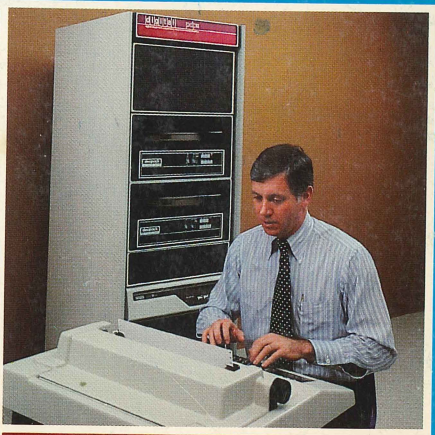
digital

pdp11

processor handbook



04/34/45/55/60

digital

pdp11

processor handbook

04/34/45/55/60

digital

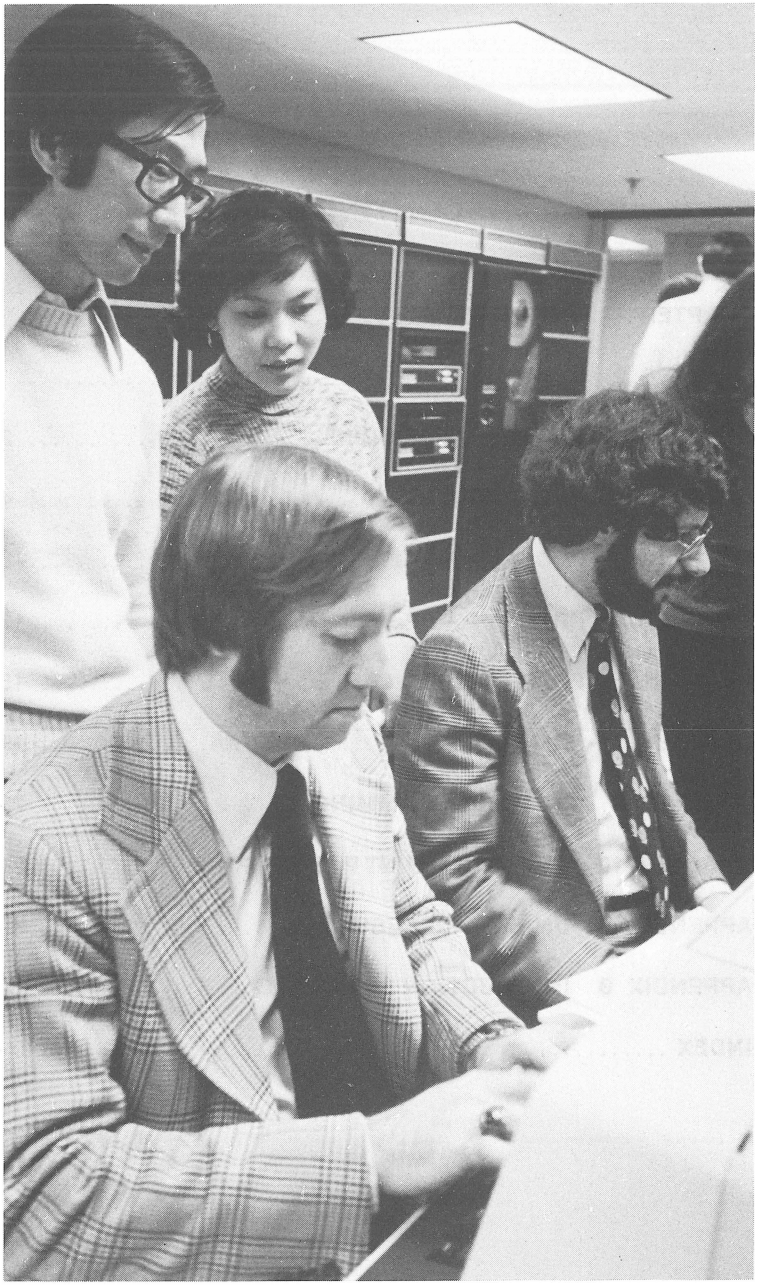
pdp11

04/34/45/55/60 processor handbook

digital equipment corporation

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.

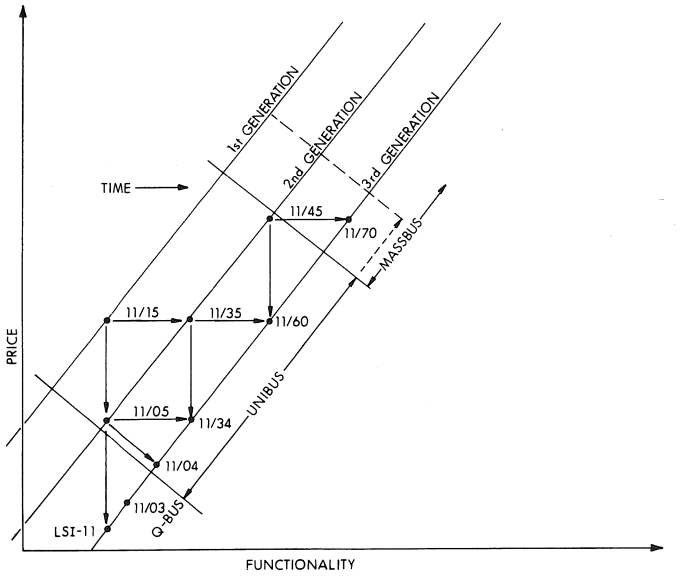


Figure 1-1 PDP-11 Family Development

The processors specifically discussed in this handbook are the:

- PDP-11/04
- PDP-11/34
- PDP-11/45
- PDP-11/55
- PDP-11/60

PDP is the acronym for Programmable Data Processor; 11 is the number of the series of the processors designed by DIGITAL. The numeral following the 11 refers to the *general* relative power of the processor. The PDP-11/60 is, for example, a more powerful processor than the PDP-11/04.

Historically, there were PDP-1s through PDP-10s designed before the PDP-11 family appeared. Although the PDP-8 family continues to be one of DIGITAL's most successful and stable product lines, it is in the PDP-11 family that there has been the greatest range of growth and development. 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. It is the UNIBUS and its unique capabilities which have provided the flexibility and growth options for the PDP-11 family members discussed in this handbook. Figure 1-2 illustrates the block structure of the PDP-11.

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, especially in areas like memory management. 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.

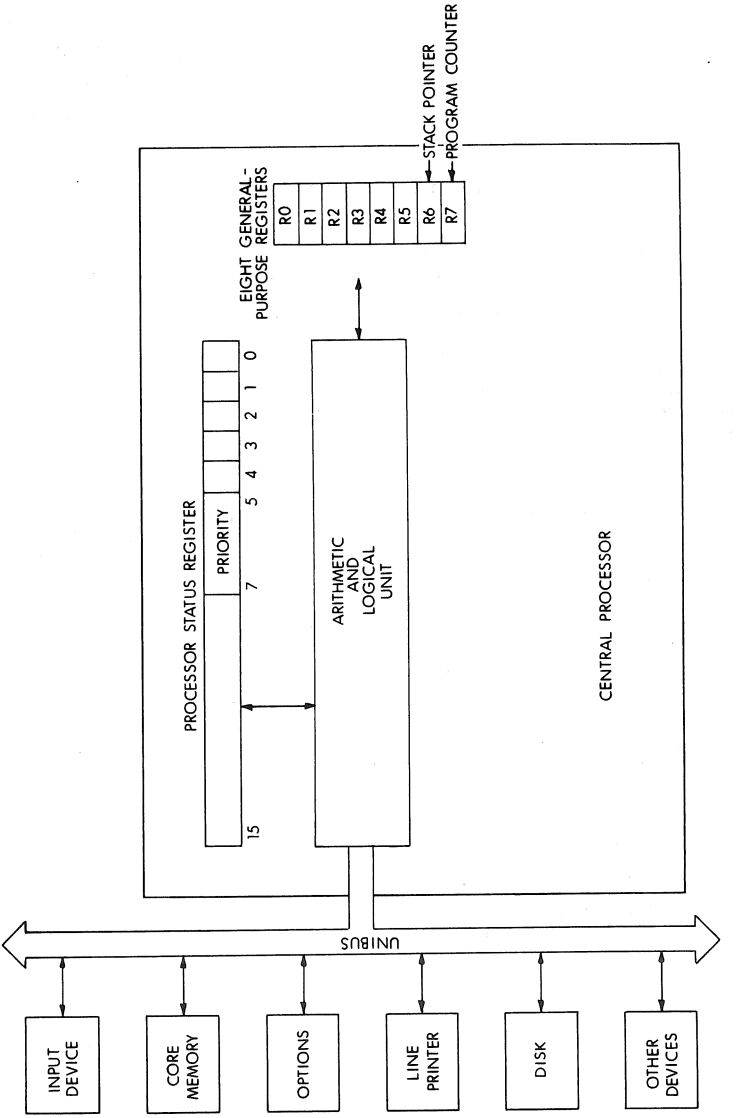


Figure 1-2 PDP-11 Block Structure

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 1-1 illustrates these software products.

Table 1-2 PDP-11 Software Systems Summary

LSI-11 based	11/04	11/34	11/55	11/60	11/70
RT-11 Foreground/Background or Single Job Operating System 16K to 56K bytes of memory. In 16K bytes: Single Job (SJ) operation; subset MACRO included; BASIC, FORTRAN IV, FOCAL as options. In 32K bytes: Foreground/Background (F/B) or SJ operation; languages can support string operations, laboratory and graphics peripherals; full MACRO assembler included; multi-user BASIC available as option supporting as many as 4 users (under SJ monitor). MU BASIC supports as many as 8 users in 48K bytes under SJ monitor; and as many as 4 in 56K bytes under F/B monitor. **Languages:** MACRO included; FORTRAN IV; BASIC, MU BASIC, FOCAL, and APL are options.					
	MUMPS-11 Multi-User Data Base Management System 56K to 248K bytes of memory. A 56K system supports 2-4 users. At least 64K bytes are needed to support 6 users. Supports maximum of 65 timesharing users, 30-40 simultaneously (depending on processor) **Languages:**MUMPS included.				
		RSTS/E General-purpose Timesharing System 96K to 248K bytes of memory, or 256K to 4096K bytes on 11/70. Depending on disk and memory configuration, RSTS/E can support a maximum of 63 users. **Languages:** BASIC-PLUS included; COBOL, FORTRAN IV, DIBOL, APL, and MACRO are options.			
RSX-11S Execute-only Real-Time Multi-programming System 16K to 248K of memory. 8K (bytes) system allows 4K for user tasks. 16K bytes required for on-line task loading or support for tasks written in FORTRAN. **Languages:** Program development on host RSX-11M or IAS system.					
	RSX-11M Small-to-Moderate-sized Real-Time Multi-programming System 32K to 248K bytes of memory or 256K to 4096K bytes on 11/70. 16K (bytes) system allows up to 8K for user tasks; includes a subset of MACRO. At least 48K bytes are required for full MACRO support, concurrent program development and application tasks execution, or memory management support. Error logging supported. **Languages:** MACRO included; FORTRAN IV and FORTRAN IV-PLUS and BASIC are options.				
			IAS Multi-purpose Multi-programming System 128K to 248 K bytes of memory or 256K to 4096K bytes on 11/70. Timeshared interactive and batch job processing with concurrent real-time applications execution. Depending on disk and memory configuration, as many as 10 interactive users can be supported on an 11/60; as many as 20 interactive users on an 11/70. Error logging supported. **Languages:** MACRO included; FORTRAN IV, FORTRAN IV-PLUS, COBOL, and BASIC are options.	

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 paper tape readers 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, to provide the complete answer to customer needs in all product line areas — business, education, industry, laboratory, and medicine.

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 applications 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, bootstrap loader, clock, expander boxes, cabinets, and all required cables. 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 Package Systems program. For example, all the operating systems listed are available as a package system with the PDP-11/60 processor.

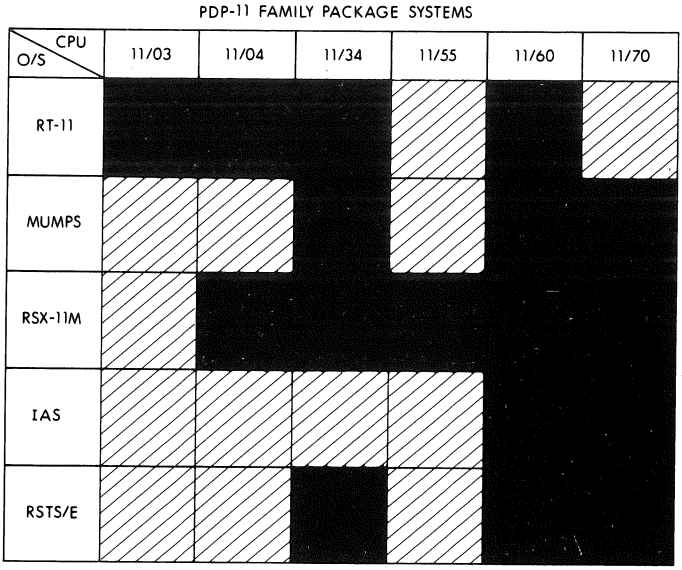


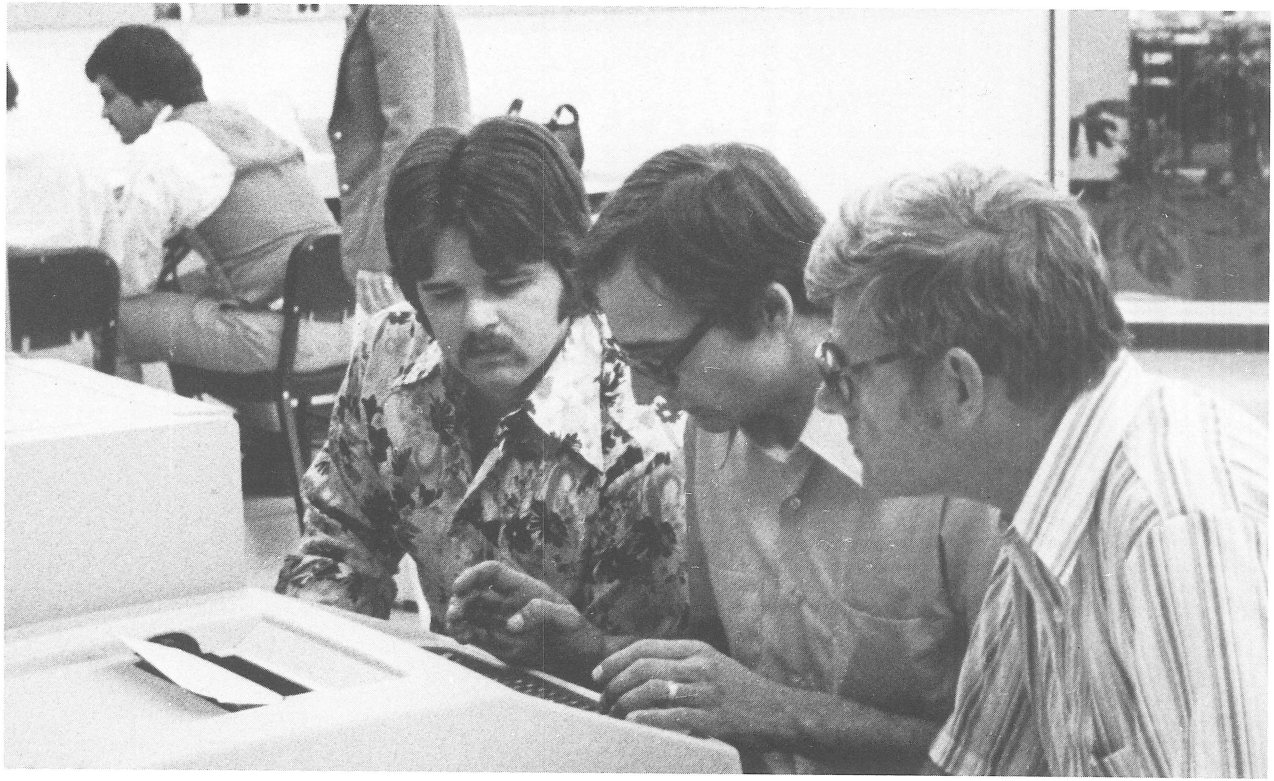
Figure 1-3 Package Systems

DOCUMENTATION

DIGITAL offers several levels of technical 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 like microprogramming are also well-covered in commercially available books. If you have a specific documentation need, discuss the issue with a DIGITAL salesperson, 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.



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, 34, 45, 55, and 60 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 4K words 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 five 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 closest to the processor on the bus takes precedence over those further away.

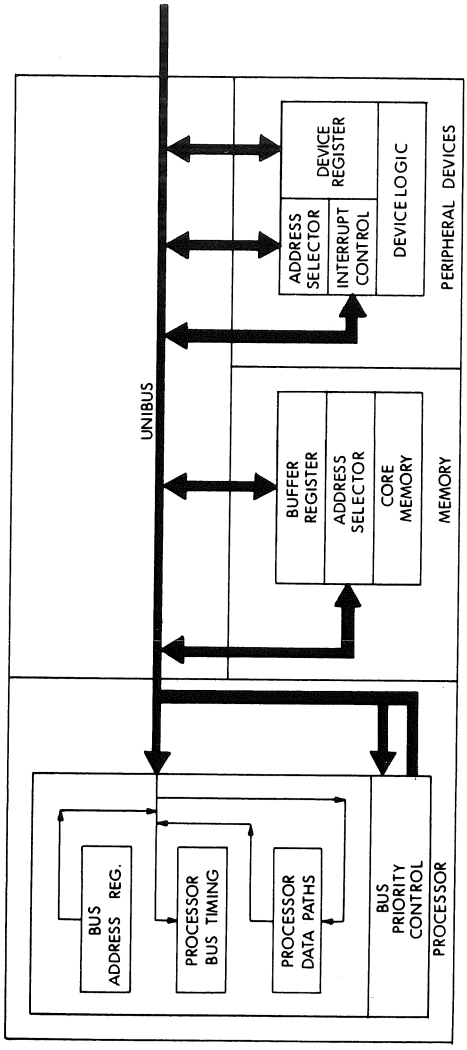


Figure 2-1 UNIBUS

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. 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 have 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 typical transfer rate on the UNIBUS is one 16-bit word every 400 ns, or about 2.5 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 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).

A NPR is issued when a device wishes to perform a data transaction. A NPR does not use the CPU; 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.

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 transaction.

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 which contains the starting address of an interrupt service routine. ("I need to interrupt.") The CPU accepts the interrupt vector and asserts SSYN (Slave YNc) 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 word.

3. The new PC and PS (interrupt vector) are taken from the specified address. The old PS and PC are then pushed onto the current stack. The service routine is then entered.

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.

Interrupt Servicing

Every hardware device capable of interrupting the processor has a unique pair of locations (2 words) 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, 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) 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 closest to the CPU has the highest priority.

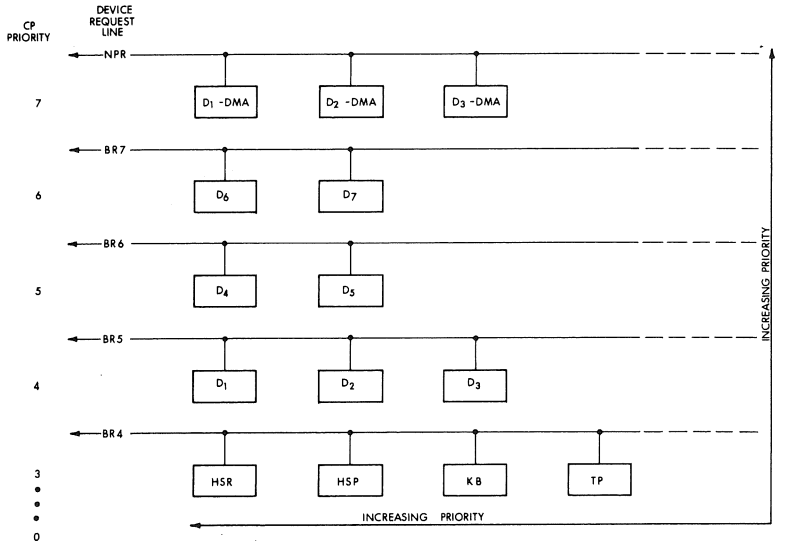


Figure 2-2 Priority Control

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 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 is 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 priori-

ties 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 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 *or* byte is transferred *into* the master.
- 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). Must be 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.

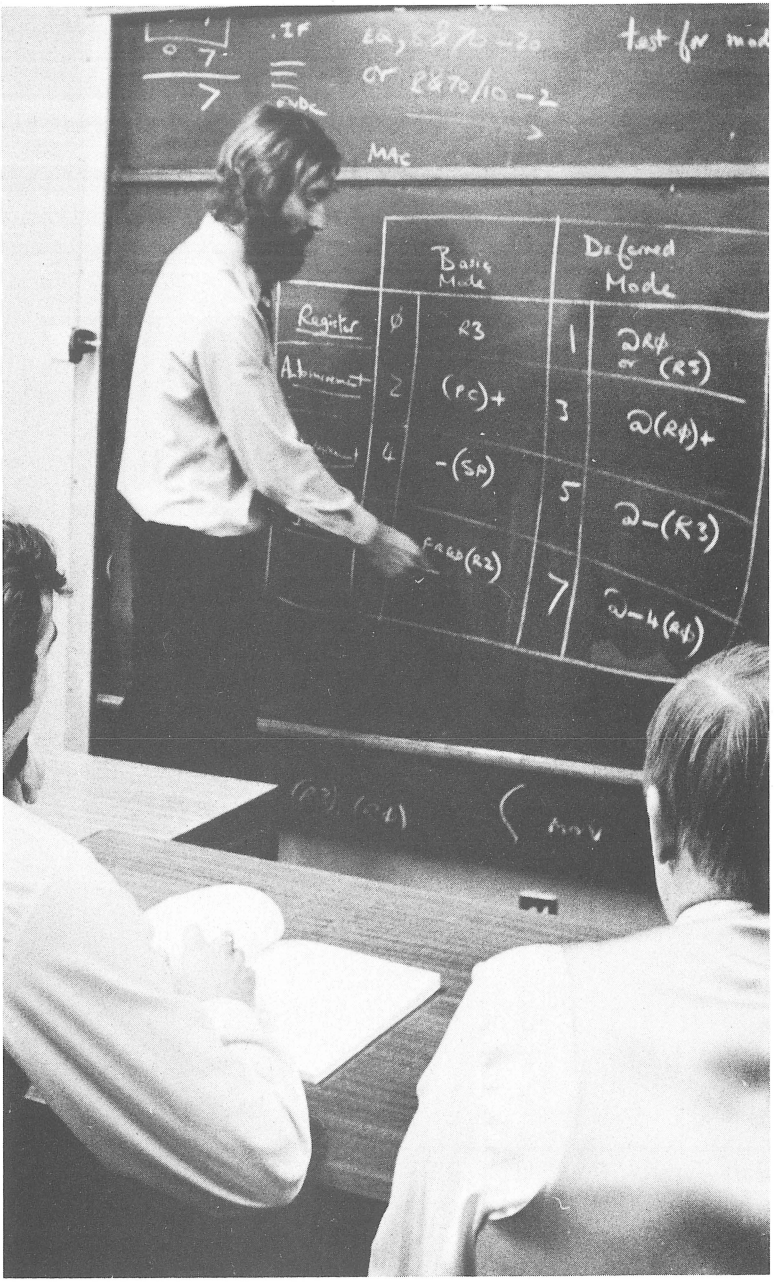
Data transactions between a master and slave device are synchronized by master sync (MSYN) and slave signals. Below is an example of how these signals are used during a typical DATI transaction:

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 2-1 Bus Control

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


Basic Mode
Deferred Mode
Register
Addressing
0
2
4
R3
(PC)+
-(SP)
1
3
5
7
(R3)
@(R4)+
@-(R3)
MOV

CHAPTER 3

ADDRESSING MODES

In the PDP-11, 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 and one of several addressing modes. Each PDP-11 memory reference instruction specifies the:

- function to be performed (operation code)
- general purpose register to be used when locating the source and/or destination operand
- 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 in the combination of addressing modes with the instruction set that the PDP-11 provides a unique number of capabilities. The PDP-11 is designed to handle structured data both efficiently and with flexiblity. 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, allowing automatic stepping through memory locations.
- 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.

Utilization of the 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 in the PDP-11 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 PDP-11 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 utilize the following format.

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

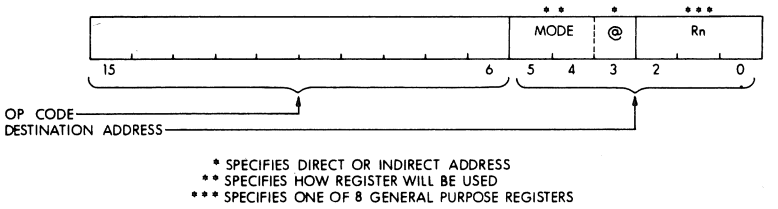


Figure 3-1 Single Operand Instruction Format

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

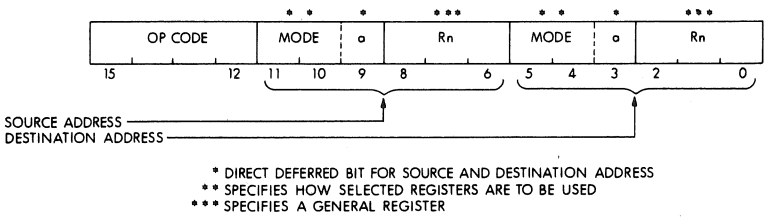


Figure 3-2 Double Operand Instruction Format

Bits 3-5 specify the binary code of the addressing mode chosen.

The four direct addressing modes are:

- register
- autoincrement
- autodecrement
- index

When bit 3 of the instruction is set, indirect addressing is specified and the four basic modes become deferred modes. In a register deferred mode, the content of the selected register is taken as the address of the operand. In the other deferred modes, the content of the register specifies the address of the operand, rather than the operand itself. 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 PDP-11 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. Assembler syntax requires that a general register be defined as follows:

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

% sign indicates register definition.

Register Mode Examples

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

Represented as:

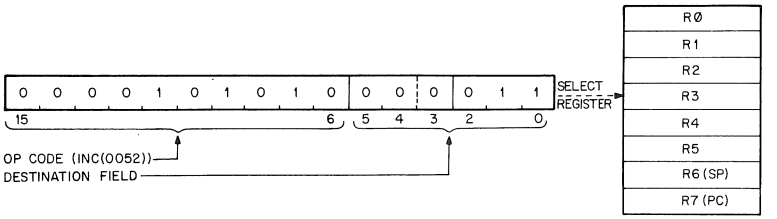
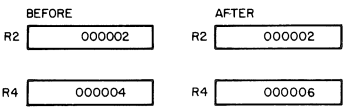


Figure 3-3 Register Mode Example

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)**

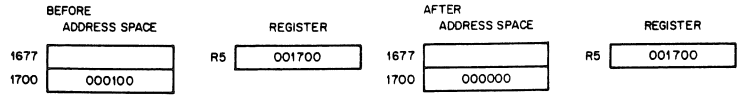
In register deferred mode, the address of the operand is stored in a general purpose register. The address contained in the general purpose register directs the CPU to the operand. The operand is located outside the CPU, either in memory, or in an I/O register.

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:



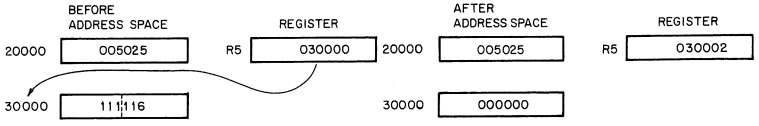
AUTOINCREMENT MODE MODE 2 (Rn)+

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 autoincremented each time the instruction is executed. It is autoincremented by 1 if you are using byte instructions, by 2 if you are using word instructions.

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:



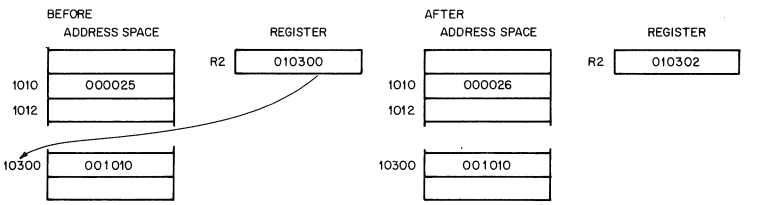
AUTOINCREMENT DEFERRED MODE MODE 3 @(Rn)+

In autoincrement deferred mode, the register contains a pointer to an address. The "+" indicates that the pointer in R2 is incremented by 2 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 volumes, 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:



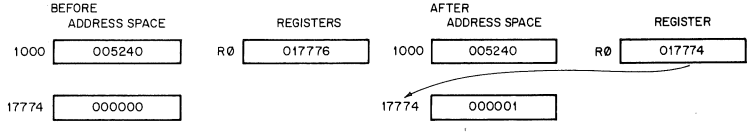
AUTODECREMENT MODE MODE 4 -(Rn)

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 autodecremented by 1 for bytes, by 2 for words.

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:



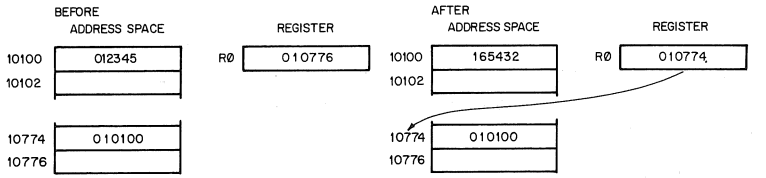
AUTODECREMENT DEFERRED MODE MODE 5 @-(Rn)

In autodecrement deferred mode, the register contains a pointer. The pointer is first decremented by 2, 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:



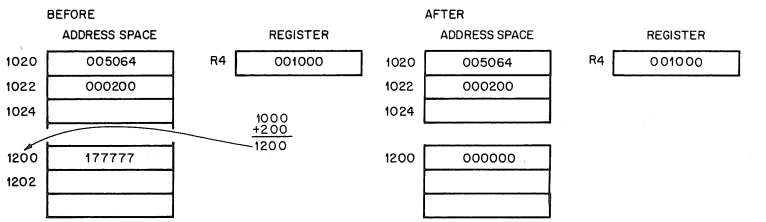
INDEX MODE MODE 6 ± X(Rn)

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 002000	The address of the operand is determined by adding 200 to the contents of R4. The location is then cleared.

