	1111100
125	175	1111101
126	176	1111110
127	177	1111111
128	200	10000000

INDEX