Represented as:



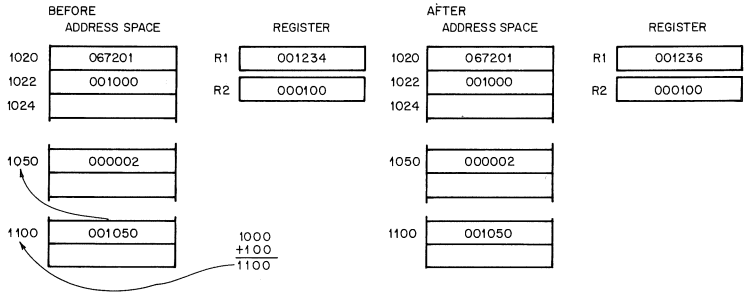
INDEX DEFERRED MODE MODE 7 @X(Rn)

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 a PDP-11. When the CPU uses the PC to access a word from memory, the PC is automatically incremented by 2 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 2.

The PC can be used with all the PDP-11 addressing modes. There are four modes in which the PC can provide advantages for handling position-independent code (see Chapter 5) and for handling unstructured data. These modes refer to the PC and are termed immediate, absolute (or immediate deferred), relative, and relative deferred.

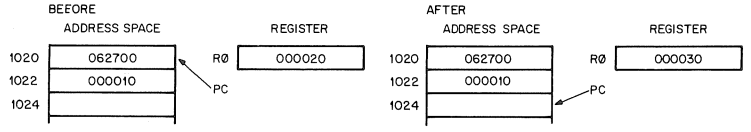
PC IMMEDIATE MODE **MODE 2** **#n**

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:



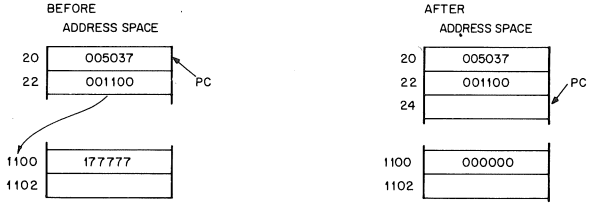
PC ABSOLUTE MODE MODE 3 @#A

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.

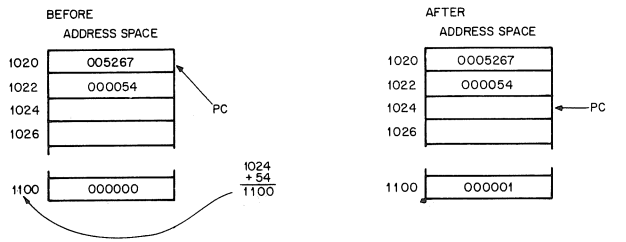
With the relative addressing mode, the address of the operand is always determined with respect to the updated PC. Therefore, when the instruction is relocated, the operand remains the same relative distance away.

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 (see Chapter 5).

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:

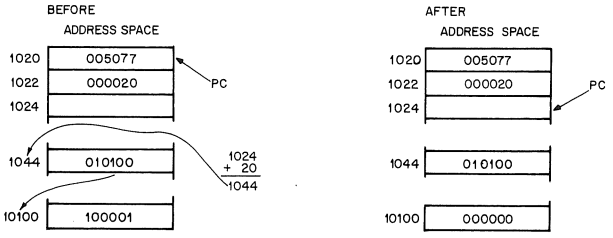


Table 3-1 summarizes the four basic modes used with direct addressing:

Table 3-1 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.

Binary Code	Mode	Name	Symbolic	Function
100	4	Autodecrement	−(Rn)	Register is decremented and then used as a pointer to sequential data.
110	6	Index	X(Rn)	Value X is added to (Rn) to produce address of operand. Neither X nor (Rn) is modified.

Table 3-2 summarizes the same four basic modes used with indirect addressing.

Table 3-2 Indirect Addressing Modes

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

Binary Code	Mode	Name	Symbolic	Function
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 (located in a word contained in the instruction) 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 inmmediate, absolute (or immediate deferred), relative, and relative deferred, and are summarized in Table 3-3.

Table 3-3 PC Register Addressing Modes

Binary Code	Mode	Name	Symbolic	Function
010	2	Immediate	#n	Operand is contained in the instruction.

Binary Code	Mode	Name	Symbolic	Function
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.

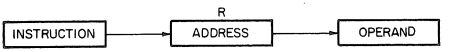
GRAPHIC SUMMARY OF PDP-11 ADDRESSING MODES

General Register Addressing Modes

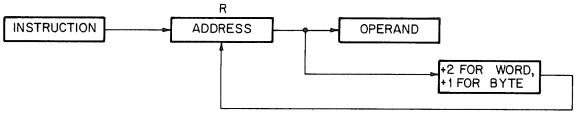
R is a general register, 0 to 7.
(R) is the contents of that register.

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

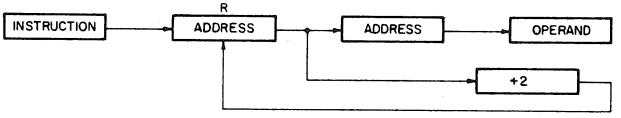
Mode 1 **Register deferred** OPR (R) R contains address.



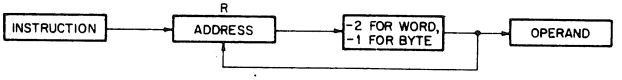
Mode 2 **Autoincrement** OPR (R)+ R contains address, then increment (R).



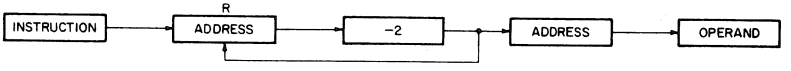
Mode 3 **Autoincrement deferred** OPR @(R)+ R contains address of address, then increment (R) by 2.



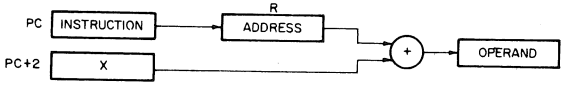
Mode 4 **Autodecrement** OPR −(R) Decrement (R), then R contains address.



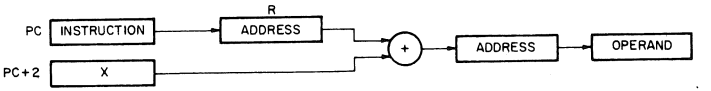
Mode 5 **Autodecrement deferred** OPR @−(R) Decrement (R) by 2, then R contains address of address.



Mode 6 **Index** OPR X(R) (R)+X is address, second word of instruction.



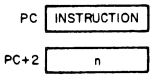
Mode 7 **Index deferred** OPR @X(R) (R)+X is address (second word) of address.



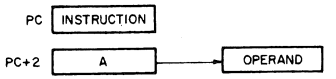
Program Counter Addressing Modes

Register = 7

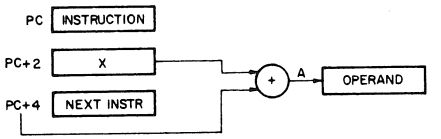
Mode 2 **Immediate** OPR #n Literal operand n is contained in the instruction.



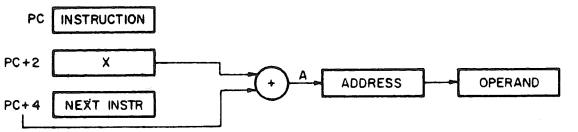
Mode 3 **Absolute** OPR @#A Address A is contained in the instruction.

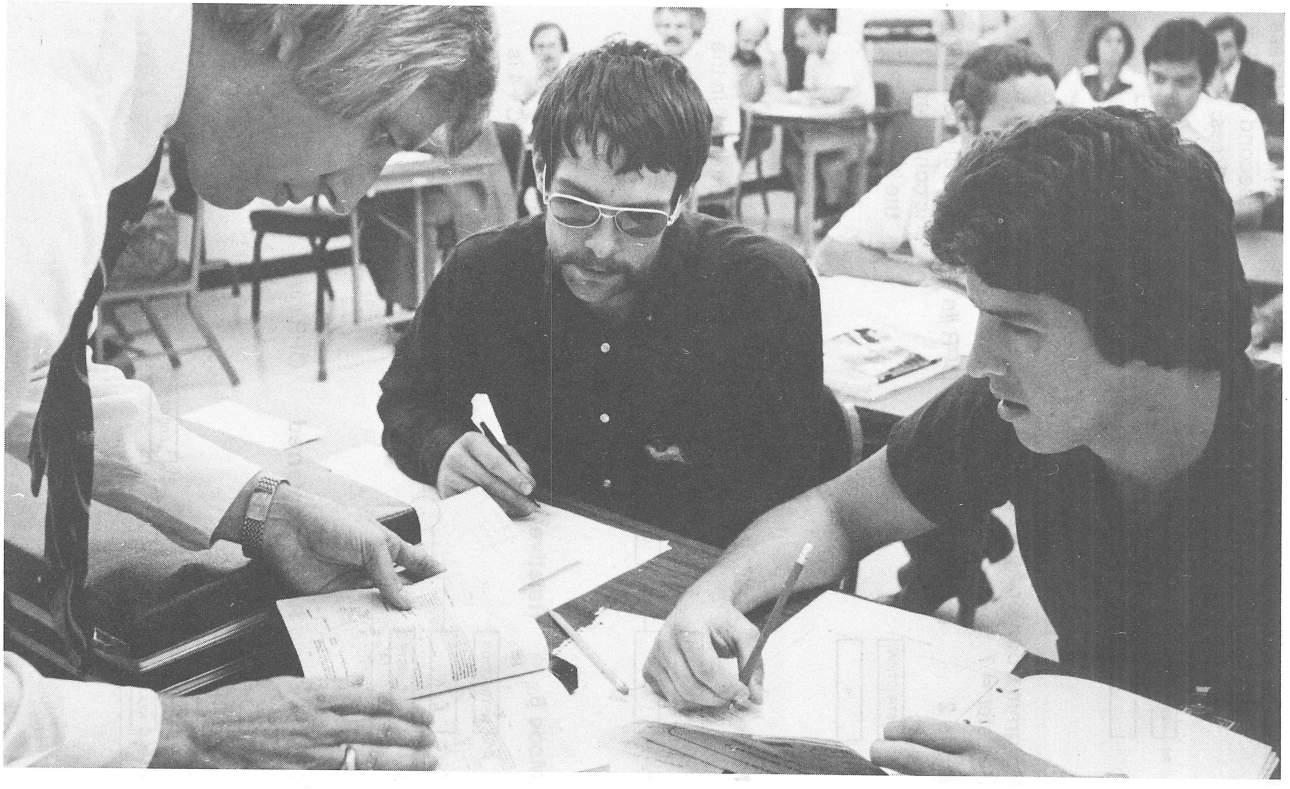


Mode 6 **Relative** OPR A PC+4 + X is address. PC+4 is updated PC.



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:

```
CLR A,C
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. All PDP-11 instructions fall into one of three 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.
- *Program Control* — 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.

SINGLE OPERAND INSTRUCTIONS

	Mnemonic	Instruction
General		
	CLR(B)	clear destination
	COM(B)	1's complement dst
	INC(B)	increment dst
	DEC(B)	decrement dst
	NEG(B)	2's complement negate dst
	TST(B)	test dst

	Mnemonic	Instruction
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
	MFPS	move byte from processor status
	MTPS	Move byte to processor status

Instruction Format

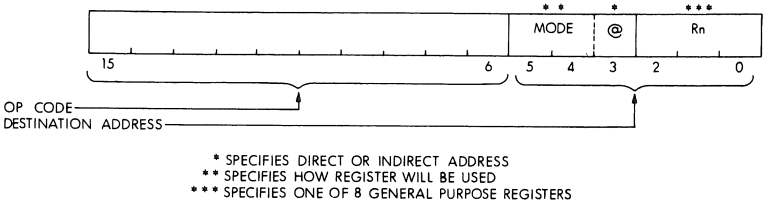


Figure 4-1 Single Operand Instruction Format

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 src to dst
	SUB	subtract src from dst
	ASH	shift arithmetically
	ASHC	arithmetic shift combined

	Mnemonic	Instruction
Logical		
	BIT(B)	bit test
	BIC(B)	bit clear
	BIS(B)	bit set
	XOR	exclusive OR

Instruction Format

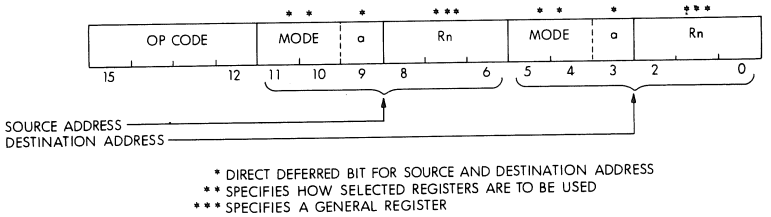


Figure 4-2 Double Operand Instruction Format

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. 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.

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.

PROGRAM CONTROL INSTRUCTIONS

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

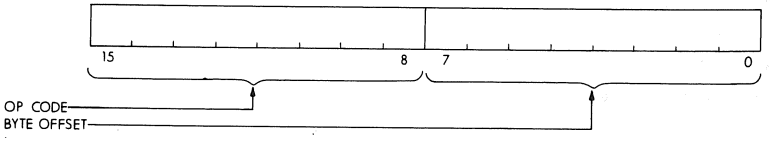


Figure 4-3 Branch Instruction Format

- The high byte (bits 8-15) of the instruction is an op code specifying the conditions to be listed.
- The low byte (bits 0-7) of the instruction is the offset value in words that determines the new program location if the branch is taken.

JUMP AND SUBROUTINE INSTRUCTIONS

	Mnemonic	Instruction
Jump & Subroutine		
	JMP	jump
	JSR	jump to subroutine
	RTS	return from subroutine

Instruction Format

JSR Format

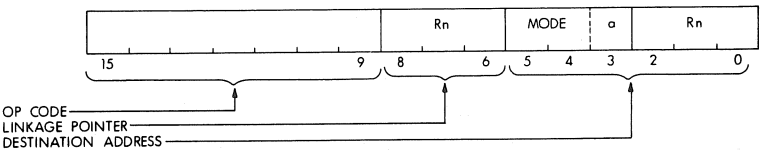


Figure 4-4 JSR Instruction Format

- Bits 9-15 are always octal 004 indicating the op code for JSR.
- Bits 6-8 specify the link register. Any general purpose register may be used in the link, except R6.
- Bits 0-5 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

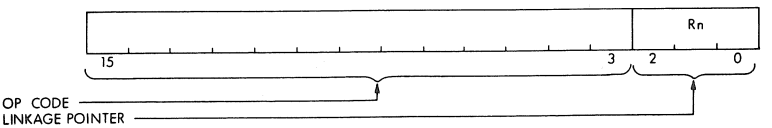


Figure 4-5 RTS Instruction Format

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

- Bits 3-15 always contain octal 00020, which is the op code for RTS.
- Bits 0-2 specify any one of the general purpose registers.

- The register specified by bits 0-2 must be the same register used as the link between the JSR causing the jump and the RTS returning control.

Interrupts and Traps

Mnemonic	Instruction
EMT	emulator trap
TRAP	trap
BPT	breakpoint trap
IOT	input/output trap
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

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 codition 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 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. When the instruction results in a carry out of the most significant bit of the result, the carry itself is usually moved into the C bit. 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.

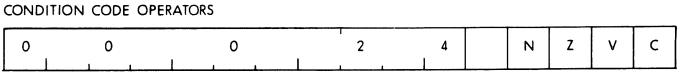


Figure 4-6 Condition Code Operators' Format

Instruction Format

The format of the condition code operators is as follows:

- Bits 15-5 — the "op" code
- 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 "select" 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_8byte locations beginning at memory address 600.

(R0) = 600
(R1) = 30

```
LOOP:   CLRB(R0)+
        DEC R1
        BNE R1
        LOOP
        HALT
```

Program Description

- The CLRB (R0)+ instruction clears the content of the location specified by R0 and increments R1.
- R0 is the pointer.
- Because the auto-increment 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 start to clear another location. If the counter is zero, indicating done, then the program executes the next instruction, HALT.

Double Operand Instruction Example

This routine prints out a portion of a payroll program for review by the supervisor. It is known that 76 locations are to be printed and the locations start at address 600.

```
INIT:   MOV #600, R0
        MOV #76, R1
START:  MOVB (R0)+, I/O
        DEC R1
        BNE START
        HALT
```

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 sets up R1 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 (I/O) for printing. The data comes from the location specified by R0. The pointer R0 is then incremented to point to the next sequential location.

- The counter (R1) 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 (R1) reaches zero, indicating all data has been transferred, the branch does not occur and the program executes the next instruction, HALT.

Branch Instruction Example

NOTE: Branch instructions are limited from $+177_8$ 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. Unforunately, 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 #400, R1

START:    CMPB(R0)+,#3

          BHI CONT

STACK:    MOVB (R0),-(R1)
```

```
CONT:   INC R0
        CMP #1264, R0
        BHIS START
        HALT
```

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 (4264), 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

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) is 077777 and C = 1 C: set if (dst) is 177777 and C = 1	Adds the contents of the C bit into the destination. This permits the carry from the addition of the low order words/bytes to be carried into the high order result.
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)	N: set if result < 0 Z: set if result = 0 V: set if sign of register changed during shift C: loaded from last bit shift out of register	The contents of the register are shifted right or left the number of times specified by 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	R,Rv1 ← R,Rv1 The double word is shifted NN places to the right or left, where NN = (src)	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 right shift (loaded with the last bit shifted out of the 32-bit operand)	The contents of the register and the register OR-ed with one are treated as one 32-bit word. Rv1 (bits 0-15) and R (bits 16-31) are shifted right or left the number of times specified by the shift count. 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. When the register chosen is an odd number, the register and the register OR-ed with one are the same. In this case, the right shift becomes a rotate. The 16-bit word is rotated right the number of bits specified by the shift count.

ASL ASLB Arithmetic Shift Left	SO SO	0063DD 1063DD	(dst) ← (dst) shifted one place to the left	N: set if high order bit of the 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	Shifts all bits of the destination left one place. The low order bit is 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.
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 order bit of the destination	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. **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	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.

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

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
BCS Branch if carry set	PC	103400	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 Branch if equal	PC	001400	PC ← PC + (2 × offset) if Z = 1	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of the Z bit and 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	PC ← PC + (2 × offset) if NvV = 0	N: unaffected Z: unaffected V: unaffected C: unaffected	Causes a branch if N and V are 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	PC ← PC + (2 × offset) if $\bar{Z} \vee (N \oplus V) = 0$	N: unaffected Z: unaffected V: unaffected C: unaffected	Causes a branch if the exclusive OR of the N and V bits is 1. Thus, BGT always branches following an operation that added two negative numbers, even if overflow occurred. In particular, BGT always causes a branch if it follows a CMP instruction operating on a negative source and a positive destination (even if overflow occurred). Further, BGT never 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	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.
BHIS Branch if higher than the same	PC	103000	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.

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

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
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 destination. The contents of the destination are lost.
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	PC ← PC + (2 × offset) if Zv(N⊻V) = 1	N: unaffected Z: unaffected V: unaffected C: unaffected	Causes a branch if the exclusive OR of the N and V bits is 1. 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 if follows a CMP instruction operating on a positive source and negative destination. BLE does not cause a branch if the result of the previous operation was 0 (without overflow).
BLO Branch if lower	PC	103400	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	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. Comparison of unsigned values with the CMP instruction to be tested for "higher or same" and "higher" by a simple test of the C bit.

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

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
BLT Branch if less than	PC	002400	PC ← PC + (2 × offset) if NvV = 1	N: unaffected Z: unaffected V: unaffected C: unaffected	Causes a branch if the exclusive 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	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.

BNE Branch if not equal	PC	001000	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	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 vector C: loaded from trap vector	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 these debugging aids. No information is transmitted in the low byte.
BR Branch	PC	000400	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	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.

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

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
BVS Branch if V bit set	PC	102400	PC ← PC + (2 × offset) if V = 1	N: unaffected Z: unaffected V: unaffected C: unaffected	Tests the state of V bit (overflow) and causes a branch if the V bit is set. BVS is used to detect arithmetic overflow in the previous operation.
CCC Clear all condition code bits	CC	000257	Sets and clears condition code bits. Selectable combinations of these bits may be cleared or set together. Condition code bits corresponding to bits in the condition code operator (bits 0-3) are modified according to the sense of bit 4, the set/clear bit of the operator; i.e., sets the bit specified by bit 0, 1, 2, or 3, if bit 4 is a 1. Clears corresponding bits if bit 4 = 0.		
CLC Clear C	CC	000241	Sets and clears condition code bits. Selectable combinations of these bits may be cleared or set together. Condition code bits corresponding to bits in the condition code operator (bits 0-3) are modified according to the sense of bit 4, the set/clear bit of the operator; i.e., sets the bit specified by bit 0, 1, 2, or 3, if bit 4 is a 1. Clears corresponding bits if bit 4 = 0.		
CLN Clear N	CC	000250	Sets and clears condition code bits. Selectable combinations of these bits may be cleared or set together. Condition code bits corresponding to bits in the condition code operator (bits 0-3) are modified according to the sense of bit 4, the set/clear bit of the operator; i.e., sets the bit specified by bit 0, 1, 2, or 3, if bit 4 is a 1. Clears corresponding bits if bit 4 = 0.		

CLR CLRB Clear	SO	0050DD 1050DD	(dst) ← 0	N: cleared Z: set V: cleared C: cleared	Contents of specified destination are replaced with zeros.
CLV Clear V	CC	000242	Sets and clears condition code bits. Selectable combinations of these bits may be cleared or set together. Condition code bits corresponding to bits in the condition code operator (bits 0-3) are modified according to the sense of bit 4, the set/clear bit of the operator; i.e., sets the bit specified by bit 0, 1, 2, or 3, if bit 4 is a 1. Clears corresponding bits if bit 4 = 0.		
CLZ Clear Z	CC	000244	Sets and clears condition code bits. Selectable combinations of these bits may be cleared or set together. Condition code bits corresponding to bits in the condition code operator (bits 0-3) are modified according to the sense of bit 4, the set/clear bit of the operator; i.e., sets the bit specified by bit 0, 1, 2, or 3, if bit 4 is a 1. Clears corresponding bits if bit 4 = 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: cleared if there is a carry from the most significant bit of the result	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.

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

Mnemonic/ Instruction	Type	OP Code	Operation	Condition Codes	Description
COM COMB Complement	SO	0051DD 1051DD	(dst) ← n (dst)	N: set if most significant bit of result = 0 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).
DEC DECB Decrement	SO	0053DD 1053DD	(dst) ← (dst) − 1	N: set if result < 0 Z: set if result = 0 V: set if (dst) was 100000 C: not affected	Subtracts 1 from the contents of the destination.
DIV Divide	DO	071RSS	R,Rv1 ← R,Rv1/(src)	N: set if quotient < 0 Z: set if quotient = 0 V: set if source = 0 or if the absolute value of the register is larger than the absolute value of the source. (In this case the instruction is aborted because the quotient would exceed 15 bits.) C: set if divide 0 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 is of the same sign as the dividend. R must be even.

EMT Emulator Trap	PC	104000	−(SP) ← PS −(SP) ← PC PC ← (30) PS ← (32)	N: loaded from trap vector Z: loaded from trap vector V: loaded from trap vector C: loaded from trap vector	All operation codes from 104000 to 104377 are EMT instructions and may be used to transmit information 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 (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.
FADD Floating Add	FP	07500R	[(R)+4,(R)+6] ← [(R)+4,(R)+6] + [(R),(R)+2], if result ≥ 2^{-128}; else [(R)+4,(R)+6] ← 0	N: set if result < 0 Z: set if result = 0 V: cleared C: cleared	Adds the A argument to the B argument and stores the result in the A argument position on the stack. General register R is used as the stack pointer for the operation. A ← A + B Used on 11/03.
FDIV Floating Divide	FP	07503R	[(R)+4,(R)+6] ← [(R)+4,(R)+6]/ [(R),(R)+2] if result ≥ 2^{-128}; else [(R)+4,(R)+6] ← 0	N: set if result < 0 Z: set if result = 0 V: cleared C: cleared	Divides the A argument by the B argument and stores the result in the A argument position on the stack. If the divisor (B argument) is equal to zero, the stack is left untouched. A ← A/B Used on 11/03.

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

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
FMUL Floating Multiply	FP	07502R	[(R)+4,(R)+6] ← [(R)+4,(R)+6] × [(R),(R)+2] if result ≥ 2^{-128}; else [(R)+4,(R)+6] ← 0	N: set if result < 0 Z: set if result = 0 V: cleared C: cleared	Multiplies the A argument by the B argument and stores the result in the A argument position on the stack. A ← A × B Used on 11/03.
FSUB Floating Subtract	FP	07501R	[(R)+4,(R)+6] ← [(R)+4,(R)+6] − [(R),(R)+2]if result ≥ 2^{-128}; else [(R)+4,(R)+6] ← 0	N: set if result < 0 Z: set if result = 0 V: cleared C: cleared	Subtracts the B argument from the A argument and stores the result in the A argument position on the stack. A ← A − B Used on 11/03.

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 address of the HALT instruction plus 2. Transfers on the UNIBUS are terminated immediately. The PC points to the next instruction to be executed. Pressing the continue 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 077777 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 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 location following JSR; this address now put in reg PC ← (tmp) PC now points to subroutine 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 processor 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 instructions in each routine to save and restore the linkage pointer. Further, since all linkages are saved in a re-entrant manner on the processor 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. JSR, PC saves the use of an extra register. In both JSR and JMP the address is used to load the program counter, R7. Thus, for example, a JSR in destination mode 1 for general register R1 (where (R1) = 100) will access a subroutine at location 100. This is effectively one level less of deferral than operate instructions such as add. In the PDP-11/60, a JSR mode 0 will result in an illegal instruction and a trap through the trap vector address 4.

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

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
LDUB Load Micro-break 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 exception 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 purposes through the EXAM function. Instruction execution in user mode will result in a trap to 10. The instruction also allows, through the write code, an alteration of registers. **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. RO, 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.

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

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description

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

MED CODE	REGISTER AND FUNCTION
XXX00X	LOW HALF ASP LOW (READ)
XXX01X	HIGH HALF ASP LOW (READ)
XXX02X	LOW HALF ASP HIGH (READ)
XXX03X	HIGH HALF ASP HIGH (READ)
XXX04X	LOW HALF BSP LOW (READ)
XXX05X	HIGH HALF BSP LOW (READ)
XXX06X	LOW HALF BSP HIGH (READ)
XXX07X	HIGH HALF BSP HIGH (READ)
XXX100	CSP(0) (READ)
XXX101	CSP(1) (READ)
XXX102	CSP(2) (READ)
XXX103	CSP(3) (READ)
XXX104	CSP(4) (READ)
XXX105	CSP(5) (READ)
XXX106	CSP(6) (READ)
XXX107	CSP(7) (READ)
XXX110	CSP(10) (READ)
XXX111	CSP(11) (READ)
XXX112	CSP(12) (READ)
XXX113	CSP(13) (READ)
XXX114	CSP(14) (READ)

MED CODE	REGISTER AND FUNCTION
XXX154	CACHE INVALIDATE
XXX155	READ CACHE TAG
XXX20X	LOW HALF ASP LOW (WRITE)
XXX21X	HIGH HALF ASP LOW (WRITE)
XXX22X	LOW HALP ASP HIGH (WRITE)
XXX23X	HIGH HALF ASP HIGH (WRITE)
XXX24X	LOW HALF BSP LOW (WRITE)
XXX25X	HIGH HALF BSP LOW (WRITE)
XXX26X	LOW HALF BSP HIGH (WRITE)
XXX27	HIGH HALF BSP HIGH (WRITE)
XXX300	CSP(0) (WRITE)
XXX301	CSP(1) (WRITE)
XXX302	CSP(2) (WRITE)
XXX303	CSP(3) (WRITE)
XXX304	CSP(4) (WRITE)
XXX305	CSP(5) (WRITE)
XXX306	CSP(6) (WRITE)
XXX307	CSP(7) (WRITE)
XXX310	CSP(10) (WRITE)
XXX311	CSP(11) (WRITE)
XXX312	CSP(12) (WRITE)

				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)

				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 data space MFPI Move from previous instruction space	MS	0065SS 0065SS	(temp) ← (src) −(SP) ← (temp)	N: set if the source < 0 Z: set if the source = 0 V: cleared C: unaffected	Pushes a word onto the current stack from an address in previous space. The source address is computed using the current registers and memory map. MFPI: 11/60; MFPD: 11/45/55

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

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
MFPS	MS	1067DD	(dst) ← PS dst lower 8 bits	N: set if PSS 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 upper byte of the register. The destination operand is treated as a byte address. 11/34 only.
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).		
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 resistor (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	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		

Partial 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) <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	1066SS 0066SS	(temp) ← (SP)+ (dst) ← (temp)	N: set if the source < 0 Z: set if the source = 0 V: cleared C: unaffected	This instruction pops a word off the current stack determined by PS (bits 15,14) and stores that word into an address in previous space PS (bits 13,12). The destination address is computed using the current registers and memory map. MTPI: 11/45/55; MTPD: Interpreted as MTPI in 11/60.
MTPS	MS	1064SS	PS ← (src)	N: set according to effective src operand 0-3 Z: same V: same C: same	The 8 bits of the effective operand replace the current contents of the PS. 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. 11/34 only.

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

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
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 that or equal to 2^{-1}	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.)
NEG NEGB Negate	SO	0054DD 1054DD	(dst) ← (dst)	N: set if result < 0 Z: set if result = 0 V: set if result = 100000 C: cleared if result = 0	Replaces the contents of the destination address by its 2's complement. Note that 100000 is replaced by itself.
RESET	MS	000005	PC (SP) PS (SP)	N: unaffected Z: unaffected V: unaffected C: unaffected	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 word = 0 V: loaded with the exclusive OR of the N bit and C bit (as set by the completion of the rotate operation) C: loaded with the high order bit of the destination	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.
ROR RORB Rotate Right	SO	0060DD	(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: loaded with the low order bit of the destination	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.
RTI	MS	000002	PC ← (SP)+ PS ← (SP)+	N: loaded from processor stack Z: loaded from processor stack V: loaded from processor stack C: loaded from processor stack	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.

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

Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
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 processor stack into the specified register. Return from a non-reentrant subroutine is typically made through the same 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	MS	000006	PC ← (SP)+ PS ← (SP)+	N: loaded from processor stack Z: loaded from processor stack V: loaded from processor stack C: loaded from processor 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 next "T" trap. In the case of the RTI instruction, the "T" trap will occur immediately after the RTI.

SBC SBCB Subtract Carry	SO	0056DD 1056DD	(dst) ← (dst) − C	N: set if result < 0 Z: set if result = 0 V: set if (dst) = 100000 C: cleared if (dst) = 0 and C = 1	Subtracts the contents of the C bit from the destination. This permits the carry from the subtraction of the low order words/bytes to be subtracted from the high order part of the result.
SCC Set all Cs	CC	000277	Sets and clears condition code bits. Selectable combinations of these bits may be cleared or set together. Condition code bits corresponding to bits in the condition code operator (bits 0-3) are modified according to the sense of bit 4, the set/clear bit of the operator; i.e., sets the bit specified by bit 0, 1, 2, or 3, if bit 4 is a 1. Clears corresponding bits if bit 4 = 0.		
SEC Set C	CC	000261	Sets and clears condition code bits. Selectable combinations of these bits may be cleared or set together. Condition code bits corresponding to bits in the condition code operator (bits 0-3) are modified according to the sense of bit 4, the set/clear bit of the operator; i.e., sets the bit specified by bit 0, 1, 2, or 3, if bit 4 is a 1. Clears corresponding bits if bit 4 = 0.		
SEN Set N	CC	000270	Sets and clears condition code bits. Selectable combinations of these bits may be cleared or set together. Condition code bits corresponding to bits in the condition code operator (bits 0-3) are modified according to the sense of bit 4, the set/clear bit of the operator; i.e., sets the bit specified by bit 0, 1, 2, or 3, if bit 4 is a 1. Clears corresponding bits if bit 4 = 0.		
SEV Set V	CC	000270	Sets and clears condition code bits. Selectable combinations of these bits may be cleared or set together. Condition code bits corresponding to bits in the condition code operator (bits 0-3) are modified according to the sense of bit 4, the set/clear bit of the operator; i.e., sets the bit specified by bit 0, 1, 2, or 3, if bit 4 is a 1. Clears corresponding bits if bit 4 = 0.		
SEZ Set Z	CC	000264	Sets and clears condition code bits. Selectable combinations of these bits may be cleared or set together. Condition code bits corresponding to bits in the condition code operator (bits 0-3) are modified according to the sense of bit 4, the set/clear bit of the operator; i.e., sets the bit specified by bit 0, 1, 2, or 3, if bit 4 is a 1. Clears corresponding bits if bit 4 = 0.		

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 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 can not be used to transfer control in the forward direction.
SPL Set priority level	PC	00023N	PS (bits 7-5) ←Priority (priority =n n 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. Assembler syntax is: SPL N **Note:** This instruction is a no op in user and supervisor modes (only in the 11/45); if used in 11/60, results in a processor trap through vector address 10.

SUB Subtract	DO	16SSDD	(dst) ← (dst) − (src) (src)	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: cleared if there is a carry from the most significant bit of the result	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.
SWAB Swap Byte	SO	0003DD	Byte 1/Byte 0 Byte 0/Byte 1	N: set if high order bit 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 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 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.

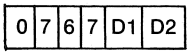
Mnemonic/ Instruction	Type	OPCode	Operation	Condition Codes	Description
TRAP	PC	10400 to 104777	−(SP) ← PS −(SP) ← PC PC ← (34) PS ← (36)	N: loaded from trap vector Z: loaded from trap vector V: loaded from trap vector C: loaded from trap vector	Operation codes from 104400 to 104777 are TRAP instructions. 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	(dst) ← (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	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 give an WAIT command, the processor will not compete for bus by fetching instructions or 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 follow-

ing 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
Extended
Function
Code

The UCS (User Control Store) option for the PDP-11/60 utilizes the XFC instruction. Details on use are contained in documentation associated with the USC 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 determiation by D2 field or additional macro instruction words. If the option is not enabled, a trap through vector address 10 occurs.

XOR	DO	074RDD	(dst) ← Rv(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 offers you a great deal of programming flexiblity and power. Utilizing the combination of the instruction set, the addressing modes, and the programming techniques makes it possible to 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. 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 virtual 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 use of virtual address space and in 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 virtual 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 you are familiar with the addressing modes described in Chapter 3.

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

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 the contents of the registers have been supplied independent of a particular virtual 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	relative mode
@A	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 independent. 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 excutes 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=R2SP
        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$:                                    ;
        MOV     R2,F.PF(R4)             ;SET AST ENTRY POINT
        MOV     PC,-(SP)                ;PUSH CURRENT LOCATION
        ADD     #BA-.,(SP)              ;COMPUTE ACTUAL LOCA-
TION
                                        ;OF AST
20$:
        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, 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. In the PDP-11, 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 in the PDP-11 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.

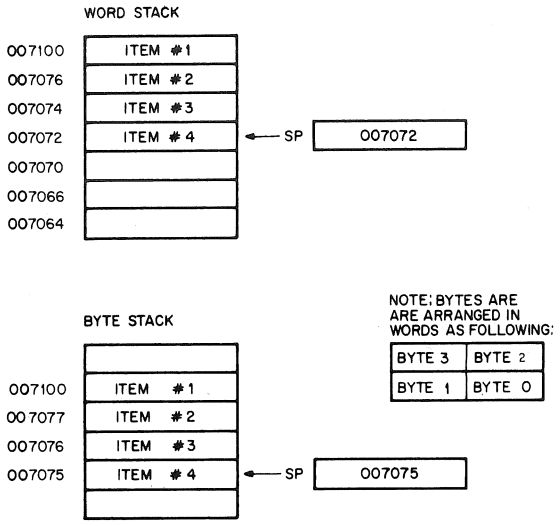


Figure 5-1 Word and Byte Stacks

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 sharable temporary storage locations.

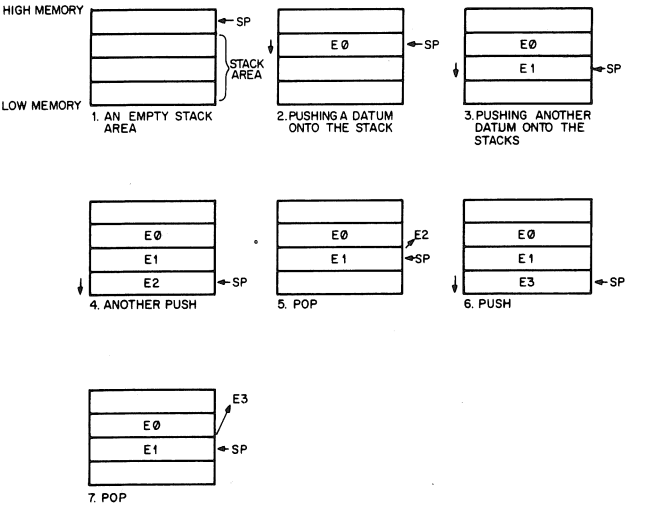


Figure 5-2 Illustration of Push and Pop Operations

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 PDP-11 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 which was 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.

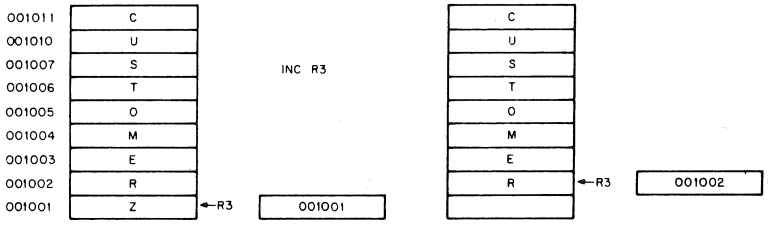


Figure 5-3 Byte Stack used as a Character Buffer

NOTE that 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:

INCSP 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.

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

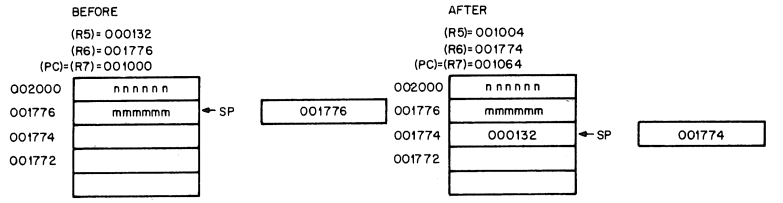


Figure 5-4 JSR

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, a RTS PC has the effect of returning to the address specified on the top of the stack.

PDP-11 Subroutine Advantages

There are several advantages to the PDP-11 subroutine calling procedure, affected 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 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."

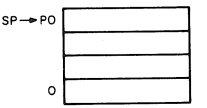
The first word contains the interrupt service routine entry address (the address of the service routine program sequence), and the second word contains the new PS which will determine the machine status, including the operational mode and register set to be used by the interrupt service routine. The 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.

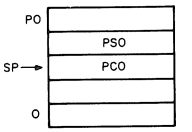
Nesting

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.

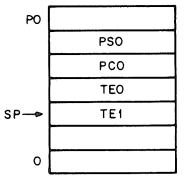
1. Process 0 is running; SP is pointing to location P0.



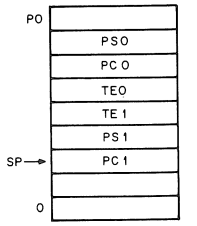
2. Interrupt stops process 0 with PC = PC0, and status = PS0; starts process 1.



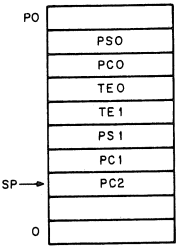
3. Process 1 uses stack for temporary storage (TE0, TE1).



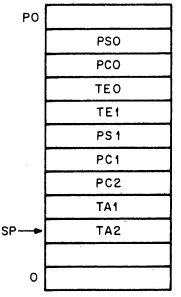
4. Process 1 interrupted with PC = PC1 and status = PS1; process 2 is started.



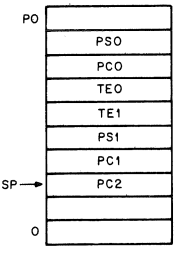
5. Process 2 is running and does a JSR R7,A to subroutine A with PC = PC2.



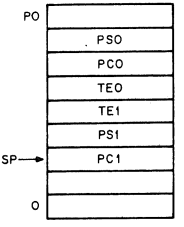
6. Subroutine A is running and uses stack for temporary storage.



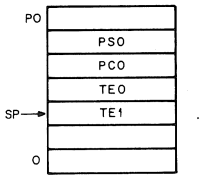
7. Subroutine A releases the temporary storage holding TA1 and TA2.



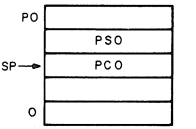
8. Subroutine A returns control to process 2 with an RTS R7; PC is reset to PC2.



9. Process 2 completes with an RTI instructions (dismisses interrupt) PC is reset to PC(1) 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.

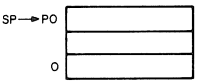


Figure 5-5 Nested Interrupt Service Routines and Subroutines

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.

PDP-11 Approach

Programs 1, 2, and 3 can share Subroutine A.

Conventional Approach

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:

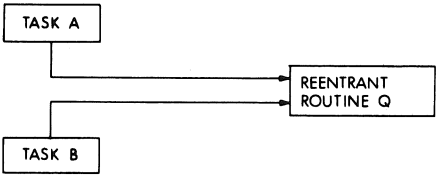


Figure 5-7 Sharing Control of a Routine

- 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 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 reeentrant 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, either 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 coroutine call.
.			
.			

				LOC:	
JSR PC,@(SP)+ (PC0)	PC0	←SP		. . .	When the call is executed, the PC from routine A is pushed on the stack and execution continues at LOC.
. . .	PC1	SP	JSR PC,@(SP)+ (PC1)	. . .	Routine B can 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,

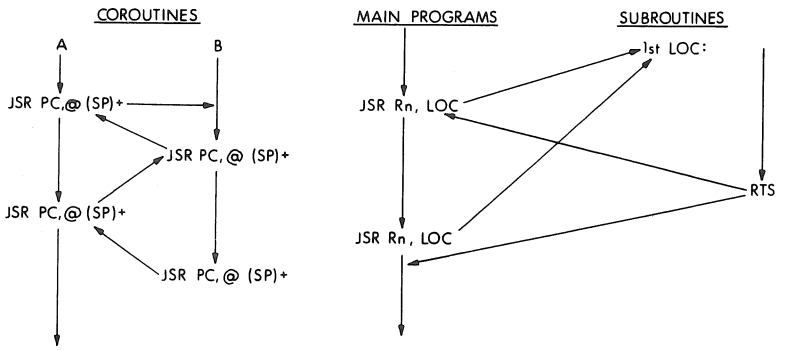


Figure 5-9 Coroutines vs. Subroutines

- 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.

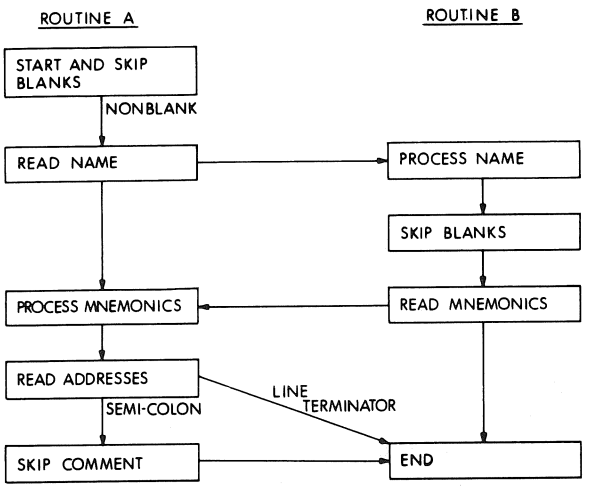


Figure 5-10 Coroutine Path

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 I1          concurrently
        Process I2
then
        Write 02
        Read I2          concurrently
        Process I1
```

Figure 5-11 illustrates a coroutine swapping interaction.

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.

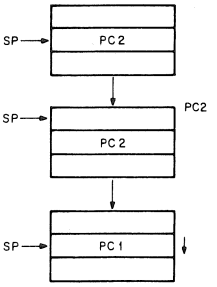


Figure 5-11 Coroutine Interaction

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:

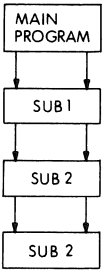


Figure 5-12 Recursive Routine Flow

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 exam-

ple, 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 condition peripherals for power fail.

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 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
```

2. 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 10^2=1.
 -100

   21   count = 2
 -100
```

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

```
   21
  -10

   11   count for 10^1=1.

   11
  -10

    1   count=2
  -10
```

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

No lower power, so remainder is 1.

Answer = 022, 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
```

PDP-11 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				

```
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
001004  000006
001006  000007
001010  000010

        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 #VALUE,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 COUN-
                                        ;COUNTER
NEXT:           CMP R1,R2               ;YES, FINISHED?
                BNE CHECK               ;NO, GO BACK
                HALT                    ;YES, STOP

VALUES:         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
```

```
R0=%0           ;PROGRAMMING EXAMPLE
R1=%1           ;ACCEPT (IMMEDIATE ECHO) AND
SP=%6           ;STORE 20. CHARS
CR=15           ;FROM THE KEYBOARD, OUTPUT CR & LF
LF=12           ;ECHO ENTIRE STRING FROM STORAGE
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)

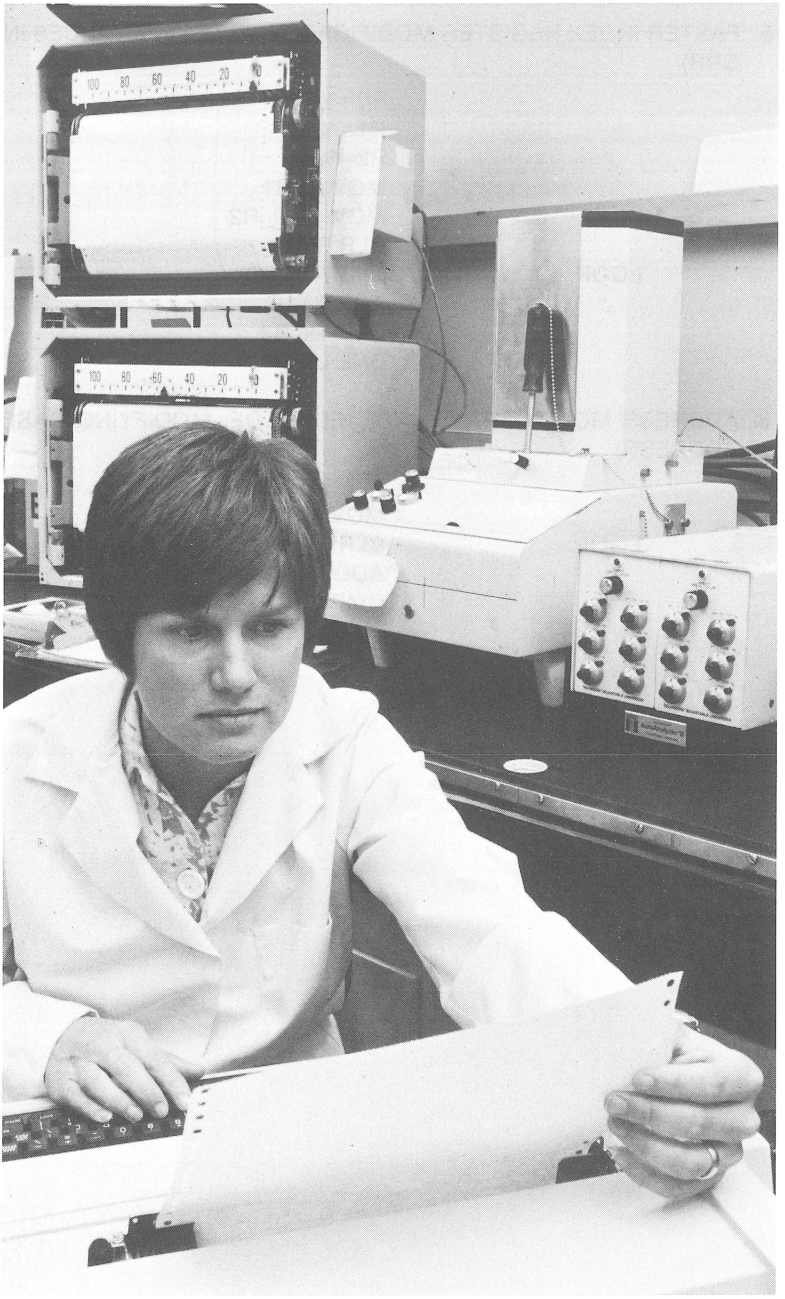
```
                R0=%0
                CLR R0
LOOP:           CLR TBL (R0)
                ADD #2,R0
                CMP R0,#100.
                BNE LOOP
```

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

PDP-11/04, PDP-11/34

PDP-11/04

The PDP-11/04 is the low-end member of the PDP-11 family of processors. It has most of the capabilities and features of the 11/34, and, except for the CPU circuit boards, is nearly the same, which is why the two processors are discussed together. The PDP-11/04 CPU is so compact that the entire CPU logic is contained on one circuit board. This feature allows for flexibility of system expansion because of the extra chassis space available. Features of the 11/04 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.
- Operator front panel with built-in CPU console emulator that allows control from any ASCII terminal without the need for the conventional front panel with display lights and switches.
- Automatic bootstrap loader which 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 8K bytes of memory to as much as 56K bytes.
- Choice of 5¼-inch or 10½-inch high mounting chassis.

MEMORY

The PDP-11/04 is available with MOS, core memory, or a mixture of the two. MOS (metalic oxide semiconductor) memory uses industry standard 4K random access memory chips with cycle time of 700 nanoseconds. MOS packaging provides up to 16K words on a single circuit board, which can be located in any available backplane slot. Optional battery backup is available to maintain MOS memory contents during a power failure.

Memory

Min size:	4K words
Max size:	28K words
Type:	MOS, core
Access time:	500 nsec, typ
Cycle time:	725 nsec, typ

CONSOLE

The CPU console emulator feature permits control of the PDP-11/04 from any ASCII terminal connected to the processor. Console emulator operations include the normal memory LOAD, EXAMINE, and DEPOSIT, in addition to START or BOOT. This ROM-resident virtual console routine emulates all the functions of a normal programmers' console and provides at the keyboard the equivalent capability of any serial ASCII terminal connected to the system.

The operational programmers' console is a useful aid for program development. The 11/04 includes a maintenance feature which aids in system error diagnostics. When in maintenance mode, the programmers' console enables the CPU's microcode to be single stepped and the UNIBUS addresses and data to be displayed or printed. Detailed information about the operators' and programmers' consoles is presented in the section of this chapter which discusses the PDP-11/34.

Usual Mechanical Conditions

These requirements may vary or be altered according to site conditions; a DIGITAL salesperson can offer appropriate information about any specific situation.

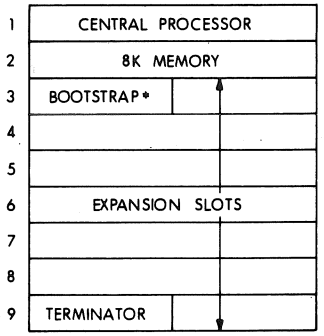


Figure 6-1 PDP-11/04 Backplane

PDP-11/34

The PDP-11/34 is a mid-range member of the PDP-11 family of processors. Features include:

- Integral memory management hardware that provides program protection, memory relocation, and addressing of up to 124K 16-bit words

- Integral extended instruction set (EIS) that provides hardware fixed-point arithmetic in double-precision mode (32-bit operands).
- 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.
- Operator front panel with built-in CPU console emulator that allows control from any ASCII terminal without the need for the conventional front panel with display lights and switches.
- Automatic bootstrap loader which allows system restart from a variety of peripherals devices without manual switch toggling or key-pad operations.
- Choice of 5¼-inch or 10½-inch high mounting chassis.

MEMORY

The PDP-11/34 is available with MOS memory, core memory, a mixture of the two, or cache. MOS (metallic oxide semiconductor) memory uses industry standard 4K random access memory chips with a cycle time of 725 nanoseconds. MOS packaging provides up to 16K words on a single circuit board.

Optional battery backup is available to maintain MOS memory contents during a power failure.

8K or 16K words of core memory are provided on a single board, which mounts in one slot and overhangs the adjacent slot.

Parity memory, MOS or core, is standard on all PDP-11/34s, as is memory management and protection. This hardware feature is designed for systems where the memory size is greater than 28K words and for multi-programming systems where protection and relocation facilities are necessary.

Memory

Max size:	124K words
Type:	core or MOS
Parity:	standard

Cache Memory

The cache memory option utilizes a 2K byte direct mapping approach with an expected "hit" ratio of 86%. Detailed information on cache is presented in Chapter 8.

MOS

The basic unit of MOS memory, MS11-JP, contains 16K words of parity MOS memory. Each 16K words of MOS requires 1 hex mounting space.

Core

The basic unit of core memory, MM11-DP, contains 16K words of parity core memory. Each 16K words of core memory requires 2 hex mounting spaces.

Parity

All main memory in a PDP-11/34 system contains 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.

Battery Backup

Core memory is non-volatile; the contents are preserved when power is removed. However, MOS memory is volatile. If power is interrupted, an auxiliary power supply must be provided if information in the memory is to be saved. With the 5½" and 10½" CPU assemblies there is an optional battery backup unit that can preserve the contents of 32K words of MOS memory for about 2 hours. This auxiliary power unit is a battery that is charged up by the main AC power when the computer system is operating normally. In this normal mode, the battery backup has no effect on the MOS memory. But if power is interrupted, voltage-sensing circuitry within the backup option will automatically cause the MOS to be powered from this auxiliary power. The MOS information will be retained by being refreshed at a low cycle rate, using minimum power.

M9301 MODULE

The M9301 module, which is included with the PDP-11/34, provides four functions.

- It contains a read-only memory (ROM) that holds diagnostic routines for verifying computer operation.
- It contains, also in ROM, the several bootstrap loader programs 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.

There are two versions of the M9301 module available:

	M9301-YA	M9301-YB
Main user	OEM	End User
Able to run secondary bootstrap program directly upon power up or reboot	yes*	no
Automatic entry to console emulator routine	yes*	yes
Needs an ASCII terminal	no	yes

* Selection of one of these two operations is made by setting of switches contained on the module.

Diagnostics

Both versions of the M9301 contain diagnostics to check both the processor and memory in a Go/No-Go mode. Execution of the diagnostics occurs automatically but may be disabled by switches on the M9301.

Bootstrap Loader

The M9301-YA contains independent bootstrap 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 bootstrap program for the peripheral device is determined by switches on the M9301 board. This is especially useful in remote applications where no operator is present.

After execution of the CPU diagnostics, the M9301-YB 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 or 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.

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 M9301 module contains a console emulator routine. When this routine is used in conjunction with the terminal, functions quite similar to those found on the programmers' console of traditional PDP-11 family computers are generated.

Summary of the Console Emulator Functions

LOAD	Loads the address to be manipulated into the system.
EXAMINE	Allows the operator to examine the contents of the address that was loaded and/or deposited.
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.

Console Emulator Operation

The console emulator allows the user to perform LOAD, EXAMINE, DEPOSIT, START, and BOOT functions by typing in the appropriate code on the keyboard.

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
- automatic entry on return from a power failure
- load 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 by the terminal. This sequence will be follwed by a $ on the next line.

Example — a typical printout on power up:

```
XXXXXX      XXXXXX      XXXXXX      XXXXXX
$
```

R0	R4	R6	PC
		STACK	PROGRAM
PROMPT		POINTER	COUNTER
CHARACTER		(SP)	

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 11/34 *Users' Guide* and the associated hardware manuals.

Termination

The M9301 contains resistors for proper impedance termination at the end of the UNIBUS.

OPERATOR'S CONSOLE

The operator's console is the front panel link between the user and the computer. It contains a minimum number of switches and lights. All normally used console functions are available through the combination of the operator's console and an ASCII terminal, such as an LA36 DECwriter.

POWER	OFF	DC power to the computer is off.
	ON	Power is applied to the computer (and the system).
	STNBY	Standby; no DC power to the computer, but DC power is applied to MOS memory (to retain data). 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 INI-Tialize and then re-turn to BOOT, the operation depends on the setting of the CONT/HALT switch.
	HALT	Only the processor is initialized and no "UNIBUS INIT" is generated. Upon lifting the CONT/HALT switch, the M9301 routine is executed allowing examination of sys-tem peripherals without clearing their contents with "UNIBUS INIT."
	CONT	Initialize and then execute the M9301 program.

When the BOOT switch is released, the following action takes place:

A. For both M9301-YA and M9301-YB (when the switches are set for this operation):

1. Run basic CPU diagnostics.
2. Print out (on the console terminal) contents of R0, R4, SP, and PC at the time of power up, following by a dollar sign ($) on the next line.
3. Enter console emulator routine, awaiting keyboard commands.
4. When a device bootstrap command is issued, first run processor memory diagnostics, then execute secondary bootstrap program from the designated peripheral device.

B. For the M9301-YA (OEM) version only (when M9301-YA switches are set for this operation):

1. Run basic CPU diagnostics.
2. Run memory diagnostics.
3. Run secondary bootstrap program from the preselected peripheral device.

> **NOTE:** When utilizing the stand-alone switch setting described as alternative b, above, the switches must be reset to enable execution of the console emulator routine.

PDP-11/34 PROCESSOR BACKPLANE CONFIGURATION

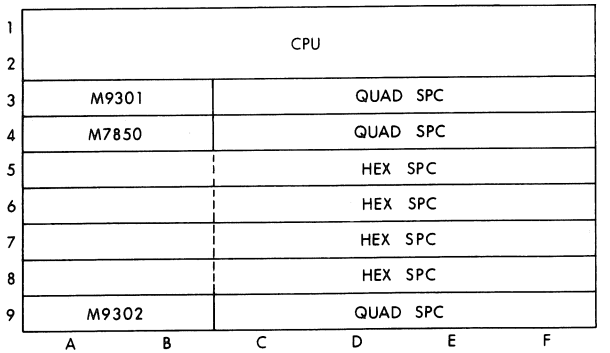
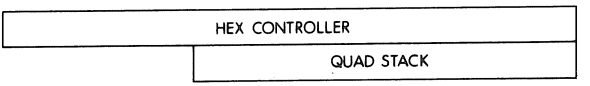


Figure 6-2 Processor Backplane

The processor backplane consists of a double system unit (SU) comprising 9 hex slots. All PDP-11/34 systems contain the CPU, M9301 Bootstrap/Terminator, M7850 parity control, and M9302 (or a UNIBUS jumper to the next SU) as shown in Figure 6-2. Memory is added as follows depending on whether the system uses core or MOS.

Core Memory:
Core memory is available in two size increments, 8K and 16K words.

The 8K core, MM11-C, consists of a hex and a quad module as follows:



The 16K core, designed MM11-D, consists of 2 hex modules as follows:

HEX CONTROLLER
HEX STACK

MOS Memory:
MOS memory is available in 8K or 16K increments and all increments consist of a single hex module.

8K and 16K increments are MS11-F and MS11-J.

The following backplane configurations constitute the basic PDP-11/34 computer.

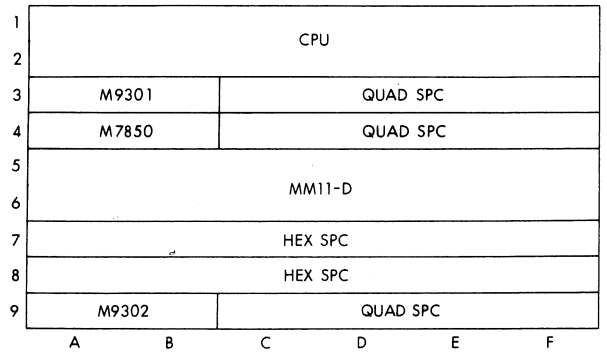


Figure 6-3 16K Core using MM11-D

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.

MEMORY MANAGEMENT ON THE PDP-11/34

Memory Management and User Protection

The PDP-11/34's integral memory management facility allows a 16-bit machine to provide 18-bit capability for a four-fold extension of addressable memory. Access to memory is in as many as 32K units through eight programmable registers. These registers assign (or map) the virtual addresses, in 4K-word pages, to 4K-word physical

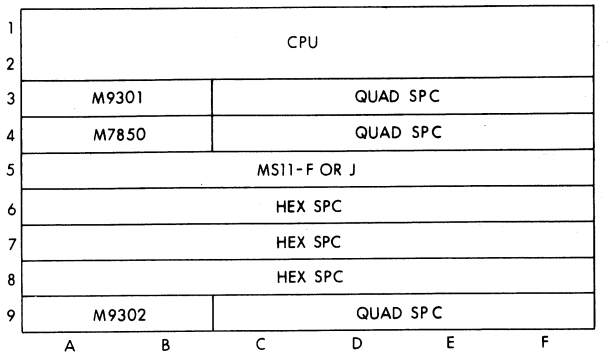


Figure 6-4 16K MOS using MS11-R or J

addresses anywhere within physical memory. The starting address of each 4K-word physical segment is stored in the registers.

Only virtual addresses need to be provided; transformation to physical addresses takes place automatically and transparently.

Programming

The memory management hardware has been optimized for a multi-programming environment. The processor can operate in two modes, **kernel** and **user**.

When in kernel mode, the program has complete control and can execute all instructions. Monitors and supervisory programs are executed in this mode.

When in 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

In a multi-programming environment several user programs would be resident in memory at any given time. The task of the supervisory program would be to:

- Control the execution of the various user programs.
- Allocate memory and peripheral device resources.
- Safeguard the integrity of the system as a whole by careful control of each user program.

In a multi-programming system, the management unit assigns pages (relocatable memory segments) to your program and prevents you from making any unauthorized access to those pages outside your assigned area. Thus, you can effectively be prevented from accidental or willful destruction of any other user program or of the system executive program.

Hardware-implemented features enable the operating system to dynamically allocate memory upon demand, while a program is being run.

Basic Addressing

18-bit direct byte addresses are generated by PDP-11/34 and larger family members. Although the PDP-11 family word length is 16 bits, the UNIBUS and CPU addressing logic is actually 18 bits. Thus, while the PDP-11 word can contain address references only up to 32K words (64K bytes) the CPU and UNIBUS can reference addresses up to 128K words (256K bytes). These extra two bits of addressing logic provide the basic framework for expanding memory references.

In addition to the word length constraint on basic memory addressing space, the uppermost 4K words of address space are always reserved for UNIBUS I/O device registers. In a basic PDP-11 memory configuration (without management), all address references to the uppermost 4K words of 16-bit address space (160000-177777) are converted to full 18-bit references with bits 17 and 16 always set to 1. Thus, a 16-bit reference to the I/O device register at address 173224 is automatically converted internally to a full 18-bit reference to the register at address 773223. The basic PDP-11 configuration can then directly address up to 28K words of true memory and 4K words of UNIBUS I/O device registers, and, with memory-managed systems, 128K words.

Active Page Registers

The memory management unit uses two sets of eight 32-bit Active Page Registers (APR). An APR is actually a pair of 16-bit registers: a Page Address Register (PAR) and a Page Descriptor Register (PDR). These registers are always used as a pair and contain all the information needed to describe and relocate the currently active memory pages.

One set of APRs is used in kernel mode, and the other in user mode. The set to be used is determined by the current CPU mode contained in the processor status word.

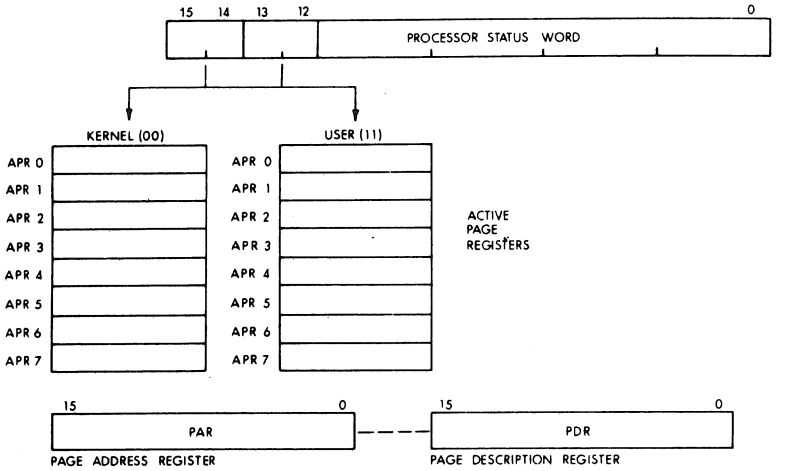


Figure 6-5 Active Page Registers

Capabilities Provided by Memory Management

Memory Size (words):	124K, max (plus 4K for I/O & registers)
Address Space:	Virtual (16 bits)
	Physical (18 bits)
Modes of Operation:	Kernel & User
Stack Pointers:	2 (one for each mode)
Memory Relocation:	
Number of Pages:	16 (8 for each mode)
Page Length:	32 to 4,096 words
Memory Protection:	no access
	read only
	read/write

Virtual Addressing

When the memory management unit is operating, the normal 16-bit direct byte address is no longer interpreted as a direct physical address (PA) but as a virtual address (VA) containing information to be used in constructing a new 18-bit physical address. The information

contained in the virtual address is combined with relocation and description information contained in the active page register to yield an 18-bit physical address.

Because addresses are relocated automatically, the computer may be considered to be operating in virtual address space. This means that no matter where a program is loaded into physical memory, it will not have to be re-linked; it always appears to be at the same virtual location in memory.

The virtual address space is divided into eight 4K-word pages. Each page is relocated separately. This is a useful feature in multi-progammmed timesharing systems. It permits a new large program to be loaded into discontinuous blocks of physical memory.

A basic function is to perform memory relocation and provide extended memory addressing capability for systems with more than 28K of physical memory. Two sets of page address registers are used to relocate virtual addresses to physical addresses in memory. These sets are used as hardware relocation registers that permit several users' programs, each starting at virtual address 0, to reside simultaneously in physical memory.

Program Relocation

The page address registers are used to determine the starting physical address of each relocated program in physical memory. Figure 6-6 shows a simplified example of the relocation concept.

Program A starting address 0 is relocated by a constant to provide physical address 6400_8.

If the next program virtual address is 2, the relocation constant will then cause physical address 6402_8, which is the second item of Program A, to be accessed. When Program B is running, the relocation constant is changed to 100000_8. Then Program B virtual addresses starting at 0 are relocated to access physical addresses starting at 100000_8. Using the active page address registers to provide relocation eliminates the need to re-link a program each time it is loaded into a different physical memory location. The program always appears to start at the same address.

A program is relocated in pages consisting of from 1 to 128 blocks. Each block is 32 words in length. Thus, the maximum length of a page is 4096 (128 × 32) words. Using all of the eight available active page registers in a set, a maximum program length of 32,768 words can be accommodated. Each of the eight pages can be relocated anywhere in the physical memory, as long as each relocated page begins on a

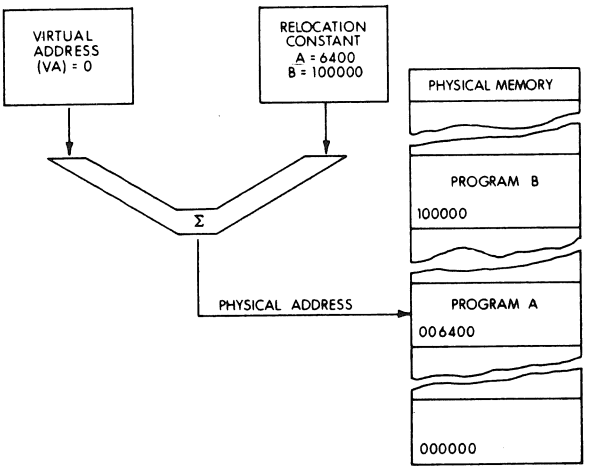


Figure 6-6 Simplified Memory Relocation

boundary that is a multiple of 32 words. However, for pages that are smaller then 4K words, only the memory actually allocated to the page may be accessed.

The relocation example shown in Figure 6-7 illustrates several points about memory relocation.

a) Although the program appears to the processor to be in contiguous address space, the 32K-word physical address space is actually scattered through several separate areas of physical memory. As long as the total available physical memory space is adequate, a program can be loaded.

b) Pages may be relocated to higher or lower physical addresses with respect to their virtual address ranges. In the example in Figure 6-7, page 1 is relocated to a higher range of physical addresses, page 4 is relocated to a lower range, and page 3 is not relocated at all (even though its relocation constant is non-zero).

c) All of the pages shown in the example start on 32-word boundaries.

d) Each page is relocated independently. There is no reason why two or more pages could not be relocated to the same physical memory space. Using more than one page address register in the set to access the same space would be one way of providing different memory access rights to the same data, depending upon which part of the program was referencing that data.

Memory Units

Block:	32 words
Page:	1 to 128 blocks (32 to 4,096 words)
No. of pages:	8 per mode
Size of relocatable memory	32,768 words, max (8 × 4,096)

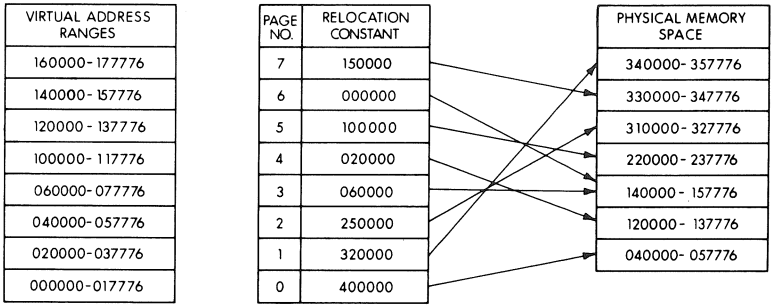


Figure 6-7 Relocation of a 32K-Word Program into 124K-Word Physical Memory

Protection

A timesharing system performs multiprogramming; it allows several programs to reside in memory simultaneously and to operate sequentially. Access to these programs, and the memory space they occupy, must be strictly defined and controlled. A timesharing system requires several types of memory protection. For example:

- User programs must not be allowed to expand beyond allocated space, unless authorized by the system.
- Users must be prevented from modifying common subroutines and algorithms that are resident for all users.
- Users must be prevented from gaining control of or modifying the operating system software.
- Users must be prevented from accessing or modifying memory occupied by other users.

The PDP-11 memory management option provides the hardware facilities to implement all of the above types of memory protection.

Inaccessible Memory
Each page has a 2-bit access control key associated with it. The key is assigned under program control. When the key is set to 0, the page is defined as non-resident. Any attempt by a user program to access a non-resident page is prevented by an immediate abort. Using this feature to provide memory protection, only those pages associated with the current program are set to legal access keys. The access control keys of all other program pages are set to 0, which prevents illegal memory references.

Read-Only Memory
The access control key for a page can be set to 2, which allows read (fetch) memory references to the page, but immediately aborts any attempt to write into that page. This read-only type of memory protection can be afforded to pages that contain common data, subroutines, or shared algorithms. This type of memory protection allows the access rights to a given information module to be user-dependent. That is, the access right to a given information module may be varied for different users by altering the access control key.

A page address register in each of the sets (kernel and user modes) may be set up to reference the same physical page in memory and each may be keyed for different access rights. For example, the user access control key might be 2 (read-only access), and the kernel access control key might be 4 (allowing complete read/write access).

Multiple Address Space
There are two complete PAR/PDR sets provided, one set for kernel mode and one set for user mode. This affords the timesharing system another type of memory protection. The mode of operation is specified by the processor status word current mode field, or previous mode field, as determined by the current instruction.

Assuming the current mode PS bits are valid, the active page register sets are enabled as follows:

PS(bits 15,14)	PAR/PDR Set Enabled
00	Kernel mode
01	Illegal (all references aborted on access
10	
11	User mode

Thus, a user mode program is relocated by its own PAR/PDR set, as are kernel programs. This makes it impossible for a program running in one mode to reference space allocated to another mode accidentally when the active page registers are set correctly. For example, a user cannot transfer to kernel space. The kernel mode address space may be reserved for resident system monitor functions, such as the basic Input/Output control routines, memory management trap handlers, and timesharing scheduling modules. By dividing the types of timesharing system programs functionally between the kernel and user modes, a minimum of space control housekeeping is required as the timeshared operating system sequences from one user program to the next. For example, only the user PAR/PDR set needs to be updated as each new user program is serviced. The two PAR/PDR sets implemented in the memory management unit are shown in Figure 6-8 and Figure 6-9.

Table 6-1 PAR/PDR Address Assignments

Kernel Active Page Registers			User Active Page Registers		
No.	PAR	PDR	No.	PAR	PDR
0	772340	772300	0	777640	777600
1	772342	772302	1	777642	777602
2	772344	772304	2	777644	777604
3	772346	772306	3	777646	777606
4	772350	772310	4	777650	777610
5	772352	772312	5	777652	777612
6	772354	772314	6	777654	777614
7	772356	772316	7	777656	777616

Page Address Register (PAR)

The page address register (PAR), shown in Figure 6-8, contains the 12-bit page address field (PAF) that specified the base address of the page.

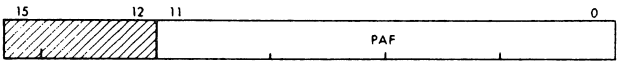


Figure 6-8 Page Address Register

Bits 15-12 are unused and reserved for possible future use.

The page address register may be thought of alternatively as a relocation constant, or a base register containing a base address. Either interpretation indicates the basic function of the page address register (PAR) in the relocation scheme.

Page Descriptor Register (PDR)

The Page Descriptor Register (PDR), shown in Figure 6-9, contains information relative to page expansion, page length, and access control.

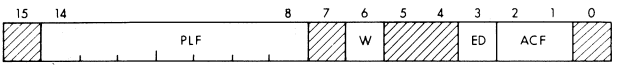


Figure 6-9 Page Descriptor Register

Access Control Field (ACF)

This 2-bit field, bits 2 and 1, of the PDR describes 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 an abort of the current operation. A memory reference that causes an abort is not completed and is terminated immediately.

Aborts are caused by attempts to access non-resident pages, by page length errors, or by access violations, such as attempting to write into a read-only page. 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. A write is synonymous with what is usually called a store or modify in many computer systems. Table 6-2 lists the ACF keys and their functions. The ACF is written into the PDR under program control.

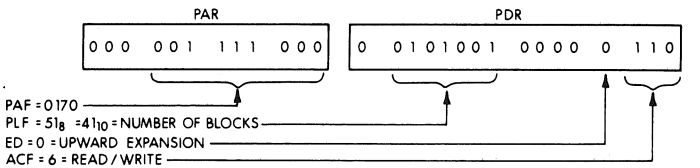
Table 6-2 Access Control Field Keys

AFC	Key	Description	Function
00	0	Non-resident	Abort any attempt to access this non-resident page
01	2	Resident read-only	Abort any attempt to write into this page.
10	4	(unused)	Abort all Accesses.
11	6	Resident read/write	Read or Write allowed. No trap or abort occurs.

Expansion Direction (ED)

The ED bit located in PDR bit position 3 indicates the authorized direction in which the page can expand. A logic 0 in the bit (ED = 0) indicates the page can expand upward from relative zero. A logic 1 in this bit (ED = 1) indicates the page can expand downward toward relative zero. The ED bit is written into the PDR under program control. When the expansion direction is upward (ED = 0), the page length is increased by adding blocks with higher relative addresses. Upward expansion is usually specified for program or data pages to add more program or table space. An example of page expansion upward is shown in Figure 6-10.

When the expansion direction is downward (ED = 1), the page length is increased by adding blocks with lower relative addresses. Downward expansion is specified for stack pages so that more stack space can be added. An example of page expansion downward is shown in Figure 6-11.



NOTE: To specify a block length of 42 for an upward expandable page, write highest authorized block number directly into high byte of PDR. Bit 15 is not used because the highest allowable block number is 177_8.

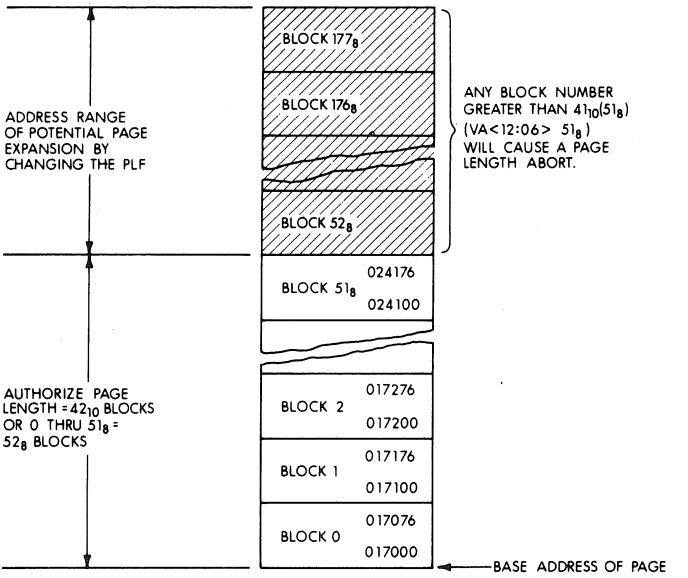


Figure 6-10 Example of an Upward Expandable Page

Written Into (W)

The W bit located in PDR bit position 6 indicates whether the page has been written into since it was loaded into memory. W = 1 is affirmative. The W bit is automatically cleared when the PAR or PDR of that page is written into. It can be set only by the control logic.

In disk swapping and memory overlay applications, the W bit (bit 6) can be used to determine which pages in memory have been modified by a user. Those that have been written into must be saved in their current form. Those that have not been written into (W = 0), need not be saved and can be overlayed with new pages, if necessary.

Page Length Field (PLF)

The 7-bit PLF located in PDR (bits 14-8) specifies the authorized length of the page, in 32-word blocks. The PLF holds block numbers from 0 to 177_8, thus allowing any page length from 1 to 128_{10} blocks. The PLF is written in the PDR under program control.

PLF for an Upward Expandable Page

When the page expands upward, the PLF must be set to one less than the intended number of blocks authorized for that page. For example, if 52_8 (42_{10}) blocks are authorized, the PLF is set to 51_8 (41_{10}) (Figure 6-9). The hardware compares the virtual address block number, VA (bits 12-6) with the PLF to determine if the virtual address is within the authorized page length.

When the virtual address block number is less than or equal to the PLF, the virtual address is within the authorized page length. If the virtual address is greater than the PLF, a page length fault (address too high) is detected by the hardware and an abort occurs. In this case, the virtual address space legal to the program is non-contiguous because the three most significant bits of the virtual address are used to select the PAR/PDR set.

PLF for a Downward Expandable Page

The capability of providing downward expansion for a page is intended specifically for those pages that are to be used as stacks. In the PDP-11, a stack starts at the highest location reserved for it and expands downward toward the lowest address as items are added to the stack.

When the page is to be downward expandable, the PLF must be set to authorize a page length, in blocks, that starts at the highest address of the page. That is always Block 177_8. Refer to Figure 6-11, which shows an example of a downward expandable page. A page length of 42_{10} blocks is arbitrarily chosen so that the example can be compared with the upward expandable example shown in Figure 6-10.

> **NOTE:** The same PAF is used in both examples. This is done to emphasize that the PAF, as the base address, always determines the lowest address of the page, whether it is upward or downward expandable.

To specify page length for a downward expandable page, write complement of blocks required into high byte of PDR.

In this example, a 42-block page is required.
PLF is derived as follows:

$42_{10} = 52_8$; 2's complement = 126_8

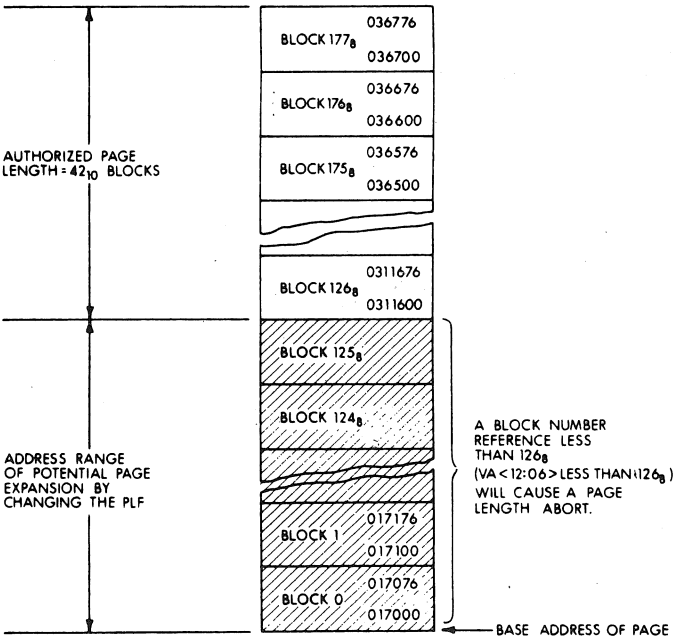


Figure 6-11 Example of a Downward Expandable Page

The calculations for complementing the number of blocks required to obtain the PLF is as follows:

MAXIMUM BLOCK NO.	MINUS	REQUIRED LENGTH	EQUALS	PLF
177_8	–	52_8	=	125_8
127_{10}	–	42_{10}	=	85_{10}

Virtual & Physical Addresses

The memory management unit is located between the central processor unit and the UNIBUS address lines. When memory management is enabled, the processor ceases to supply physical address information to the UNIBUS. Instead, virtual addresses are sent to the memory management unit where they are relocated by various constants computed within the memory management unit.

Construction of a Physical Address

The basic information needed for the construction of a Physical Address (PA) comes from the Virtual Address (VA), which is illustrated in Figure 6-12, and the appropriate APR set.

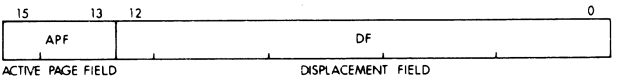


Figure 6-12 Interpretation of a Virtual Address

The virtual address consists of:

1. The Active Page Field (APF). This 3-bit field determines which of eight active page registers (APR0-APR7) will be used to form the physical address (PA).
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 (2^{13} = 8K bytes). The DF is further subdivided into two fields as shown in Figure 6-13.

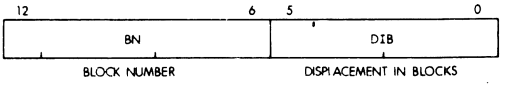


Figure 6-13 Displacement Field of Virtual Address

The displacement field (DF) 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 16-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 12-bit Page Address Field (PAF) (part of the

active page register) and specifies the starting address of the memory which that APR describes. The PAF is actually a block number in the physical memory, e.g. PAF = 3 indicates a starting address of 96 (3 × 32 = 96) in physical memory.

The formation of the physical address is illustrated in Figure 6-14.

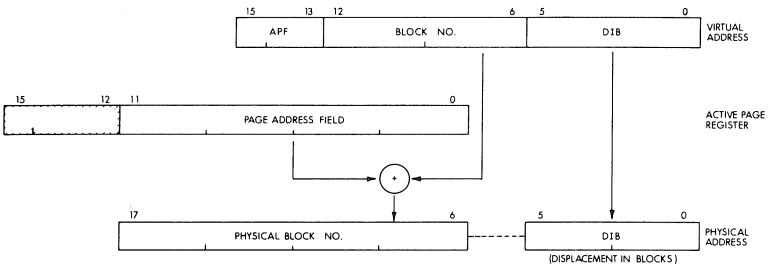


Figure 6-14 Construction of a Physical Address

The logical sequence involved in constructing a physical address is as follows:

1. Select a set of active page registers depending on current mode.
2. The active page field of the virtual address is used to select an active page register (APR0-APR7).
3. The page address field of the selected active page 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 block number from the page address field to yield the number of the block in physical memory 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 true 18-bit physical address.

Determining the Program Physical Address

A 16-bit virtual address can specify up to 32K words, in the range from Q to 177776_8 (word boundaries are even numbers). The three most significant virtual address bits designate the PAR/PDR pair to be referenced during page address relocation. Table 6-3 lists the virtual address ranges that specify each of the PAR/PDR sets.

Table 6-3 Relating Virtual Address to PAR/PDR Set

Virtual Address Range	PAR/PDR Set
000000-17776	0
020000-37776	1
040000-57776	2
060000-77776	3
100000-117776	4
120000-137776	5
140000-157775	6
160000-177776	7

NOTE Any use of page lengths less than 4K words causes holes to be left in the virtual address space.

Status Registers

Aborts generated by protection hardware are vectored through kernel virtual location 250. Status registers 0 and 2 are used to determine why the abort occured. Note that an abort to a location which is itself an invalid address will cause another abort. Thus the kernel program must insure that kernel virtual address 250 is mapped into a valid adddress, otherwise a loop will occur which will require console intervention.

Status Register 0 (SR0)

SR0 contains abort error flags, memory management enable, plus other essential information required by an operating system to recover from an abort or service a memory management trap. The SR0 format is shown in Figure 6-15. Its address is 777 572.

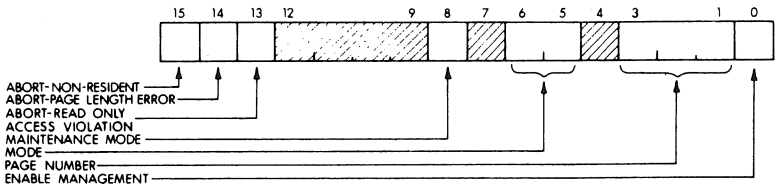


Figure 6-15 Format of Staus Register 0 (SR0)

Bits 15-13 are the abort flags. They may be considered to be in priority order in that flags to the right are less significant and should be ig-

nored. For example, a non-resident abort service routine would ignore page length and access control flags. A page length abort service routine would ignore an access control fault.

> **NOTE** Bit 15, 14, or 13, when set (abort conditions) cause the logic to freeze the contents of SR0 bits 1 to 6 and status register SR2. This is done to facilitate recovery from the abort.

Protection is enabled when an address is relocated. This implies that either SR0, bit 0 is equal to 1 (memory management enabled) or that SR0, bit 8, is equal to 1 and the memory reference is the final one of a destination calculation (maintenance/destination mode).

Note that SR0 bits 0 and 8 can be set under program control to provide meaningful memory management control information. However, information written into all other bits is not meaningful. Only that information which is automatically written into these remaining bits as a result of hardware actions is useful as a monitor of the status of the memory management unit. Setting bits 15-13 under program control will not cause traps to occur. These bits, however, must be reset to 0 after an abort or trap has occurred in order to resume monitoring memory management.

Abort Nonresident
Bit 15 is the abort nonresident bit. It is set by attempting to access a page with an access control field (ACF) key equal to 0 or 4 or by enabling relocation with an illegal mode in the PS.

Abort Page Length
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 (PFL) of the PDR for that page.

Abort Read Only
Bit 13 is the abort read-only bit. It is set by attempting to write in a read-only page, access key 2.

> **NOTE** There are no restrictions against abort bits' being set simultaneously by the same access attempt.

Maintenance/Destination Mode
Bit 8 specifies maintenance use of the memory management unit. It is used for diagnostic purposes. For the instructions used in the initial

diagnostic program, bit 8 is set so that only the final destination reference is relocated. It is useful to prove the capability of relocating addresses.

Mode of Operation
Bit 5 and 6 indicate the CPU mode (user or kernel) associated with the page causing the abort. (Kernel = 00, User = 11).

Page Number
Bits 3-1 contain the page number of reference. Pages, like blocks, are numbered from 0 upwards. The page number bit is used by the error recovery routine to identify the page being accessed if an abort occurs.

Enable Relocation and Protection
Bit 0 is the enable bit. When it is set to 1, all addresses are relocated and protected by the memory management unit. When bit 0 is set to 0, the memory management unit is disabled and addresses are neither relocated nor protected.

Status Register 2 (SR2)
SR2 is loaded with the 16-bit virtual address (VA) at the beginning of each instruction fetch, but is not updated if the instruction fetch fails. SR2 is read-only; a write attempt will not modify its contents. SR2 is the Virtual Address Program Counter. Upon an abort, the result of SR0 bits' 15, 14, or 13 being set will freeze SR2 until the SR0 abort flags are cleared. The address of SR2 is 777 576.

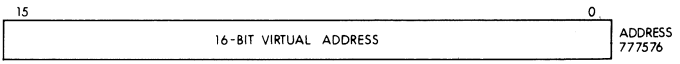


Figure 6-16 Format of Status Register 2 (SR2)

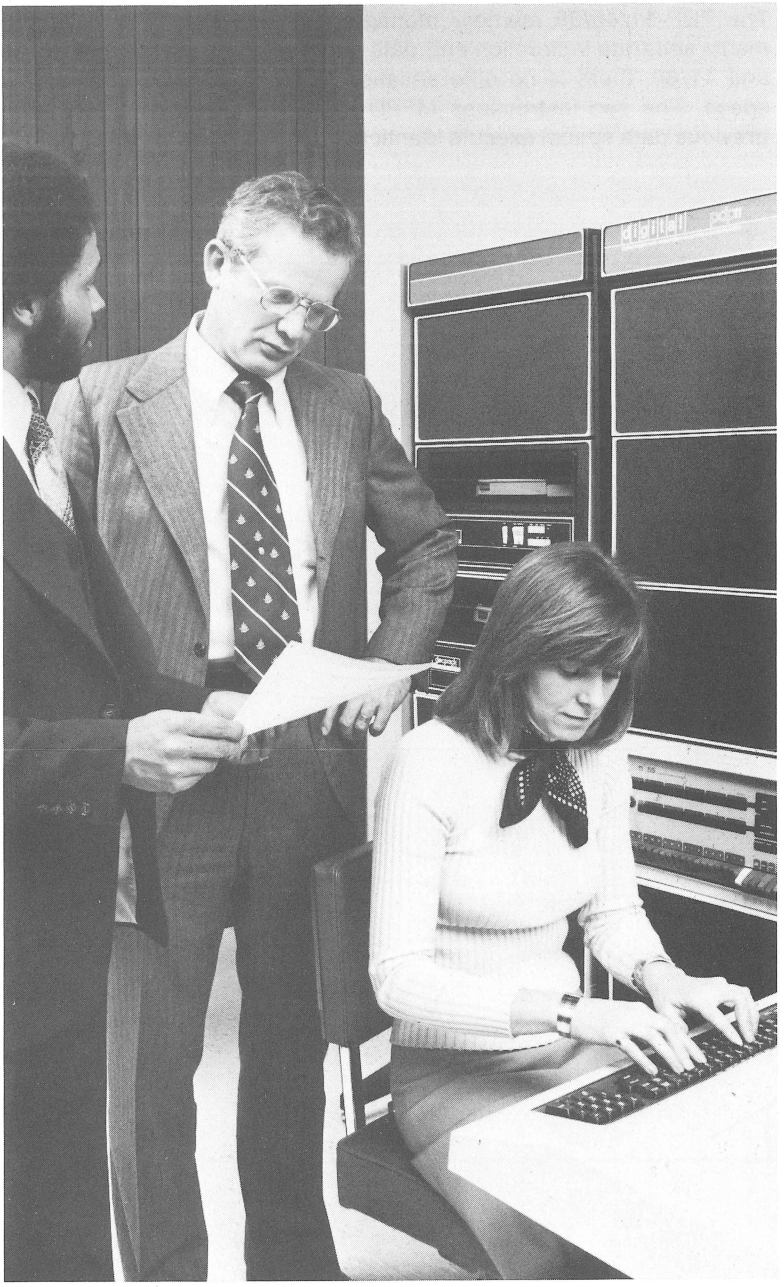
MEMORY MANAGEMENT INSTRUCTIONS

Memory management provides communication between two spaces, as determined by the current and previous modes of the processor status word (PS).

Mnemonic	Instruction	Op Code
MFPI	move from previous instruction space	0065SS
MTPI	move to previous instruction space	0066DD
MFPD	move from previous data space	1065SS
MTPD	move to previous data space	1066DD

These instructions are directly compatible with the larger 11 computers.

The PDP-11/45/55 memory management unit, the KT11-C, implements separate instruction and data address space. In the PDP-11/34 and 11/60, there is no differentiation between instruction and data space. The two instructions MFPD and MTPD (Move to and from previous data space) execute identically to MFPI and MTPI.



CHAPTER 7

PDP-11/45, PDP-11/55

The PDP-11/45 and the PDP-11/55 are mid-range, very high speed computers. They are designed for *fast* program execution and applications involving FORTRAN-compiled tasks as well as scientific and engineering calculations.

The PDP-11/45 is a core memory based processor, the PDP-11/55 is a bipolar memory based machine.

PDP-11/45 FEATURES

The PDP-11/45 is designed for speed. The high speed CPU circuits allow program execution speeds in excess of three million instructions per second. Bipolar memory is available at a speed of 300 nanoseconds. Core memory is available at a speed of 980 nanoseconds. Both may be combined on a single system up to a total of 32K words bipolar, and core up to 128K words total memory. High speed computations may be performed by an independent floating point processor, standard in the 11/55, optional in the 11/45. It overlaps its operations with that of the central processor and offers, for example, average double precision multiply times of 5.43 microseconds, working with its own set of 64-bit registers.

The PDP-11/45 provides solutions to multiple task applications where the computer must solve many similar problems or run multiple programs concurrently, for example, the automation of industrial processes, monitoring and controlling multiple operations in real-time while simultaneously preparing and printing production reports for management. Memory size may be as small as 16K bytes or as large as 248K bytes to accommodate several programs in memory simultaneously.

PDP-11/45 features include:

- memory expandable to 248K bytes
- optional memory segmentation, protection, and relocation
- optional FP11-C floating point processor with advanced features and operation with 32-bit and 64-bit numbers
- reliable core memory
- fast secondary bus between processor and solid-state bipolar memory which operates in parallel with the UNIBUS
- bootstrap
- real-time clock

PDP-11/55 FEATURES

The PDP-11/55 is a bipolar memory based computer designed for greater processor and system performance through the use of a dedicated internal semiconductor memory bus. This high speed bus allows the PDP-11/55 to fetch and execute instructions at 300 nanoseconds. Two separate semiconductor controllers allow simultaneous data transfers for increased system throughput (i.e., the CPU transfers to one controller while DMA devices transfer to the other.) The PDP-11/55 can be expanded up to 248K bytes with the aid of the memory management, which is an integral part of the central processor. The fast floating point processor option operates as an integral part of the central processor, yet interacts with the CPU only when data must be transferred to or from memory.

PDP-11/55 features include:

- A central processor unit with 64K bytes of 300 nsec bipolar memory, or 32K bytes of 980 nsec core memory combined with 32K bytes of 300 nsec bipolar memory.
- An optional floating point processor (FP11-C) which provides very fast arithmetic processing capabilities. It performs a single precision (32 bit) add in 1.65 microseconds, and a double precision (64 bit) multiply in only 5.43 microseconds.
- A dual-bus structure that allows you to intermix core and bipolar memory to optimize system performance.
- Integral memory management hardware which provides 18-bit addressing capability (up to 248K bytes) as well as memory protection.
- An automatic bootstrap loader which initiates system startup at the flick of a single switch.
- A real-time clock.

The PDP-11/55 and 11/45 hardware has been optimized towards a multi-programming environment; the processor operates in three modes (kernel, supervisor, and user) and has two sets of general registers.

The PDP-11/55, 11/45 CPUs perform all arithmetic and logical operations required in the system. They also act 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 logic for a wide range of operations. These include high-speed fixed point arithmetic, including hardware multiply and divide, extensive test and branch operations, and other

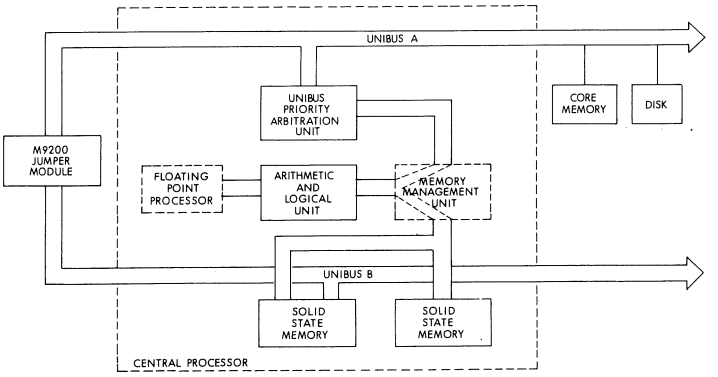


Figure 7-1 PDP-11/55, PDP-11/45 System Block Diagram

control operations. It also provides for the addition of the high-speed floating point and memory management units.

The CPU operates in three modes: kernel, supervisor, and user. When in kernel mode, a program has complete control of the processor; when in any other mode the processor is prohibited 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 two sets of eight 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. A special instruction, MARK, is provided to further facilitate re-entrant programming. One of the general registers is used as the program counter. Three others are used as processor stack pointers, one for each operational mode.

Figure7-2 illustrates the data paths in the CPU.

The 11/55 and 11/45 CPUs perform all the computer's computation and logic operations in a parallel binary mode through step by step execution of individual instructions. The instructions are stored in either core or solid state memory.

General Registers

The general registers (see Figure 7-3) can be used for a variety of purposes, the uses varying with requirements.

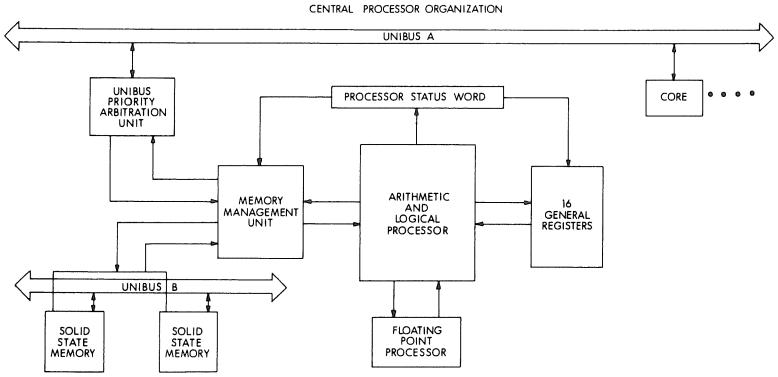


Figure7-2 Central Processor Data Paths

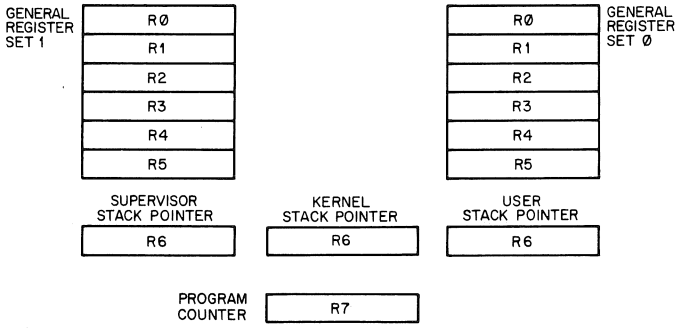


Figure 7-3 The General Registers

Register 7 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 often used only for addressing purposes and is not used as an accumulator for arithmetic operations.

Register 6 is normally used as the processor stack pointer indicating the last entry in the appropriate stack (a common temporary storage area 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 an interrupt or trap occurs, the central processor automatically

saves its current status on the processor stack selected by the service routine. This stack-based architecture facilitates re-entrant programming.

The other 12 registers consist of two sets of unrestricted registers, R0-R5. The register set currently 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 to facilitate multi-programming. The six general registers in 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 stack from user programs, or from other parts of the supervisor.

Processor Status Word

The processor status word, located at location 777776, contains information on the current status of the PDP-11/55, 11/45. See Figure 7-4. 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.

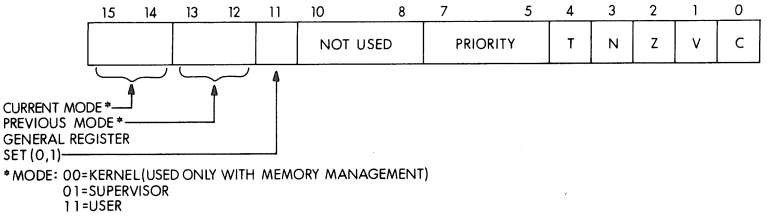


Figure 7-4 Processor Status Word

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 to 1) is currently being used (bit 11).

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. The processor will also ignore the RESET and SPL instructions. In kernel mode, the processor will execute all instructions.

A program operating in kernel mode can map users' programs anywhere in core 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 which can only be used by the kernel. This instruction allows a kernel mode program to alter the central processor's priority without affecting the rest of the processor status word.

Stack Limit Register

All PDP-11s have a stack overflow boundary at location 400. The kernel stack boundary in the PDP-11/55, 11/45 is a variable boundary set through the stack limit register found in location 777775.

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).

Floating Point Processor

The PDP-11/55, 11/45 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 conversions in parallel with the CPU. It is described in detail in Chapter 10.

MEMORY

Memory is the primary storage medium for instructions and data. Two types are available on the 11/45, 11/55 processors:

Solid State Bipolar memory with a cycle time of 300 nsec

Core Magnetic core memory with a cycle time of 980 nsec, access at 360 nsec (450 nsec at the UNIBUS)

The PDP-11/45 is a core based machine, and the PDP-11/55 is a bipolar memory based machine containing 32K or 64K bytes (maximum) of bipolar memory. Any system can be expanded to 248K bytes in increments of 32K bytes. The system can be configured with various mixtures of core and bipolar memory up to a maximum limit of 64K bytes of bipolar memory.

Solid State Memory

The central processor communicates directly with bipolar memory through a UNIBUS that is internal to the PDP-11/55, 11/45 processing system. The CPU can control up to two independent solid state memory controllers. Each controller can have from one to four 2K-byte increments (8K byte maximum) or from one to four 8K-byte increments (32K byte maximum). 2K- and 8K-byte increments cannot be mixed in the same bipolar memory controller.

Each controller has dual ports and provides one interface to the CPU and another to a second UNIBUS. See Figure 7-5.

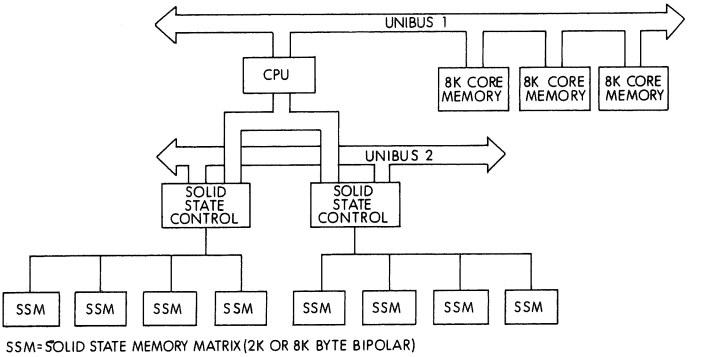


Figure 7-5 Memory Configuration

There are two UNIBUSes on the PDP-11/55, 11/45 but in a single processing environment the second UNIBUS is generally connected to the first and becomes part of it. If the two UNIBUSes are connected, DMA devices on both UNIBUSes can access bipolar memory. If the two UNIBUSes are not connected, only DMA devices on UNIBUS 2 can

access bipolar memory and must include UNIBUS arbitration logic which lends itself to multiprocessor environments (Figure 7-6).

The UNIBUS and data path to the solid state memory are independent. While the central processor is operating on data in one solid state memory controller through the direct data path, any device could be using the UNIBUS to transfer information to core, to another device, or to the other solid state memory controller. This autonomy significantly increases the throughput of the system.

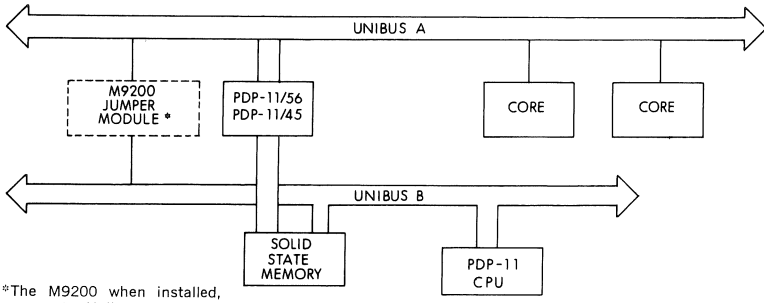


Figure 7-6 Multiprocessor Use of the Second UNIBUS

Core Memory

The central processor communicates with core memory through the UNIBUS.

Each memory bank operates independently from other banks through its own controller, which interfaces directly to the UNIBUS. Core memory can be attached to the UNIBUS until the system contains a total of 248K (253,952) bytes of memory.

An external device may use the UNIBUS to read or to write to core memory completely independently of or simultaneously with the central processor's access of solid state memory. Core memory and solid state memory may be used by the processor interchangeably.

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, float-

ing 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. Traps are discussed in Chapter 5.

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 fail.

The processor traps to location 24 when power is restored 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) or tries to fetch any instruction from an odd address. 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 5 to 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.

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

MULTIPROGRAMMING

The PDP-11/55, 11/45 architecture, with its three modes of operation, the two sets of general registers, its memory management capability, and its program interrupt request facility, provides an ideal environment for multi-programming systems.

In any multi-programming system there must be some method of transferring information and control between programs operating in the same or different modes. The PDP-11/55 and 11/45 provide these communication paths.

Control Information
Control is passed inwards (in user, supervisor, or kernel mode) by all traps and interrupts. Kernel routines can map into low physical core

where the vector space is. Thus all traps and interrupts pass through kernel space to pick up their new PC and PS and to determine the new mode of processing.

Control is passed outwards (kernel, supervisor, user) by the RTI and RTT instructions (described in Chapter 4).

Data

Data is transferred between modes by four instructions: Move From Previous Instruction Space (MFPI), Move From Previous Data Space (MFPD), Move to Previous Instruction Space (MTPI) and Move to Previous Data Space (MTPD). There are four instructions rather than two, as 11/45 and 11/55 memory management distinguishes between instruction and data space. The instructions are described fully in Chapter 4. However, it should be noted that these instructions have been designed to allow data transfers to be under the control of the innermost mode (kernel, supervisor, user) and not the outermost, thus providing protection of an inner program from an outer.

Processor Status Word

The PDP 11/55, 11/45 protect the PS from implicit references by supervisor and user programs which could result in damage to an inner level program.

A program operating in kernel mode can perform any manipulation of the PS. Programs operating at outer levels (supervisor and user) are inhibited from changing bits 5-7 (the processor's priority). They are also restricted in their treatment of bits 15, 14 (current mode); bits 13, 12 (previous mode); and bit 11 (register set); these bits may be set in user or supervisor mode. However, in order to clear these bits, a trap or interrupt must be issued which returns the system to kernel mode.

PDP-11/45, 11/55 SPECIFICATIONS

Memory

Min size:	64K bytes
Max size:	248K bytes
Type:	bipolar, core
Parity:	optional

Central Processor

Instructions	basic set + XOR, SOB, MARK, SXT, RTT, MUL, DIV, ASH, ASHC, SPL
Programming modes:	3
No.of general registers:	16 (two sets of eight)

Auto hardware interrupts: yes

Auto software interrupts: yes

Power fail/auto restart: yes

NOTES:

- CPU fastbus activity does not degrade data transfer speed of either bus, except when both buses are simultaneously accessing the same MS11 control board.
- If there are two MS11 controls in a CPU, transfers on one bus to one memory do not interact with transfers on the other bus to the other memory.
- Data transfer rates for the PDP-11/55, 11/45:

  Configuration #1
  The maximum system data transfer rate with UNIBUS controllers transferring to interleaved MM11-UP core memory over the UNIBUS while the CPU transfers to bipolar memory over the fastbus is 9.0 megabytes per second.

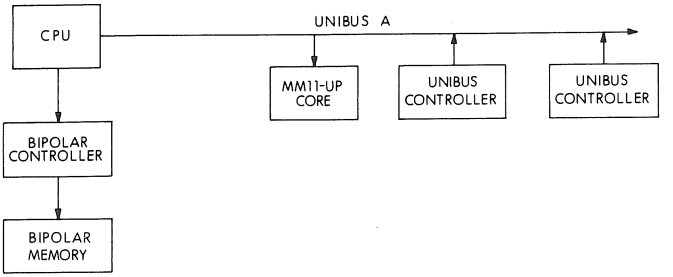


Figure 7-7 Configuration #1

Configuration #2
The maximum system data transfer rate with a UNIBUS controller transferring to bipolar memory while the CPU transfers to the same bipolar memory (same bipolar memory controller) is 7.14 megabytes per second.

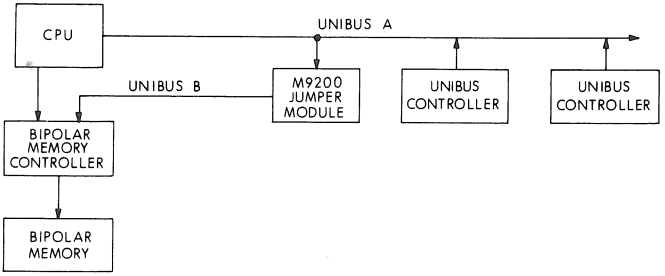


Figure 7-8 Configuration #2

Configuration #3
The maximum system data transfer rate with a UNIBUS controller transferring to one bipolar controller while the CPU transfers to the other bipolar controller is 10.78 megabytes per second.

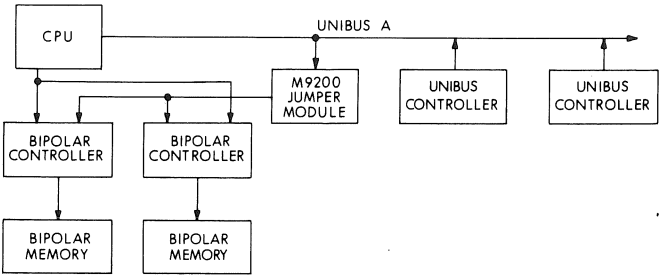


Figure 7-9 Configuration#3

- The two MS11 solid state memory controls are connected to a single UNIBUS (UNIBUS B) that can be easily separated from the 11/45 CPU UNIBUS (UNIBUS A) by removing a simple jumper module (M9200), thus facilitating dual UNIBUS systems. UNIBUS B does not have its own UNIBUS arbitration control logic; thus, a second PDP-11 CPU is required for other than NPR transfers from a single device.

CONSOLE OPERATION

The PDP-11/55, 11/45 system operator's console is designed for convenient system control. A complete set of function switches and display indicators provide comprehensive status monitoring and control facilities.

Console

The operator's console for the PDP-11/55, 11/45 provides the following facilities:

- a system key switch (OFF/ON/LOCK)
- a bank of seven indicator lights, indicating the following central processor states: RUN, PAUSE, MASTER(UNIBUS), USER, SUPERVISOR, KERNEL, DATA.
- an 18-bit address register display
- an addressing error indicator light (ADRS ERR)
- a 16-bit data register display
- an 18-bit switch register
- control knobs
- address display select
  USER I VIRTUAL
  USER D VIRTUAL
  SUPERVISOR I VIRTUAL
  SUPERVISOR D VIRTUAL
  KERNEL I VIRTUAL
  KERNEL D VIRTUAL
  PROGRAM PHYSICAL
  CONSOLE PHYSICAL
- data display select
  DATA PATHS
  BUS REGISTER
  FPP μADRS.CPU μADRS.
  DISPLAY REGISTER
- control switches
  LOAD ADRS (Load Address)
  EXAM (Examine)
  CONT (Continue)
  ENABLE/HALT
  S-INST/S-BUS CYCLE (Single Instruction/Single Bus Cycle)
  START
  DEPOSIT
  REG EXAM (Register Examine)
  REG DEPOSIT (Register Deposit)

System Power Switch

The system power switch controls central processor power as follows:

OFF	Power off for CPU. Solid state memory still receives power in order to insure data retention.

POWER	Power ON for CPU — normal use, all console controls operable.
PANEL LOCK	Power ON for CPU — all console controls not operable except switch register.

Central Processor State Indicators

This bank of indicator lights shows the current major system state as follows:

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, console intervention, or power down.
PAUSE	The CPU is inactive because the current instruction execution has been completed as far as possible without more data from the UNIBUS, and the CPU is waiting to regain control of the UNIBUS (UNIBUS mastership).
MASTER	1) The CPU is in control of the UNIBUS (UNIBUS Master). The CPU relinquishes control of the UNIBUS during DMA and NPR data transfers. 2)The CPU has been HALTed from the system operator's console.
USER	The CPU is executing program instructions in user mode. When the memory management unit is enabled all address references are in user virtual address space.
SUPERVISOR	The CPU is executing program instructions in supervisor mode. When the memory management unit is enabled, all address references are in supervisor virtual addressing space.
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 a 0, the last memory reference was to I address space in the current CPU mode.

Address Display Register

The address display register is primarily a software development and maintenance aid. The contents of this 18-bit indicator are controlled by the address select knob as follows:

VIRTUAL	The address display register indicates the current address reference as a 16-bit virtual address when the memory management unit is enabled; otherwise, it indicates the true 16-bit physical address. Bits 17 and 16 will be off unless the memory management unit is disabled and the current address references some UNIBUS device register in the uppermost 8K bytes of basic address space (i.e., 248K-256K).
PROGRAM PHYSICAL	The address display register indicates the current address reference as a true 18-bit physical address.
CONSOLE PHYSICAL	The address display register indicates the current address reference as a 16-bit virtual address when the memory management unit is enabled; otherwise, it indicates the true 16-bit physical address. Bits 17 and 16 indicate the contents of corresponding bits of the switch register as of the last LOAD ADRS console operation.

Addressing Error Display

This 1-bit display indicates the occurrence of any addressing errors. The following address references are invalid:

- non-existent memory
- access control violations
- unassigned memory pages

Data Display Register

DATA PATHS	The data display register indicates the current output of the PDP-11/55, 11/45 arithmetic/logical unit subsystem (SHFR).
BUS REGISTER	The data display register indicates the current output of the PDP-11/55, 11/45 CPU UNIBUS, semiconductor memory, or of the internal bus.
FPP μADRS.CPU μADRS.	The data display register indicates the current ROM address, FPP control micro-program (bits 15-8), and the CPU control micro-program (bits 7-0).

DISPLAY — The data display register indicates the current contents of the 16-bit write-only switch register located at physical address 777570. This register is generally used to display diagnostic information, although it can be used for other purposes.

Switch Registers

The functions of this 18-bit bank of switches are determined by:

- control switches
- address display select knob

These functions will be described in the next section with the appropriate control switch.

Note that the current setting of the switch register may be read under program control from a read-only register at physical address 777570.

Control Switches

LOAD ADRS (Load Address)

When the LOAD ADRS switch is depressed, the contents of the switch register are loaded into the CPU bus address register and displayed in the address display register lights. If the memory management unit is disabled, the address display is the true physical address.

If the memory management unit is enabled, the interpretation of the address indicated by the switch register is determined by the address display select knob.

Note that the LOAD ADRS function does not distinguish between PROGRAM PHYSICAL and CONSOLE PHYSICAL.

EXAM (Examine)

Depressing the EXAM switch causes the contents of the current location specified in the CPU bus address register to be displayed in the DATA display register.

Depressing the EXAM switch again causes a EXAM-STEP operation to occur. The result is the same as the EXAM except that the contents of the CPU Bus Address Register are incremented by two before the current location has been selected for display. An EXAM-STEP will not cross a 64K byte memory block boundary.

An EXAM operation which causes an ADRS ERR (Addressing Error) must be corrected by performing a new LOAD ADRS operation with a valid address.

REG EXAM (Register Examine)

Depressing the REG EXAM switch causes the contents of the general purpose register specified by the low order five bits of the bus address

register to be displayed in the data display register. In the PDP-11/55, consecutive register examines will automatically increment to the next general purpose register.

The switch register is interpreted as follows:

Contents	Register Displayed
0-5	general registers 0-5 (set 0)
6	kernel mode register 6
7	program counter (PC)
108—158	general registers 0-5 (set 1)
168	supervisor mode register 6
179	user mode register R6

CONT (Continue)

Depressing the CONT switch causes the CPU to resume executing instructions of bus cycles at the address specified in the Program Counter. The CONT switch has no effect when the CPU is in RUN state.

The function of the CONT switch is modified by the setting of the ENABLE/HALT and S/INST-S/BUS cycles switches as follows:

ENABLE (up) CPU resumes normal operation under program control.

HALT (down) S/INST (up) — CPU executes next instruction then stops.

S/BUS cycle (down) — CPU executes next address reference, then stops (i.e., one UNIBUS cycle).

ENABLE/HALT

The ENABLE/HALT switch is a 2-position switch with the following functions.

ENABLE (up) The CPU is able to perform normal operations under program control.

HALT (down) The CPU is stopped and is operable only by the console switches.

The setting of the ENABLE/HALT switch modifies the function of the CONTINUE and START switches.

S/INST—S/BUS CYCLE (Single Instruction/Single Bus Cycle)

The S/INST-S/BUS CYCLE switch affects only the operation of the CONTINUE switch. This switch has no effect on any other switch when the ENABLE/HALT switch is set to ENABLE.

START

The functions of the START switch depend upon the setting of the ENABLE/HALT switch as follows:

ENABLE	Depressing the START switch causes the CPU to start executing program instructions at the address specified by the current contents of the CPU bus address register. The START switch has no effect when the CPU is in run state.
HALT	Depressing the START switch causes a console reset to occur.

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 CPU bus address register.

Raising the DEP switch again causes a DEP-STEP operation to occur. The result is the same as the DEP, except that the contents of the CPU bus address register are incremented by two before the current location has been selected for the deposit operation. A DEP-STEP will not cross a 64K byte memory block boundary.

A DEP operation which causes an ADRS ERR (Addressing Error) is aborted and must be corrected by performing a new LOAD ADRS operation with a valid address.

REG DEP (Register Deposit)

Raising the REG DEP causes the contents of the switch register to be deposited into the general purpose register specified by the lowest four bits of the current contents of the CPU bus address register. In the PDP-11/55, 11/45, consecutive register deposits will automatically increment to the next general purpose register (GPR).

The CPU bus address register should have been loaded previously by a LOAD ADRS operation according to the switch register settings mentioned in the section describing REG EXAM.

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 a LOAD ADRS. The following sequence is possible:

EXAM	
DEP	ADDRESS A
EXAM	
STEP EXAM	
DEP	ADDRESS A + 1
EXAM	

ADDRESS SELECT

The ADDRESS SELECT knob is used for two functions. It provides an interpretation for the address display register. It also determines for EXAM, STEP-EXAM, DEP, and STEP-DEP which set of page address registers, if any, will be used to relocate the address loaded by the LD ADRS function.

KERNEL I, KERNEL D, SUPER I, SUPER D, USER I and USER D positions cause the address loaded into the switch register to be relocated if the memory management option is installed and operating. Which set of the six sets of page address registers (PARs) is used is determined by the ADDRESS SELECT switch. EXAMs, STEP-EXAMs, DEPs and STEP-DEPs under these conditions are relocated to the physical address specified by the appropriate PAR. If the action attempted from the console is not allowed (for example, attempting to DEP into a read-only page) the ADRS ERROR indicator will come on. A new LD ADRS must be done to clear this condition. Note that, in the general case, the physical location accessed is different from the virtual address loaded into the switch register. The address display register will always, in these six positions, show exactly what was loaded from the switch register. These positions make it convenient to examine and change programs which are subject to relocation, without requiring any knowledge of where they have actually been relocated in physical memory.

PROGRAM PHYSICAL — This position is provided to allow monitoring the physical addresses being accessed by a program when "single stepping" through the program. It is most useful when the accesses are being relocated by the memory management option. In this case, the address shown in the address display register is different than that shown in the other positions. This position should not be used to perform EXAM, STEP-EXAM, DEP or STEP-DEP functions.

CONSOLE PHYSICAL — This position is provided to allow EXAM, STEP EXAM, DEP and STEP-DEP functions to physical memory locations whether or not the memory management option is installed or operating. In this position the address display register indicates the physical address loaded from the switch register.

MEMORY MANAGEMENT ON THE PDP-11/55, 11/45

The PDP-11/55, 11/45 memory management unit provides the hardware facilities necessary for complete memory management, protection, and relocation. It is designed to be a memory management facility for systems with the memory size greater than 56K bytes and for multi-user, multi-programming systems requiring memory protection and relocation facilities.

Although some of the material in this section duplicates that in the section on 11/34 memory management, it is repeated here so that the reader does not have to refer to the previous section.

The power and efficiency of the PDP-11/55, 11/45 are most effectively utilized when several programs are run simultaneously. Several user programs are resident in memory at any given time in a multi-programming environment. The tasks of the supervisory program are: to control the execution of the various user programs, to manage the allocation of memory and peripheral device resources, and to control each program carefully, safeguarding the integrity of the system.

In a multi-programming system, the memory management unit provides the means for assigning memory pages to a user program and preventing that user from making any unauthorized access to these pages outside the assigned area. Thus, a user can effectively be prevented from accidental or willful destruction of any other user program or the system executive program.

The basic characteristics of the PDP-11/55, 11/45 memory management unit are:

- 16 user mode memory pages
- 16 supervisor mode memory pages
- 16 kernel mode memory pages
- 8 pages in each mode for instructions
- 8 pages in each mode for data
- page lengths from 32 to 4096 words
- each page provided with full protection and relocation
- transparent operation
- 6 modes of memory access control
- memory extension to 124K words (248K bytes)

PDP-11 FAMILY BASIC ADDRESSING LOGIC

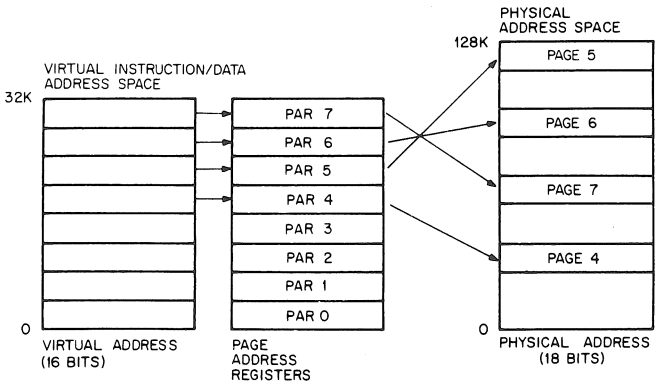
The addresses generated by all PDP-11 family central processor units (CPUs) are 18-bit direct byte addresses. Although the PDP-11 family word length and operational logic are all 16-bit length, the UNIBUS and CPU addressing logic is actually 18-bit length. Thus, while the PDP-11 word can contain address references only up to 64K bytes, the CPU and UNIBUS can reference addresses up to 256K bytes. These extra two bits of addressing logic provide the basic framework for expanded memory operation.

In addition to the word length constraint on basic memory addressing space, the uppermost 8K bytes of address space are always reserved

for UNIBUS I/O device registers. In a basic PDP-11/55, 11/45 memory configuration, (without the memory management option), all address references to the uppermost 8K bytes of 16-bit address space (170000-177777) are converted to full 18-bit references with bits 17 and 16 always set to 1. Thus, a 16-bit reference to the I/O device register at address 173224 is automatically internally converted to a full 18-bit reference to the register at address 773224. The basic PDP-11/55, 11-/45 configuration can address up to 56K bytes of true memory and 8K bytes of UNIBUS I/O device registers directly. Memory configurations beyond this require the PDP-11/55, 11/45 memory management unit.

VIRTUAL ADDRESSING

When the PDP-11/45 memory management unit is operating, the normal 16-bit direct byte address is no longer interpreted as a direct physical address (PA) but as a virtual address (VA) containing information to be used in constructing a new 18-bit physical address. The information contained in the virtual address is combined with relocation information contained in the page address register (PAR) to yield an 18-bit physical address (PA). Using the memory management unit, memory can be dynamically allocated in pages, each composed of from 1 to 128 integral blocks of 64 bytes.



PAR = Page Address Register

Figure 7-10 Virtual Address Mapping into Physical Address

The starting physical address for each page is an integral multiple of 64 bytes, and each page has a maximum size of 8198 bytes. Pages may be located anywhere within the 256K bytes physical address

space. The determination of which set of 16 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.

INTERRUPT CONDITIONS UNDER MEMORY MANAGEMENT CONTROL

The memory management unit relocates all addresses. 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 physical bank of memory.

When a trap, abort, or interrupt occurs, the push of the old PC, old PS is to the user/supervisor/kernel R6 stack specified by CPU mode bits 15,14 of the new PS in the vector (bits 15,14: 00 = kernel, 01 = supervisor, 11 = user). The CPU mode bits also determine the new PAR set. In this manner 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 simply by setting the CPU mode bits of the new PS in the vector to return control to the appropriate mode.

CONSTRUCTION OF A PHYSICAL ADDRESS

All addresses with memory relocation enabled 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 8 PARs in each mode of CPU operation, kernel, supervisor, and user. Using status register #3, the operating system may elect to disable D space and map all references (instructions and data) through I space, or to use both I and D space.

The basic information needed for the construction of a physical address (PA) comes from the virtual address (VA), which is illustrated in Figure 7-11, and the appropriate PAR set.

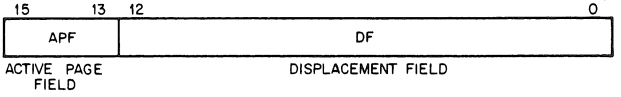


Figure 7-11 Interpretation of a Virtual Address

The virtual address (VA) consists of:

- 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 (PA).

- 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 (2_{13} = 8K bytes). The DF is further subdivided into two fields as shown in Figure 7-12).

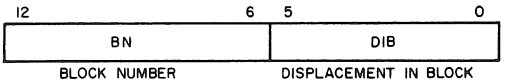


Figure 7-12 Displacement Field of Virtual Address

The displacement field (DF) consists of:

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

The remaining information needed to construct the physical address comes from the 12-bit page address field (PAF), part of the page adddress register (PAR), and specifies the starting address of the memory page that the PAR describes. The PAF is actually a block number in the physical memory, PAF = 3 indicates a starting address of 96 (3 × 32) words in physical memory.

The formation of a physical address (PA) takes 90 ns. Thus, in situations which do not require the facilities of the memory management unit, it should be disabled to permit time savings.

The logical sequence involved in constructing a physical address (PA) is as follows:

1. Select a set of page address registers depending on the space to be referenced.
2. The active page field (APF) of the virtual address is used to select a page address register (PAR0-PAR7).
3. The page address field (PAF) of the selected page address register (PAR) contains the starting address of the currently active page as a block number in physical memory.

4. The block number (BN) from the virtual address (VA) is added to the block number from the page address field (PAF) to yield the number of the block in physical memory (PBN-Physical Block Number) which will contain the physical address (PA) being constructed.
5. The displacement in block (DIB) from the displacement field (DF) of the virtual address (VA) is joined to the physical block number (PBN) to yield a true 18-bit PDP-11/55, 11/45 physical address (PA).

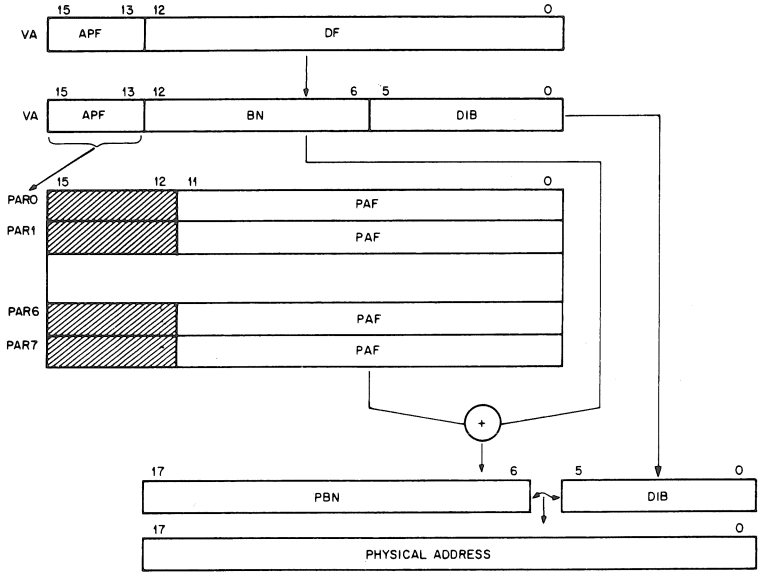


Figure 7-13 Construction of a Physical Address

MANAGEMENT REGISTERS

The PDP-11/55, 11/45 memory management unit implements three sets of registers. There are 32 registers in all, two groups of 16 per set. Each register contains 16 bits. One set of registers is used in kernel mode, another in supervisor, and the other 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 status register number 3. Each group is further subdivided into two parts of 8 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 various memory management registers are located in the uppermost 4K of PDP-11 physical address space along with the UNIBUS I/O device registers. For the actual addresses of these registers, refer to memory management unit register map at the end of this chapter.

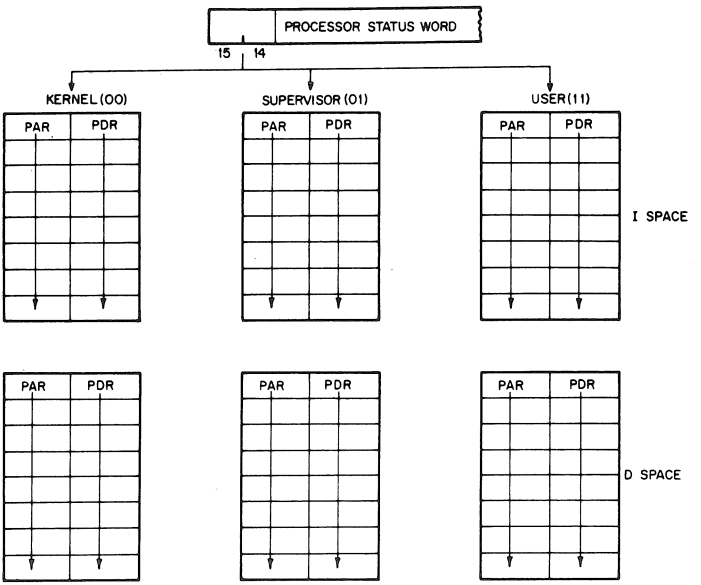


Figure 7-14 Active Page Registers

Page Address Registers (PAR)

The page address register (PAR) contains the page address field (PAF), a 12-bit field which specifies the starting address of the page as a block number in physical memory.

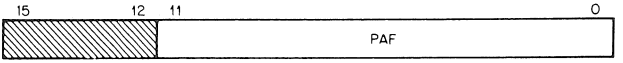


Figure 7-15 Page Address Register

Bits 15-12 of the PAR are unused and reserved for possible future use.

The page address register (PAR) which contains the page address field (PAF) may be 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 (PAR) as a relocation tool.

Page Descriptor Register

The page descriptor register (PDR) contains information relative to page expansion, page length, and access control.

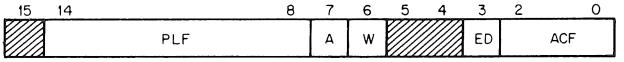


Figure 7-16 Page Descriptor Register

Access Control Field (ACF)

This 3-bit field, occupying bits 2-0 of the page descriptor register (PDR), 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, whereas a reference causing a trap is completed. 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 and prevent illegal access.

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

It should be noted that the use of I space provides a further form of protection, execute only.

Access Information Bits

A Bit (bit 7) — This bit 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 (i.e. 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 must be saved in their new form and which pages have not been modified and can simply be overlaid.

Note that A and W bits are reset to 0 whenever either PAR or PDR is modified.

Expansion Direction (ED)

This 1-bit field, located at bit 3 of the page descriptor register (PDR), specifies whether the page expands upward from relative zero (ED = 0) or downwards towards relative zero (ED = 1). Relative zero, in this case, is the PAF (Page Address Field). Expansion is done by changing the page length field. In expanding upwards, blocks with higher relative addresses are added; in expanding downwards, blocks with lower relative addresses are added to the page. Upward expansion is usual-

ly used to add more program space, while downward expansion is used to add more stack space.

Page Length Field (PLF)
The 7-bit field, occupying bits 14-8 of the page descriptor register (PDR), specifies the number of blocks in the page. A page consists of at least one and of at most 128 blocks, and occupies contiguous core locations. If the page expands upwards, this field contains the length of the page minus one (in blocks). If the page expands downwards, this field contains 128 minus the length of the page (in blocks).

A length error occurs when the block number (BN) of the virtual address (VA) is greater than the page length field (PLF), if the page expands upwards; or if the page expands downwards, when the BN is less than the PLF.

Reserved Bits
Bits 15, 4, and 5 are reserved for future use, and are always 0.

FAULT RECOVERY REGISTERS

Aborts and traps generated by the memory management hardware are vectored through kernel virtual location 250. Status registers 0, 1, 2, and 3 are used in order to differentiate an abort from a trap, to determine why the abort or trap occurred, and to 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.

Status Register 0 (SRO) (status and error indicators)
SRO contains error flags, the page number whose reference caused the abort, and various other status flags. The register is organized as shown in Figure 7-17.

Bits 15-12 are the error flags. They may be considered to be in a priority queue; 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. When set (error conditions), these bits cause memory management to freeze the contents of bits 1-7 and status registers 1 and 2. This is done to facilitate error recovery. Bits 15-12 are enabled by a signal called RELOC. RELOC is true when an address is being relocated by the memory management unit. This implies that either SR0, bit 0 is equal to 1 (relocation operating) or that SR0, bit 8 (MAINTENANCE)

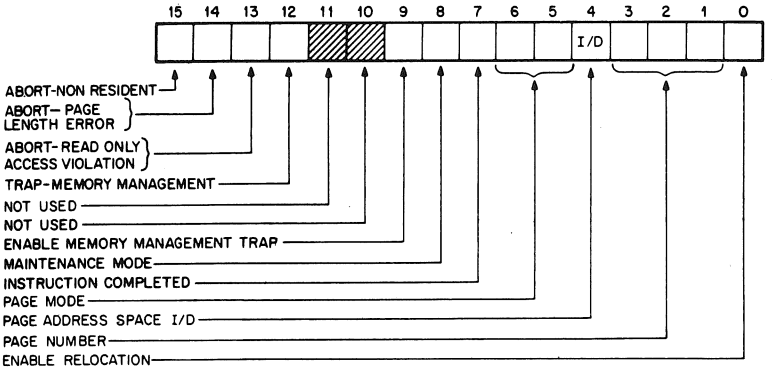


Figure 7-17 Format of Status Register 0 (SR0)

is equal to 1 and the memory reference is the final one of a destination calculation (maintenance/destination mode).

> **NOTE:** Status register 0 (SR0) bits 0, 8, and 9 can be set under program control to provide meaningful control information. However, information written into all other bits is not meaningful. Only that information which is automatically written into these remaining bits as a result of hardware actions is useful as a monitor of the status of the memory management unit. Setting bits 15-12 under program control will not cause traps to occur; however, these bits must be reset to 0 after an abort or trap has occurred in order to resume status monitoring.

Abort Non-resident

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.

Abort Page Length

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 (PLF) of the page descriptor register (PDR) for that page. Bits 14 and 15 may be set simultaneously by the same access attempt.

Abort Read-Only
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 is the trap memory management bit. It is set by a read operation which references a page with an access control field (ACF) of 1 or 4, or by a write operation to a page with an ACF key of 4 or 5.

Bits 11 and 10 are spare locations and are always equal to 0. They are unused and reserved for possible future expansion.

Enable Memory Management Traps
Bit 9 is the enable memory management traps bit. It can be set or cleared by doing a direct write into SR0. If bit 9 is 0, no memory management traps will occur. The A and W bits will, however, continue to log potential memory management traps. When bit 9 is set to 1, the next potential memory management trap will cause a trap vector through kernel virtual address 250.

> **NOTE:** If an instruction which sets bit 9 to 0 (disable memory management trap) causes a potential memory management trap in the course of any of its memory references prior to the one actually changing SR0, then the trap will occur at the end of the instruction anyway.

Maintenance/Destination Mode
Bit 8 specifies maintenance use of the memory management unit. It is provided for diagnostic purposes only and must not be used for other purposes.

Instruction Completed
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 5 and 6 indicate the CPU mode (user/supervisor/kernel) associated with the page causing the abort. (kernel = 00, supervisor = 01, user = 11). If an illegal mode (10) is specified, bit 15 will be set and an abort will occur.

Page Address Space
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 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 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.

Status Register 1 (SR1)
SR1 records are autoincrement/decrement of the general purpose registers, including explicit references through the PC. SR1 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 (in 2's complement notation) by which the register was modified are written into SR1.

The information contained in SR1 is necessary to accomplish an effective recovery from an error resulting in an abort. The low order byte is written first. 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 on R6(SP) that can be caused by the MARK instruction cannot occur if the instruction is aborted.

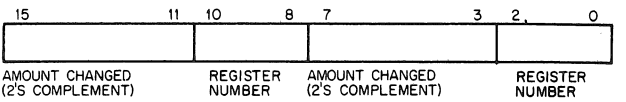


Figure 7-18 Format of Status Register 1 (SR1)

Status Register 2
SR2 is located 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 trap, parity, odd address, and time-out traps. Note that SR2 does not get the trap vector on EMT, TRAP, BPT and IOT instructions. SR2 is read-only; it can not be written into. SR2 is the virtual address program counter.

Status Register 3

Status Register 3 (SR3) enables or disables the use of the D-space PARs and PDRs. 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 registers, and bit 2 to the kernel's registers. When the appropriate bits are set, D space is enabled; when clear, it is disabled. Bits 3-15 are unused. On initialization this register is set to 0 and only I space is in use.

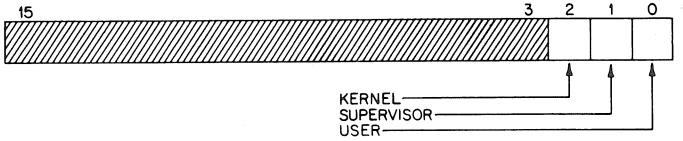


Figure 7-19 Format of Status Register 3 (SR3)

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 (loading a missing page, etc.).
2. Restoring the general purpose registers indicated in SR1 to their original contents at the start of the instruction by subtracting the modify value specified in SR1.
3. Restoring the PC to the abort-time PC by loading R7 with the contents of SR2, 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.

Clearing Status Registers Following Trap/Abort

At the end of a fault service routine bits 15-12 of SR0 must be cleared (set to 0) to resume error checking. On the next memory reference

following the clearing of these bits, the various status registers will resume monitoring the status of the addressing operations, (SR2) will be loaded with the next instruction address, SR1 will store register change information, and SR0 will log memory management status information.

Typical Memory Page

When the memory management unit is enabled, the kernel mode program, a supervisor mode program and a user mode program each have 8 active pages (described by the appropriate page address registers and page descriptor registers) for data; and 8 for instructions. Each segment is made up of from 1 to 128 blocks and is pointed to by the page address field (PAF) of the corresponding page address register (PAR) as illustrated in Figure 7-20.

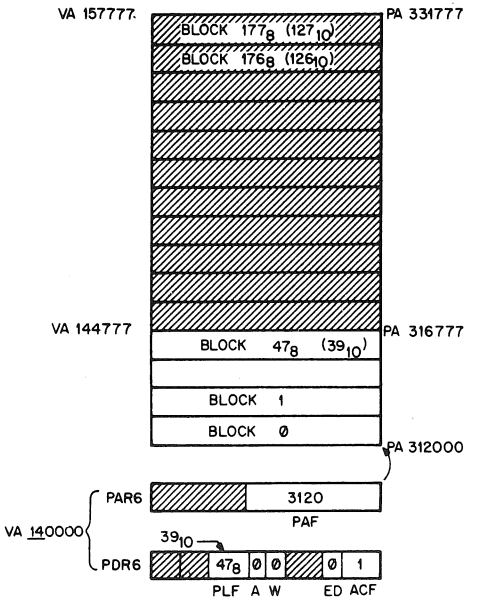


Figure 7-20 Typical Memory Page

The memory segment illustrated in Figure 7-20 has the following attributes:

- page length: 40 blocks
- virtual address range: 140000 — 144777
- physical address range: 312000 — 316777

- No trapped access has been made to this page.
- Nothing has been modified (i.e. written) in this page.
- read-only protection
- upward expansion

These attributes were determined according to the following scheme:

1. Page address register (PAR6) and page descriptor register (PDR6) were selected by the active page field (APF) of the virtual address (VA). (Bits 15-13 of the VA = 68.)
2. The initial address of the page was determined from the page address field (PAF) of APR6 (312000 = 31208 blocks × 408 (3210) 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 (478 + 1 = 4010 blocks) was determined from the page length field (PLF) contained in page descriptor register PDR6. Any attempts to reference beyond these 4010 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 7-21.
5. The access bit (A bit) of PDR6 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 access control field ACF for this page), or a trapped reference i.e., read, in this case, occurs, the A bit will be set to a 1.
6. The written bit (W bit) indicates that no locations in this page have been modified. If an attempt is made to modify any location in this particular page, an access control violation abort will occur. If this page were involved in a disk swapping or memory overlay scheme, the W bit would be used to determine whether it had been modified and thus required saving before it could be overlaid.
7. This page is read-only protected; i.e. 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 (ACF) 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.

Note that 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 (PAR) and page descriptor register (PDR) under program control. In a normal application, it is assumed that the particular page which contains these registers would be assigned to the control of a supervisory type program operating in kernel mode.

Non-Consecutive Memory Pages

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 (PAF) of the PARs in such a way that higher virtual address blocks may be located in lower physical address blocks as illustrated in Figure 7-21.

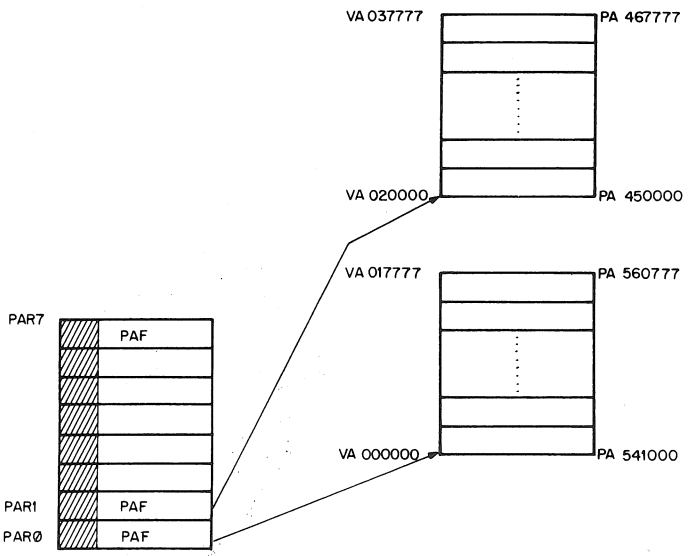


Figure 7-21 Non-Consecutive Memory Pages

Note that although a single memory page must consist of a block of contiguous locations, memory pages as macro units do not have to be located in consecutive physical address locations. It also should be realized that the assignment of memory pages is not limited to consecutive non-overlapping physical address locations.

Stack Memory Pages

When constructing PDP-11/55, 11/45 programs it is often desirable to isolate all program variables from pure core (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. 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 (ED) bit of the appropriate page descriptor register (PDR) to a 1. Figure 7-22 illustrates a typical stack memory page. This page will have the following parameters.

PAR6: PAF = 3120
PDR6: PLF = 1758 or 12510 (12810-3)
ED = 1
A = 0 or 1
W = 0 or 1
ACF = nnn (to be determined by programmer as the need dictates).

NOTE: The A, W bits will normally be set by hardware.

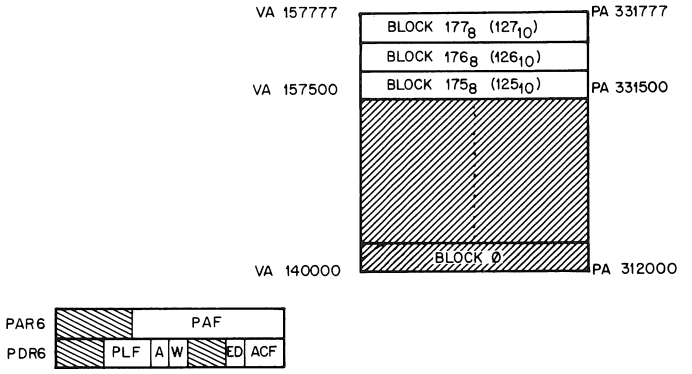


Figure 7-22 Typical Stack Memory Page

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 (BN) from the virtual address is less than the page length field of the appropriate descriptor register.

TRANSPARENT OPERATION OF MEMORY MANAGEMENT

It should be clear at this point that in a multiprogramming application it is possible for memory pages to be allocated in such a way that a particular program seems to have a complete 32K basic PDP-11/55, 11/45 memory configuration. Using relocation, a kernel mode supervisory-type program can easily perform all memory management tasks in a manner entirely transparent to a supervisor or user mode program. In effect, a PDP-11/55, 11/45 system can utilize its resources to provide maximum throughput and response to a variety of users.

MEMORY MANAGEMENT UNIT — REGISTER MAP

REGISTER	ADDRESS
Status Register #0(SR0)	777572
Status Register #1(SR1)	777574
Status Register #2(SR2)	777576
Status Register #3(SR3)	772516
User I Space Descriptor Register (UISDR0)	777600
.	.
.	.
.	.
User I Space Descriptor Register (UISDR7)	777616
User D Space Descriptor Register (UDSDR0)	777620
.	.
.	.
.	.
User D Space Descriptor Register (UDSDR7)	777636
User I Space Address Register (UISAR0)	777640
.	.
.	.
.	.
User I Space Address Register (UISAR7)	777656
User D Space Address Register (UDSAR0)	777660
.	.
.	.
.	.
User D Space Address Register (UDSAR7)	777676

REGISTER	ADDRESS
Supervisor I Space Descriptor Register (SISDR0)	772200
.	.
.	.
.	.
Supervisor I Space Descriptor Register (SISDR7)	772216
Supervisor D Space Descriptor Register (SDSDR0)	772226
.	.
.	.
.	.
Supervisor D Space Descriptor Register (SDSDR7)	772236
Supervisor I Space Address Register (SISAR0)	772240
Supervisor I Space Address Register (SISAR7)	772256


digital pdp11/60
RUN
PROC
USER
CONSOLE
BATT
POWER ON ACTION
BOOT RUN HALT
POWER
LOCK
R1
R2
STD BY
(D)ADRS
EXAM
DEP
HALT/SI
(L)ADRS
CONT
BOOT
(D)SWR
(L)SWR
MAINT
DIAG
CNTRL
START
7
4
5
6
1
2
3
0

CHAPTER 8

PDP-11/60

FEATURES

The PDP-11/60 is at the top of the mid-range PDP-11 processors, and is the most powerful processor described in this handbook. 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. The cache/UNIBUS memory design provides a system-oriented computer that can handle both single and multi-user systems at high speed.

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 of the PDP-11/60 that are explained in detail in this chapter include:

- cache memory system
- memory management
- 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 microprograming capability (described in Chapter 9)

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 MOS or core memory. Cache memory provides for rapid execution of instructions, while the main memory provides cost-effective bulk storage.

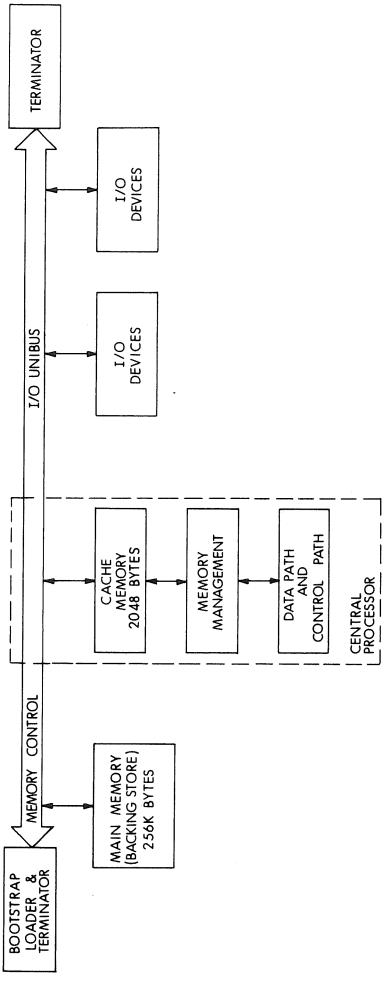


Figure 8-1 Simplified PDP-11/60 System

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 8-2. 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.

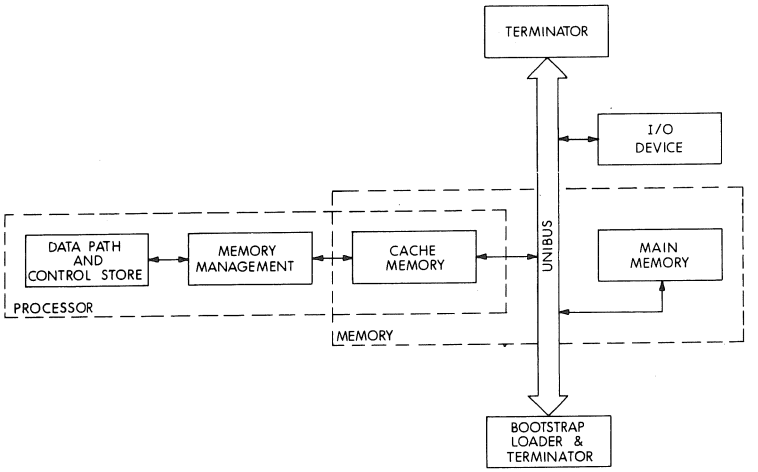


Figure 8-2 Cache Memory System Relationships

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 more available 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 8-1.

Table 8-1 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
Allocatión 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, only 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 8-3 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 throught 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.

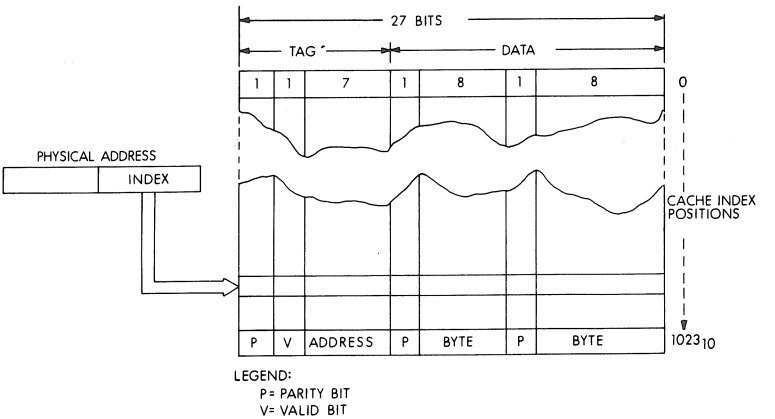


Figure 8-3 Cache Memory Data Format

Physical and Cache Address

Since the physical address space is 128K words, 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 8-4.

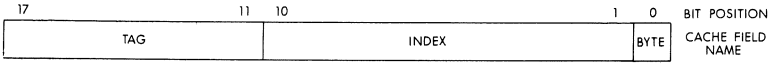


Figure 8-4 Physical Address Format

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 8-3).

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 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 instruc-

tion 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.

In a typical program, writes occur on only 10% of memory references, as compared to 90% for reads. Upon these reads, hits will average 77% to 92%.

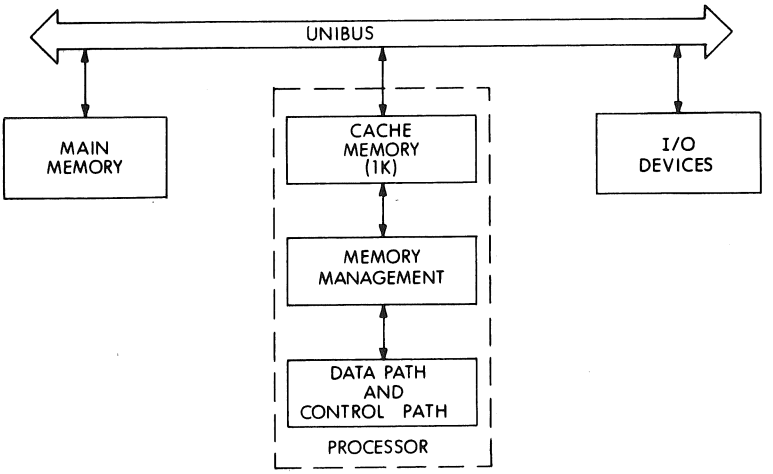


Figure 8-5 Cache Adressing Scheme

Table 8-2 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*	No change
Write (word)		
Hit	Updated**	Written Into
Miss	Allocated*	Written Into
Write (byte)		
Hit	Updated**	Written Into
Miss	No change	Written Into
NPR Operations	**What Happens In Cache**	**What Happen 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***	Written Into
Miss	No change	Written Into

* Allocated — The data and tag of the cache location are changed to correspond to the main memory location.

** Updated — The data in cache is changed to correspond to the data in main memory.

*** 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 8-2).

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 parity 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.

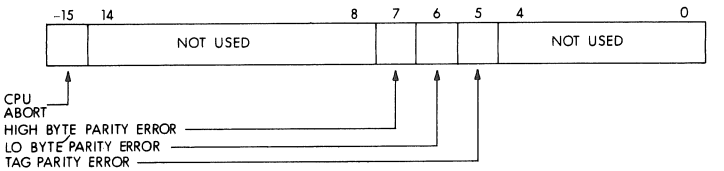


Figure 8-6 Memory System Register 777744

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.

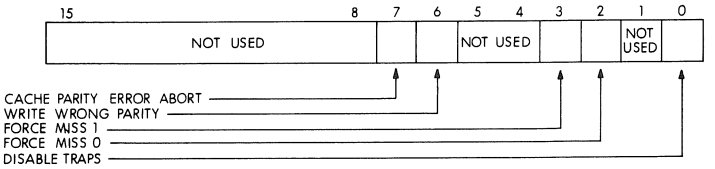


Figure 8-7 Cache Control Register 777746

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.

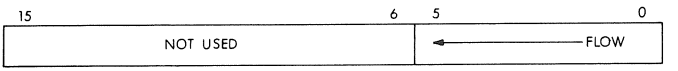


Figure 8-8 Hit/Miss Register 777752

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.

MEMORY MANAGEMENT ON THE PDP-11/60

Unlike the memory management units discussed in the PDP-11/34, 11/45, and 11/55 sections, the memory management (KT11) logic is an integral part of the PDP-11/60 cache memory module. It performs two basic functions:

1. The relocation of virtual memory addresses to physical memory addresses; i.e., the transformation from a symbolic to an absolute addressing scheme
2. Protection of active user programs against unauthorized access

Because the KT11, when enabled, relocates all addresses automatically, the 11/60 may be considered to be operating in a virtual address space. This means that, regardless of where a program is loaded into physical memory, it will *not* have to be re-linked — it always appears to be at the same virtual location in memory.

Memory Relocation

The primary memory management function is to perform memory relocation and provide expanded memory addressing capability for systems with more than 28K of physical memory. The KT11 uses two sets of page address registers to relocate virtual addresses to physical addresses in memory. These sets are used as hardware relocation registers that permit several user programs, each starting at virtual address 0, to reside in physical memory simultaneously.

Program Relocation

The page address registers are used to determine the starting address of each relocated program in physical memory. Figure 8-9 shows a simplified example of the relocation concept.

In Figure 8-9, Program A starting address 0 is relocated by a constant to provide physical address 6400_8.

If the next processor virtual address is 2, the relocation constant will then cause physical address 6402_8, which is the second item of Program A, to be accessed. When Program B is running, the relocation constant is changed to 100000_8. Then Program B virtual addresses starting at 0 are relocated to access physical addresses starting at

100000_8. Using the active page address registers to provide relocation eliminates the need to re-link a program each time it is loaded into a different physical memory location. The program always appears to start at the same address.

In PDP-11/60 systems, a program is relocated in pages. A page can consist of from 1 to 128 blocks. Each block is 32 words in length. Thus, the maximum length of a page is 4096 (128 × 32) words. Using all of the eight available active page registers in a set, a maximum program length of 32,768 words can be accommodated. Each of the eight pages can be relocated anywhere in the physical memory, as long as each relocated page begins on a boundary that is a multiple of 32 words. However, for pages that are smaller than 4K words, only the memory actually allocated to the page may be accessed.

The relocation example shown in Figure 8-9 illustrates several points about memory relocation. These are:

- Although the program appears to the processor to be in contiguous address space, the 32K-word virtual address space is actually scattered through several separate areas of phsical memory. As long as the total available physical memory space is adequate, a program can be loaded. The physical memory space need not be contiguous.
- Pages may be relocated to higher or lower physical addresses with respect to their virtual address ranges. In the example in Figure 8-9, page 1 is relocated to a higher range of physical addresses, page 4 is relocated to a lower range, and page 3 is not relocated at all (even though its relocation constant is non-zero).
- All of the pages shown in the example start on 32-word boundaries.
- Each page is relocated independently. There is no reason why two or more pages could not be relocated to the same physical memory space. Using more than one page address register in the set to access the same space would be one way of providing different memory access rights to the same data, depending upon which part of a program was referencing that data. In the example shown in Figure 8-9, note the relocation constant assigned to pages 4 and 6. As a result, virtual addresses within both address ranges access the same physical addresses in memory, using separate page address registers.

Virtual to Physical Address Conversion

With the KT11 memory management logic as a standard component of the 11/60 cache module, the address output from the data path module cannot be considered as the direct physical address of a memory location. Instead it is viewed as a 16-bit virtual address that

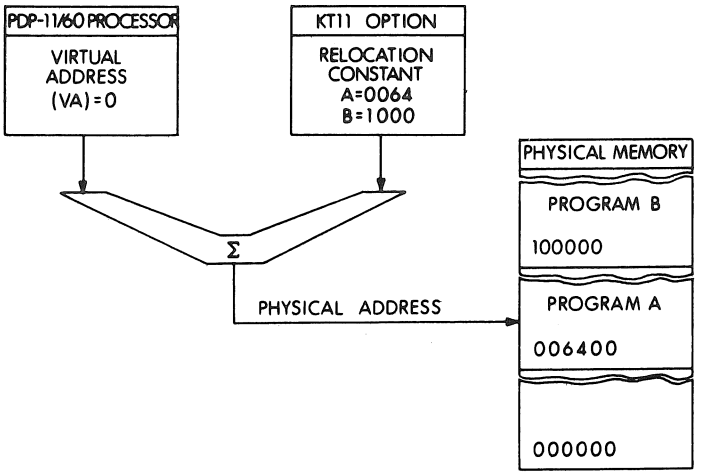


Figure 8-9 Simplified Memory Relocation Example

contains information to be used by the KT11 to construct a 18-bit physical address. (Bit 0 is not used in the physical address configuration to cache.)

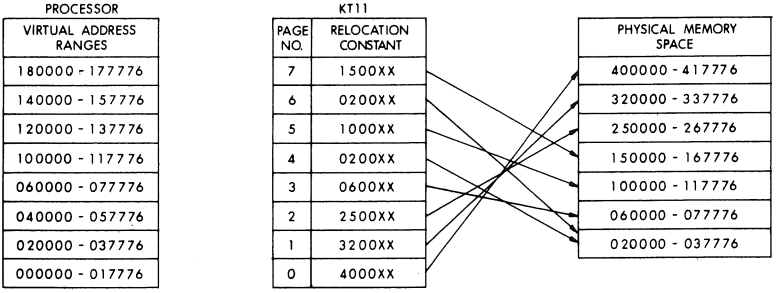


Figure 8-10 Relocation of a 32K-Word Program Into 124K-Word Physical Memory

PDP-11/60 MAIN MEMORY

The 11/60 is available with MOS and core configurations. The use of MOS memory provides the following advantages:

- lower power consumption

- greater packaging density
- faster cycle time
- 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 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.

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. Double 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 a part or all of cache using the force miss bits of the cache control register. Part or all of the read data is brought from the cache; 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 double MOS memory errors.

The error correcting logic will correct the 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 8-3 summarizes cache parity operations.

Table 8-3 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 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 own 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.

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.

> **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.*

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 contents 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 8-4.
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 8-4.
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 8-4, 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 8-4 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
FAC3[3]	013/213
FAC3[4]	014/214
FAC3[5]	015/215
USER R6	016/216
FDST3	017/217

ASPHI: 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

BSPLO: 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

BSPHI: 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
	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 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:

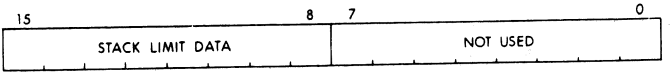


Figure 8-11 Stack Limit Register Format

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.

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 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.

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 10.

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-11/34.

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 AND MAINTENANCE

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 (parity, error status registers, microbreak). These aids are coordinated via the Reliability and Maintenance Program (RAMP).

RAMP is a DIGITAL corporate program the purpose of which is the development of trade-off data for use by DIGITAL's engineering groups in hardware design. Reliability means minimizing failures and maintainability means planning for ease of maintenance and for minimum time spent isolating faults and making repairs.

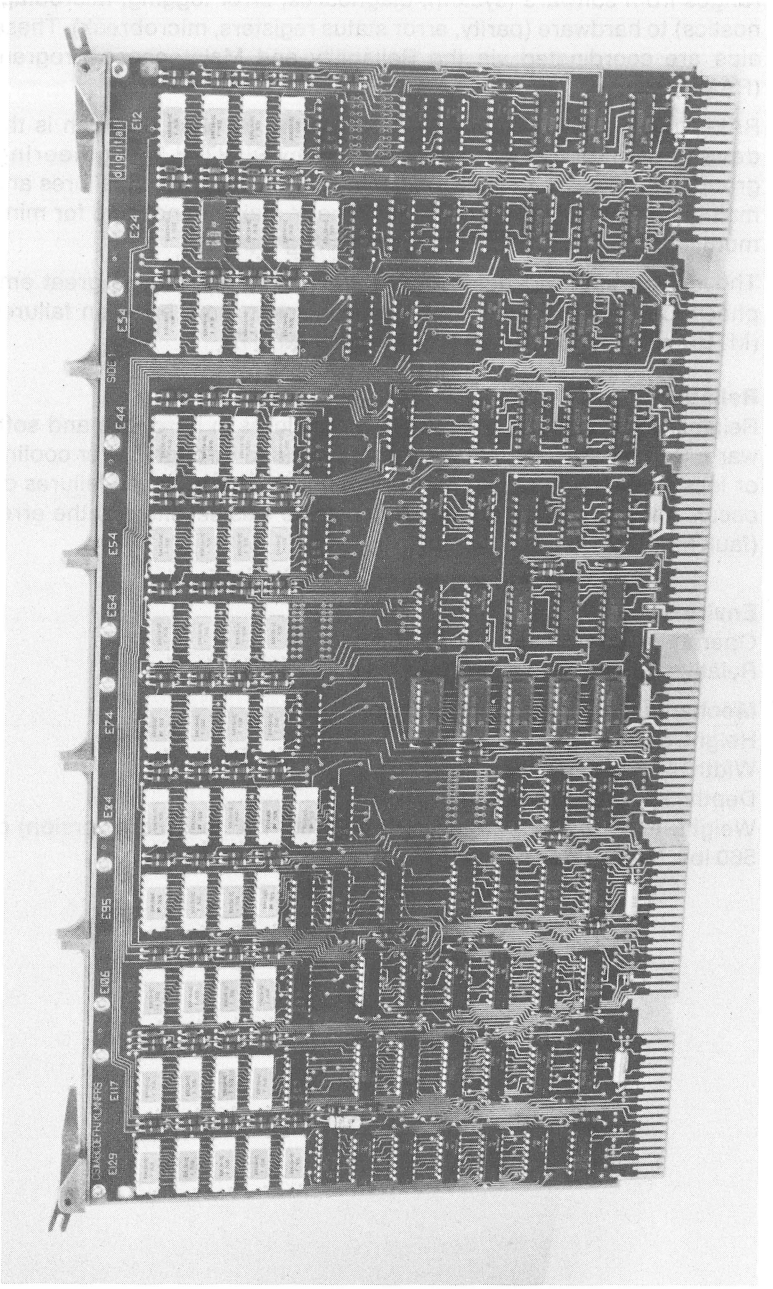
The design and packaging of the PDP-11/60 has placed great emphasis on RAMP. This means reduced mean time between failures (MTBF) and reduced mean time to repair (MTTR).

Reliability
Reliability refers to the minimization of failures in hardware and software. Some hardware failures can be avoided through better cooling or less stress upon components. In other instances, when failures do occur, it is important that the computer be less sensitive to the error (fault tolerant).

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: PDP-11S60, 930 lbs.; PDP-11T60, 710 lbs. (core version) or 560 lbs. (MOS version)



CHAPTER 9

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 micro-progamming 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 USC 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.

MICROPROGRAMMING

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 capatibility 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, technologi-

cal 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 9-1.

The micro-level architecture of the 11/60 is radically different from the standard PDP-11 structure visible to the macro-level programmer. To

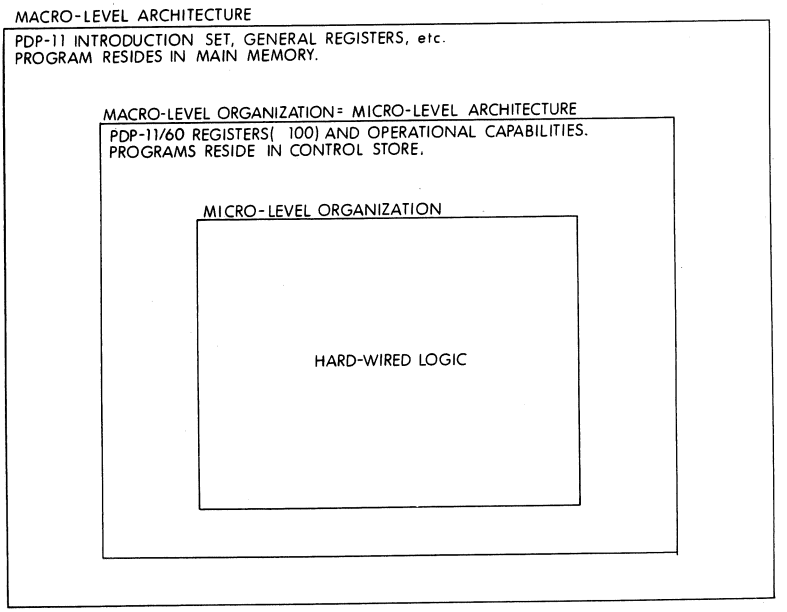


Figure 9-1 Hierarchical Structure of Memories, Architecture, and Organization

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)
- the **WCS** section, which contains additional control store for the user

microprogrammer in the form of a RAM (Random Access Memory). This RAM can also be used as a high-speed local store with the aid of routines stored in the transfer micro store (TMS) ROM.

USER CONTROL STORE OPTION

The principal use of the 11/60 microprocessor is the implementation of the PDP-11 instruction set. However, the processor has been designed with a dynamic control structure so that other functions can be implemented. The UCS option provides additonal and alterable control store for the 11/60, enabling you to extend the capabilities of the PDP-11. Possible applications range from extending the PDP-11 instruction set to emulating a computer with a different instruction set.

The Writable Control Store is a 1-board hardware option for the 11/60 central processor, which includes a 1K-by-48-bit Random Access Memory (RAM). This hardware by itself is not the complete product.

To use the WCS hardware, that is, to do microprogram development and debugging, DIGITAL provides the following software tools:

- the Microprogram Assembler: MICRO-11/60
- the Microprogram Loader: MLD
- the Microprogram Debugging Tool: MDT

MICRO-11/60

The MICRO-11/60 assembler converts microprograms written in its source language to absolute object code. The source language of MICRO-11/60 allows the symbolic definition of fields and macros and the use of these names in specifying the actions to be performed by the microprogram.

The MICRO-11/60 assembler performs two logical functions: translation and address selection. In translating names within a microinstruction to the appropriate set of bits, the assembler also performs valuable syntax and error checking. In assigning addresses, the assembler aids the programmer in laying out branches and in allocating storage in an effective manner.

MICROPROGRAM LOADER

The Microprogram Loader (MLD) performs three functions in loading the Writable Control Store:

- initialization of the Writable Control Store to a special pattern
- loading of the resident section of the Writable Control Store
- loading of the set of object modules that make up the microprogram

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, 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.

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.
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 instruction address are all explicitly identified in each instruction. There is no microprogram counter analogous to the PDP-11 PC.

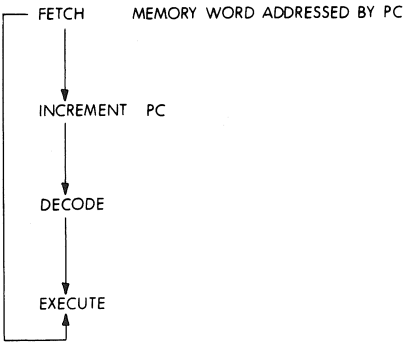
Sequencing and Branching

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 9-2. 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.

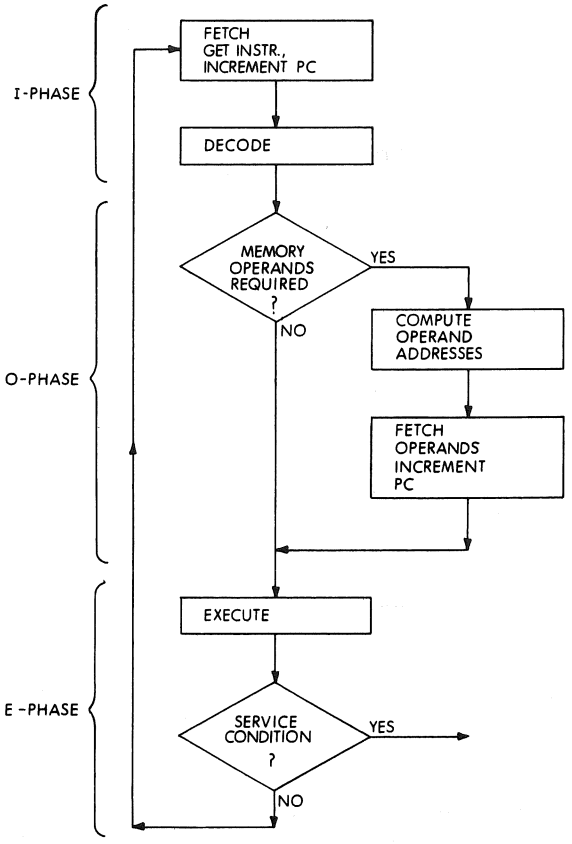


Figure 9-2 Program Flow in the PDP-11/60

CHAPTER 10

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 three FPPs available for the PDP-11 family: the FP11-A, used with the PDP-11/34; the FP11-C, used with the PDP-11/45, 11/55, and 11/70; and the FP11-E, used with the PDP-11/60.

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
- 6 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.

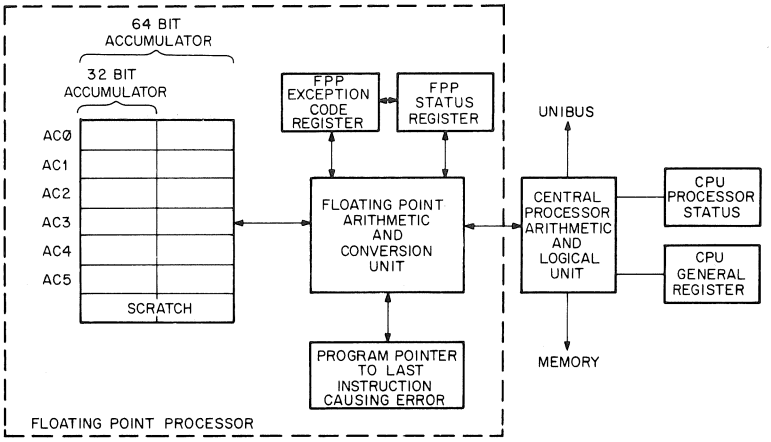


Figure 10-1 Floating Point Processor

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 a 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 a 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)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 octal) notation (i.e., as K+200 octal). Thus exponents from −128 to +127 can be represented by 0 to 377 (octal), or 0 to 255 (decimal). For reasons given below, a biased EXP of 0 (true exponent of −200(octal)) is reserved for floating point zero. Thus exponents are

restricted to the range −127 to +127 inclusive (−117 to 177 (octal)) or, in excess 200(octal) notation, 1 to 377 (octal). 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 (octal) is regarded as producing a floating overflow; if the true exponent is less than −177 (octal) 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

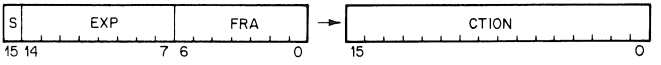
The undefined variable is any bit pattern with a sign bit of one and a biased exponent of zero. The term undefined variable is used to indicate that these bit patterns are not assigned a corresponding floating point arithmetic value. An undefined variable is frequently referred to as "−0" elsewhere in 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.

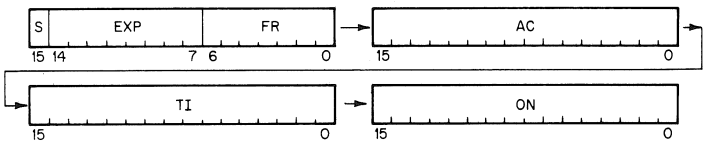
FLOATING POINT DATA

Floating point data is stored in words of memory as illustrated below.

F Format, single precision



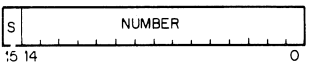
D Format, double precision



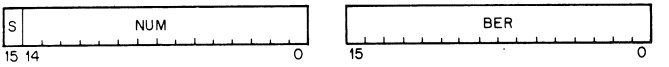
S = Sign of fraction
EXP = Exponent in excess 200 notation, restricted to 1 to 377 octal for non-vanishing numbers.
FRACTION = 23 bits in F Format, 55 bits in D Format, + one hidden bit (normalization). The binary radix point is to the left.

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 FFP 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.

> **NOTES:** 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 (octal), 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 (octal).

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 code 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 arithmetic 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), 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 carries 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 8 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 8 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 octal
- on underflow: if the biased exponent is 0, it is correct. If it is not 0, it is too large by 400 octal.

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 occurence of underflow or overflow.

FLOATING POINT INSTRUCTIONS

Each instruction that references a floating point number can operate 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 integer (FL = 1)or a 16-bit integer (FL=0) is 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, 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
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. If overflow occurs and FIV is not enabled, AC ← exact 0. For all other cases, AC ← SUM.	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: 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 octal 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) 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 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 results in AC. The fractional parts are correctly stored. The exponent part is too small by 400 octal for overflow. It is too large by 400 octal 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: None: 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. 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.	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 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, 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 $\neq -200$, (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:
Special Comment:

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, the FPS is loaded with the source. The user is cautioned not to use bits 12 and 13 (in FP11-C, FP11-E, and the FP11-A) or bit 4 (in the FP11-A) 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 Floing/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) octal
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 (octal), 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 octal (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*1n 2))

   The reduced argument is g*1n2<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 octal 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 Double or from Double to Floating	176AC-FDST	If EXP(AC) = 0, FDST ← 0 and rounding causes overflow, FDST ← exact 0. 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.	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 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 Integer or Long Integer	175(AC + 4)DST	DST ← C_{xj} (AC) if − JL − 1<C×ffi(AC) < JL + 1, else DST ← 0, where C_{xj} specifies conversion from floating mode x to integer mode j: 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	C ← FC ← 0 if − JL − 1 < C×ffi (AC) < JL + 1, else FC ← 1. V ← FV ← 0. Z ← FZ ← 1 if (DST) = 0, else FZ ← 0. 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 octal	C ← FC ← 0. V ← FV ← 0. Z ← FZ ← 1 if (DST) = 0, else 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 (refer to Section 11.6) 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 ← o. 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 octal for overflow. It is too large by 400 octal 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 oper-

ands 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) 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, and FP11-E.

The following table summarizes the floating point execution time of the FP11-A, FP11-E, and FP11-C.

Table 10-1 Comparison of Floating Point Processor Instruction Timing (sec)

Operation (register-to-register)	11/34 FP11-A	11/55/45 FP11-C	11/60 FP11-E
Single Precision			
Add/Subtract	8.91	1.65	1.02.
Multiply	16.2	3.27	1.53
Divide	16.2	4.29	7.00
Double Precision			
Add/Subtract	8.91	1.68	1.02
Multiply	25.36	5.43	3.74
Divide	35.36	6.73	12.75

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 10-2 provides the basic instruction times for mode 0. Tables 10-3 through 10-7 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 10-2 which gives MULF, mode 0, execution time of 13.4 μseconds.
2. Refer to Note 1 as specified in the notes column of Table 10-2. Note 1 specifies an additional 0.84 μseconds is to be added if rounding mode is specified. This yields 14.24 μseconds.
3. The modes 1-7 column of Table 10-2 refers to Table 10-3 to determine the additional time required for mode 1 through 7 instructions. In this example, mode 3 specifies an additional 3 μseconds for single precision yielding 17.34 μseconds.

All timing information is in microseconds unless otherwise noted. Times are typical; processor timing can vary ± 10%.

NOTE: Add .13 μseconds for each memory cycle if MS11-JP MOS memory is utilized. Add .12 μseconds for each DATI memory cycle if memory management is enabled.

Table 10-2 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 10-3
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 10-4
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 10-5
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 10-6
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 10-7
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 10-3 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 10-4 Floating Destination Store Time

Addressing Mode	Memory Cycles		Time(μs)	
	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 10-5 Floating Destination Fetch And Store Time

Addressing Mode	Memory Cycles		Time(μs)	
	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 10-6 Source Fetch Time

Addressing Mode	Memory Cycles		Time(μs)	
	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 10-7 Destination Store Time

Addressing Mode	Memory Cycles		Time(μs)	
	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 μseconds when in rounding mode (FT = 0).
- 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 zeros of the result.

- 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 zeros in the fractional result.

- 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 zeros; for negative integers, the number of shifts required for normalization is equivalent to the number of leading ones.

- 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

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 OP Code 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.

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 10-8
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 0.
(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 450ns must be added to the effective execution 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 10-2 through 10-7.
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 10-8 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 10-9 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

INSTRUCTION	FP11-A	FP11-E	FP11-C
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
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	I4350	
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 10-2 gives a simplified picture of how a load class floating point instruction is executed.

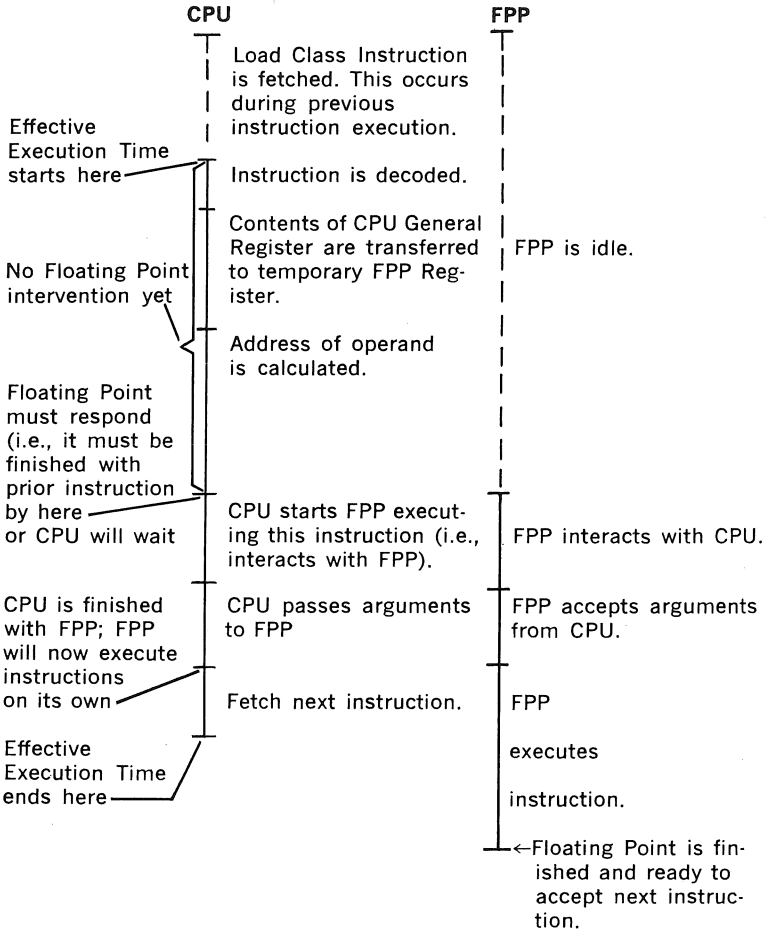


Figure 10-2 Load Class Floating Point Instruction

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.

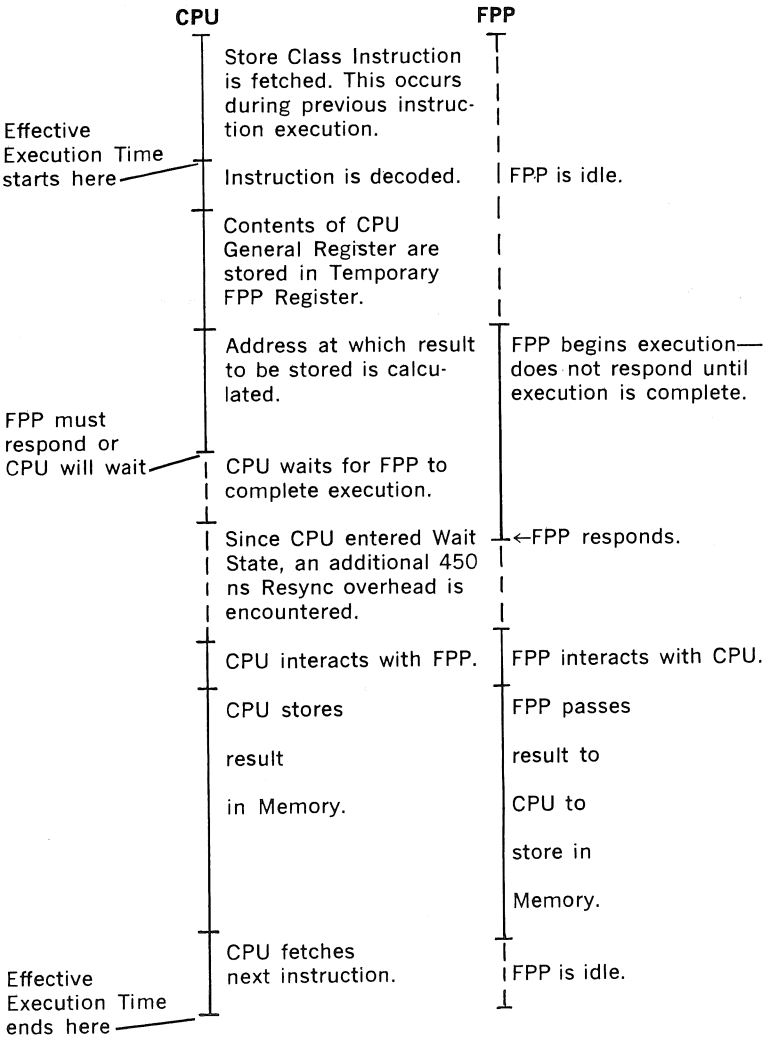


Figure 10- 3 Store Class Floating Instruction

Figures 10-2 and 10-3 show how timing associated with a typical load class and store class instruction is derived.

Figures 10-4 and 10-5 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 10-4, 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 10-8)	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 nsDisengage
and Fetch Time	300 ns
Effective Execution Time	1950

for LDF X(R3),ACLO (Ref. Figure 10-4):

First we calculate Wait Time:

Wait Time = [Floating Point Execution (previous FP instruction)(MULF)]	1800 ns
− [Disengage and Fetch Time (previous FPT instruction)]	− 300
− [Execution time of interposing nonFPT instruction (SOB)]	− 750
− [Preinteraction Time]	− 450
− [Address Calculation (Mode 6 from Table 10-8)]	− 600
	− 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 10-8)	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 instruction 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 10-11 and 10-12.

TERM	DEFINITION
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:
(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 10-13.
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 has received all operands (load class). Refer to Table 10-14.
Effective execution time	Total CPU time required to execute a floating point instruction. Effective execution time = instruction decode + address calculation + wait time + arguement transfer time + shared execution time + disengage and fetch time.

Table 10-10 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

Table 10-10 Floating Point Instructions, cont.

Category	Instruction
NULL	LDFPS STFPS STST CFCC SETF, SETD SETI, SETL

Table 10-11 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 10-12 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 10-13 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 10-14 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	

* Requires CPU shared code execution. For Mode 0 address calculation, add 4 cycles.

	MO			Not MO	
	Min.	Max.	Typical	Min.	Max.
34.NEGD	340		0	0	
35.ABSF	340		0		
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		

Table 10-15 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.	
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 10-16 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 10-17 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 received 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 10-8 and 10-9 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.

for MULF (RO), AC1:

Instruction Fetch	340 nsec
Address Calculation Time (Mode 1 from Table 10-11)	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), ACO (Ref. Figure 10-5):

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 10-11)]	– 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 10-11)	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

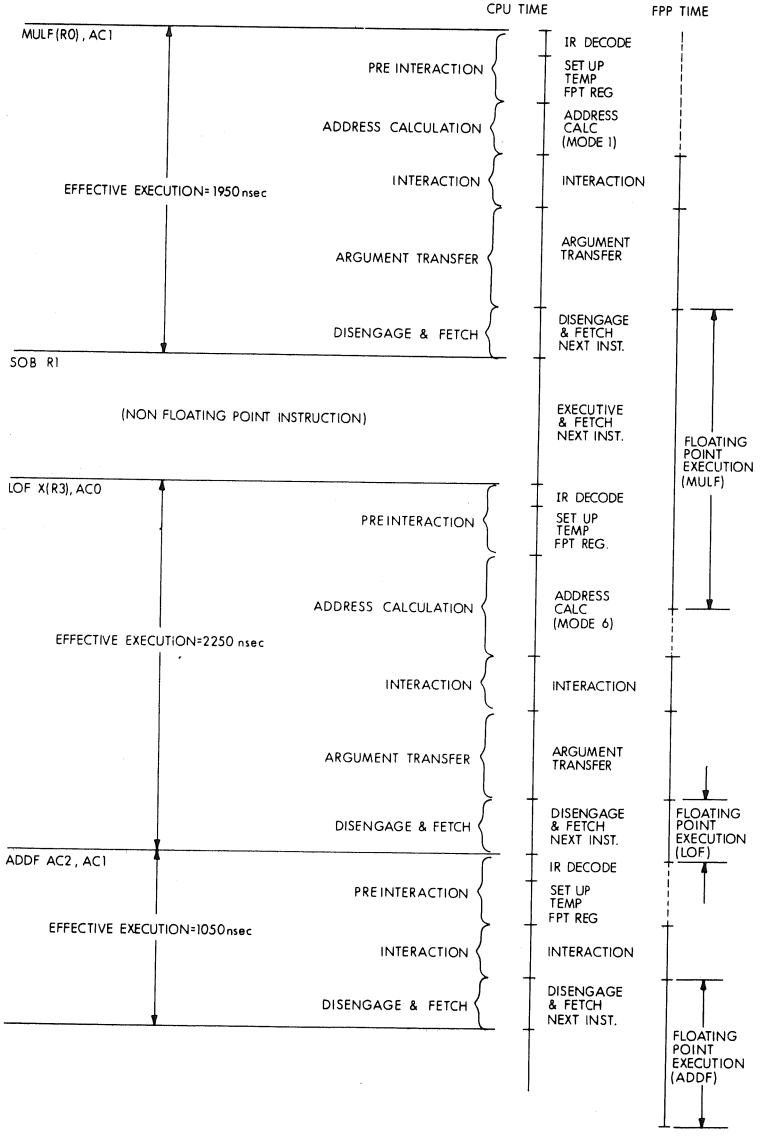


Figure 10-4 Calculation of Effective Execution Times for Load Class Instructions (FP11-C)

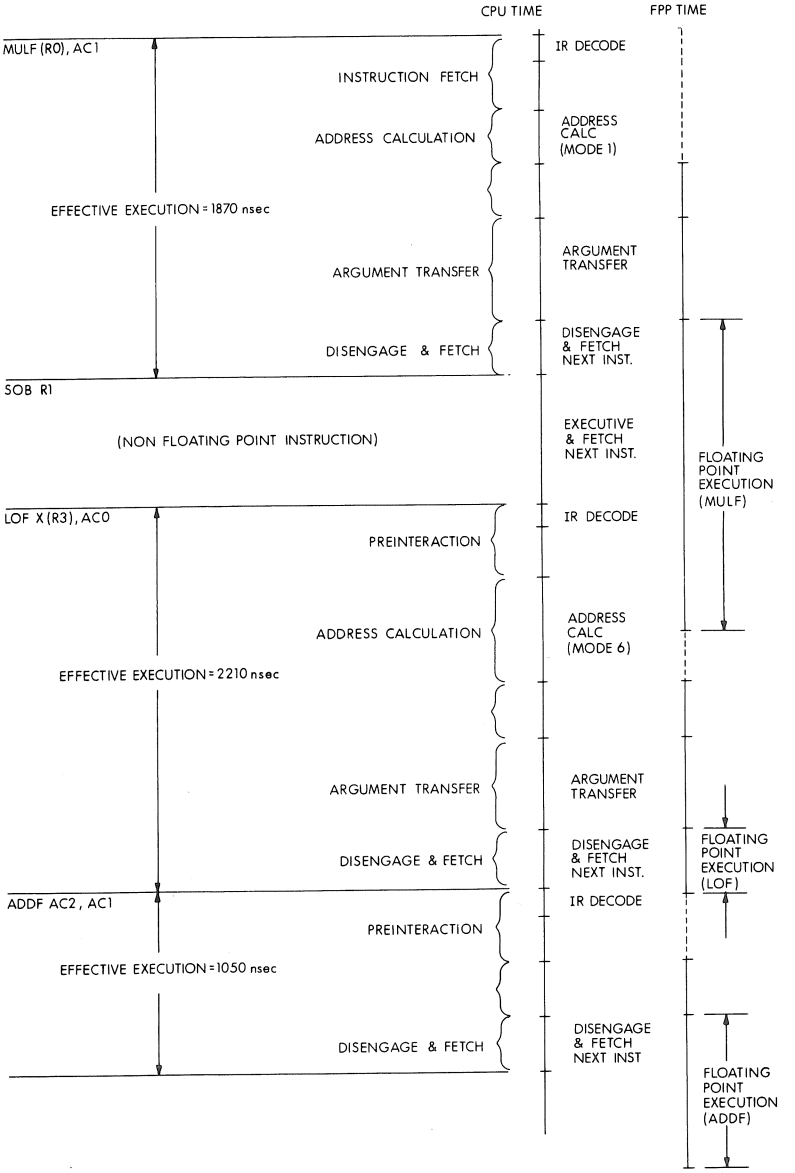


Figure 10-5 Calculation of Effective Execution Times for Load Class Instructions (FP11-E)

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-11/34	777570	68	1
PDP-11/35	777570	68	1
PDP-11/40	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

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.

**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		
776 200	DX11	
776 176		#31
775 610	DL11-C, -D, -E,	#1
775 576		#4
775 400	DS11,	#1

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

777 676 777 660	User Data PAR, reg 0-7
777 656 777 640	User Instruction PAR, reg 0-7
777 636 777 620	User Data PDR, reg 0-7
777 616 777 600	User Instruction PDR, reg 0-7
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

775 376 775 200	DN11,	#16 #1
775 176 775 000	DM11,	#16 #1 (DV11, #1-4)
774 776 774 400	DP11,	#1 #32
774 376 774 000	DC11,	#32 #1
773 766 773 000	BM792, BM873 ROM PDP-11 diagnostic bootstrap (half of it)	
772 776 772 700	PA611 typeset punch	
772-676 772 600	PA611 typeset reader	
772 576		maintenance (AFMR)
772 574	AFC11,	MX channel/gain (AFCG)
772 572		flying cap data (AFBR)
772 570		flying cap status (AFCS)
772 556 772 550	XY11 plotter	
772 546		
772 544		counter
772 542	KW11-P,	count set
772 540		clock status
772 536		
772 534		
772 532		read lines (MTRD)
772 530		tape data (MTD)
772 526	TM11,	memory address (MTCMA)
772 524		byte record counter (MTBRC)
772 522		command (MTC)
772 500		tape status (MTS)
772 516	Memory Mgt reg (MMR3)	
772 476		cont & status #3 (MTCS3)
772 474		bus adrs ext (MTBAE)
772 472		tape control (MTTC)
772 470		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	
772 136 – 772 110	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
```

770 366 770 200	UNIBUS Map	
767 776 766 000	GT40 bootstrap	
765 776 765 000	PDP-11 diagnostic bootstrap (half of it)	User & Special Systems
763 776	(top of floating addresses)	
760 010	(start of floating addresses)	

NOTE

All presently unused UNIBUS addresses are reserved by Digital.

datascope

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)

	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

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

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-4 PDP-11/55, 11/45 CENTRAL PROCESSORS

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. 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 + EF Time
MOV Instruction: Instr Time = SRC Time + EF Time

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.

NOTES

1. The times specified generally apply to Word instructions. In most cases Even Byte instructions have the same times, with some Odd Byte instructions taking longer. All exceptions are noted.
2. Timing is given without regard for NPR or BR servicing. Core memory is assumed to be located within the CPU mounting assembly.
3. If the Memory Management option is installed and operating, instruction execution times increase by .09 μsec for each memory cycle used.
4. All times are in microseconds.

SOURCE ADDRESS TIME

Instruction	Source Mode	SRC Time Bipolar	SRC Time 8K Core	SRC Time 16K Core	Memory Cycles
Double Operand	0	.00	.00	.00	0
	1	.30	.83	.89	1
	2	.30	.83	.89	1
	3	.75	1.81	1.92	2
	4	.45	.98	1.04	1
	5	.90	1.96	2.07	2
	6	.60	1.73	1.86	2
	7	1.05	2.71	2.89	3

DESTINATION ADDRESS TIME

Instruction	DST Mode	DST Time (A) Bipolar	8K Core	16K Core	Memory Cycles
Single Operand and Double Operand (except MOV, MTP, JMP, JSR)	0	.00	.00	.00	0
	1	.30	.83(B)	.86(B)	1
	2	.30	.83(B)	.86(B)	1
	3	.75	1.81(B)	1.92(B)	2
	4	.45	.98	1.04	1
	5	.90	1.96	2.07	2
	6	.60	1.73(B)	1.86(B)	2
	7	1.05	2.71(B)	2.89(B)	3

NOTE (A): Add .15 μsec for odd byte instructions, except DST Mode 0.
NOTE (B): For 8K core, add .07 μsec if SRC Mode = 1-7; for 16K core, add .085 μsec if SRC Mode = 1-7.

EXECUTE, FETCH TIME
Double Operand

Instruction (Use with SRC Time and DST Time)	SRC Mode 0 DST Mode 0 EF Time Bipolar	8K Core	16K Core	Mem Cyc	SRC Mode 1-7 DST Mode 0 ET Time Bipolar	8K Core	16K Core	Mem Cyc	SRC Mode 0 to 7 DST Mode 1 to 7 EF Time Bipolar	8K Core	16K Core	Mem Cyc
ADD, SUB, BIC, BIS	.30 (D)	.90 (C)	.97 (C)	1	.45 (D)	1.05 (E)	1.12 (E)	2	.75	1.82	1.81	2
CMP, BIT	.30 (D)	.90 (C)	.97 (C)	1	.45 (D)	1.05 (E)	1.12 (E)	1	.45	1.13	1.19	1
XOR	.30 (D)	.90 (C)	.97 (C)	1	—	—	—		.75	1.82	1.81	2

NOTE (C): For 8K, add .23 μsec if DST is R7; for 16 K, add .22 μsec if DST is R7.
NOTE (D): Add .3 μsec if DST is R7.
NOTE (E): For 8K, add .23 μsec if DST is R7, add .08 μsec if DST is odd byte and not R7; for 16K, add .65 μsec if DST is odd byte *not* R7.

Double Operand (Cont.)

Instruction (Use with SRC Time)	DST Mode	DST Register	EF Time (SRC MODE = 0) Bipolar	EF Time (SRC MODE = 0) 8K Core	EF Time (SRC MODE = 0) 16K Core	EF Time (SRC MODE = 1-7) Bipolar	EF Time (SRC MODE = 1-7) 8K Core	EF Time (SRC MODE = 1-7) 16K Core	Memory Cycles
MOV	0	0-6	.30	.9	.97	.45	1.05	1.12	1
	0	7	.60	1.13	1.19	.75	1.28	1.34	1
	1	0-7	.75	2.00	2.13	.75	1.95	2.09	2
	2	0-7	.75	2.00	2.13	.75	1.95	2.09	2
	3	0-7	1.20	2.98	3.16	1.20	3.05	3.25	3
	4	0-7	.90	2.15	2.28	.90	2.03	2.16	2
	5	0-7	1.35	3.13	3.31	1.35	3.13	3.31	3
	6	0-7	1.05	2.90	3.09	1.20	3.05	3.25	3
	7	0-7	1.50	3.88	4.13	1.65	4.03	4.28	4

Single Operand

Instruction (Use with DST Time)	DST MODE = 0 EF Time Bipolar	8K Core	16K Core	Memory Cycles	DST MODE 1 to 7 EF Time Bipolar	8K Core	16K Core	Memory Cycles
CLR COM, INC, DEC, ADC, SBC, ROL, ASL, SWAB, SXT	.30 (J)	.90 (G)	.97 (G)	1	.75	1.82	1.81	2
NEG	.75	1.28	1.34	1	1.05	2.10 (F)	1.99 (F)	2
TST	.30 (J)	.90 (G)	.97 (G)	1	.45	1.13	1.19	1
ROR, ASR	.30 (J)	.90 (G)	.97 (G)	1	.75	1.82 (H)	1.81 (H)	2
ASH, ASHC	.75 (I)	1.28 (I)	1.34 (I)	1	.90 (I)	1.43 (I)	1.49 (I)	1

NOTE (F): Add .12 μsec if odd byte.
NOTE (G): For 8K, add .23 μsec if DST is R7; for 16K, add .22 μsec if DST is R7.
NOTE (H): Add .15 μsec if odd byte.
NOTE (I): Add .15 μsec per shift.
NOTE (J): Add .30 μsec if DST is R7.

Single Operand (Cont.)

Instruction (Use with SRC Times)	Bipolar	8K Core	16K Core	Memory Cycles
MUL	3.30	3.83	3.89	1
DIV				
by zero	.90	1.43	1.49	1
shortest	7.05	7.58	7.64	1
longest	8.55	9.08	9.14	1

Instruction	Bipolar	8K Core	16K Core	Memory Cycles	
MFPI	1.05	2.18	2.31	2	use with SRC times
MFPD	1.05	2.18	2.31	2	

Instruction	DST Mode	Instruction Time			Memory Cycles
		Bipolar	8K Core	16K Core	
MTPI MTPD	0	.90	2.03	2.16	2
	1	1.20	2.93	3.13	3
	2	1.20	2.93	3.13	3
	3	1.65	4.03	4.28	4
	4	1.35	3.01	3.19	3
	5	1.80	4.11	4.35	4
	6	1.65	4.03	4.28	4
	7	2.10	5.01	5.32	5

Branch Instructions

Instruction	Instr Time (Branch)			Instr Time (No Branch)			Memory Cycles
	Bipolar	8K Core	16K Core	Bipolar	8K Core	16K Core	
BR, BNE, BEQ, BPL, BMI, BVC, BVS, BCC, BCS, BGE, BLT, BGT, BLE, BHI, BLOS, BHIS, BLO	.60	1.13	1.18	.30	.90	.98	
SOB	.60	1.13	1.18	.75	1.28	1.32	1

Jump Instructions

Instruction	DST Mode	Instr Time Bipolar	Instr Time 8K Core	Instr Time 16K Core	Memory Cycles
JMP	1	.90	1.43	1.49	1
	2	.90	1.43	1.49	1
	3	1.20	2.26	2.37	2
	4	.90	1.43	1.49	1
	5	1.35	2.41	2.52	2
	6	1.05	2.18	2.31	2
	7	1.50	3.16	3.34	3
JSR	1	1.50	2.63	2.76	2
	2	1.50	2.63	2.76	2
	3	1.80	3.46	3.64	3
	4	1.50	2.63	2.76	2
	5	1.95	3.61	3.79	3
	6	1.65	3.38	3.58	3
	7	2.10	4.36	4.61	4

Control, Trap & Miscellaneous Instructions

Instruction	Instr Time Bipolar	Instr Time 8K Core	Instr Time 16K Core	Memory Cycles
RTS	1.05	2.11	2.22	2
MARK	.90	2.03	2.16	2
RTI, RTT	1.50	3.16	3.34	3
SET N, Z, V, C				
CLR, N, Z, V, C	.60	1.13	1.28	1
HALT	1.05	1.58	1.64	0
WAIT WAIT Loop for a BR is .3 μsec.	.45	.45	.45	0
RESET	10ms	10ms	10ms	1
IOT, EMT, TRAP, BRT	2.40	5.08	5.27	5
SPL	.60	1.13	1.19	1
INTERRUPT First Device	2.25	4.95	5.07	4

LATENCY

Interrupts (BR requests) are acknowledged at the end of the current instruction. For a typical instruction execution time of 3 μsec, the average time to request acknowledgement would be one-half this or 1.5 μsec. The worst case (longest) instruction time (Negative Divide with SRC Mode 7) and hence, the longest request acknowledgement would be 12.62 μsec max with 16K core (11.79 μsec with 8K core, and 9.00 μsec with Bipolar).

The Interrupt service time, which is the time from BR request acknowledgement to the fetch of the first subroutine instruction, is 5.44 μsec max with 16K core, 4.95 μsec with 8K core, and 2.25 μsec with Bipolar.

Hence, the total *worst case* time from BR request to begin the fetch of the first service routine instruction is:

	Bipolar	8K Core	16K Core
Normal	11.25	16.74	18.41
Memory Management Operating	11.70	17.19	18.96

The total *average* time for BR request to begin the fetch of the first service routine instruction is:

	Bipolar	8K Core	16K Core
Normal	3.95	8.45	9.30
Memory Management Operating	4.40	8.90	9.75

NPR Latency is 3.5 μsec worst case.

PDP-11/60 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.

Instr Time = SRC Time + DST Time + 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.

B.2 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

FLOATING POINT INSTRUCTION TIMING

Floating point instruction times are calculated in a manner similar to the calculation of CPU instruction timing. However, due to the fact that the FP11-E is a separate processor, calculation of floating point instruction times must take this parallel or independent processing into account.

The following paragraphs provide a description of the method used to calculate effective instruction execution times.

NOTE

Resync and Interaction Times are not considered since handshaking synchronization overhead has been eliminated by the use of decoding and instruction fetch logic. That is, instruction fetch of floating point 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).

(continued)

V -Z

NOTES

NOTES

(please fold here)

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY
IF MAILED IN THE UNITED STATES

Postage will be paid by:

DIGITAL EQUIPMENT CORPORATION
PRODUCT PROMOTION GROUP
PK3-2/M18
MAYNARD, MASS. 01754

(staple here)

PDP-11/04/34/45/55/60 HANDBOOK READER'S COMMENTS

Your comments and suggestions will help us in our continuous effort to improve the quality and usefulness of our handbooks.

What is your general reaction to this manual? (format, accuracy, completeness, organization, etc.) _____

_____

_____

_____

What features are most useful? _____

_____

_____

_____

Does the publication satisfy your needs? _____

_____

_____

What errors have you found? _____

_____

_____

_____

Additional comments _____

_____

_____

_____

Name _____

Company _____ Dept. _____

Title _____

City _____ State _____ Zip _____

(staple here)

digital

DIGITAL EQUIPMENT CORPORATION, Corporate Headquarters: Maynard, Massachusetts 01754, Telephone (617) 897-5111—SALES AND SERVICE OFFICES; UNITED STATES—ALABAMA, Birmingham and Huntsville • ARIZONA, Phoenix and Tucson • CALIFORNIA, Los Angeles, Oakland, Sacramento, San Diego, San Francisco, Santa Ana, Santa Barbara, Santa Clara, Sunnyvale • COLORADO, Denver • CONNECTICUT, Fairfield and Meriden • DISTRICT OF COLUMBIA, Washington, D.C. (Lanham, MD) • FLORIDA, Miami, Orlando, Tampa • GEORGIA, Atlanta • HAWAII, Honolulu • ILLINOIS, Chicago, Peoria, Rolling Meadows • INDIANA, Indianapolis • IOWA, Bettendorf • KENTUCKY, Louisville • LOUISIANA, New Orleans • MASSACHUSETTS, Springfield and Waltham • MICHIGAN, Detroit • MINNESOTA, Minneapolis • MISSOURI, Kansas City and St. Louis • NEBRASKA, Omaha • NEW HAMPSHIRE, Manchester • NEW JERSEY, Cherry Hill, Fairfield, Princeton, Somerset • NEW MEXICO, Albuquerque • NEW YORK, Albany, Buffalo, Long Island, Manhattan, Rochester, Syracuse • NORTH CAROLINA, Charlotte and Durham/Chapel Hill • OHIO, Cincinnati, Cleveland, Columbus, Dayton • OKLAHOMA, Tulsa • OREGON, Portland • PENNSYLVANIA, Harrisburg, Philadelphia (Blue Bell), Pittsburgh • RHODE ISLAND, Providence • SOUTH CAROLINA, Columbia • TENNESSEE, Knoxville and Nashville • TEXAS, Austin, Dallas, El Paso, Houston • UTAH, Salt Lake City • VIRGINIA, Richmond • WASHINGTON, Seattle • WEST VIRGINIA, Charleston • WISCONSIN, Milwaukee • INTERNATIONAL—ARGENTINA, Buenos Aires • AUSTRALIA, Adelaide, Brisbane, Canberra, Melbourne, Perth, Sydney • AUSTRIA, Vienna • BELGIUM, Brussels • BOLIVIA, La Paz • BRAZIL, Rio de Janeiro and Sao Paulo • CANADA, Calgary, Edmonton, Halifax, London, Montreal, Ottawa, Toronto, Vancouver, Winnipeg • CHILE, Santiago • DENMARK, Copenhagen • EGYPT (A.R.E.), Cairo • FINLAND, Espoo • FRANCE, Lyon, Paris, Puteaux • HONG KONG • INDIA, Bombay • INDONESIA, Djakarta • IRAN, Tehran • IRELAND, Dublin • ISRAEL, Tel Aviv • ITALY, Milan, Rome, Turin • JAPAN, Osaka and Tokyo • MALAYSIA, Kuala Lumpur • MEXICO, Mexico City • NETHERLANDS, Amstelveen, Rijswijk, Utrecht • NEW ZEALAND, Auckland and Christchurch • NORTHERN IRELAND, Belfast • NORWAY, Oslo • PUERTO RICO, San Juan • SINGAPORE • SOUTH KOREA, Seoul • SPAIN, Madrid • SWEDEN, Gothenburg and Stockholm • SWITZERLAND, Geneva and Zurich • TAIWAN, Taipei • UNITED KINGDOM, Birmingham, Bristol, Ealing, Epsom, Edinburgh, Leeds, Leicester, London, Manchester, Reading • VENEZUELA, Caracas • WEST GERMANY, Berlin, Cologne, Frankfurt, Hamburg, Hannover, Munich, Nurnberg, Stuttgart • YUGOSLAVIA, Belgrade and Ljubljana •

PRINTED IN USA EB 09430 20/78 030 02 80

DIGITAL EQUIPMENT CORPORATION, Corporate Headquarters: Maynard, Massachusetts 01754, Telephone (617) 897-5111—SALES AND SERVICE OFFICES; UNITED STATES—ALABAMA, Birmingham and Huntsville • ARIZONA, Phoenix and Tucson • CALIFORNIA, Los Angeles, Oakland, Sacramento, San Diego, San Francisco, Santa Ana, Santa Barbara, Santa Clara, Sunnyvale • COLORADO, Denver • CONNECTICUT, Fairfield and Meriden • DISTRICT OF COLUMBIA, Washington, D.C. (Lanham, MD) • FLORIDA, Miami, Orlando, Tampa • GEORGIA, Atlanta • HAWAII, Honolulu • ILLINOIS, Chicago, Peoria, Rolling Meadows • INDIANA, Indianapolis • IOWA, Bettendorf • KENTUCKY, Louisville • LOUISIANA, New Orleans • MASSACHUSETTS, Springfield and Waltham • MICHIGAN, Detroit • MINNESOTA, Minneapolis • MISSOURI, Kansas City and St. Louis • NEBRASKA, Omaha • NEW HAMPSHIRE, Manchester • NEW JERSEY, Cherry Hill, Fairfield, Princeton, Somerset • NEW MEXICO, Albuquerque • NEW YORK, Albany, Buffalo, Long Island, Manhattan, Rochester, Syracuse • NORTH CAROLINA, Charlotte and Durham/Chapel Hill • OHIO, Cincinnati, Cleveland, Columbus, Dayton • OKLAHOMA, Tulsa • OREGON, Portland • PENNSYLVANIA, Harrisburg, Philadelphia (Blue Bell), Pittsburgh • RHODE ISLAND, Providence • SOUTH CAROLINA, Columbia • TENNESSEE, Knoxville and Nashville • TEXAS, Austin, Dallas, El Paso, Houston • UTAH, Salt Lake City • VIRGINIA, Richmond • WASHINGTON, Seattle • WEST VIRGINIA, Charleston • WISCONSIN, Milwaukee • INTERNATIONAL—ARGENTINA, Buenos Aires • AUSTRALIA, Adelaide, Brisbane, Canberra, Melbourne, Perth, Sydney • AUSTRIA, Vienna • BELGIUM, Brussels • BOLIVIA, La Paz • BRAZIL, Rio de Janeiro and Sao Paulo • CANADA, Calgary, Edmonton, Halifax, London, Montreal, Ottawa, Toronto, Vancouver, Winnipeg • CHILE, Santiago • DENMARK, Copenhagen • EGYPT (A.R.E.), Cairo • FINLAND, Espoo • FRANCE, Lyon, Paris, Puteaux • HONG KONG • INDIA, Bombay • INDONESIA, Djakarta • IRAN, Tehran • IRELAND, Dublin • ISRAEL, Tel Aviv • ITALY, Milan, Rome, Turin • JAPAN, Osaka and Tokyo • MALAYSIA, Kuala Lumpur • MEXICO, Mexico City • NETHERLANDS, Amstelveen, Rijswijk, Utrecht • NEW ZEALAND, Auckland and Christchurch • NORTHERN IRELAND, Belfast • NORWAY, Oslo • PUERTO RICO, San Juan • SINGAPORE • SOUTH KOREA, Seoul • SPAIN, Madrid • SWEDEN, Gothenburg and Stockholm • SWITZERLAND, Geneva and Zurich • TAIWAN, Taipei • UNITED KINGDOM, Birmingham, Bristol, Ealing, Epsom, Edinburgh, Leeds, Leicester, London, Manchester, Reading • VENEZUELA, Caracas • WEST GERMANY, Berlin, Cologne, Frankfurt, Hamburg, Hannover, Munich, Nurnberg, Stuttgart • YUGOSLAVIA, Belgrade and Ljubljana •

PRINTED IN USA EB 09430 20/78 030 02 80